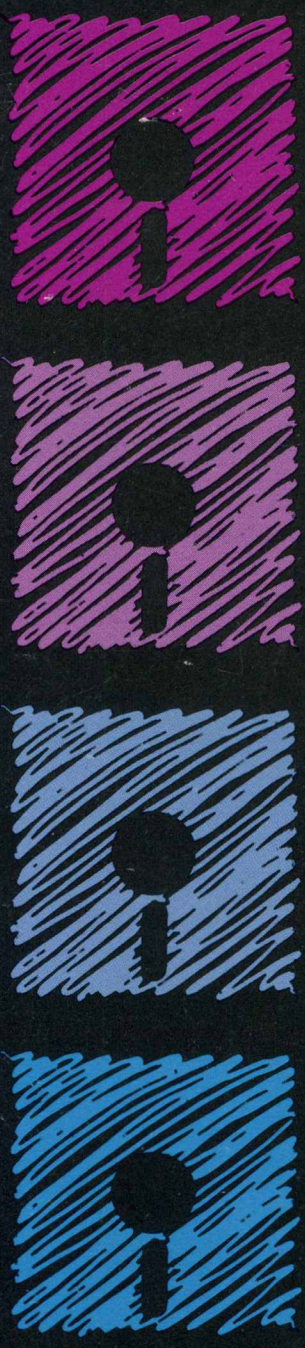
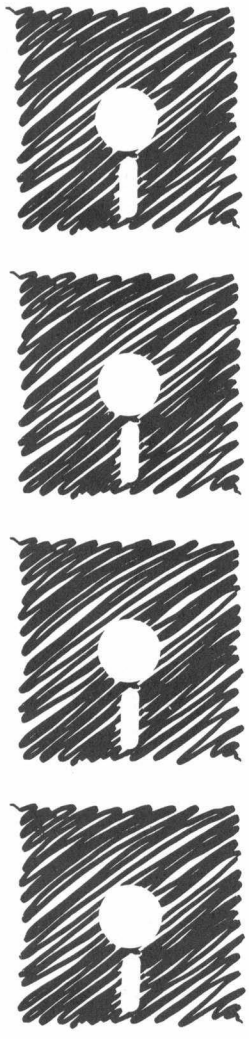
Dr. Dobb's Journal



VOLUME 10

Software Tools

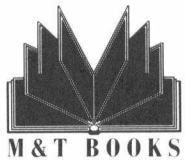
Dr. Dobb's Journal



VOLUME

10

Software Tools



A Publication of M&T Publishing, Redwood City, CA

Acknowledgments

We have always used this space to thank our authors, programmers all, who contribute their time, knowledge, and source code for the edification of other programmers. They deserve all our gratitude: to broadcast hard-earned knowledge like seeds on the wind is extraordinary generosity. We of *Dr. Dobb's* staff are pleased to be the breeze in the metaphor.

Bound Volume Staff

Production Coordinator:
Ann Roskey

Production:
Joe Sikoryak

Cover Design:
Michael Hollister

Index:
Rhoda Simmons

COPYRIGHT © 1986 by M&T Publishing Inc. All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

Printed in the United States of America

About M&T Publishing and *Dr. Dobb's Journal*

M&T Publishing was founded in 1981 as the subsidiary of Markt & Technik A.G., the leading West German publisher of computer magazines, books, and software. In addition to its active role in the international microcomputer marketplace, M&T also publishes two monthly magazines. *Business Software* was aquired in 1983, and in 1984 M&T acquired the rights to publish *Dr. Dobb's Journal* from People's Computer Company.

Since its first issue in 1976, *Dr. Dobb's Journal* has maintained a unique voice among microcomputer publications. The first issue was published to provide short-term distribution of the newly written Tiny BASIC language. Reader response was so strong that it went immediately into monthly production. *DDJ* has led the way since then by focusing on important advances in microcomputers, printing public-domain software, and fostering vital reader interaction on technical matters.

Today, *Dr. Dobb's Journal* remains the prime source of information and software tools for advanced programmers. Its early issues are still in demand, and no part of the editorial content of *DDJ* has ever gone out of print. Volumes One through Ten of this series contain the entire editorial content of *DDJ* 's first ten years and are available in bound volumes through M & T Publishing.

Through *Dr. Dobb's Journal* and its other software publications, M&T Publishing is providing microcomputer professionals and enthusiasts with the most advanced and useful software information available.

Editor's Preface

It was the year of living dangerously.

This book is a collection of the complete editorial content of *Dr. Dobb's Journal of Software Tools* for one year – its tenth year, 1985. It was an exciting year, and a year in which the magazine pulled the beards of some giants.

Noticing that Apple had not put enough memory in the Macintosh, we showed the world how to add more. While we were at it, we also beat Apple to the punch with a hard-disk SCSI interface for the Mac.

We dared to point out in detail how a much-praised implementation of Pascal deviated from the language standard. We challenged the Unix establishment with plans for a free Unix. We asked hard questions about privacy and control in the Information Age.

But mostly we published tools for programmers. That's what *Dr. Dobb's Journal of Software Tools* has been doing since the personal computer revolution began in 1975. In this volume you'll find exhaustive reviews of programmers' editors and C compilers. You'll find thousands of lines of useful source code in C, Modula-2, Pascal, Forth, assembly language, and Prolog. And you'll read professional programmers' shoptalk about the most important issues in programming.

Michael Swaine, editor-in-chief
Dr. Dobb's Journal of Software Tools

Contents

VOLUME 10, NUMBER 99, 1985

4 Editorial
5 Letters
8 Dr. Dobb's Clinic D. E. CORTESI
11 Fatten Your Mac TOM LAFLEUR and SUSAN RAAB
17 QuickDraw Meets ImageWriter THOM MAYER
24 Archiving Files with CP/M-80 and CP/M-86 IAN E. ASHDOWN
42 MBOOT and MODEM7 for the C-64's CP/M WALT PIOTROWSKI
57 Unstructured Forth Programming RICHARD WILTON
59 Software Reviews
APL*PLUS/PC STEFAN H. UNGER
Tools Financial & Statistical STEFAN H. UNGER
VSI—Virtual Screen Interface RONALD G. PARSONS
Fancy Font System, The DAVID D. CLARK
73 Book Reviews
75 16-Bit Software Toolbox RAY DUNCAN
85 The Software Designer MICHAEL SWAINE
87 Of Interest R. P. SUTHERLAND

VOLUME 10, NUMBER 100, 1985

92 Editorial
93 Letters
98 Dr. Dobb's Clinic D. E. CORTESI
103 C/Unix Programmer's Notebook A. SKJELLUM
106 Festschrift for Doctor Dobb SUZANNE RODRIGUEZ, TOM PITTMAN, BOB ALBRECHT
110 Fire in the Valley PAUL FREIBERGER and MICHAEL SWAINE
117 Tiny BASIC for the 68000 GORDON BRANDLY
121 An Enhanced ADFGVX Cipher System C. E. BURTON
140 More dBASE Tips and Techniques GENE HEAD
143 Software Reviews
Modula-2/86 MICHAEL SCHMIDT
155 The Software Designer MICHAEL SWAINE
156 CP/M Exchange ROBERT BLUM
164 Book Reviews
166 Of Interest R. P. SUTHERLAND

VOLUME 10, NUMBER 101, 1985

172 Editorial
173 Letters
174 C Chest ALLEN HOLUB
187 Realizable Fantasies RICHARD STALLMAN
191 Programming in Logic JOHN MALPAS
196 A Tour of Prolog D. E. CORTESI
210 Tax Advisor DEAN SCHLOBOHM
234 16-Bit Software Toolbox RAY DUNCAN
241 CP/M Exchange ROBERT BLUM
247 Software Reviews
SAVVY PC STEVE KING
250 Of Interest R. P. SUTHERLAND

VOLUME 10, NUMBER 102, 1985

256 Editorial
257 Letters
258 Realizable Fantasies MICHAEL SWAINE and BOB ALBRECHT
260 Dr. Dobb's Clinic D. E. CORTESI
264 MONTY Plays Scrabble TERRY A. WARD
269 Shortcut to SCISTAR: For Prowriter Users AMER W. NELSON
273 fx80char: A Character Editor for Epson FX-80 Printers DAVID D. CLARK
293 A Magic Mushroom for the Epson Alice GARY B. PALMER
305 Let's Mouse Around VINCENT and RONALD G. PARSONS
310 Software Reviews
Turbo Toolbox KARL R. KACHIGAN
SPSS/PC STEVEN ANTHONY SOLA
314 The Software Designer MICHAEL SWAINE
316 C Chest ALLEN HOLUB
323 16-Bit Software Toolbox RAY DUNCAN
328 CP/M Exchange ROBERT BLUM
333 Of Interest R. P. SUTHERLAND

VOLUME 10, NUMBER 103, 1985

340 Editorial
341 Letters
345 Dr. Dobb's Clinic D. E. CORTESI
350 Realizable Fantasies D. E. CORTESI
353 C Chest ALLEN HOLUB
360 Using Decision Variables in Graphics Primitives THOM HOGAN
367 Solid Shape Drawing on the Commodore 64 RICHARD RYLANDER
389 A Compiler Written in Prolog G. A. EDGAR
399 Software Reviews
Toolworks C/80 D. C. SHOEMAKER
Disk Maker I JIM KRONMAN
AMPRO Little Board RICHARD CONN
BetterBASIC MATTHEW TRASK
413 16-Bit Software Toolbox RAY DUNCAN

VOLUME 10, NUMBER 104, 1985

422 Editorial
423 Letters
428 Dr. Dobb's Clinic D. E. CORTESI
436 C Chest ALLEN HOLUB
446 Unix Exchange AXEL SCHREINER
452 Information Age Issues DEAN GENGLE
456 Modems: 2400 Bit/Sec and Beyond DALE WALSH
458 C UART Controller DON GAY
463 Christensen Protocols in C DONALD KRANTZ
479 Software Reviews
Envoy DAVID W. CARROLL
481 16-Bit Software Toolbox RAY DUNCAN
489 CP/M Exchange ROBERT BLUM
495 Of Interest ALEX RAGEN

VOLUME 10, NUMBER 105, 1985

502 Editorial
503 Letters
505 Dr. Dobb's Clinic D. E. CORTESI
510 C Chest ALLEN HOLUB
524 Build a Custom PC or Clone JIM KRONMAN
527 The Ultimate Parallel Print Spooler DON RINDSBERG
534 Designing a Real-Time Clock for the S-100 Bus ALAN D. WILCOX
559 Realizable Fantasies MICHAEL SWAINE and BOB ALBRECHT
560 16-Bit Software Toolbox RAY DUNCAN
572 Computer Calisthenics MICHAEL WIESENBERG
578 Of Interest MICHAEL SWAINE

VOLUME 10, NUMBER 106, 1985

586 Editorial
587 Letters
590 Dr. Dobb's Clinic D. E. CORTESI
594 C Compilers for MS-DOS C SPECIAL INTEREST GROUP and RICHARD RELPH
609 A Peephole Optimizer for Assembly Language Source Code DAVID L. FOX
623 Small-C Update J. E. HENDRIX
629 Asynchronous Protocols DAVID W. CARROLL
633 C Chest ALLEN HOLUB
648 16-Bit Software Toolbox RAY DUNCAN
652 CP/M Exchange ROBERT BLUM

VOLUME 10, NUMBER 107, 1985

658 Editorial
659 Letters
660 Dr. Dobb's Clinic D. E. CORTESI
664 C Chest ALLEN HOLUB
671 Unix Exchange AXEL SCHREINER
678 The Software Designer MICHAEL SWAINE
680 Parallel Pattern Matching and Fgrep IAN ASHDOWN
698 Bose-Nelson Sort JOE CELKO
704 Two TEX Implementations for the IBM PC RICHARD FURUTA and PIERRE A. MACKAY
711 Mac Toolbox JOHN BASS
721 CP/M Exchange ROBERT BLUM
723 16-Bit Software Toolbox RAY DUNCAN
730 Of Interest ALEX RAGEN

VOLUME 10, NUMBER 108, 1985

736 Editorial
737 Letters
738 Dr. Dobb's Clinic D. E. CORTESI
743 C Chest ALLEN HOLUB
759 A Threaded-Code Microprocessor Bursts Forth LEO BRODIE
765 Design of a Forth Target Compiler H. ROBINSON, P. MORSE II, S. BOWHILL
785 Software Reviews
Neon BRUCE HORN
MacFORTH ALAN CLUTE
791 Mac Toolbox JOHN BASS
794 CP/M Exchange ROBERT BLUM

VOLUME 10, NUMBER 109, 1985

804 Editorial
805 Letters
808 Dr. Dobb's Clinic D. E. CORTESI
816 Modula-2 versus Pascal for Microcomputers: An Update DAVID W. CARROLL
822 Bit Manipulation in Modula-2 BRIAN R. ANDERSON
827 The Software Designer PAUL HECKEL
833 Software Reviews
Programming Editors MARK U. EDWARDS
849 16-Bit Software Toolbox RAY DUNCAN
855 CP/M Exchange ROBERT BLUM
868 Of Interest ALEX RAGEN

VOLUME 10, NUMBER 110, 1985

874 Editorial
875 Letters
877 Dr. Dobb's Clinic D. E. CORTESI
879 C Chest ALLEN HOLUB
889 Windowing Operating Environments MICAHEL SWAINE
893 BANKSWAP ALBERT S. WOODHULL
904 Adding a COPY Command to ProDOS SHAWN DAY
911 CP/M-68K Conditional Batch Processing ROGER E. DONAIS
918 ProDOS and the RAM Disk ALFRED STEELE
925 16-Bit Software Toolbox RAY DUNCAN
929 Mac Toolbox JOHN BASS
931 CP/M Exchange W. E. WILSON

938 Index

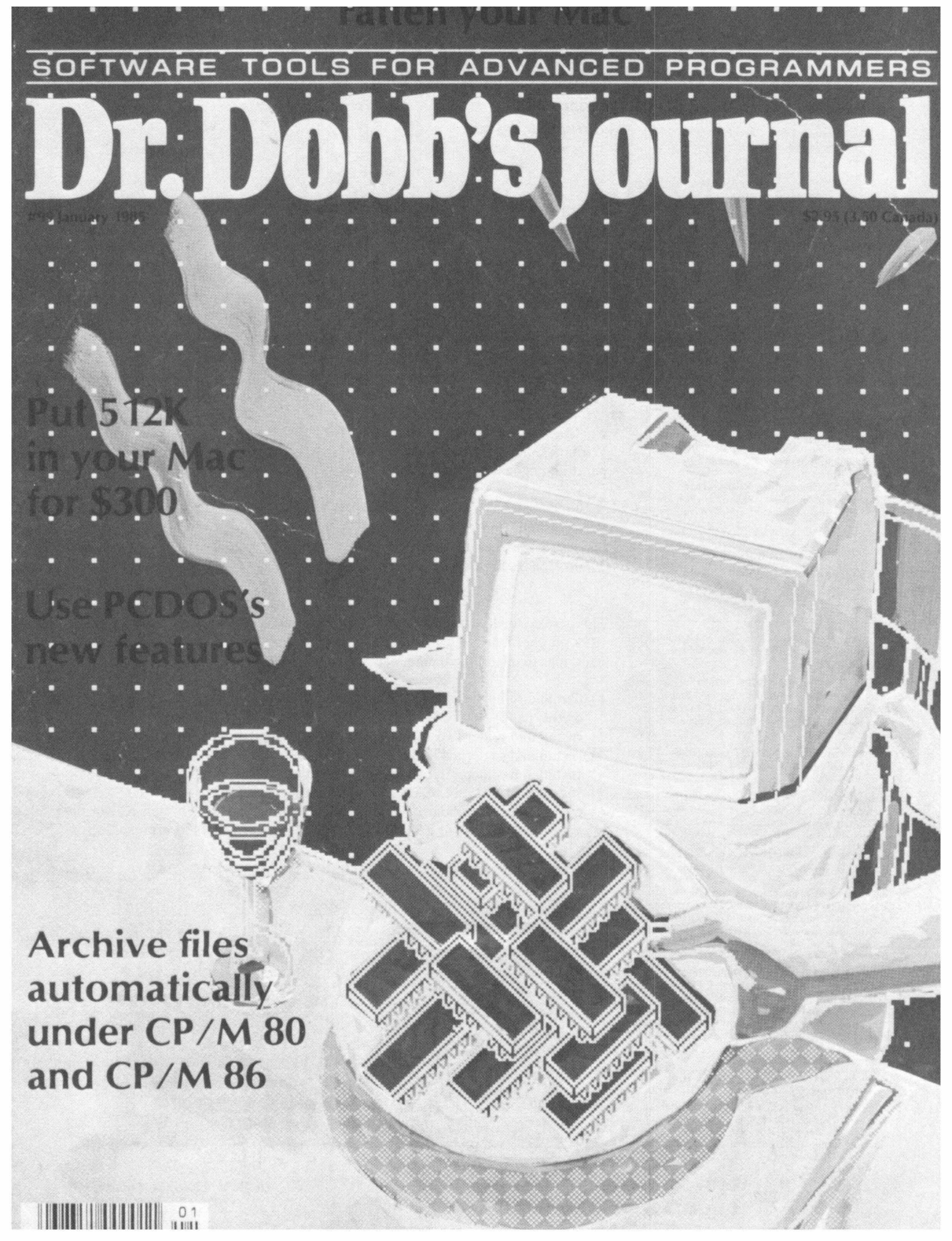
SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

#99 January 1985 $2.95 ($3.50 Canada)

Put 512K in your Mac for $300

Use PCDOS's new features



Archive files automatically under CP/M 80 and CP/M 86

Dr. Dobb's Journal

Editorial

Editor-in-Chief	*Michael Swaine*
Managing Editor	*Randy Sutherland*
Assistant Editor	*Frank DeRose*
Contributing Editors	*Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors	*Polly Koch, Cindy Martin*
Typesetter	*Jean Aring*
Editorial Intern	*Mark Johnson*

Production

Design/Production Director	*Detta Penna*
Art Director	*Shelley Rae Doeden*
Production Assistant	*Alida Hinton*
Cover	*William Cone*

Advertising

Advertising Director	*Stephen Friedman*
Advertising Sales	*Walter Andrzejewski, Shawn Horst, Beth Dudas*
Advertising Coordinators	*Alison Milne, Lisa Boudreau*

Circulation

Circulation and Promotions Director	*Beatrice Blatteis*
Fulfillment Manager	*Stephanie Barber*
Direct Response Coordinator	*Maureen Snee*
Promotions Coordinator	*Jane Sharninghouse*
Circulation Assistant	*Kathleen Boyd*

M&T Publishing, Inc.

Chairman of the Board	*Otmar Weber*
Director	*C.F. von Quadt*
President	*Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal *is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.*

January 1985
Volume 10, Issue 1

Newcomers

We are fortunate to have found a part time technical editor in Alex Ragen, a senior systems analyst and project leader at Tandem Computers, Inc. Alex's extensive programming experience and writing ability help us to ensure the accuracy and readability of the editorial content of *DDJ*. Another newcomer is Frank DeRose, assistant editor. Frank is an experienced writer and teacher. He was using the Unix system at U.C. Berkeley to compose his dissertation before we lured him away from academia to assist the Doctor.

This Month's Cover

William Cone, this month's cover artist, used a Macintosh drawing with underpainting in gouache and acrylic to depict this hungry Mac.

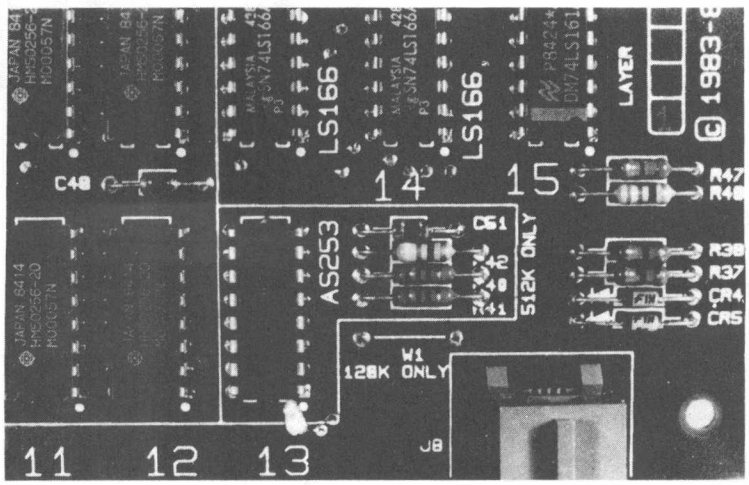
This Month's Referees

Robert Blum, Contributing Editor
David Cortesi, Resident Intern
Thom Hogan, Editor-in-Chief, Business Software

Mac Addendum

This arrived too late to make it into the "Fat Mac" article. If you have the newer 128K motherboard, when you get to page 20 you will need to follow these instructions to install the memory select logic.

1) Cut and remove jumper W1 (marked 128K only) located at G14 (see photo on this page)
2) Install and solder two 2.2K ¼ w resistors at locations R40 and R41
3) Install and solder a 47 ohm ¼ w resistor at location R42
4) Install and solder a .luf capacitor at location C51
5) Install a 74AS253N IC into the socket at location G13
6) Check the motherboard for any solder splashes, or broken etch. Clean the board with alcohol or a TG degreaser to remove any flux left after soldering



SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal

ARTICLES

Fatten Your Mac — 18
by Thomas Lafleur and Susan Raab
Step by step instructions to increase the RAM in a Macintosh to 512K; diagrams, photographs, and warnings are included **(Reader Ballot No. 191)**

QuickDraw Meets ImageWriter — 26
by Thom Mayer
How QuickDraw stores graphics and how ImageWriter prints graphics **(Reader Ballot No. 192)**

Archiving Files with CP/M 80 and CP/M 86 — 36
by Ian E. Ashdown
A utility that determines which files on a disk have been changed and automatically backs them up **(Reader Ballot No. 193)**

MBOOT and MODEM7 for the C-64's CP/M — 62
by Walt Piotrowski
All you need to download CP/M software **(Reader Ballot No. 194)**

Unstructured Forth Programming: An Introduction — 86
by Richard Wilton
A modest proposal **(Reader Ballot No. 195)**

COLUMNS

Dr. Dobb's Clinic — 12
by D.E. Cortesi
Review of *A C Reference Manual*, structured search, ragged patches, Z80 assembly riddle **(Reader Ballot No. 190)**

16-Bit Software Toolbox — 108
by Ray Duncan
PCDOS Version 3.0, graphics routines for IBM PC **(Reader Ballot No. 196)**

The Software Designer — 124
by Michael Swaine
Interview with Tom Evslin of Solutions Inc.; designing for the Mac: some experiences **(Reader Ballot No. 197)**

DEPARTMENTS

Editorial 6

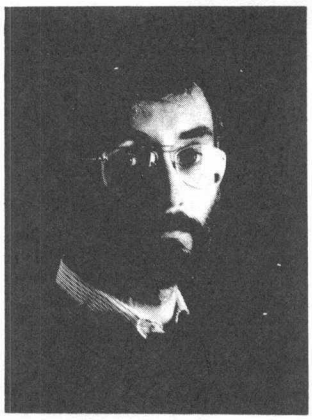
Letters 8

Software Reviews 88 — STSC's APL Interpreter, VSI, Fancy Font

Book Reviews 106

Of Interest 126 **(Reader Ballot No. 198)**

EDITORIAL



I'd been looking forward to this conference for months and wasn't about to let the power failure darken or the rain dampen my spirits. From the cheerful arguments going on around me as we stood dripping in line outside the dining hall, I gathered that others shared my enthusiasm. The organizers of the Hackers' Conference had brought a hundred-odd insanely great programmers and a few writers to this isolated stretch of Marin headlands to discuss the future of the hacker ethic, and discuss the future of the hacker ethic we would; the weather was irrelevant.

The first session dispelled any illusion that we would find consensus on the meaning of the hacker ethic, outside of its having something to do with programming for the joy of it, and nothing to do with breaking into systems. There was a wide span of opinion about the commercial aspect of software development, from the belief that all software should be free to a what-the-traffic-will-bear attitude. Bill Atkinson, admitting that he wanted his QuickDraw routines for the Mac to remain proprietary for a while, looked uneasy when Lee Felsenstein encouraged the attendees to develop, in anarchic concert, a reverse-engineered, non-proprietary Hackers' Mac. Bob Wallace, whose shareware pay-if-you-like-it plan for PC Write was keeping food on his table, seemed amused by the debate. But there was consensus on several significant points: that hacking was worthy in and of itself, that the attendees ought to stay in communication with one another and that it was important to pass the torch to the next generation of hackers. That those present shared something worthy of passing on to others. At the end of the conference I walked out to the ocean and stood on the rocks above the surf, savoring the newfound sense of rational community, as though I had been through a Woodstock for grownups.

Three days later, in Las Vegas, cesspool of the American spirit, I stop midaisle to consult my Comdex program, but a heavy diet of suites has thickened my discriminative faculties until all the booths look alike, nor does the program identify the players, 80 thousand jaded innocents flown in from the undifferentiated heartland to dance the shuck and jive in tight suits, unsensible shoes, flat borrowed jargon. There are, an anonymous passerby pontificates as I pore through the program book, some Interesting Products here this year, though Nothing Revolutionary; this passing summary strikes me as the true smug theme of this year's Comdex. Some fashion war has been won, it seems: the pros, the working press, the veterans, having traded in last year's cynicism for smartmoney conservatism, even the quest for innovation now out of style, pose midaisle, sniffing out the Interesting Products.

One area still open to innovation, I suggest hopefully that evening in some smoky Comdex suite, is software marketing; I'm thinking of shareware, and Bob Wallace, who seems to be everywhere, smiles. The next day I meet Ramon Zamora midaisle in the MGM Grand and he explains that his new company is soliciting grant money to develop shareware, adding that "we call it 'public sector' software."

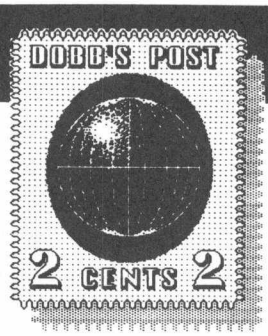
In another suite Lee Felsenstein elaborates on the philosophy behind the Hackers' Mac. It is a mistake to think of it as a product, he explains; it's a tendency; an appeal to the hobbyist willingness to try things out for no reward but the chance to learn. I promise to mention the idea in *Dr. Dobb's*.

Back in the office, I check with Randy on the update to the fatten-your-Mac article and the status of our in-house public-domain software bulletin board. Return a call from Bob Albrecht, who's excited about some plan to lure kids into programming in C. Read in Business Week that Steve Jobs thinks "[i]f Apple falters, innovation will cease." And smile.

Michael Swaine

Michael Swaine

LETTERS



Spleen Ventilations

Dear *Dr. Dobb's*,
One very annoying thing that has come up recently is the holy war involving computer languages. It was never more apparent than in your October 1984 issue in the column "The Software Designer." Typical were the statements of Philippe Kahn, who was quoted as saying "C is a disease," and "When I see people writing spreadsheets in C, I think 'They're out of their minds.' It was designed to write operating systems," and "In Europe C is considered an American disease."

Really? C is a disease? How? Why does the fact that C's first job was operating systems disqualify it from spreadsheets? Forth was originally written to manipulate radio receivers. Does this mean that *Dr. Dobb's* should have rejected the article that demonstrated using Forth for the Fast Fourier Transform? *All* of Europe hates C? There hasn't been that much unanimity since WWII.

Now it may be true that C has some fatal flaw that will doom it to die, dinosaur-like, in some tar pit of abandoned operating systems. But Kahn certainly hasn't demonstrated it, and he has merely made himself look foolish by making statements that have no correlation with his thesis. And the sad thing is that he is not the only person doing this.

One of the most irresponsible statements I have ever read was that of Edsger W. Dijkstra, in his "How Do We Tell Truths That Might Hurt?" It read: "It is practically impossible to teach good programming to students who have had prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."

Now, this is patently untrue—some of the best programmers I know started out self-taught in BASIC and write very clear, structured code. What I assume he meant, in all that rhetoric, was "BASIC is not a good programming language"—which is true in my opinion. But look at that quote again. I'll wager that there were hundreds of mediocre computer science teachers who leapt for joy when they read that. BASIC is implemented on almost every system, which means that it is hard to avoid coming across it and learning it. Having a hard time teaching that student certain programming practices? Well, he must have learned some BASIC—must be he's mentally mutilated. Might as well give up on him; he'll only flunk anyway.

If all we get are diatribes and spleen ventilations, what chance is there for the future of program language design? When a computer scientist merely rants instead of reasons, how seriously can one take claims of the designers of a new langauge?

It is incredibly annoying reading such nonsense because there are valid and rational reasons for not using a programming language. I personally will not use Forth, because I do not like reverse Polish notation, and because it is too close to assembly language for me to be comfortable with it. But I certainly will not condemn those who use it, and if they write good, efficient programs with it, more power to them.

Perhaps *DDJ* could run a series of interviews with software designers, discussing what language they use and emphasizing the positive. It might let people see more clearly just what they might be missing.

Thank you for a very interesting magazine.

John M. Gamble
4471 Eastwood Dr.
#18103
Batavia, OH 45103

grep.c

Dear *DDJ*,
Thanks for your wonderful public domain contribution of grep.c in the October issue. It has become a well-worn tool in my MSDOS 2.0 programming toolbox. There was an error in the program listing that you might warn your readers about to avoid many hours of "bug" hunting. At the middle of page 61 there is a call to *omatch* contained in a *while* statement passing only two parameters when three are required. The line should be:

```
while ( *lin && omatch(&lin,
    pat, boln) )
```

Thanks again for the grep.c utility.

Sincerely,
Michael H. Cox
3659 Gas Light Curve #5
Montgomery, AL 36116

Dear *DDJ*:
It was with interest in learning more about structures and pointers in C that I studied Allen Holub's article on grep.c in the October 1984 issue. Perhaps you would be interested in knowing about an error in Listing Two, page 64. In the *while* loop of the *dodash* module there is a Boolean test on whether *dstart* − *dest* < *maxccl*. The last statement of the module is *return* (*dest–dstart*);. Can you tell that one of these is out of order? It turns out that if the former expression is adjusted to read *dest* − *dstart* < *maxccl*, then *dodash* will not try to expand a character class to more than *maxccl* characters.

You might think this bug is innocuous, but with my particular setup, this turns out to be fatal to the expansion of sets, like [a − e], to character classes, like [abcde]. I think what's happening is that my compiler sees the pointers *dstart* and *dest* as unsigned, so that

when they are out of order the difference becomes a large unsigned value instead of a small signed value. This makes the *while* stop after the first pass! For the record, I use CI C86 on an IBM PC.

Signed,
Scott D. Thomas
354 Colorado Avenue
Palo Alto, CA 94306

Complex Numbers

Dear *DDJ*:
After publication of my article "Simple Calculations with Complex Numbers" in the October 1984 issue of *DDJ*, Mr. Alan Tracht of Cleveland Heights, OH, noticed problems with the routine Polar. For certain arguments, it would return incorrect results. He was kind enough to call me with suggestions for correcting the errors and also told me about a less cumbersome method for preventing floating-point underflow and overflow. A section of the revised program is presented in the Listing (page 10).

The CONSTANT section in the listing is a slightly revised version of the constants from the implementation section of Listing One in the original article. The new version of Polar is much simpler since it passes on most of the hard work to a new FUNCTION called ATan2. This function is a Pascal counterpart to a function found in most Fortran libraries. It takes two arguments, the first a dividend and the second a divisor, and returns the arctangent of the quotient. Because of the extra information inherent in the two arguments, the ATan2 function can correctly handle right angles and determine the correct quadrant of the result. It is tailor made for conversions to polar coordinates.

During testing of the new function, I discovered that the UCSD version of ATan is not quite as "bullet-proof" as I had thought. It was necessary to add some protection for that function as noted in the listing. Such extra protection was not needed by Turbo Pascal or any of the C or Fortran compilers I used to check the new ATan2 algorithm.

The ATan2 function calls another new function, SafeDivide, which attempts to divide its first argument by the second. This function employs the overflow and underflow checking method suggested by Mr. Tracht. It is much less expensive in terms of execution time than the original, which made calls to the natural logarithm function. If anyone has an allergic reaction to the GOTOs, they can be replaced by a copy of the division statement that they jump to. It just takes more space and is redundant.

Finally, I noticed a typographical error I made in the text of the article. In equation 6 on page 31, the calculation of the amplitude is given as taking the arctangent of the real coefficient divided by the imaginary coefficient. In fact, one must take the arctangent of the imaginary coefficient divided by the real coefficient. The order of the dividend and divisor in the original (and new) listing is correct, however.

I hope the erroneous Polar routine did not greatly inconvenience anyone. I would also like to thank Mr. Tracht for his helpful suggestions.

Sincerely,
David D. Clark
246 S. Fraser St. #2
State College, PA 16801

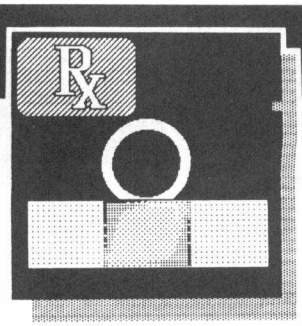
Letters Listing (Text begins on page 8)

```
CONST MAX_REAL   = 9.99999999999999999999E+37;
      PI         = 3.14159265358979323846;          { pi }
      PI_OVER_2  = 1.57079632679489661923;          { pi/2.0 }
      BIG_SQRT   = 1.0E+19;                         { Sqrt(MAX_REAL) }
      CLOSEST    = 1.0E-19;                         { Sqrt(MIN_REAL) }


FUNCTION SafeDivide(x, y : Real) : Real;
{ divide x by y and intercept incipient underflow or overflow }

LABEL 1;

BEGIN { SafeDivide }
  IF Abs(y) < 1.0 THEN                      { overflow possible }
     IF Abs(x) > Abs(y)*MAX_REAL THEN
        SafeDivide := MAX_REAL              { catch overflow }
     ELSE
        GOTO 1
  ELSE IF Abs(x) < 1.0 THEN                 { underflow possible }
     IF Abs(y) > Abs(x)*MAX_REAL THEN
        SafeDivide := 0.0                   { catch underflow }
     ELSE
        GOTO 1
  ELSE
1:   SafeDivide := x/y                      { a normal calculation }
END { of SafeDivide };
```

```
FUNCTION ATan2(x, y : Real) : Real;
{ take arctangent of x/y and account for right angles and proper quadrant }

VAR   a : Real;

BEGIN { ATan2 }
  IF y = 0.0 THEN              { check for exceptional cases with y = 0 }
     IF x = 0.0 THEN           { if both arguments are zero, return zero }
        a := 0.0
     ELSE
        a := PI_OVER_2         { otherwise it's a right angle }
  ELSE BEGIN
     a := SafeDivide(x, y);    { try the divide, preventing unseemly errors }
     IF a >= MAX_REAL THEN     { probably tried to overflow }
        a := PI_OVER_2
     ELSE BEGIN
        IF a <> 0.0 THEN BEGIN

          { kludge to protect UCSD ATan }
          a := Abs(a);
          IF a > BIG_SQRT THEN
             a := PI_OVER_2
          ELSE IF a < CLOSEST THEN
             a := 0.0
          ELSE
             a := ATan(a);
       END;
       IF y < 0.0 THEN         { choose upper or lower half of plane }
          a := PI - a
    END
  END;
  IF x < 0.0 THEN              { choose left or right half of plane }
     a := -a;
  ATan2 := a
END { of ATan2 };
PROCEDURE Polar{Arg: Complex; VAR Modulus, Amplitude: Real};

BEGIN { Polar }
  WITH Arg DO BEGIN
       IF Abs(Re) < CLOSEST THEN
          Re := 0.0;
       IF Abs(Im) < CLOSEST THEN
          Im := 0.0;
       Modulus := Sqrt(Sqr(Re) + Sqr(Im));
       Amplitude := ATan2(Im, Re)
  END { of WITH Arg }
END { of Polar };
```

End Listing

DR. DOBB'S CLINIC



by D. E. Cortesi

The compiled and analyzed results of your throughput measurements aren't quite ready for this January column as we promised. Next month.

We C A Good Book

A C Reference Manual, by Samuel P. Harbison and Guy L. Steele, Jr. (Prentice-Hall, 1984; $19.95) was published just too late to be mentioned in the C Resource List of a few months back. That's a pity because it is the best C resource we've come across yet.

The book is just what its title implies: a complete, authoritative, and (so far as we can tell) accurate reference to the C language. It is beautifully organized, with many small topics grouped logically into chapters. Each topic ends with a list in small type of all the other topics that are related to it, so that no matter where you start you can follow the threads of a concept throughout the book. Differences between C compilers are covered as they occur, matters of coding style and dangerous constructs are discussed, and there are many examples, mostly illuminating.

That it's useful and accurate is a credit to its authors, but that the book is so thorough is the result of its genesis: the authors say it "grew out of our effort to write a family of C compilers." They found that, " In spite of C's popularity . . . there was no description of C precise enough to guide us in designing the new compilers . . . [and none] precise enough for our programmer-customers, who would be using compilers that analyzed C programs more thoroughly than was the custom." So they compiled this one, and a good job they made of it.

Structured Search

Deep in the code of the DIFF program (presented as a separate article in the August *DDJ*), we posed a small problem in structured programming. At heart, it was nothing more than the old "loop with two exits," a problem that you're sure to meet whenever you confine yourself to the fundamental control shapes permitted by structured design.

In the context of that program, the problem went like this. A symbol table named ST (an array indexed from 0 to MaxSym-1) is being treated as a hash table to store Lines. Hash (Line) produces the initial probe of the table for any Line. Whenever we probe an entry of the table, we encounter one of three results:

(1) If ST[S].HashVal is negative, then entry S is free and the present Line may be installed in it.

(2) If ST[S].HashVal equals Hash-(Line) and if ST[S].LineVal equals Line, then this Line has already been entered and S is its index.

(3) Otherwise, some other Line is hashed to entry S, so we must try the next entry in succession, wrapping around at the end of the table.

The problem fits awkwardly into conventional program structures because the loop must terminate under either case (1) or case (2), but if it terminates under case (1) the new Line must be installed in ST[S]. In the original program (which was hacked together in a hurry), this was all done with Goto statements in an efficient but inelegant way. Several readers responded with rewritten functions, and the variety of their solutions is interesting.

Before looking at them, let's look at a side issue. We stated the equality test in case (2) in two parts for performance reasons. It takes little time to compare two hash signatures (integers). Different Lines, however, will occasionally hash to the same value, in which case (and only then) the slower test of comparing two variable-length strings is applied. In the published program, this was handled with a compound IF statement,

```
if (ST[S].HashVal = H)
   and (ST[S].LineVal^ = Line)
   then . . .
```

Only Paul Sand of Dover, NH, noticed that such a statement is not good Pascal. "I want to bring to your attention a portability bug," he wrote. "The problem is that standard Pascal does not guarantee short-circuit evaluation of Boolean expressions. For example, in good old Apple Pascal, both tests will *always* be done no matter what the outcome of the first test, slowing things down considerably. Other versions of Pascal might do the tests in reverse order."

Sand is dead right. C promises to short-circuit the second (slower) relation when the first test fails, and Ada offers the CAND (conditional and) operator for explicit control of the sequence of tests. In Pascal, the only way to get the desired effect is to write:

```
if ST[S].HashVal = H then
   if ST[S].LineVal^ = Line then
```

When you do, you end up duplicating code in the ELSE leg of the now doubled IF.

Sand's version of the search code (Listing One, page 17) is a straightforward encoding of the specifications given above. An auxiliary Boolean variable, Done, is introduced to control the loop. Case (1) is handled as soon as it is discovered so that, on termination, cases (1) and (2) have been made identical.

A. Salemma (our approximation from a handwritten signature) of Washington, DC, and Wayne Rivers of Springfield, VA, sent solutions much like Sand's except that they used the more intuitive "Repeat . . . Until

Done" structure instead of "While Not Done Do."

It isn't necessary to handle all three conditions inside the loop. Fred Marchand of Bellingham, WA, rewrote the program in C for his own uses, but his implementation of the symbol-search function translates back to Pascal as shown in Listing Two (page 17). The loop ends when either case (1) or case (2) is discovered. The special case (1), entry of a new symbol, is handled outside the loop.

Gary Dale of Toronto, Ontario, took an approach which is similar to Marchand's but he made better use of Pascal's treatment of Boolean data (Listing Three, page 17). He was the only one to use a Boolean expression to set the value of the auxiliary variable. Dale made a number of other changes in the program, including implementing his own heap storage so that it could store more lines.

Ragged Patches

No subject, it seems, inspires more imaginative, not to say panicky, adventures in typography than the program patch presented in a glossy magazine. In the past year, various PC-related magazines have carried articles on patching WordStar for this or that purpose as well as articles on patching other bits of MSDOS or PCDOS; each has used a different way of showing the necessary Debug commands.

It's just nerves, we think. The author knows that making a patch is a risky business; if it isn't done just right, the results will be unpredictable (and, fairly or not, blamed on the author). So the author writes down *exactly* what is to be done, making up typographic conventions for "what the computer will show" and "what the reader should enter" on the fly.

The editors also know that patches are risky things that must be done just right—and that they haven't the expertise to tell which parts of the author's presentation are essential and which are decoration. So they change the author's careful instructions as little as possible; this usually means fitting them to the Procrustean bed of a specific column width, reducing blanks and commas to proportionally spaced insignificance, changing the ASCII apostrophe to an inverted comma, and shrinking fat computer-printed asterisks to eentsy specks above the line.

What's left is a nearly unreadable account of a hypothetical Debug session. Even if it were readable, it wouldn't be relevant. Not everyone uses Debug; other debuggers, and other utilities for modifying disk files, are available. And some patches can be usefully made on the fly by another program. The problem with showing patches in terms of Debug operations is that it confuses the *process* with the desired *results*. All that is relevant about a patch is the address of the target area, what's in that area now, and what to change it to. You can put this data neatly in a three-column table.

The Table on page 17 is an example. It shows a patch for ZDOS version 2.13 (MSDOS 2 for the Zenith 100 series). The patch corrects a tendency of serial output to drop characters. It appeared first in *BUSS, The Independent News-*

letter of Heath Co. Computers as a column and a half of tortuous instructions. The table is clearer and less likely to lead to error. It informs the reader what is to be changed, but leaves the how of the change unspecified.

(Charles Floto's *BUSS*, incidentally, is a valuable resource for anyone who owns Heath hardware. It costs $28 for a year of 20 gossip- and bargain-filled issues; call (202) 544-0900 to subscribe.)

Wot Duzzit Dew?

David S. Tilton sent us the following sequence of Z80 assembly code. It's an absolutely astonishing implementation of . . . but you can figure it out. Unfortunately, it requires such narrowly defined conditions that it's probably useless, despite being the fastest one of its kind. What does it do, and what are its narrow requirements for utility?

```
Loop:
        cp      (hl)
        ret     z
        rl      L
        djnz    Loop
        ld      L,O
        ret
```

Might there be a way to use this gimmick in another architecture (e.g., 8086, 68000) to get the same result with more flexibility? **DDJ**

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

Load IO.SYS with the debug command: L 1000:0 0 A 20

At . . .	Find . . .	Replace . . .
1000:1E32	MOVE AH,AL	MOV BH,AL
1000:1E47	POP BX MOV AL,AH	MOV AL,BH POP BX
1000:1E91	MOV AH,AL	MOV BH,AL
1000:1EA6	POP BX MOV AL,AH	MOV AL,BH POP BX

Table
A patch to the BIOS of ZDOS 2.13 (*not* PCDOS) to prevent dropping bytes in serial output. Addresses based on loading IO.SYS from disk under Debug; not correct for patching the active system.

Dr. Dobb's Clinic (Text begins on page 12)

Listing One.
Structured hash-table search by Paul Sand

```
h := hash(Line);
s := h mod MaxSym;
done := FALSE;

while not done do
    if (ST[s].HashVal < 0) then begin
        with ST[s] do begin
            hashval := h;
            new(LineVal);
            LineVal^ := Line
        end;
        done := TRUE
    end
    else
        if ST[s].HashVal = h then
            if ST[s].LineVal^ = Line then
                done := TRUE
            else
                s := (s + 1) mod MaxSym
        else
            s := (s + 1) mod MaxSym;
store := s; { result }
```

End Listing One

Listing Two.
Structured hash-table search by Fred Marchand

```
h := Hash(Line);
s := h mod MaxSym;
found := FALSE;

while (ST[s].HashVal > 0) and (not found) do
    if (ST[s].HashVal = h) and
       (ST[s].LineVal^ = Line) then
        found := TRUE
    else
        s := (s+1) mod MaxSym;

if (ST[s].HashVal < 0) then
    with ST[s] do begin
        HashVal := h;
        new(LineVal);
        LineVal^ := Line
    end;

store := s;
```

End Listing Two

Listing Three.
Structured hash-table search by Gary Dale

```
h := Hash(Line);
s := h mod MaxSym;

repeat
    with ST[s] do
        if (HashVal = h) then
            found := LineVal^ = Line
        else
            found := HashVal < 0;
    if not found then
        s := (s+1) mod MaxSym
until found;

if (ST[s].HashVal < 0) then
    with ST[s] do begin
        HashVal := h;
        new(LineVal);
        LineVal^ := Line
    end;

store := s;
```

End Listings

Fatten Your Mac

by Tom Lafleur and Susan Raab

Veteran microcomputer hardware hackers have denounced Apple for abandoning the principle of an open architecture with the Mac. But to the true hacker, all architectures are open.

The modifications described in the article by Lafleur and Raab are not for amateurs. In fact, we can't with a clear conscience recommend that you void your warranty and risk destroying your motherboard to save a few dollars. Although we've taken steps to convince ourselves of the technical accuracy of the article (and we know that Tom Lafleur has used this procedure on several Macs), we haven't yet had the nerve to fatten a Mac ourselves. Don't take this on as a first hardware project. Don't undertake it if you aren't sure of the risks involved. And don't blame us if anything goes wrong. We are providing this information strictly as a service to those who know how to use it. We take no responsibility for fried Macs. If you have the newer 128 motherboard, read the addendum on page 4.

Are you tired of those pesky disk writes while using MacPaint? Does the performance of even your new hard disk drive leave you longing for the lightning response of a RAM disk? Want to run Lotus 1-2-3 and can't, or are you developing MAC software and just plain running out of room?

For these and many other reasons, Macintosh owners everywhere are rushing to Apple for a FAT MAC upgrade that packs a full 512K of memory into the system. Most will (and should) wait for Apple to add the necessary chips to the Macintosh motherboard. However, if you are one of the adventurous few willing to sacrifice your Apple warranty and wager the life of your Macintosh against your soldering skills, here's how to perform the upgrade yourself for half the cost.

You'll Need. . .

The checklist in Table I (page 21) shows the parts and tools you'll need to complete the job. To minimize your MAC's down time, make sure you have everything on the list before you begin. All of the parts can be found at your local electronics supply house or by looking for ads in your favorite computer magazine.

Because the 256K RAM chips are a little harder to find, I have listed a few sources. We have paid $15 to $45 each for the memory chips: they are not cheap! We've tested only NEC memory chips in this conversion, but other equivalent 256K memory chips should work as well. Once you've got the parts and equipment together, you're ready to begin.

The Procedure

You'll need about four hours of quiet concentration to complete this upgrade yourself. Before you begin, read through the entire procedure. Then disconnect all cables from the Macintosh, including the keyboard, mouse, printer, extra drive, power, and anything else you might have plugged in.

(1) **Remove the case screws.** Use the Xcelite XTD-10 Torx screwdriver to remove all five of the case screws (see Photo 1 on page 20). There are two near the bottom, one inside the battery case, and two deep inside the handle.

Tom Lafleur, P.O. Box 490, Del Mar, CA 92014.

Susan Raab, Digital Research International, 160 Central Avenue, Pacific Grove, CA 93950.

(2) **Remove the case.** Turn the Macintosh facedown on a table. Gently press down on the battery compartment, power connector, and I/O connectors while you lift up on the back of the case. The back of the case should come off in your hands, leaving the faceplate in position around the screen and disk drive door. If the case is stubborn, insert a long metal ruler into the the seam between the faceplate and the back of the case (see Photo 2, page 20). (*Do not* use a screwdriver here.) Use the ruler to gently pry the seam open as you pull up on the back of the case.

(3) **Remove the motherboard.** Find and disconnect the two cables that attach the display board and disk drive to the motherboard (see Photo 3, page 20). Ease the motherboard from its connections and remove it from the chassis. Check the revision number of the motherboard. We've successfully performed this upgrade on motherboards that have the revision numbers 630-0101 screened on top and 820-0086C etched on the back.

(4) **Locate and remove the memory chips.** You'll find sixteen memory chips with part number MT–4264 at IC locations F5 through F12 and G5 through G12 (see Photo 4, page 20) Remove them from the motherboard as follows:

- Using a small pair of wire cutters, clip all the pins off each chip as close to the chip as you can (see Photo 5, page 20). Then throw away all 128K of these memory chips—they aren't as valuable as the Macintosh motherboard!
- Using a low-temperature soldering iron (700°), remove all the memory chip pins from the motherboard (see Photo 6, page 21). It's handy to use a Weller WTCPN soldering station because it has a magnetic tip that can pull the pin out of the board as soon as the pin is heated, but other soldering irons will work if you use care. This is a repetitive task (256 pins!) and you'll soon develop a rhythm. To keep the rhythm going, skip over pins 8 and 16 from each chip on the first pass. Because they're connected to the inner power and ground layers of the motherboard, they require more heat and time to remove. If you remove them on a second pass, they'll establish their own rhythm.
- Clean out all the holes with a solder sucker (see Photo 7, page 21). Again skip holes 8 and 16 on your first pass to keep your rhythm going, then clean them out on a second pass.
- Clean the motherboard with a fine brush. Look the board over for any solder splashes or broken etch.

(5) **Insert IC sockets.** Carefully solder a good IC socket in each memory chip location. When you're done, clean the back side of the motherboard with alcohol or a TF degreaser to remove all of the flux left after soldering. Examine the motherboard again for short or broken etch.

(6) **Install new memory chips.** Carefully insert the sixteen 41256 memory chips into the sockets (see Photo 8, page 21).

(7) **Test motherboard.** Before you go any further, check your work by starting up the system. At this point, it should operate as a normal 128K Macintosh. To complete the test, follow these steps:

- Insert the motherboard back into the Macintosh chassis. Connect the display and drive cables to the motherboard, then connect the power cable.
- Start up the system. If all is well, you'll see the normal question-mark disk icon. If the system's diagnostic software finds a problem, it will display a sad icon (or as much of one as it can!). If the system does not start nor-

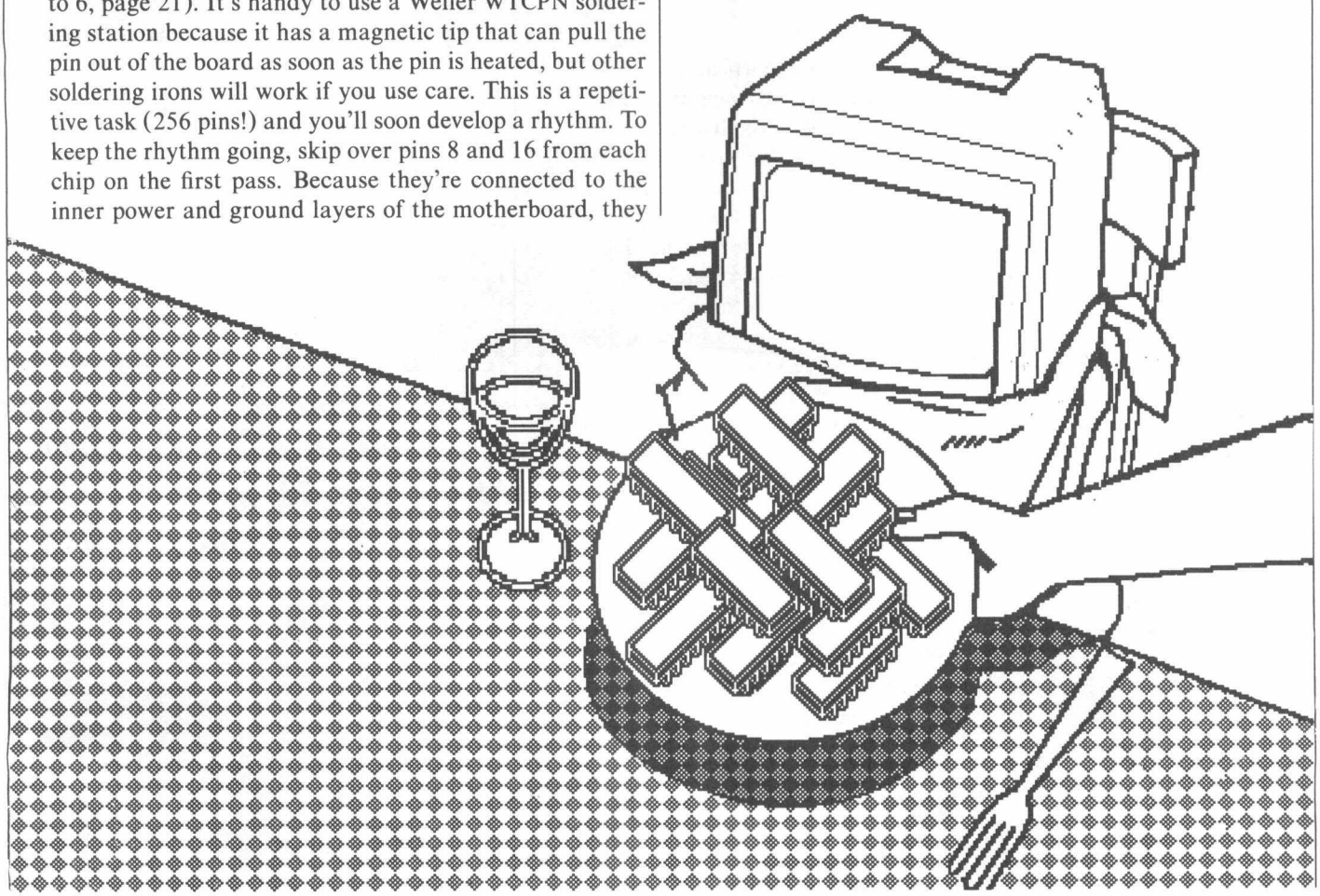


Photo 1

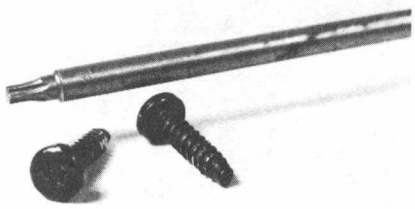


Photo 2

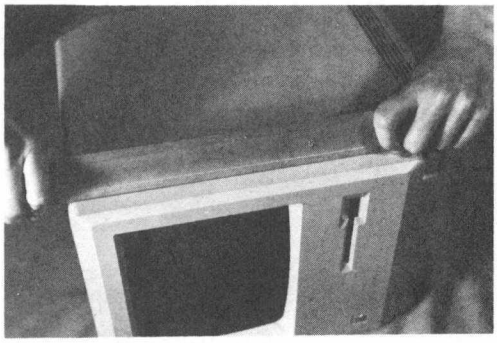


Photo 3

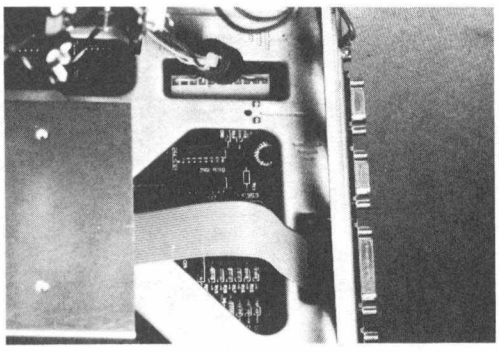


Photo 4

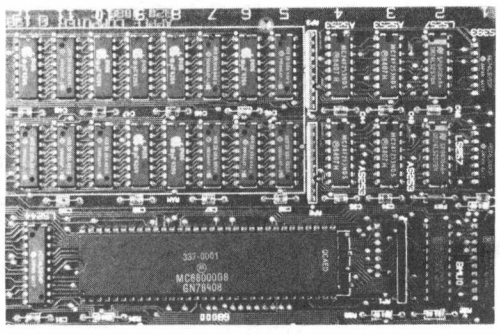
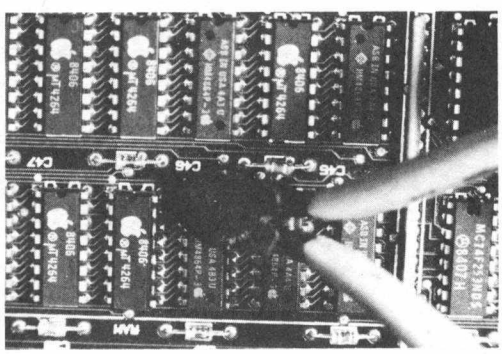


Photo 5



mally, check the trouble-shooting section at the end of this article. *Do not go on to the next step until the system boots up normally.*

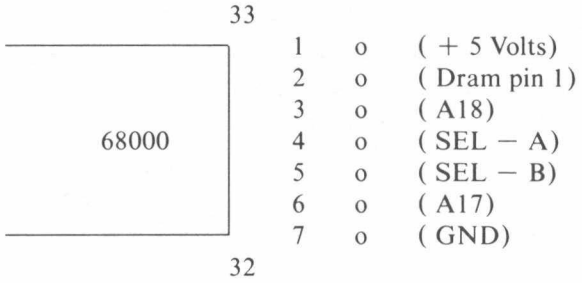
(8) **Remove the motherboard.** Again disconnect the display, disk, and power cables from the motherboard. Ease the motherboard from its connectors and remove it from the chassis.

(9) **Assemble a new memory select IC.** You must build a memory select IC that lets the Macintosh access the extra memory you've just installed. Create the new IC as follows, using Photos 9 and 10 (page 21) and Figures One and Two (page 23) for reference.

- On a good quality 16-pin IC socket, bend out all the pins except 2, 7, 14, and 16.
- Use solder and some small-gauge wire (30 awg) to connect pins 1, 10, 11, 12, and 13 to pin 8.
- Solder a 2.2K 1/4 watt resistor between pins 15 and 16.
- Use solder and some small-gauge wire to connect pins 3 and 4 to pin 15.
- Insert a 74F253 or a SN74AS253 dual 4-to-1 multiplexer into this modified IC socket.

(10) **Install the new memory select IC.** You must connect the new IC assembly to the 68000 address bus, memory select logic, and other lines on the motherboard as follows:

- Mount your new IC assembly on top of the 74F253 (or SN74AS253) located at F3 on the motherboard. Solder pins 2, 7, 14, and 16 of the new IC assembly to the same pins on the 74F253 at F3.
- Locate the seven IC pads at E3, next to pins 32 and 33 of the 68000 microprocessor (see below):



- On the back side of the motherboard, cut the etch between pins 1 and 2 at location E3.
- Solder a 47-ohm 1/4 watt resistor between pin 7 of the new IC assembly and pin 2 of the IC pad at location E3. (See Figure One.)
- Using solder and small-gauge wire, connect pin 5 of the new IC assembly to pin 3 of the IC pads at location E3.
- Using solder and small-gauge wire, connect pin 6 of the new IC assembly to pin 6 of the IC pads at location E3.
- Check the motherboard for any solder splashes, missed wiring, or broken etch. Clean the board with alcohol or a TF degreaser to remove any flux left after soldering.

(11) **Test your FAT MAC!** Insert the upgraded motherboard into the Macintosh chassis. Reconnect the disk, display, and power cables. Power up your Macintosh and check for the normal question-mark disk icon. If your MAC does not appear normal, review the trouble-shooting section at the end of this article. If the normal question-mark disk icon appears, insert your system disk and open the disk copy program. The disk copy program should display a message that

says it only works with a standard 128K MAC. To see how much memory you have, start up BASIC (if you have it) and enter the free memory command: PRINT FRE(0). On our system, BASIC reports about 340K of memory. You may also want to run the MAC memory diagnostics as outlined below for a few hours to check for any long-term problems.

Trouble Shooting

If the system does not start normally, use a multimeter to check that all the connections you've made are connected properly. If all the connections check out, use the system diagnostic program in the Macintosh ROM to determine if

Parts:

Qty.	Part No.	Description	Vendor
16	41256-200	256K 200-ns memory chips	NEC
17		16-pin IC sockets	
1	SN74AS253N or 74F253N	Dual 4-to-1 multiplexer	TI,Motorola Fairchild
1		2.2K 1/4 watt resistor (any resistor from 1K to 4.7K will work)	
1		47-ohm 1/4 watt resistor	

Memory chip vendors:

The first five vendors sell their products retail and have a small or no minimum order; the other vendors are industrial suppliers and may have a minimum order.

Jameco electronics	(415) 592-8097
JDR microdevices	(408) 995-5430
Advance computer products	(800) 854-8230 (714) 558-8822
Jade computer products	(800) 421-5500 (800) 262-1710
DoKay computer products	(800) 538-8800 (800) 848-8008
NARA	(408) 748-9200
TAKA	(415) 952-9000
Japan electronics	(818) 369-1833
D-L-C	(213) 938-2677

Tools:

Xcelite XTD-10 Torx screwdriver, 6-inch shaft
Long metal ruler
Low-temperature soldering iron
(Weller WTCPN recommended)
Solder sucker
Small-gauge wire (30 awg)
Multimeter (for problem solving)

Table I.
Supplies Checklist

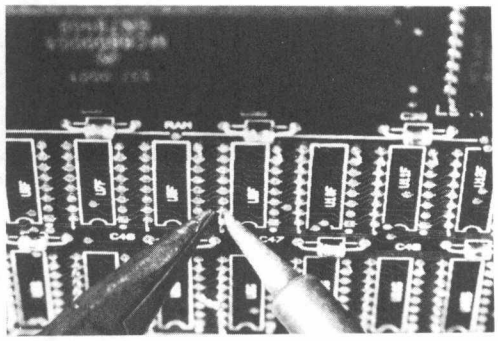


Photo 6

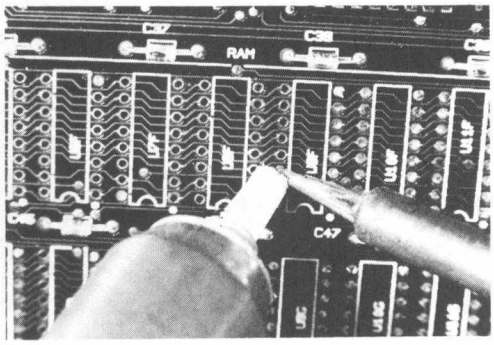


Photo 7

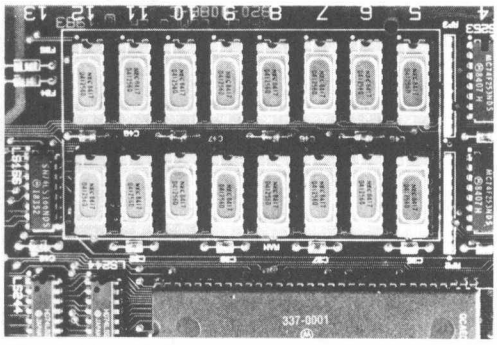


Photo 8

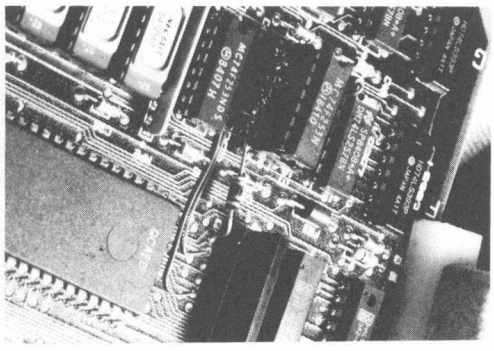


Photo 9

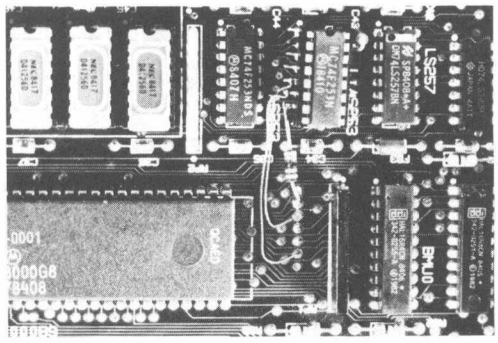


Photo 10

Chip Location	Data Bit Number	Pin 2 to 68000 Pin	Pin 14 to Memory Buffer Pin
F5	D0	5	E12 – 2
F6	D1	4	4
F7	D2	3	6
F8	D3	2	8
F9	D4	1	17
F10	D5	64	15
F11	D6	63	13
F12	D7	62	11
G5	D8	61	E13 – 2
G6	D9	60	4
G7	D10	59	6
G8	D11	58	8
G9	D12	57	17
G10	D13	56	15
G11	D14	55	13
G12	D15	54	11

Table II.
Connections for Pins 2 and 14

	Class Code	Sub Code
1 =	ROM test failed	Meaningless
2 =	Memory test – bus subtest	Identifies bad chips
3 =	Memory test – byte write	Identifies bad chips
4 =	Memory test – Mod3 test	Identifies bad chips
5 =	Memory test – address uniqueness	Identifies bad chips

Table III.
Diagnostic Codes

Data Bit	Location	Sub Code Bits
0	F5	0001
1	F6	0002
2	F7	0004
3	F8	0008
4	F9	0010
5	F10	0020
6	F11	0040
7	F12	0080
8	G5	0100
9	G6	0200
10	G7	0400
11	G8	0800
12	G9	1000
13	G10	2000
14	G11	4000
15	G12	8000

Table IV.
Chip Identification

any of the memory chips you've inserted are bad.

Checking the connections is another repetitive task: you must make sure that all of the 256 new connections you've made carry signal to the appropriate destinations. And most of the connections carry signal to more than one place!

For example, a signal on pin 0 on one memory chip should be connected to pin 0 on every other memory chip. A signal on pin 1 should appear on pin 1 of all the other memory chips. Check for this continuity on all pins except 2, 14, and 15.

Pin 15 is common among chips in the same row. For example, pin 15 on a chip in row F should be connected to every other pin 15 in row F but not in row G. Pin 15 on a chip in row G should be connected to every other pin in row G. Pin 2 on each of the sixteen memory chips is directly connected to one of the sixteen data lines of the 68000 microprocessor. Pin 14 connects the memory chips to the memory buffer circuits at locations E12 and E13.

Table II (at left) shows how pins 2 and 14 should be connected. Each row in the table gives information about one of the memory chips. The first column lists the chip's location. The second column lists the data bit of the 68000. The third column lists the pin on the 68000 to which pin 2 of the memory chip should be connected. The fourth column lists the pin on the memory buffers to which pin 14 of the memory chip should be attached.

If you discover that one of these connections is not connected properly, find and correct the broken etch, or add some small-gauge wire until the connection is restored. If all the connections are in working order and you're still having trouble, use the system diagnostics in the Macintosh ROM to identify the bad memory chips. Look at all signals connected to the suspected chip for a bad connection.

Diagnostics

Before starting the diagnostics, you must have installed the programmer's buttons Interrupt and Reset on the left side of your Macintosh. Hold down the Interrupt button and either press the Reset button or power on your Macintosh.

A sad Macintosh icon appears with a numeric code under it. If all is working well, the code will be 0F 000D, and some small bits will cycle under the code to indicate that the Macintosh is running the memory diagnostic program. The numeric code that you will see has two parts; for example, 0F is the class code and 000D is the sub code. As shown in Table III (at left), the class code tells what part of the diagnostic program found an error, and the sub code tells what the error was. Each of the sixteen bits in the sub code identifies one of the sixteen memory chips. Table IV (at left) maps the sub code bits to their respective chip's location.

If the diagnostics discover more than one bad chip, the sub code displays multiple bits. For example, if bit 3 is bad, the diagnostics display sub code 0008. If both bit 3 and bit 10 are bad, the diagnostics display sub code 0408. If the diagnostics identify a bad chip, you'll have to replace it with a good one or find the problem on the board.

After all the diagnostics have passed, the program displays an exception code giving the current state of your MAC. You should normally see a sub code of 000D, NMI. The others are listed in Table V (at right) for your information only.

Removing the Modification

If you need to convert your Macintosh back into a standard 128K unit, simply remove the IC assembly you added at location F3, remove the sixteen 41256 memory chips, and replace them with standard 200-ns 4164 64K memory chips. You must also connect the jumper at location E3, between pins 1 and 2.

We've completed the conversion on over 10 MACs and have had no problems with this upgrade. So good luck with your new FAT MAC.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 191**.

	Class Code	Sub Code
F =	Exception	0001 Bus error
		0002 Address error
		0003 Illegal instruction
		0004 Zero divide
		0005 Check instruction
		0006 Traps instruction
		0007 Privilege violation
		0008 Trace
		0009 Line 1010
		000A Line 1111
		000B Other exception
		000C Nothing
		000D NMI (normal indication)

Table V.
Exception Codes

READ.ME

The 256K dynamic RAM chips that turn a Mac into a Fat Mac are very sensitive to static electricity. Please note the following advice on handling 256K RAMs, adapted from information from John Gilchrist of Microprocessors Unlimited in Beggs, Oklahoma.

To damage a true LSI device like a 256K RAM chip, you don't have to touch it. Being close to it with a high potential voltage on your body will do the job. You can, for example, generate 3000 volts by walking across a carpet in leather shoes or by peeling off a foot or so of cellophane tape, and you won't feel a thing as the tiny spark jumps to the IC, doing its hidden damage.

Following these steps should lessen the risk to the chips from static electricity.

1. If you have a choice of workspace, almost any floor covering is better (for the present purpose) than carpeting.

2. Take off your shoes. Don't take this as a personal remark, but your feet sweat enough to make it unlikely that a high static charge can build up when you are standing barefoot on a noncarpet surface. And don't wear any nylon clothing.

3. Spread out a large sheet of aluminum foil (about a three-foot length) and work on that. Wrap a corner of the foil around the Mac chassis or motherboard and poke the computer's power cord through the foil. Keep your body (e.g., one elbow) on the foil throughout the process of preparing the Mac to receive its new chips.

4. Don't handle the 256K chips until, or any more than, you have to. When you are ready for them, slide the chips out of their factory tube onto the foil. Keep your body in constant contact with the foil as you install the chips.

5. When you've finished the installation and are ready to test your work, be sure to remove the wall plug from the foil before plugging it into the wall. Otherwise you could fry more than a chip.

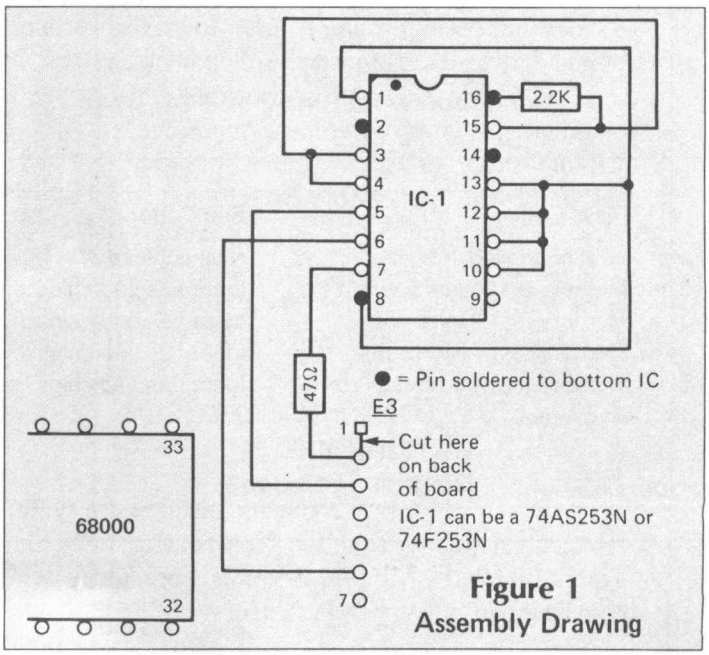


Figure 1
Assembly Drawing

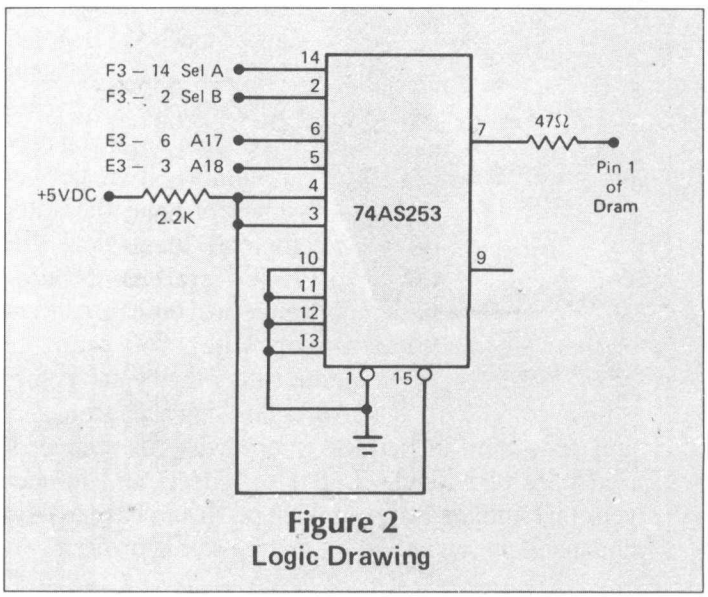


Figure 2
Logic Drawing

QuickDraw Meets ImageWriter

by Thom Mayer

Printing sideways with a Macintosh.

Apple's new crop of computers, the Macintosh and the Lisa, makes extensive use of bit-mapped graphics. Less visible than the graphics hardware, but just as necessary, is the well thought-out, well crafted, integrated collection of graphics software primitives called QuickDraw. These routines make it a simple matter to put graphics into programs. Because of QuickDraw, programmers have an easier job, users get better software, and Apple—who paid for its development—racks up sales. Unfortunately, the Lisa QuickDraw routines lack any support for the Apple ImageWriter dot matrix printer; hence, this article. The ImageWriter exhibits the quality features we have begun to associate with Apple hardware (e.g., well-crafted built-in software primitives and complete concise documentation). I would recommend the ImageWriter to anyone in the market for a printer.

Figures 1 and 2 (page 27) were drawn on a Lisa by QuickDraw and were printed on an ImageWriter dot matrix printer with the enclosed routine. Listing One (page 31) gives a Pascal unit containing the function IMAGE_PRINT, and Listing Two (page 35) demonstrates its use. In this article we will briefly review the way QuickDraw stores a graphics image and the way the ImageWriter prints graphics. Then we will take a cursory look at the routine IMAGE_PRINT and discuss some of the considerations that influenced its design.

QuickDraw stores an image in a continuous piece of memory with each bit corresponding to one pixel: a one for black, a zero for white. For example, the Lisa screen, which measures 720 × 364 dots, requires 262,080 bits or 32K bytes. The exact relationship between the data in memory and the pixels on the screen depends on the values stored in the bitMap. A bitMap is a data structure specified in Pascal as follows:

```
BitMap = RECORD
   baseAddr : QDPtr;
   rowBytes : Integer;
   bounds : Rect;
END;
```

The field baseAddr is a pointer (memory address) to the beginning of the block of memory where the image is stored. We should imagine this memory grouped by lines, each line containing the number of bytes specified by the field rowBytes. Two dots on the screen that have the same vertical coordinate and differ in the horizontal coordinate by one are neighboring bits in memory. Two dots on a QuickDraw image that have the same horizontal coordinate and differ in the vertical coordinate by one are separated by exactly rowBytes bytes in memory (Figure 3, page 27).

The ImageWriter, like all dot matrix printers, forms characters and graphics by printing a pattern of dots. The printer head is composed of a vertical array of nine dot-forming devices spaced 1/72 inch apart; at one stroke, the Im-

Thom Mayer, Tigre Designs, 3006 Lafayette, Austin, TX 78722.

ageWriter produces a pattern of nine vertical pixels. Normal characters are composed of seven such patterns, with a blank eighth pattern forming the space between the characters. The horizontal spacing between successive patterns can be set by sending the printer control codes: A spacing of 1/72 inch yields nine characters per inch, a spacing of 1/136 inch yields 17 characters per inch, and so on (Figure 4a, page 28).

When the ImageWriter is in graphics mode, each byte received produces an eight-pixel pattern determined by the binary representation of the byte (recall 1 byte = 8 bits). The top pixel corresponds to the least significant bit, and the bottom pixel corresponds to the most significant bit. A dot is printed if there is a one in the corresponding bit; no dot is printed if there is a zero (Figure 4b, page 28).

Because a byte in QuickDraw corresponds to eight horizontal pixels on the screen, and a byte to the ImageWriter prints eight pixels aligned vertically, the image will come out sideways on the paper. Each printed line of graphics corresponds to a vertical strip on the screen, eight pixels wide.

The pixels on the Lisa screen are not square: they are 1.5 times as high as they are wide. On the printer, the vertical spacing between dots is fixed by the printer head to 1/72 inch, so to match the Lisa screen aspect ratio we need a horizontal dot length 1/48 of an inch, which we make out of two dots on 1/96-inch spacing. At this spacing, an image the size of the Lisa screen nearly fills an 8 × 10 sheet of paper.

The printer algorithm naturally breaks into several steps, shown here in the simple form of the main routine:

```
function image_print ({parame-
    ters} ) : boolean;
    begin
        size_of_image;
        init_printer;
        init_buffer;
        for line_number:=1
            to (row_width) do
                begin
                buffer_a_line
                    (line_number);
                spew_buffer;
                end;
        close_printer;
        image_print:=true;
```

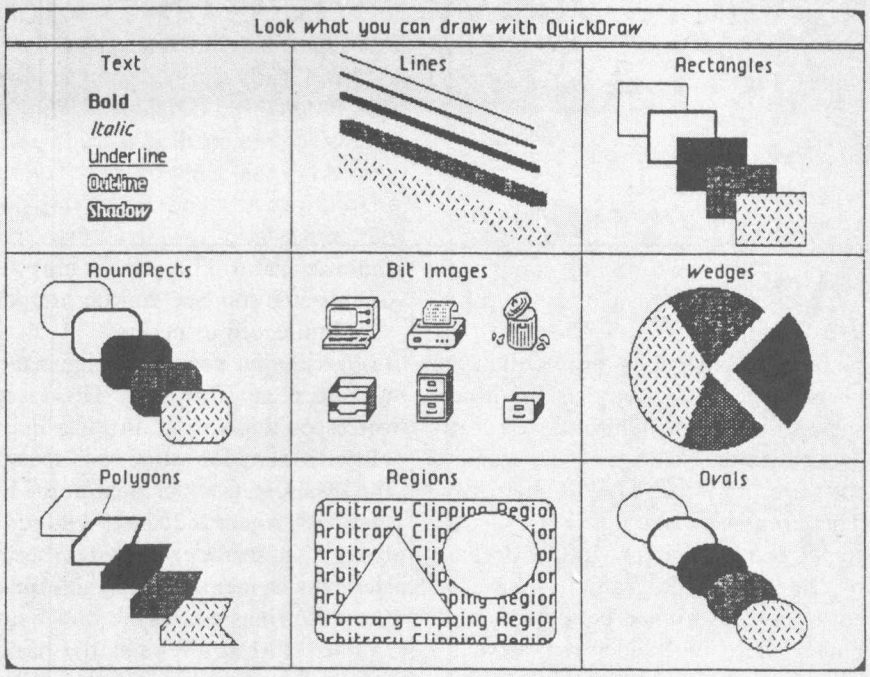


Figure 1.

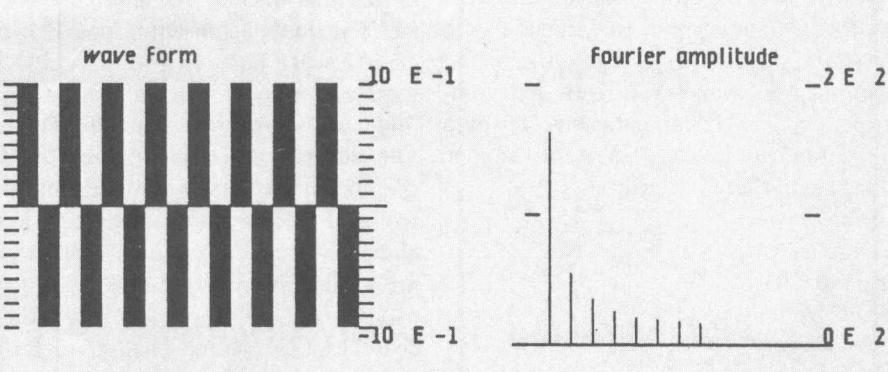


Figure 2.

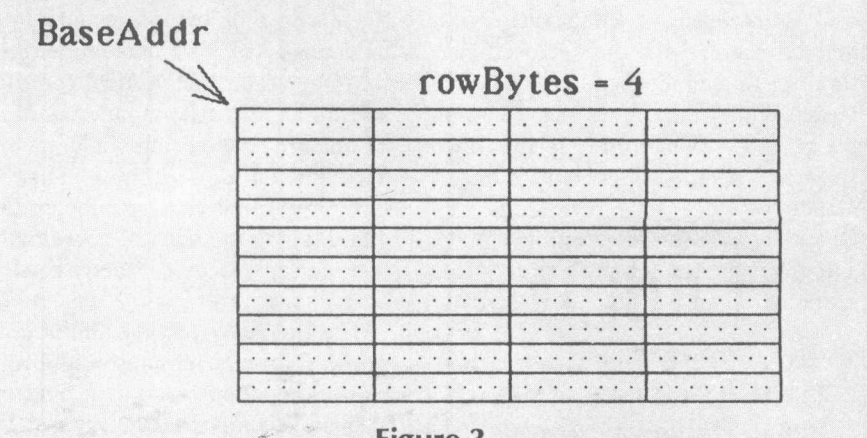


Figure 3.
This piece of memory can hold an image as large as 32 × 10 pixels.

```
end;
```

This algorithm did not spring out of my head fully formed in top down style, but rather I developed it through successive refinement of a crude prototype. At some point in the development, however, you must translate your work into the structured programming form. This process may give you insights into how to improve your work, and it will certainly lengthen the lifespan of your programs by simplifying future modifications. Good software is too costly to be disposable.

Briefly let's go through each routine:
(1) SIZE_OF_IMAGE looks at the bitMap to see where the image is stored in memory (baseAddr), how it is organized (rowBytes), and what size image we are to print (bounds).
(2) INIT_PRINTER sends the printer the initialization codes. To change from the default settings, which are appropriate for text, to the proper settings for graphics takes 16 bytes of data, so most of this first block is NULs.
(3) The FOR loop prints one line of graphics per pass. BUFFER_A_LINE gathers the bytes needed to print the line and stores them in the buffer. SPEW_BUFFER sends the buffer to the printer and listens for confirmation from the printer. At the beginning and end of the buffer are control codes that are initialized by INIT_BUFFER.
(4) CLOSE_PRINTER close files, switches the printer back to default settings, and form-feeds the paper. If the routine gets this far, it has detected no problems and returns the value true.

There we have it, the means to print a QuickDraw image on the ImageWriter dot matrix printer. Now we will look more closely at the program design, so have your listing handy.

IMAGE_PRINT is set up as a Boolean-valued function that returns false only if the routine couldn't print the image. The usual cause for failure is the printer being off, which is detected when the blockwrite routine says it was unable to complete the transfer of blocks. All failures are channeled through the procedure NO_GOOD :

```
procedure no_good;
  begin
    image_print:=false;
    exit(image_print);
  end;
```

The not well-known but extremely handy procedure exit satisfies those programming urges that in other environments might be handled by a GOTO statement. It is the calling program's responsibility to respond appropriately to failures. The following code fragment, which uses the system call PAbortFlag, gives the user the ability to recover or abort:

```
while not ( PAbortFlag or
  IMAGE_PRINT
  (grafptr,printerPort) )
  begin
  writeln ( ' fix printer or press
    abort key ');
  writeln ( ' press RETURN to
    continue ');
  readln ( );
  end;
```

Filling the buffer with the bytes needed to print one line is like buying groceries at the supermarket. BUFFER_LINE loads a byte, skips row_bytes bytes, loads a byte, and so on; clearly we need random access to memory by the byte. This calls for a packed array of char. But we don't want to make a new array: we want to use the memory that QuickDraw has already set up. So rather than declaring an array variable, we declare a pointer variable, CH, of type pointer to packed array of char. Now we are all set. Assign CH the value baseAddr (i.e., the beginning of the image memory), and an expression like CH^[n] gives access to the nth byte of the image memory. The memory has not been changed—only the way the program views it.

Pascal is a strongly typed language, and most compilers check the type of a pointer before allowing assignment. Just as you can't mix apples with oranges, you can't mix pointers to apples with pointers to oranges. Lisa Pascal, however, includes a wonderful extension, the unary operator @, which returns the pointer to the operand (just like the & operator in C). For example, if Z is an array, then @Z is a pointer that points to Z. Simple, yes ? The @ operator has another important feature: the pointer resulting from an @ operation, like the nil pointer, is assignment-compatible with any other type pointer. To make CH point to the same place as bitAddr, use the assignment CH:= @bitAddr^. I call this detyping a pointer. It is like a CAST operation in C.

Between the routine IMAGE_PRINT and the printer sits the device driver routine that actually transfers the bytes. A feature of the Lisa port driver is the automatic insertion of a line feed after each carriage return, an option that can be enabled and disabled with system calls. The automatic line feed may be handy if you have a vintage 1960 printer, but in a graphics setting, adding the byte 00001010 behind every occurrence of the byte

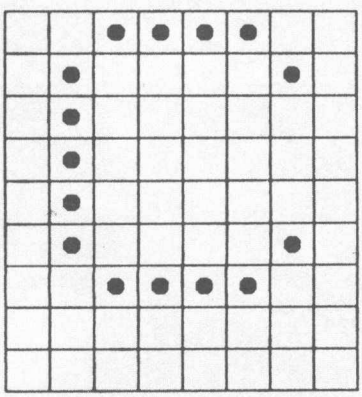


Figure 4a.
The printer head produces nine vertical pixels per shot. The standard characters are constructed from eight such vertical patterns.

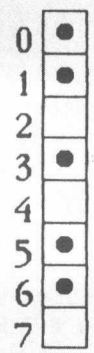


Figure 4b.
In graphics mode, each byte produces a vertical line of eight pixels. This pattern was produced by the byte 01101011 binary (i.e., 107 decimal).

0001101 is a bug. To track down this bug, without the 20-20 vision of hindsight, required a long session with the debugger. Among other capabilities, the debugger allows you to halt the program and to go in and look at the memory—even change it if you wish—and then resume operation of your program. My search went as follows.

First I went in and looked at the memory where the screen image was stored to make sure that it looked as I had expected. Then I looked at the memory corresponding to my buffer to make sure it was correct. Upon closely inspecting the buffer for the line that was giving me trouble, I found out that the bug happened only when the byte 0D was in the buffer. With the debugger I could change the occurrences of 0D to some other byte, or vice versa, and ascertain that 0D was the guilty party. 0D is the code for carriage return, and it was instantly clear to me that somebody was adding an LF after every carriage return. But who?

I spent a long time thinking the printer was the culprit but finally convinced myself otherwise and uncovered the automatic line feed feature of the port driver. The system calls necessary to reset this appear in the procedure AUTO_LF, which can enable or disable the automatic line feed on either serial port. Unfortunately, the routine now needs to know which port the printer is connected to. I was tempted to circumvent this need by temporarily resetting both port drivers, but this would be unwise on a multitasking machine like the Lisa because some background task might be using the other port; for example, a modem connected to the other serial port might be in use by a BBS running in the background.

In the interest of simplicity, this version of IMAGE_PRINT makes no attempt at treating large blank spaces intelligently, an addition that can improve speed substantially. As presented, IMAGE_PRINT requires 188 seconds to print a 720 × 364 picture. Also in interest of simplicity, the width of the printed image is rounded up to the nearest multiple of eight, i.e., on a byte boundary. For those desiring sample QuickDraw programs and a version of IMAGE_PRINT that does not make the above two *simplifications*, mail me a $25 check, and I will send you a 3.5-inch Lisa format disk with two versions of IMAGE_PRINT, other graphics utilities, and complete programs that use QuickDraw and the utilities. Source and compiled code is included. You are welcome to distribute the code for noncommercial purposes as long as you do not remove my copyright notice. Please identify any modifications you make, and I encourage you to share any successes you have. We are the neurons of our cultural brain.

DDJ

QuickDraw Meets ImageWriter

(Text begins on page 26)

Listing One

```
{
{$R-}
unit GPHU;   {graphics utilities }
{ COPYRIGHT T.MAYER 1984  --  DISTRIBUTION FOR NON-COMERCIAL PURPOSES ALLOWED   }
interface
  uses
{$U-}
     {$U QD/QuickDraw }  QuickDraw,
     {$U QD/QDSupport }  QDSupport,
     {$U syscall}        syscall;
{$U+}

  function image_print(tp:grafptr;printerPort:integer) : boolean;

implementation

function image_print;
type cars = packed array[0..maxint] of char;
var
     printer:text;p:file;
     row_bytes,bytes_to_print,height,line_number,index,i,buff_top:integer;
     ch:^cars;
     buff : packed array[0 .. 1023] of char;

     procedure AutoLinefeed(whichPort:integer;enabled:boolean);
     { whichPort=0 -> portA,  whichPort=1 -> portB  }
     var Ecode:integer;path:pathname;Cparm:Dctype;
     begin
          if (whichport mod 2=0) then path:='-RS232A' else  path:='-RS232B';
          CParm.dcVersion:=2; CParm.dcCode:=17;
          if enabled then CParm.dcData[0]:=1 else CParm.dcData[0]:=0;
          DEVICE_CONTROL(Ecode,path,Cparm);
     end;
```

(Continued on next page)

Listing One

```
      procedure no_good;
      begin
           image_print:=false;
           exit(image_print);
      end;

      procedure init_printer;
      var i:integer;
      begin
 { first we disable auto line feed in RS232 device driver }
           AutoLineFeed(PrinterPort,false);

 { the printer file is to be accessed via blockwrite. }
           reset(p,'-printer');

 { send one block, mostly nuls, with initialization codes :)

 (this tells printer to recognize 8th bit - neccessary for graphics )
           buff[0]:=chr(27);
           buff[1]:=chr(90);
           buff[2]:=chr(0);
           buff[3]:=chr(50);
(make sure printer adds no LF after CR )
          buff[4]:=chr(27);
          buff[5]:=chr(90);
          buff[6]:=chr(128);
          buff[7]:=chr(0);

{ this sets horizontal spacing to 96 dots per inch }
          buff[8]:=chr(27);
          buff[9]:='E';

{ this sets vertical line feed to 1/9 inch, i.e. height of 8 dots }
          buff[10]:=chr(27);
          buff[11]:='T';
          buff[12]:='1';
          buff[13]:='6';

(this sets printer head motion left to right only.  yields better quality output )
          buff[14]:=chr(27);
          buff[15]:=')';

          for i:=16 to 511 do buff[i]:=chr(0); (most of block is NULs )

{ now actually send the initialization block }
          if (blockwrite(p,buff,1)<>1) then no_good;

     end;

     procedure init_buffer;
     var i:integer;
          begin
{ these first six bytes say "here comes 728 graphics codes"  }
          buff[0]:=chr(27);
          buff[1]:='G';
          buff[2]:='0';
          buff[3]:='7';
          buff[4]:='2';
          buff[5]:='8';

          for i:=6 to 1023 do buff[i]:=chr(0); {blank if not otherwise reset }

          buff[734]:=chr(13);buff[735]:=chr(10); { CR LF ends each line}

          buff_top:=6;
```

(Continued on page 34)

Listing One

```
     end;

     procedure enbuff(a:char);
     begin
          if buff_top=1022 then exit(enbuff);
          buff[buff_top]:=a;buff[buff_top+1]:=a;
          buff_top:=buff_top+2;
     end;

     procedure spew_buffer;
     const nblocks=2;
     begin
          if (blockwrite(p,buff,nblocks)<>nblocks) then no_good;
          buff_top:=6;
     end;

     procedure close_printer;
     var printer:text;
     begin
          close(p);
          rewrite(printer,'-printer');
          write(printer,chr(27),'c');        {return printer to default set up}
          write(printer,chr(12));            {form feed}
          close(printer);
          AutoLineFeed(PrinterPort,true);  { enable port driver autoLF function }
     end;

     procedure buffer_a_line(line_number:integer);
     var i:integer;
     begin
          index:=row_bytes - line_number;
          for i:=1 to height do   {buffer a line}
               begin
               enbuff(ch^[index]);
               index:=index+row_bytes;
               end;
     end;

     procedure size_of_image;
     var width : integer;
     begin
          with tp^.portBits do
          begin
               row_bytes:=rowbytes;
               height:=bounds.bottom-bounds.top;
               width := bounds.right - bounds.left;
               ch:=@baseAddr^;
          end;
          bytes_to_print:=width div 8;
          if width mod 8 <>0 then bytes_to_print:=bytes_to_print+1;
     end;

begin
     size_of_image;
     init_printer;
     init_buffer;
     for line_number:=1 to (bytes_to_print) do
          begin
               buffer_a_line(line_number);
               spew_buffer;
          end;
     close_printer;
     image_print:=true;
end;  { of image_print }
end.  { of implementation }
```

End Listing One

Listing Two

```
program example;   { this sample program creates ILLUSTRATION 1 }
uses
      {$U QD/QuickDraw }  QuickDraw,
      {$U QD/QDSupport }  QDSupport,
      {$U syscall}        syscall,
      {$U gphu    }       gphu,        { graphics utilities   -  here we use :
                                              init_graphics
                                              bar_plot
                                              image_print }

      {$U mathlib}        mathlib;     { math library         -  here we use :
                                              square_wave
                                              famp  { i.e. FourierAMPlitudes )   }
 var
      wave : array[1 .. 256] of real;
      i,j:integer;
begin
      init_graphics;   erase_screen;

{ make a square wave and graph it : }
square_wave(@wave,256,16);
bar_graph ( 256, @wave[1], 10,40, 256,128, 0.0, 1.0, true,false, 'wave form','');

{ fourier transform it and graph that : }
famp(256,@wave);
bar_graph ( 129,  @wave, 400,40, 200,128, 0.0, 1.0, true,false,'fourier amplitude','');

{ print the whole screen }
if not image_print ( thePort, 0 ) then writeln('turn on the printer');

end.
```

End Listings

Archiving Files with CP/M-80 and CP/M-86

by Ian E. Ashdown

If you use CP/M-80 versions 2.x or CP/M-86, you know the problem well: sitting there at two in the morning trying to remember which files you worked on so you can copy them to a backup disk. If you have a hard disk in your system, the problem can be an acute pain. Which of several hundred files did you update or otherwise modify during your marathon programming session?

The ideal solution would be to have a utility program that somehow determines which files have been changed on a disk and automatically copies them to a backup disk. This procedure, known as "incremental backup," is superior by far to the usual methods of either relying on your memory (at two in the morning?) or copying the entire disk.

Although Digital Research's documentation for their CP/M-80 and CP/M-86 operating systems gives no indication that a file is marked in any manner when it is written to or renamed, you nevertheless can implement an incremental backup utility exactly as described above—the ideal solution. BU is one example of such an implementation.

An expedition into the jungle of Undocumented Features comes home with an orphan attribute bit.

Theory and Practice

For a detailed explanation of the inner workings of BU, you should read the comments accompanying the source code in the Listing (page 39). These have been written to give even the novice C programmer a clear understanding of what is going on each step of the way. What follows is a general description that covers the salient features of BU from a user's viewpoint.

The *CP/M-80 Version 2 Interface Guide*'s description of BDOS Service 30 (Set File Attributes) states that the file attribute bit t3-prime is "reserved for future system expansion." However, if you use a disk utility to set t3-prime to true in the file's directory entry, you will find that the BDOS resets it to false (zero) whenever the directory entry is changed. Since this means that the file has been opened, written to, and closed (or else renamed) by the BDOS, t3-prime is apparently an undocumented attribute bit that indicates when a file has been updated.

This behavior is not an unreliable artifact of some other process; DRI added a very similar feature, called the "Archive" attribute, to its multiuser MP/M 2 operating system. The version of PIP.COM supplied with MP/M 2 features an "A" option, which causes PIP to copy only those files that have their Archive bits set false. After copying each file, PIP sets the bit true. It seems logical, then, that in writing CP/M-80 versions 2.x and CP/M-86, DRI intended to rewrite its version of PIP to include an "A" option but for whatever reason never got around to doing so. This leaves the user to come up with a utility that takes advantage of this orphaned attribute.

A variety of such utilities are avail-

Ian Ashdown, byHeart Software, 2 – 2016 West First Avenue, Vancouver, B.C. V6J 1G8.

able, including one in the public domain and at least two commercial products. BU's advantage is that it is written in C, thus presenting you with the opportunity to customize it to your particular needs. The source code has been profusely commented for precisely this reason.

BU accepts command lines of the following form:

```
BU x[:afn] y [-AFHQSn]
where x = drive name of disk to be
          backed up,
      y = drive name of backup disk,
```

and the optional arguments are:

-A	All files, regardless of status
-F	Fast copy (without verification)
-H	Hard disk (files may be split)
-Q	Query each file before backup
-S	System attribute copied to backup
-n	Backup USER *n* files only (0 – 31)
afn	Any legal CP/M ambiguous fileref (can only be used with -n option)

If the above is a bit confusing, some examples may help in explaining the various options:

BU a b – Copy updated files in all user areas from drive A: to drive B:.

BU c a -f – Copy updated files in all user areas from drive C: to drive A: without verification of copied files.

BU a:file.typ m -5 – Copy file A:FILE.TYP (user area 5) to same user area on drive M:.

BU a:file*.t? c -0q – Copy any files in user area 0 matching ambiguous file reference A:FILE*.t? to the same user area on drive C:. The operator is queried before each file is copied. Answering y or Y for "Yes" results in the file being archived; anything else results in the file being skipped.

BU b a -ah – Copy all files from all user areas from drive B: to drive A:. If BU runs out of backup disk space while copying a file, the file will be split across two disks.

BU a b -a -s – Copy all files from all user areas from drive A: to drive B:. Those files with the System attribute set are copied as System files to drive B:. (Note that the dash options can be separated.)

A fair amount of the code involved in BU has to do with defensive programming: always assume that the user will make a mistake. The command line is validated as thoroughly as possible. Any errors detected are displayed with an appropriate message, along with the previously listed explanation of what command lines are valid.

Once in operation, assuming no options have been selected or ambiguous file references specified, the program will scan the directory of the disk in drive x, note which files have been changed since the last time BU was run on that disk, then copy only those files to the disk in drive y. Existing files with the same fileref and user number on the backup disk are automatically erased.

Because the new files are backup copies, each one is read after it is written and verified character by character with the original file. All available memory is used to buffer the data for disk read and write so that BU can copy and verify as quickly as possible. Once the new file has been fully verified, its

file attributes are set to "directory" (DIR) and "read-only" (R/O) to ensure that it can be displayed in a directory listing of the backup disk and that it cannot be accidentally erased.

If the combined size of the files to be backed up exceeds the available space on the backup disk, BU will take one of two actions, depending on whether or not the –H option has been selected. In the default mode, BU will stop when it runs out of disk space and erase the current, partially written, backup file, It will then ask the operator to insert a fresh disk in the backup drive. When this is done, BU will begin to copy files to the new disk.

The –H option is intended primarily for use with hard disks, where the size of the files may exceed the capacity of new backup disks. When BU runs out of disk space with this option active, it will close the current, partially written, backup file, set its attributes to DIR and R/O, then ask the operator to insert a fresh disk in the backup drive. When this is done, BU will open a sequentially numbered fileref (e.g., FILE.TYP would become FILE--01.TYP) and continue writing the current file to this new backup file from where it left off. The file in effect will be split across two or more backup disks, with no wasted disk space.

Reassembling these split files is quite simple. In principle, you need only open the first file for write access, use lseek( ) to find its end, open the second file for read access, then append it to the first file. The C code required to do this is left to the reader as an exercise. Alternatively, you can always use the concatenation feature of DRI's PIP.COM utility. The command line would be:

```
PIP rebuilt.fil = first.fil,second.fil
```

The disadvantage of this approach is that all three files must be on-line at the same time, which in a two-drive system means shuffling one of them to the destination disk before you can concatenate and rebuild your original file.

What BU does not address is archival file maintenance procedures. If BU is used with a dual floppy drive system and every working disk has its own backup copy, there is no problem. At the end of your programming session, simply insert the backup disk in the second drive, type "BU x y," and every changed file is automatically updated. The backup disk becomes a duplicate of the working disk (although you do have to manually erase files that were deleted from the working disk).

The problem arises when you want to maintain backup copies of hard disk files. Depending on whether or not the –H option is used, files will be on different disks and possibly split across disks after you run BU. When the files later are changed again, it may happen that BU will archive them on different disks than the backup copies. BU automatically erases existing files with the same fileref and user number on the backup disk before copying a file, but it can't do that when the file is on another disk. This leaves the responsibility of deleting (or otherwise archiving) obsolete versions of files to the user.

It is a very simple matter to extend BU so that a disk identifier file is added to each backup disk. This would be especially useful when using multiple floppies to archive hard disk files; the identifier files could be numbered sequentially. However, because disk naming and storage procedures are very much a matter of individual taste, this has been left to the reader to implement.

Etymological Nuisances

BU will mishandle random access files that have been created with unwritten records or unallocated blocks or extents. Because it uses the BDOS sequential read service to access the files, BU will stop reading random access files at the first unallocated block or extent. Happily, very few programs behave in such an unfriendly manner. (This is perhaps because most file copy utilities will balk at such files as well.)

Speaking of file copy utilities, a few are available that, under certain circumstances, will write a file without resetting its *Archive attribute*. One of these, oddly enough, is DRI's own PIP.COM. If a file on a disk has its Archive and Read-Only attribute bits set true, you can copy another file with the same fileref and user number to the disk with PIP only by using the "W" option. However, the BDOS will not reset the Archive attribute bit afterwards, so BU will be unable to recog-

nize the file as changed. The only solution here is to be aware of the problem and, if necessary, to perform a manual file backup immediately after using the utility.

BU accepts ambiguous file references only when a user number is also specified. In one sense, this is an aspect of the user interface design: a user normally should be allowed access to files in one user area only, especially when operations using ambiguous filenames are being performed. More truthfully, BDOS Services 17 (Search for First File) and 18 (Search for Next File) will not accept file references for all user areas. Either they search for all files in all user areas (including erased files), or they search a particular user area only. The only way BU could find an unambiguous or ambiguously specified file in all user areas would be to search the disk directory 32 times!

Suggested Enhancements

Those of you with the ambition and the time can extend BU to become a complete file archiving utility. If your system has a hard disk, BU could maintain a catalog file that records each fileref and version number (without erasing previous versions), any pertinent file statistics such as size in kilobytes and assigned file attributes, the backup disk identification number, the time and date the backup copy was made, and the operator who made the backup. Even if you don't own a hard disk, you could maintain a similar sort of file on each backup disk for record-keeping purposes. As a starting point for such a program, you might consider the "Archive" program that is developed in Kernighan & Plauger's book *Software Tools* (Addison-Wesley, ISBN 1-201-03669-X). The program is presented in source code using RATFOR, which for all practical purposes is a variation on a theme of the C programming language.

So, there you have it: a fully functional incremental backup utility for CP/M-80 and CP/M-86. Why DRI did not include an Archive option with PIP for these operating systems after going to the trouble of designing all the necessary features into them is a matter for future historians of computer software to ponder. In the meantime, I hope you enjoy using BU. **DDJ**

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 193.**

Archiving Files Listing (Text begins on page 36)

```
/* BU.C - A File Backup Utility for CP/M-80 & CP/M-86
 *
 * Copyright:    Ian Ashdown
 *               byHeart Software
 *               2 - 2016 West First Avenue
 *               Vancouver, B.C. V6J 1G8
 *               Canada
 *
 * Acknowledgment: DeSmet C code and suggestions for program
 *                 improvement courtesy of Dr. Dobb's Journal
 *                 Contributing Editor Anthony Skjellum
 *
 * Version:      1.1     Written for Aztec CII v1.06b (CP/M-80)
 *                       and DeSmet C88 v2.2 (CP/M-86)
 *
 * Date:         December 31st, 1983 (Version 1.0)
 *               September 7th, 1984 (Version 1.1)
 *
 * BU utilizes the undocumented "archive" file attribute feature
 * of CP/M-80 Versions 2.x and CP/M-86 to automatically detect
 * files that have been changed since the disk was last "backed
 * up" and copy them (with verification) to a backup disk. This
 * program performs the same action as the "A" option of PIP
 * under Digital Research's MP/M 2, for which the Archive
 * attribute is documented.
 *
 * Usage: BU x[:afn] y [-AFHQSn]
 *
 *        where x = drive name of disk to be backed up
 *              y = drive name of backup disk
 *
 *        and the optional arguments are:
 *
 *              -A        All files, regardless of status
 *              -F        Fast copy (without verification)
 *              -H        Hard disk (files may be split)
 *              -Q        Query each file before backup
 *              -S        System attribute copied to backup
```

(Continued on next page)

Archiving Files Listing (Listing Continued, text begins on page 36)

```
 *              -n        Backup USER 'n' files only (0-31)
 *              afn       Any legal CP/M ambiguous fileref
 *                        (can only be used with -n option)
 */

#include "stdio.h"
#include "ctype.h"      /* Contains macro for "isdigit()" */

/*** DEFINITIONS ***/

#define AZTEC   1       /* Aztec CII v1.06b (CP/M-80) */
#define DESMET  0       /* DeSmet C88 v2.2 (CP/M-86) */

#if DESMET
#define movmem(src,dest,len)    _mov(len,src,dest)
#endif

#define ERROR       -1
#define DEL         -1  /* Deleted fileref flag */
#define ALL         -1  /* All user numbers flag */
#define TRUE        -1
#define FALSE        0
#define SUCCESS      0
#define O_RDONLY     0
#define USER_ERR     0  /* Specified fileref must only be used
                           with -number command-line option */
#define BAD_FREF     1  /* Illegal file reference */
#define BAD_ARGS     2  /* Illegal command line */
#define BAD_OPT      3  /* Illegal option */
#define BAD_USER     4  /* Illegal user number */
#define BAD_DRV      5  /* Illegal drive names */
#define SAME_DRV     6  /* Same drive name for output as input */
#define OPN_ERR      8  /* File open error */
#define READ_ERR     9  /* File read error */
#define CLS_ERR     10  /* File close error */
#define BAD_VFY     11  /* File verify error */
#define DIR_IO       6  /* BDOS Direct I/O service */
#define RESET_DRV   13  /* BDOS Reset All Drives service */
#define SEL_DRV     14  /* BDOS Select Drive service */
#define SRCH_F      17  /* BDOS Search Next service */
#define SRCH_N      18  /* BDOS Search Next service */
#define GET_DRV     25  /* BDOS Get Default Drive service */
#define SET_DMA     26  /* BDOS Set DMA Address service */
#define SET_ATT     30  /* BDOS Set File Attributes service */
#define USER_CODE   32  /* BDOS Get/Set User Code service */
#define MAX_USER    32  /* 32 user codes under CP/M (see DR's
                           documentation for BDOS Service 32) */

/*** GLOBAL VARIABLES ***/

char ent_drv,   /* Entry drive code */
     ent_user,  /* Entry user code */
     cur_user;  /* Current user code */

/*** MAIN BODY OF CODE ***/

main(argc,argv)
int argc;
char *argv[];
{
  char in_dsk,          /* Drive name of input disk */
       out_dsk,         /* Drive name of output (backup) disk */
       in_drv,          /* Drive code of input disk */
       out_drv,         /* Drive code of output disk */
       seg_no,          /* Segment number for split files */
       in_file[15],     /* Fileref of current input file */
       out_file[15],    /* Fileref of current output file */
```

(Continued on page 42)

```
    c,                  /* Scratch variable */
    *s,                 /* Scratch string pointer */
    *buffer,            /* Pointer to directory entry returned */
                        /* by "srch_file()" */
    *srch_file(),                          \
    *malloc();

/* File control blocks of current input and output files */

static char in_fcb[33],         /* (Automatically initialized */
            out_fcb[33];        /* to zero by compiler) */

/* Structure for linked list of filerefs */

struct file_ref
{
  char name[12];                /* File reference */
  struct file_ref *next;        /* Pointer to next instance */
} root,                         /* Start of linked list */
  *fref_1,                      /* Scratch pointers to */
  *fref_2;                      /* linked list instances */

/* Initialized file control block for "srch_file()". This FCB
   is for a fully ambiguous fileref that causes "srch_file()"
   to return all directory entries for the current default
   drive. */

static char fcb[] = {'?','?','?','?','?','?','?','?',
                     '?','?','?','?','?',0,0,0};

int file_cnt = 0,       /* Count of file to be backed up */
    dup_flag,           /* Duplicate fileref flag */
    all_files,          /* All_files flag (cmd-line option) */
    fast_copy,          /* Fast copy flag (cmd-line option) */
    hard_disk,          /* Hard disk flag (cmd-line option) */
    query,              /* Query flag (cmd-line option) */
    system,             /* System flag (cmd-line option) */
    user_no,            /* User number (cmd-line option) */
    next_flag = FALSE;/* Flag to indicate to "srch_file()"
                         that a "search next" is required */

register int i,j;       /* Loop indices */

long begin,             /* Input file position variables */
     end;

/* Display program header */

printf("\nBU Version 1.1");
printf("                          Copyright 1983, 1984");
printf(" byHeart Software\n\n");

/* Initialize command-line options */

all_files = FALSE;      /* Copy only non-archived files */
fast_copy = FALSE;      /* Copy files with verification */
hard_disk = FALSE;      /* Files will not be split across backup
                           disks if remaining capacity of backup
                           disk is less than current file size */
query = FALSE;          /* Backup without query */
system = FALSE;         /* Assign directory attribute to all
                           backup files */
user_no = ALL;          /* Backup files in all user areas */

/* Parse command line for user-selected options (if any) */

if(argc < 3)
```

(Continued on page 44)

```
  error(BAD_ARGS,NULL);        /* Illegal command line */
if(argc > 3)
{
  i = 3;  /* Start with third command-line argument */
  while(i < argc)
  {
    if(*argv[i] != '-')
      error(BAD_OPT,argv[i]);  /* Missing leading '-' */
    s = argv[i]+1;
    while(*s)
    {
      if(*s == 'A')          /* Check for all_files option */
        all_files = TRUE;
      else if(*s == 'F')  /* Check for fast copy option */
        fast_copy = TRUE;
      else if(*s == 'H')  /* Check for hard disk option */
        hard_disk = TRUE;
      else if(*s == 'Q')  /* Check for query option */
        query = TRUE;
      else if(*s == 'S')  /* Check for system option */
        system = TRUE;
      else if(isdigit(*s))  /* Check for user number option */
      {
        user_no = *s++ - '0';
        if(isdigit(*s))
          user_no = user_no * 10 + *s++ - '0';
        if(user_no < 0 || user_no > 31)
          error(BAD_USER,argv[i]);
        continue;
      }
      else
        error(BAD_OPT,argv[i]);
      s++;
    }
  i++;
  }
}

/* Validate input parameters */

if(*(argv[1]+1) != '\0')  /* Check for specified fileref */
{
  if(user_no == ALL)        /* Can only use with specified */
    error(USER_ERR,NULL);  /* user number (-n option) */

  /* Modify "fcb[]" to incorporate fileref */

  if(copy_fref(fcb,argv[1]) == ERROR)
    error(BAD_FREF,argv[1]);
}
if(*argv[1] < 'A' || *argv[1] > 'P' ||
    *argv[2] < 'A' || *argv[2] > 'P')
  error(BAD_DRV,NULL);          /* Illegal drive names */
if(*argv[1] == *argv[2])
  error(SAME_DRV,NULL);         /* Drive names are same */

/* Save entry drive and user codes */

ent_drv = bdos(GET_DRV);
ent_user = bdos(USER_CODE,0xff);

/* Calculate input and output drive codes */

in_drv = (in_dsk = *argv[1]) - 'A' + 1;
out_drv = (out_dsk = *argv[2]) - 'A' + 1;

/* Set default drive to input drive */
```

```
bdos(SEL_DRV,in_drv-1);

/* Set user code to "user_no" if -n option specified */

if(user_no != ALL)
  bdos(USER_CODE,user_no);

/* Read first 12 bytes of updated active directory entries int
   linked list of filerefs. If first byte of entry is 0xe5,
   then file has been erased. */

root.next = NULL;      /* Initialize linked list root */
fref_1 = &root;        /* Initialize linked list pointer */
while(buffer = srch_file(fcb,next_flag))
{
  /* Bit 7 of third filetype byte (t3') in directory entry
     is the Archive attribute indicator. The BDOS sets this
     bit to zero whenever it updates a directory entry. */

  if(buffer[0] != 0xe5 &&
    (buffer[0] == user_no || user_no == ALL) &&
    (!(buffer[11] & 0x80) || all_files))
  {
    fref_1->next =    /* Allocate space for fileref instance */
        (struct file_ref *) malloc(sizeof(struct file_ref));
    fref_1 = fref_1->next;    /* Assign space to instance */
    movmem(buffer,fref_1->name,12);  /* Move fileref to */
                                     /* linked list instance */
    fref_1->next = NULL;  /* Indicate current end of list */
  }
  next_flag = TRUE;    /* Only first call to "srch_file()" */
                       /* should be made with "next_flag" */
}                      /* set to FALSE */

/* If no files have been backed up ... */

if(!root.next)  /* Null "root.next" indicates no files have */
  {             /* been changed */
    printf("NO FILES HAVE BEEN UPDATED");
    if(user_no != ALL)
      printf(" in user area %d\n",user_no);
    else
      putchar('\n');
    reset();  /* Reset user and drive codes to entry values */
    exit(0);
  }

/* There may be duplicate filerefs in linked list due to some
   files occupying more than one extent on the disk. These
   duplicates must be marked as "deleted" in the list.
   (Duplicate filerefs with different user codes are valid.) */

/* For all filerefs ... */

fref_1 = &root;  /* Initialize a linked list pointer */
while(fref_1->next)
{
  fref_1 = fref_1->next;  /* Root instance is NULL entry */
  dup_flag = FALSE;       /* Reset duplicate fileref flag */

 /* For all preceding filerefs ... */

 fref_2 = &root;  /* Initialize another linked list pointer */
 fref_2 = fref_2->next;  /* Skip root instance */
 while(fref_2->next != fref_1->next)
 {
   /* Compare filerefs (ignore deleted filerefs and different
      user codes) */
```

(Continued on next page)

```
      if(fref_2->name[0] != DEL && fref_1->name[0] == fref_2->name[0])
        if(!strncmp(fref_1->name+1,fref_2->name+1,11))
        {
          dup_flag = TRUE;        /* Indicate duplicate fileref */
          break;
        }
      fref_2 = fref_2->next;
    }
    if(dup_flag == TRUE)
      fref_1->name[0] = DEL;      /* Delete if duplicate fileref */
    else
      file_cnt++;                 /* Increment file count */
  }

  /* Display file copy header */

  printf("Number of files to be copied: %d\n\n",file_cnt);
  printf("User:     Files being copied to Drive %c:\n\n",
         out_dsk);

  /* Initialize current input and output fileref templates */

  in_file[0] = in_dsk;
  out_file[0] = out_dsk;
  in_file[1] = out_file[1] = ':';
  in_file[10] = out_file[10] = '.';
  in_file[14] = out_file[14] = '\0';

  /* Initialize current input and output FCB templates */

  in_fcb[0] = in_drv;
  out_fcb[0] = out_drv;

  /* For all validated filerefs do ... */

  for(cur_user = 0; cur_user < MAX_USER; cur_user++)
  {
    if(user_no != ALL)
      if(cur_user != user_no)
        continue;
    bdos(USER_CODE,cur_user);     /* Set user code to "cur_user" */
    fref_1 = &root;       /* Initialize linked list pointer */
    while(fref_1->next)
    {
      fref_1 = fref_1->next;      /* Root instance is NULL entry */
      if(fref_1->name[0] == cur_user)
      {
        /* Update the current input and output FCB's */

        movmem(fref_1->name+1,in_fcb+1,11);
        movmem(fref_1->name+1,out_fcb+1,11);

  /* Reset the Read-Only and System attribute bits of the
     FCB's so that the file can be copied and displayed
     (unless the "system" flag is TRUE) */

  out_fcb[9] &= 0x7f;       /* Read-Only attribute */
  if(!system)
    out_fcb[10] &= 0x7f;    /* System attribute */

  /* Set the Archive attribute bit of the FCB's to indicate
     that the file has been backed up */

  in_fcb[11] |= 0x80;
  out_fcb[11] |= 0x80;
```

(Continued on page 48)

Archiving Files Listing (Listing Continued, text begins on page 36)

```
/* Move the fileref from the FCB's to the initialized
   input and output fileref templates */

movmem(in_fcb+1,in_file+2,8);   /* Filename move */
movmem(out_fcb+1,out_file+2,8);
movmem(in_fcb+9,in_file+11,3);  /* Filetype move */
movmem(out_fcb+9,out_file+11,3);

/* Strip high order bits off filerefs to form proper
   ASCII characters */

for(j = 2; j <= 13; j++)
{
  in_file[j] &= 0x7f;
  out_file[j] &= 0x7f;
}

/* Display the filerefs */

printf(" %2d      %s --> %s",cur_user,in_file,out_file);

/* Query operator for backup if indicated by "query"
   flag */

if(query)
{
  printf("  O.K. to backup?  ");
  if((c = in_chr()) == 'y' || c == 'Y')
    puts("Yes");
  else
  {
    puts("No");
    continue;   /* Go do next fileref if "No" */
  }
}
else
  putchar('\n');

/* Copy file from the input disk to the output disk */

if(hard_disk)   /* Split file across backup disks if */
{               /* necessary */
  begin = 0L;   /* Initialize file position pointer */
  seg_no = 0;   /* and split file segment number */
  do
{
  /* Reset the Read-Only attribute of the output file
     (if it exists) so that the input file can be
     copied to it */

  bdos(SET_ATT,out_fcb);

  end = copy_file(in_file,out_file,begin);
  if(!fast_copy)  /* Verify file unless -F selected */
    verify_file(in_file,out_file,begin);

  /* Set the Read-Only attribute of the output file */

  out_fcb[9] |= 0x80;
  bdos(SET_ATT,out_fcb);
  out_fcb[9] &= 0x7f;    /* ... and reset the fcb */
  if(end != NULL)
  {
    /* File has been partially written on current
       backup disk - new disk required to continue */
```

```
      new_disk(out_file,hard_disk);

      /* Append segment number to filename of output
         fileref (e.g. - B:FILE.TYP will become
         B:FILE--01.TYP) */

      seg_no++;
      for(j = 2; j <= 7; j++)    /* Change spaces to */
        if(out_file[j] == ' ')   /* '-' character */
          out_file[j] = '-';
      out_file[8] = seg_no/10 + '0';  /* Append segment */
      out_file[9] = seg_no%10 + '0';  /* number */

      /* Display filerefs again */

      printf(" %2d       %s --> %s\n",
          cur_user,in_file,out_file);
    }
    begin = end;
  }
  while(end != NULL);  /* Loop until file is written */
}
else
{
  /* Reset the Read-Only attribute of the output file
     (if it exists) */

  bdos(SET_ATT,out_fcb);

  if(copy_file(in_file,out_file,0L) != NULL)
  {
    /* Disk was full - erase partially written file, back
       up fileref pointer and rewrite file to new disk */

    unlink(out_file);
    new_disk(out_file,hard_disk);
    i--;
    continue;
  }
          if(!fast_copy)  /* Verify file unless -F selected */
            verify_file(in_file,out_file,0L);

          /* Set the Read-Only attribute of the output file */

          out_fcb[9] |= 0x80;
          bdos(SET_ATT,out_fcb);
        }

        /* Set the Archive attribute of the input file to
           indicate that the file was successfully backed up */

        bdos(SET_ATT,in_fcb);
      }
    }
  }
  reset();      /* Reset user and drive codes to entry values */
}

/*** FUNCTIONS ***/

/* Search for first or next directory entry */

char *srch_file(fcb_ptr,next_flag)
char *fcb_ptr;          /* Pointer to file control block */
int next_flag;          /* Flag to indicate "search next" */
{
  static char sf_cur[32],  /* Current directory entry buffer */
              sf_fcb[36];  /* File control block buffer */
```

(Continued on next page)

```
  int index,  /* Index of directory entry in DMA buffer */
      *ptr;   /* Pointer to directory entry in DMA buffer */

  bdos(SET_DMA,0x80);   /* Set DMA address to 80h */
  if(!next_flag)
  {
    movmem(fcb_ptr,sf_fcb,16);  /* Initialize FCB buffer */
    if((index = bdos(SRCH_F,sf_fcb)) == 0xff)  /* Find first */
      return NULL;       /* Return NULL if unsuccessful */
  }
  else
    if((index = bdos(SRCH_N,NULL)) == 0xff)  /* Find next */
      return NULL;       /* Return NULL if unsuccessful */

  /* BDOS services 17 and 18 leave four consecutive directory
     entries of 32 bytes each in the 128-byte DMA buffer and also
     returns an index value of 0, 1, 2 or 3 to indicate the
     correct directory entry in the accumulator. The "bdos()"
     function returns this index value. */

  ptr = 0x80 + index * 32;  /* Calculate pointer to directory
                               entry */
  movmem(ptr,sf_cur,32);  /* Move directory entry to current
                             directory entry buffer */
  return sf_cur;
}

/* Copy file starting at "offset" from beginning */

copy_file(in_file,out_file,offset)

char *in_file,          /* Input fileref */
     *out_file;         /* Output fileref */
long offset;            /* Input file position offset */
{
  register int in_cnt,  /* Character counts for unbuffered I/O */
               out_cnt;
  int fd_in,            /* Input file descriptor */
      fd_out,           /* Output file descriptor */
      full_disk = FALSE;  /* Full disk flag */

  char *buffer,     /* Input file buffer */
       *buff_ptr,   /* Pointer to current position in "buffer[]" */
       *malloc();

  unsigned buf_size = 32768;  /* Initial memory allocation size */

  /* "read()" accepts a maximum of 32768 bytes at a time. Allocate
     as much memory as possible up to this limit for the input
     buffer, using 128 byte decrements. */

  do
    if(buffer = malloc(buf_size))
      break;
  while(buf_size -= 128);

  /* Open input file for unbuffered Read-Only access */

  if((fd_in = open(in_file,O_RDONLY)) == ERROR)
    error(OPN_ERR,in_file);

  /* Create the output file by first deleting it (if it
     exists), then opening it for unbuffered Write-Only
     access. */
```

(Continued on page 52)

```
 if((fd_out = creat(out_file,NULL)) == ERROR)
   error(OPN_ERR,out_file);

 /* Initialize input file position pointer to "offset" */

 lseek(fd_in,offset,0);

 /* Copy input file to output file by buffering data
    through "buffer[]" */

 do
 {
   if((in_cnt = read(fd_in,buffer,buf_size)) == ERROR)
     error(READ_ERR,in_file);
   buff_ptr = buffer;  /* Initialize "buffer[]" pointer */
   out_cnt = 0;        /* and "out_cnt" */
   do
   {
     /* Write contents of "buffer[]" to output file in 128
        byte records until either the buffer is written or a
        write error occurs. Since the 0x1a (^Z) character CP/M
        uses as an EOF marker is a valid file character for non-
        ASCII files, "read()" always reads the last 128 byte
        record of a file under CP/M. */

     if(write(fd_out,buff_ptr,128) != 128)
       {
         /* The standard implementation of "write()" does not
            distinguish between a full disk or directory and a
            write error in its returned error code. Thus, it is
            assumed that an error means a full disk/directory. */

         full_disk = TRUE;
         break;
       }
       buff_ptr += 128;  /* Increment "buffer[]" ptr */
       out_cnt += 128;  /* Update count of chars written */
     }
     while(in_cnt > out_cnt);     /* Until end of "buffer[]" */
     offset += out_cnt;  /* Update input file position pointer */
     if(full_disk == TRUE)
       break;
   }
   while(in_cnt == buf_size);     /* Until end of file */
   free(buffer);          /* Deallocate buffer space */
   if(close(fd_in) == ERROR)      /* Close the files */
     error(CLS_ERR,in_file);
   if(close(fd_out) == ERROR)
     error(CLS_ERR,out_file);

   /* If full disk return new offset for input file, else */
   /* return NULL to indicate completion of file copy operation */

   return (full_disk ? offset : NULL);
}

/* Compare portion of input file starting at "offset" from begin-
   ning of file with output file */

verify_file(in_file,out_file,offset)
char *in_file,          /* Input fileref */
     *out_file;         /* Output fileref */
long offset;            /* Input file position offset */
{
  register int match_cnt;       /* Scratch variable */
```

```
  int out_cnt,  /* Character counts for unbuffered I/O */
      fd_in,    /* Input file descriptor */
      fd_out;   /* Output file descriptor */

  char *buffer,          /* Dynamically-allocated buffer */
       *in_ptr,          /* Input file buffer pointer */
       *out_ptr,         /* Output file buffer pointer */
       *malloc();

  unsigned buf_size = 65280;  /* Initial memory allocation size */

  /* "read()" and "write()" accept a maximum of 32768 bytes at a
     time. Allocate as much memory as possible up to this limit
     for both the input and output buffers, using 256 byte
     decrements (128 bytes for each buffer). */

  do
    if(buffer = malloc(buf_size))
      break;
  while(buf_size -= 256);

  /* Divide "buffer[]" in two for "in_ptr" and "out_ptr" */

  buf_size /= 2;

  if((fd_in = open(in_file,O_RDONLY)) == ERROR)  /* Open files */
    error(OPN_ERR,in_file);
  if((fd_out = open(out_file,O_RDONLY)) == ERROR)
    error(OPN_ERR,out_file);
  lseek(fd_in,offset,0);  /* Initialize file position pointer */

  /* Read in characters from both files and compare */

  do
  {
    in_ptr = buffer;    /* Assign buffer pointers */
    out_ptr = in_ptr + buf_size;
    if(read(fd_in,in_ptr,buf_size) == ERROR)
      error(READ_ERR,in_file);
    if((out_cnt = read(fd_out,out_ptr,buf_size)) == ERROR)
      error(READ_ERR,out_file);
    match_cnt = out_cnt;
    while(match_cnt--)                  /* Verify character */
      if(*in_ptr++ != *out_ptr++)       /* by character, and */
      {                                 /* delete the output */
        if(close(fd_out) == ERROR)      /* file if they fail */
          error(CLS_ERR,out_file);      /* to match */
        unlink(out_file);
        error(BAD_VFY,out_file);
      }
  }
  while(out_cnt == buf_size);   /* Until end of output file */
  free(buffer);                 /* Deallocate buffer space */
  if(close(fd_in) == ERROR)     /* Close the files - verifi- */
    error(CLS_ERR,in_file);     /* cation was successful */
  if(close(fd_out) == ERROR)
    error(CLS_ERR,out_file);
}

/* Copy fileref to file control block */

copy_fref(fcb,fref)
char *fcb,              /* Pointer to file control block */
     *fref;             /* Pointer to fileref */
{
  char c;       /* Scratch variable */
  int i,        /* Fileref index variable */
      j,        /* FCB index variable */
      k,        /* Scratch variable */
      done;     /* Loop break flag */
```

(Continued on next page)

Archiving Files Listing (Listing Continued, text begins on page 36)

```
if(fref[1] != ':' || fref[2] == '\0')
  return ERROR;  /* No drive code separator or null fileref */

/* Calculate drive code from drive name and put in FCB */

fcb[0] = fref[0] - 'A' + 1;

/* Process remainder of fileref */

 done = FALSE;
 for(i = 2,j = 1;i <= 9;i++,j++)        /* Skip drive code in */
 {                                      /* fileref */
   switch(c = fref[i])
   {
     case '.':          /* Filetype separator */
       if(i == 2)
         return ERROR; /* Null filename */
       for( ; j <= 8; j++)
         fcb[j] = ' '; /* Pad filename with trailing blanks */
       done = TRUE;
       break;
     case '*':          /* Match any following string */
       for( ; j <= 8; j++)
         fcb[j] = '?'; /* Pad filename with trailing */
       i++;             /* question marks */
       done = TRUE;
       break;
     case '\0':         /* End of fileref */
       for( ; j <= 11; j++)  /* Pad FCB with trailing spaces */
         fcb[j] = ' ';
       return SUCCESS;
     case ',':          /* Illegal filename characters */
     case ';':
     case ':':
     case '=':
     case '[':
     case ']':
     case '_':
     case '<':
     case '>':
       return ERROR;
     default:
       if(c >= '!' && c <= '~')
         fcb[j] = c;    /* Copy character from fileref to FCB */
       else
         return ERROR; /* Nonprintable character or ' ' */
   }
   if(done)
     break;
 }
 c = fref[i];
 if((c = fref[i]) == '\0')      /* End of fileref */
 {
   for( ; j <= 11; j++)  /* Pad FCB with trailing spaces */
     fcb[j] = ' ';
   return SUCCESS;
 }
 else if(c == '.')      /* Filetype separator */
 {
   i++;
   k = i + 2;           /* Set limit of 3 characters */
   for( ; i <= k; i++,j++)
   {
     done = FALSE;
     switch(c = fref[i])
```

(Continued on page 56)

```
    {
      case '*':       /* Match any following string */
        for( ; j (= 11; j++)
          fcb[j] = '?';       /* Pad filetype with trailing */
        return SUCCESS;       /* question marks */
      case '\0':      /* End of fileref */
        for( ; j (= 11; j++)  /* Pad FCB with trailing */
          fcb[j] = ' ';       /* spaces */
        return SUCCESS;
      case '.':               /* Illegal filetype characters */
      case ';':
      case ',':
      case ':':
      case '=':
      case '[':
      case ']':
      case '_':
      case '(':
      case ')':
        return ERROR;
      default:
        if(c )= '!' && c (= '~')
          fcb[j] = c;  /* Copy character from fileref */
        else           /* to FCB */
          return ERROR;  /* Nonprintable character or ' ' */
    }
  }

  /* Return ERROR if filetype too long */

  return (fref[i] == '\0' ? SUCCESS : ERROR);
 }
 else
  return ERROR;  /* Filename too long  */
}

/* Error report */

error(n,s)
int n;          /* Error code */
char *s;        /* Pointer to optional string */
{
 switch(n)
 {
  case USER_ERR:
    printf("\007** ERROR - No user number specified **\n");
    break;
  case BAD_FREF:
    printf("\007** ERROR - Illegal file reference: %s **\n",s);
    break;
  case BAD_ARGS:
    printf("\007** ERROR - Illegal command line **\n");
    break;
  case BAD_OPT:
    printf("\007** ERROR - Illegal command line option:");
    printf(" %s **\n",s);
    break;
  case BAD_USER:
    printf("\007** ERROR - User number must be inside range");
    printf(" of 0 to 31 **\n");
    break;
  case BAD_DRV:
    printf("\007** ERROR - Drive names must be inside range");
    printf(" of 'A' to 'P' **\n");
```

(Continued on page 58)

```
      break;
    case SAME_DRV:
      printf("\007** ERROR - Drive names cannot be equal **\n");
      break;
    case OPN_ERR:
      printf("\007\n** ERROR - Cannot open file %s **\n",s);
      reset();
      exit(0);
    case READ_ERR:
      printf("\007\n** ERROR - Read error on file %s **\n",s);
      reset();
      exit(0);
    case CLS_ERR:
      printf("\007\n** ERROR - Cannot close file %s **\n",s);
      reset();
      exit(0);
    case BAD_VFY:
      printf("\007\n** ERROR - Failed verify on file %s **\n",s);
      reset();
      exit(0);
  }
  printf("\nUsage: BU x[:afn] y [-AFHQSn]\n\n");
  printf("       where x = drive name of disk to be backed up\n");
  printf("             y = drive name of backup disk\n\n");
  printf("       and the optional arguments are:\n\n");
  printf("            -A          All files, regardless of");
  printf(" status\n");
  printf("            -F          Fast copy (without ");
  printf(" verification)\n");
  printf("            -H          Hard disk (files may be");
  printf(" split)\n");
  printf("            -Q          Query each file before");
  printf(" backup\n");
  printf("            -S          System attribute copied to");
  printf(" backup\n");
  printf("            -n          Backup USER 'n' files only");
  printf(" (0-31)\n");
  printf("            -afn        Any legal CP/M ambiguouos");
  printf(" fileref\n");
  printf("                        (can only be used with -n");
  printf(" option)\n");
  exit(0);
}

/* Request a new backup disk to be inserted in the output drive */

new_disk(name,hard_disk)
char *name;
int hard_disk;
{
  char d;

  printf("\007\n          ** BACKUP DISK FULL **\n\n");
  if(hard_disk)
    printf("WARNING: -H option active - FILE WILL BE SPLIT\n\n");
  printf("Insert new disk into drive %c to continue.\n",name[0]);
  printf("Type 'C' when ready or 'A' to abort ... ");
  while((d = in_chr()) != 'c' && d != 'C' && d != 'a' && d != 'A')
    ;
  if(d == 'a' || d == 'A')
   {
     unlink(name);
     exit(0);
   }
   else
```

(Continued on page 60)

```
    printf("\n\n");
  bdos(RESET_DRV,NULL);          /* Reset drives */
}

/* Reset user and drive codes to entry values */

reset()
{
  bdos(USER_CODE,ent_user);
  bdos(SEL_DRV,ent_drv);
}

/* Get character from current CP/M CON: device without echo */

in_chr()
{
  int c;

  do
    c = bdos(DIR_IO,0xff);
  while(!c);
  return c;
}

/* Additional function required for DeSmet C version of BU86.C */

#if DESMET
bdos(beta,delta)
int beta,
    delta;
{
  return _os(beta,delta);
}
#endif

/*** EXPLANATION OF AZTEC CII STDIO.H FUNCTIONS ***/

/* bdos(bc,de)  ¦ DeSmet equivalent is defined under FUNCTIONS ¦
 * int bc,de;
 *
 * Calls CP/M's BDOS with 8080 CPU register pair BC set to "bc"
 * and register pair DE set to "de". The value returned by the
 * 8080 CPU accumulator is the return value.
 *
 * movmem(src,dest,length)  ¦ DeSmet equivalent is defined  ¦
 * char *dest, *src;        ¦ under DEFINITIONS             ¦
 * int length;
 *
 * Moves data from "src" to "dest". The number of bytes is
 * specified by the parameter "length".
 *
 * strncmp(str1,str2,max)  ¦ Identical for DeSmet ¦
 * char *str1,*str2;
 * int max;
 *
 * Compares "str1" to "str2" for at most "max" characters, and
 * returns NULL if strings are equal, -1 if "str1" is less than
 * "str2", and +1 if "str1" is greater than "str2".
 */

/* End of BU.C */
```

End Listing

MBOOT and MODEM7 for the C-64's CP/M

by Walt Piotrowski

If you have CP/M for your Commodore 64, you may have noticed that the ads for CP/M software never mention that the program is available in Commodore's disk format. If you have tried to buy any of this software, you have found out why the ads don't mention it: with one exception, diskettes for the C-64 are not available (the only exception that I am aware of is Turbo Pascal from Borland International).

One company, Laboratory Microsystems, tried quite hard to help me buy their product but simply did not have access to any equipment that could write a diskette for the C-64. One of their suggestions was to provide the software on an IBM PC-compatible diskette; since I have access to a PC, I could transfer the program to the C-64. Unfortunately, I did not have any software for my C-64 that could do a download either.

MBOOT on the C-64—Download Only

MBOOT, from the *DDJ* CP/M exchange column, was a good attempt at a generic program to download files using Ward Christensen's Xmodem file transfer protocol. In principle, all you have to do is take the listing, change a few equates to match your modem, type in the program, and assemble it, and you are ready to download files. If you are familiar with the way that CP/M is implemented on the C-64 (see the references at the end of the article if you aren't), you have probably guessed that modifying MBOOT for this computer isn't quite that easy.

Normal CP/M operation on the C-64 does not require access to the RS-232 port, so there is no code in the BIOS to handle this port. Because the modem is controlled through the RS-232 port, the modification for MBOOT required some 6510 code in addition to MBOOT's 8080 code. This 6510 code makes RS-232 I/O requests to the Commodore I/O kernel. The problem is made a little more complex by the fact that the kernel handles the RS-232 transfers one bit at a time by servicing nonmaskable interrupts from a 6526 Interface Adapter chip. Normally, the kernel accepts an RS-232 request and sets up the hardware for the transfer but returns control to the requestor before the actual transfer is complete. The kernel then services the interrupts as they come along; other 6510 activities proceed in parallel with

"This article is about the MBOOT download program (DDJ, July 1982) and the modifications that were necessary to make it work on the C-64."

This article is about the MBOOT download program (*DDJ*, July 1982) and the modifications that were necessary to make it work on the C-64. Once you have MBOOT working, you can use a modem to download public domain software from RCPMs, and, assuming you have access to a machine with one of the more popular formats, you can also use it to set some of the commercial software that you need.

Walt Piotrowski, State University of New York, Binghamton, NY 13901.

the RS-232 transfer.

Under CP/M, however, it is quite normal to shut off the 6510 for long periods while programs run in the Z80. But the hardware does not allow the Z80 to receive these interrupts, nor does it contain any provision for switching on the 6510 if an interrupt arrives. This means that, unless MBOOT contains some provision to ensure that the 6510 is running (almost) continuously, RS-232 transfers will become garbled. Making sure that the 6510 is running while a character is being sent is not a big problem since transmission is totally under MBOOT's control. The 6510 code that I have added to make the transmission request to the I/O kernel includes a loop that waits in the 6510 until the kernel's RS-232 status byte shows that the character has been completely transmitted. Data reception is a little more complex.

Listing One (MBOOT64, page 67) is the modification of MBOOT's original 8080 code. If you were to compare MBOOT64 to MBOOT, you would find that MBOOT's inline I/O instructions, which originally were directed to the modem port, have been replaced with calls to a set of 6510 interface routines. These routines can be found at the end of Listing One. They allow the program to open the RS-232 channel (OPMDEM), close it (CLMDEM), send a character to the modem (WRMDEM), receive a character (RDMDEM), receive a character in a time out loop (RDMDEM2), and read a keyboard character (KYBD). The interface routines do not actually perform these functions: they set up parameters in memory that eventually cause a transfer to corresponding 6510 routines that do the work required. Listing Two (CPMMD65, page 79) contains the 6510 counterparts of each interface routine.

The actual transfer of control between the Z80 and the 6510 takes place at the label GO6510 in the interface routine section of MBOOT64. The transfer of control employs a user function in the BIOS that Commodore has provided to standardize the interface between CP/M programs and user-developed 6510 programs. This user function and the standard interface are described in the Commodore 64 CP/M user's guide. The modifications to MBOOT make use of request code 9 in this standard interface, and the program should be usable even if future modifications are made to Commodore's CP/M.

MBOOT (and MBOOT64) has two modes of operation: a file receive mode and a dumb terminal mode. In the file receive mode, the XMODEM protocol provides a strict handshake sequence between the computer transmitting the file and the computer receiving the file. Because of this handshaking, data reception in the file receive mode is not much more difficult than data transmission: MBOOT (and MBOOT64) sends a character to the computer that is transmitting the file then enters a wait loop. This transmitted character (ASCII ACK or NAK) is a signal that the receiving computer expects one sector of data.

In the original MBOOT, the wait loop contained code to poll the modem and a timeout counter to prevent a permanent hangup if the data never arrived. A vestige of this loop is at the label MWTI in MBOOT64. At MWTI (location 03E5), you will find a call to RDMDEM2, which activates the "read with wait" function in the 6510; this read with wait is at label INPUT2 (line 139) in CPMMD65.

Immediately after a byte is received, control is returned from the 6510 to the Z80. At this point, the 6510 is shut off while the Z80 takes care of the received character. However, this time period is very short, and the 6510 is always back on in time to accurately receive the next character.

In the dumb terminal mode, characters arrive at random times, making it impossible to predict the exact moment when a character will begin to arrive. The modification to this mode takes advantage of the fact that the terminal code is a very short loop in which a modem status check occurs every few microseconds. The modification simply puts this status check function into the 6510. Because the 6510 will now be turned on every few microseconds, it will be able to service the RS-232 interrupt either as soon as it arrives or very shortly thereafter.

The start of the terminal loop is at label TERM (location 019A) in MBOOT64. In this loop, a call to RDMDEM, one of the 6510 interface

DeSmet C

8086/8088 Development Package $109

FULL DEVELOPMENT PACKAGE
- Full K&R C Compiler
- Assembler, Linker & Librarian
- Full-Screen Editor
- Execution Profiler
- Complete STDIO Library (>120 Func)

Automatic DOS 1.X/2.X SUPPORT

BOTH 8087 AND S/W FLOATING POINT

OVERLAYS

OUTSTANDING PERFORMANCE
- First and Second in AUG '83 BYTE benchmarks

SYMBOLIC DEBUGGER $50
- Examine & change variables by name using C expressions
- Flip between debug and display screen
- Display C source during execution
- Set multiple breakpoints by function or line number

DOS LINK SUPPORT $35
- Uses DOS .OBJ Format
- LINKs with DOS ASM
- Uses Lattice® naming conventions

Check: ☐ Dev. Pkg (109)
☐ Debugger (50)
☐ DOS Link Supt. (35)

SHIP TO: _____

CWARE CORPORATION

P.O. BOX C
Sunnyvale, CA 94087
(408) 720-9696

All orders shipped UPS surface on IBM format disks. Shipping included in price. California residents add sales tax. Canada shipping add $5, elsewhere add $15. Checks must be on US Bank and in US Dollars. Call 9 a.m. – 1 p.m. to CHARGE by VISA/MC/AMEX.

Circle no. 18 on reader service card.

routines, activates the modem status check function in the 6510. This function is located at label INPUT (line 110) in CPMMD65.

Each time INPUT is entered, it examines the kernel's status byte to see if the kernel has begun to input a character. If it has, the routine waits until the character has been completely transferred (Listing Two, lines 126 – 131). INPUT and RDMDEM also return the modem character to the caller if one is ready. The wait loop in INPUT, plus the one in the character transmit routine (Listing Two, lines 102 – 104), keeps the 6510 running a very high percentage of the time, and full duplex operation at 300 baud proceeds smoothly. (More information on data rates is included near the end of the article.)

There is an additional complicating factor involving the terminal mode. As you probably know, the C-64's native character set is not ASCII. When Commodore implemented CP/M, it chose to generate ASCII with a new keyboard scan routine. This CP/M keyboard scan runs partly in the 6510 and partly in the Z80. Unfortunately, the key debounce loop is located in the Z80. This means that whenever a key is pressed the Z80 enters this delay loop; if a modem character happens to come along while the Z80 is in this loop, the modem character gets lost or garbled.

This turns out to be a significant problem because in the terminal mode every character typed at the C-64 is echoed by the computer at the other end of the line. At normal typing speeds, this debounce delay loop causes almost every echoed character to be garbled—it becomes impossible to read what you have typed. The solution was to add a new keyboard routine in the 6510 and to use that routine instead of the CP/M keyboard routine while in the terminal loop. The interface between MBOOT64 and the new keyboard routine is at label KYBD (location 04CC) in MBOOT64, and the corresponding 6510 routine is at label KBDCHR (line 154) in CPMMD65.

The keyboard routine that I implemented is a very simple one: it uses the normal C-64 kernel keyboard routines (not the CP/M routine) and simply translates received characters from the C-64 character set into ASCII through a translation table. This routine is adequate for the terminal mode, at least for me, but you will notice that it does not allow you to switch to an all upper case keyboard mode. You will also notice that the characters with ASCII codes 32 (space) and below have an auto repeat. This repeat is built into the native C-64, and it was easier to leave it in than to try to undo it.

The modification to MBOOT includes one further change. The original MBOOT contained code to buffer 16 sectors before actually doing a disk write. I tried it both ways and, with 1541 disk drives, found no speed advantage in doing this buffering. Because the program is easier to type in without this code, I omitted it. (Incidentally, the comments in the MBOOT64 listing were not in the original *DDJ* column, but I needed them to help me to understand the program. I believe that they are correct, but it isn't easy to debug a comment, so they may contain some misinterpretations.)

The Figure (at left) shows where the 8080 and 6510 portions of the modified MBOOT reside in memory. The 8080 code begins at the beginning of the TPA ($100 in Z80 space) while the 6510 code begins at location $600. You will note from Listing Two that the 6510 code is actually assembled at $1600 due to the $1000 offset between the address spaces of the two processors. The data shared between the two parts of each function resides in memory locations $603 – $605 in Z80 space.

The figure also shows the location of the character translation table that is used in the terminal keyboard routine and the new location of the kernel's RS-232 buffers. These RS-232 buffers originally were moved from their normal location so that a 48K CP/M could be used. Since writing the program, I have discovered that the 48K CP/M will occasionally send a garbage character out the RS-232 port while doing disk transfers. Most RCPMs ignore these characters, but they cause the XMODEM program on a few RCPMs to abort while doing file transfers. I have found that it is best to use a 44K CP/M

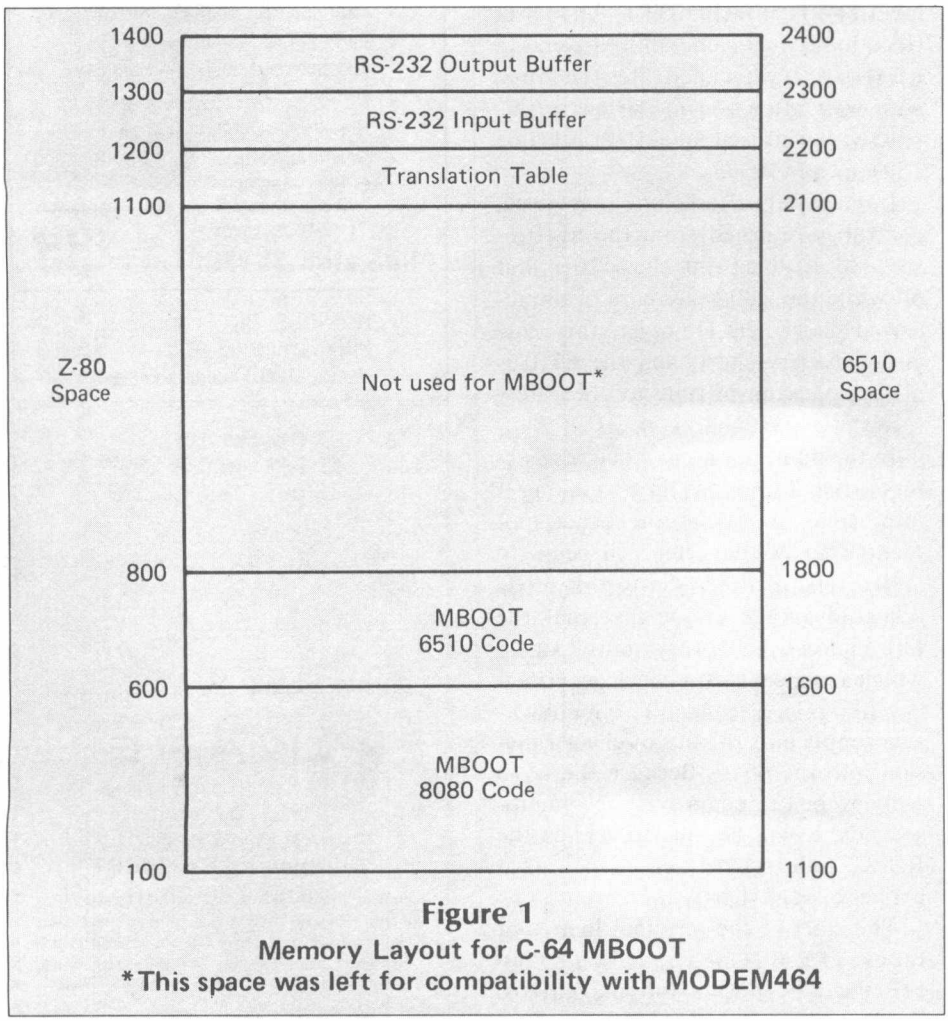


Figure 1
Memory Layout for C-64 MBOOT
***This space was left for compatibility with MODEM464**

when using MBOOT or other modem programs, even though they will work most of the time with a 48K CP/M.

There are two ways to add the 6510 code to MBOOT. The easiest, not that it works, is simply to include the code at the end of the 8080 assembly as a set of hex constants. (Don't forget that it must begin at location $600.) The second way, if you have a 6510 assembler that runs on the native 6510, is to assemble the program and load the object code into memory prior to doing a CP/M cold start; the cold start leaves these 6510 program locations unaffected. Once CP/M has been loaded, use DDT to load in MBOOT's 8080 COM file and save the combined program with "SAVE 7 C64MBT.COM."

MODEM4—Upload and Download

After I had MBOOT working, I began to run up my long distance phone bill calling RCPMs and looking for a copy of the source code for MODEM7. During the search, I found many RCPMs with copies of the object code but none with the source code. Since this search was getting rather expensive, I decided to stop and go to work on a copy of MODEM4, a program I had found and downloaded while looking for MODEM7.

MODEM4 apparently is a modem program that was part of the evolution toward MODEM7. You can use it as a terminal program as well as to upload and download files using Christensen's protocol. I modified it for the C-64 and used it to put copies of itself on several RCPMs in the Northeast. It's called MODEM464 and a short .DOC file accompanies it.

To find it, look for MODEM464.OBJ. Download it with MBOOT, rename it MODEM464.COM, and run it. If you run it by typing MODEM464 T, you will be in the terminal mode and the program will act the same as MBOOT. When you are ready to download a file (after you have set up the RCPM's XMODEM), exit from the program with CTL-E and type "MODEM464 R filename.ext." This will reload the program from disk and start it into file receive mode. Once the file has been transferred, the program goes directly back into terminal mode. If you want to send a file, type "MODEM464 S filename.ext." The program will send the file and go to terminal mode when completed, just as it does in the file receive mode.

MODEM7—Upload, Download, and More

A short time after I finished the work on MODEM4, I came across the source code for MDM730, a version of MODEM7. MODEM7 is a full-feature modem program with a terminal mode, a file upload and download capability, a terminal mode capture buffer for ASCII text, and a terminal mode ASCII file transmit function. The ASCII capture buffer is useful for saving text, such as the directory of an RCPM, that you don't want to lose but can't transfer in file (XMODEM) form. The ASCII file transmit function is useful if you have a long message that you want to leave on a bulletin board without using the time (and money) to compose it while on-line. These ASCII file features are also useful if you wish to exchange text files with someone who does not have a program with the XMODEM protocol. MDM730 will also autodial with several different kinds of modems, and it allows you to echo text directly to your printer. (The modified version will autodial only with a Hayes modem.)

The modified program, called MD730C64, is harder to learn to use than either MBOOT or MODEM4 because it has so many features. To get going, you must download three files: the executable program (MD730C64-.OBJ), the documentation for the original program (MDM730.DOC), and the notes that explain the few differences between MDM730 and MD730C64 (MD730C64.DOC).

Getting MD730C64.OBJ is easy: download it using MBOOT, rename it to a .COM file, and you are ready to run it. Getting the documentation, however, is a bit more difficult. Most RCPMs store long text files in a squeezed format to conserve disk space and to shorten the download time; you can recognize squeezed files because they have a Q in the file extension. Therefore, you may find MD730C64-.DQC instead of MD730.DOC. If you do, you must also download a program to unsqueeze the file on your own system. This unsqueeze program has many versions, but it almost always has a name that starts with USQ.

Getting the original MDM730 program documentation file may be tougher still. On most RCPMs you must download a library file called MDM730.LBR. A library file is a collection of individual files relating to the same program—MDM730.DQC is one of the individual files in MDM730-.LBR. To extract the file that you want from a library, you need a library utility program called LU (and its documentation LU.DQC). Once you have learned to use LU, you can extract MDM730.DQC from the MDM730 library file. This library also contains a program that lets you change the list of RCP/M phone numbers in MDM730 to ones that you call more frequently.

Don't be discouraged by the complexity of this whole process. Once you start to find your way around RCPMs, you will find that most of the really worthwhile programs are in libraries and have squeezed documentation. Your experiences and the utility programs that you have downloaded will pay off later.

I ran into some interesting problems when modifying MDM730. The biggest was the size of the source code. The squeezed file that I downloaded was 105K, which will fit on a C-64 diskette. When unsqueezed, it becomes 158K, which will not. To get an assembly listing, I had to use my modified MODEM4 to transfer the squeezed source code and a copy of an unsqueeze program to a Cromemco Z80 system at the university where I work. The disks on this system hold 250K, and I was able to unsqueeze the file and assemble it there. My original intent was to modify MDM730 by editing and reassembling the original source file, but I decided that editing and assembling on one machine and testing on another 30 miles away would be a nearly impossible task. Because of that, I chose to make the modification in the form of a patch overlay.

Caveats

MDM730 is a very complex program; consequently, so is MD730C64. I have tested it carefully, but you may uncover some minor problems. Also, because of equipment limitations, I have not been able to test any of these programs at data rates higher than 300 baud. Because of the switch that takes place between processors while the data is arriving, the program may need more work to run successfully at higher rates. Please let me know if you uncover any problems and if you are able (or unable) to run at rates higher than 300 baud.

A note about modems: When I began accessing RCPMs, I used the VIC modem that I had been using for CompuServe. I discovered that, while it works with CompuServe because they have a local phone number, on the typical long distance call with a noisy line the VIC modem simply doesn't work well. If you plan to call RCPMs, you will need a modem with better immunity to noise.

References

1. Commodore Business Machines. *Commodore 64 CP/M Operating System User's Guide*. Indianapolis: Howard W. Sams and Co., 1983.
2. Head, Gene. "CP/M Exchange." *Dr. Dobb's Journal*, July 1982, pages 42 – 50.
3. Hoff, Irvin M. "MDM730 (notes on how to use)." File MDM730.DOC, 1984.
4. Piotrowski, Walt. "CP/M on the Commodore 64." *Dr. Dobb's Journal*, June 1984.

RCPMs with MD730C64

1. Allentown (PA) RBBS/RCPM, Bill Earnest, (215) 398-3937.
2. Bearsville (NY) Town SJBBS, Hank Szyszka, (914) 679-6559.
3. St. Mary's College (MD) RCP/M, Jonathan Crawford and Bob Beasley, (301) 863-7165.

Thanks to the SYSOPS for permission to use their names and numbers. **DDJ**

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 194**.

MBOOT and MODEM7 (Text begins on page 62)

Listing One

```
                                        B:MBOOT64.PRN
                ;*****************************************************
                ;
                ;       MBOOT64 - FILE DOWNLOAD PROGRAM
                ;
                ;               ORIGINAL MBOOT BY GENE HEAD
                ;               D.D.J.  OCTOBER, 1982
                ;
                ;               MODIFIED FOR THE C-64 BY
                ;               WALT PIOTROWSKI
                ;
                ;
                ;*****************************************************
                ;
0000 =          BASE    EQU     0
                ;
0005 =          EXITCHR EQU     05H     ;CTL-E TO EXIT TERM MODE TO CP/M
0004 =          FILCHR  EQU     04H     ;CTL-D TO RECEIVE FILE
                ;
                ;
0600 =          MD65Z   EQU     600H    ;LOC OF 6510 MODEM RTNE (Z80 SPACE)
1600 =          MD65    EQU     MD65Z+1000H     ;(6510 SPACE)
0603 =          MDFUNC  EQU     MD65Z+3 ;MODEM FUNCTION CODE
0604 =          MDCHAR  EQU     MD65Z+4 ;MODEM CHARACTER
```

(Continued on next page)

MBOOT and MODEM7 (Listing Continued, text begins on page 62)

Listing One

```
0605 =          MDREC   EQU     MD65Z+5 ;MODEM CHAR RCVD FLAG
                ;
F900 =          BIOSFN  EQU     0F900H  ;BIOS65 FUNCTION CODE
F906 =          BIOSAD  EQU     0F906H  ;MODEM ROUTINE INDIRECT ADDRESS
CE00 =          ON6510  EQU     0CE00H  ;MEM LOC TO TURN 6510 ON
                ;
000A =          ERRLIM  EQU     10      ;NUMBER OF TRIES FOR ONE BLOCK
                ;
0001 =          SOH     EQU     1       ;ASCII CONTROLS
0004 =          EOT     EQU     4
0006 =          ACK     EQU     6
0015 =          NAK     EQU     15H
0018 =          CAN     EQU     18H
000A =          LF      EQU     10
000D =          CR      EQU     13
                ;
0005 =          BDOS    EQU     BASE+5
005C =          FCB     EQU     BASE+5CH
                ;
0100                    ORG     BASE+100H
                ;
0100 210000             LXI     H,0     ;CLEAR HL
0103 39                 DAD     SP      ;MAKE A COPY OF SP
0104 222B05             SHLD    STCK    ;SAVE FOR EXIT
0107 312B05             LXI     SP,STCK ;POINT TO LOCAL STACK
010A CDF903             CALL    INITADR ;SET UP BIOS CALLS
010D CD3604             CALL    ILPRT
0110 432D363420         DB      'C-64 MBOOT AS OF '
0121 342F312F38         DB      '4/1/84',CR,LF,0
012A 3A5D00             LDA     FCB+1   ;LOOK AT FILE NAME
012D FE20               CPI     ' '     ;BLANK?
012F C25501             JNZ     TERM1   ;NO - GO PROCESS
0132 CD3604             CALL    ILPRT   ;WRITE ERROR MSG

0135 2B2B4E4F20         DB      '++NO FILE NAME SPECIFIED++',CR,LF,0
0152 C37304             JMP     EXIT
                ;
0155 CD3604     TERM1   CALL    ILPRT   ;WRITE FIRST MSG
0158 0D0A544552         DB      CR,LF,'TERMINAL MODE',CR,LF
0169 43544C2D45         DB      'CTL-E EXITS TO CP/M'
017C 0D0A               DB      CR,LF
017E 43544C2D44         DB      'CTL-D STARTS FILE XFER'
0194 0D0A00             DB      CR,LF,0
0197 CD7E04             CALL    OPMDEM  ;OPEN FOR MODEM
                ;
                ;       DUMB TERMINAL LOOP
                ;
019A CDCC04     TERM    CALL    KYBD    ;CHECK FOR KYBD CHAR
019D FE00               CPI     0       ;CHAR RECEIVED?
019F CAAF01             JZ      TERML   ;GO CHECK FOR MODEM CHAR
01A2 FE05               CPI     EXITCHR ;EXIT TO CPM?
01A4 CA7304             JZ      EXIT
01A7 FE04               CPI     FILCHR  ;BEGIN RECEIVING FILE?
01A9 CABF01             JZ      RCVFIL
01AC CD9004             CALL    WRMDEM  ;OUTPUT KYBD CHAR TO MODEM
                ;
01AF CD9E04     TERML   CALL    RDMDEM  ;INPUT MODEM CHAR
01B2 FE00               CPI     0       ;0 - NO CHAR RECVD
01B4 CA9A01             JZ      TERM    ;BACK TO MAIN LOOP
01B7 E67F               ANI     7FH     ;REMOVE PARITY
01B9 CD1204             CALL    TYPE    ;DISPLAY RECVD CHAR
01BC C39A01             JMP     TERM    ;BACK TO MAIN LOOP
                ;
                ;       FILE RECEIVE
                ;
01BF CD1403     RCVFIL  CALL    ERASFIL ;SEE IF FILE EXISTS
01C2 CD5B03             CALL    MAKEFIL ;OPENFILE
```

(Continued on page 70)

Listing One

```
01C5 CD3604            CALL    ILPRT   ;PRINT MSG
01C8 46494C4520        DB      'FILE OPEN, READY TO RECEIVE',CR,LF,0
                ;
01E6 CD1802    RCVLP   CALL    RCVSECT ;GO RECEIVE A SECTOR
01E9 DAF801            JC      RCVEOT  ;CARRY SET-SECTOR IN OK
01EC CDBA03            CALL    WRSECT  ;WRITE TO DISK
01EF CD0C03            CALL    INCRSNO ;INCREMENT SECTOR #
01F2 CDCE02            CALL    SENDACK ;ALL OK - ACKNOWLEDGE
01F5 C3E601            JMP     RCVLP   ;GET NEXT SECTOR
                ;
01F8 CDBA03    RCVEOT  CALL    WRSECT  ;EOT RCVD-OUTPUT ALL TO DISK
01FB CDCE02            CALL    SENDACK ;SEND ACKNOWLEDGE
01FE CD9A03            CALL    CLOSFIL ;CLOSE DISK FILE
0201 CD4504            CALL    ERXIT   ;EXIT (NO ERROR)
0204 0D0A545241        DB      CR,LF,'TRANSFER COMPLETE$'
                ;
0218 AF        RCVSECT XRA     A       ;RECEIVE A SECTOR
0219 32EE04            STA     ERRCT   ;CLEAR ERROR CTR
                ;
021C 060A      RCVRPT  MVI     B,10    ;LONG DELAY
021E CDE403            CALL    RECV    ;RECEIVE ONE CHARACTER
0221 DA3102            JC      RCVSERR ;NO CHAR - ERROR
0224 FE01              CPI     SOH     ;START OF HEADER?
0226 CA7802            JZ      RCVSOH  ;YES-CONTINUE RECEIVING
0229 B7                ORA     A       ;CLEAR CARRY FOR NEXT TRY

022A CA1C02            JZ      RCVRPT  ;GO TRY AGAIN
022D FE04              CPI     EOT     ;END OF TRANSMISSION?
022F 37                STC             ;YES - CARRY=ALL DONE
0230 C8                RZ
                ;
0231 0601      RCVSERR MVI     B,1     ;REC ERROR-IGNORE REST OF TRANSMISSION
0233 CDE403            CALL    RECV    ;GET NEXT CHAR
0236 D23102            JNC     RCVSERR ;CHAR RECVD-GET ANOTHER
0239 3E15              MVI     A,NAK   ;NO CHAR-DATA ALL IN
023B CDD002            CALL    SEND    ;SEND NAK TO INDICATE REC ERROR
023E 3AEE04            LDA     ERRCT   ;GET ERROR CTR
0241 3C                INR     A       ;INCREMENT IT
0242 32EE04            STA     ERRCT   ;PUT IT BACK
0245 FE0A              CPI     ERRLIM  ;TOO MANY?
0247 C21C02            JNZ     RCVRPT  ;NO-TRY AGAIN
                ;
024A CD9A03    RCVSABT CALL    CLOSFIL
024D CD4504            CALL    ERXIT
0250 2B2B554E41        DB      '++UNABLE TO RECEIVE BLOCK'
0269 0D0A2B2B41        DB      CR,LF,'++ABORTING++$'
                ;
0278 0601      RCVSOH  MVI     B,1     ;SOH RECEIVED
027A CDE403            CALL    RECV    ;SECTOR NUM IS NEXT
027D DA3102            JC      RCVSERR ;NO CHAR - ERROR
0280 57                MOV     D,A     ;SECTOR NUM TO D
0281 0601              MVI     B,1     ;SHORT WAIT
0283 CDE403            CALL    RECV    ;NEG OF SECTOR NUM IS NEXT
0286 DA3102            JC      RCVSERR ;NO CHAR - ERROR
0289 2F                CMA             ;MAKE POSITIVE
028A BA                CMP     D       ;SAME?
028B CA9102            JZ      RCVDATA ;YES - GET DATA
028E C33102            JMP     RCVSERR ;NOT SAME - ERROR
                ;
0291 7A        RCVDATA MOV     A,D     ;SECTOR NUM TO A
0292 32EC04            STA     RCVSNO  ;SAVE FOR COMPARISON WITH EXPECTED
0295 0E00              MVI     C,0     ;CLEAR CHECKSUM
0297 218000            LXI     H,BASE+80H      ;BUFFER ADDRESS
                ;
029A 0601      RCVCHR  MVI     B,1     ;SHORT WAIT
029C CDE403            CALL    RECV    ;GET A CHARACTER
```

(Continued on page 72)

Listing One

```
029F DA3102            JC      RCVSERR ;NO CHAR - ERROR
02A2 CD1204            CALL    TYPE    ;*****SECURITY BLANKET*****
02A5 77                MOV     M,A     ;CHAR TO BUFFER
02A6 2C                INR     L       ;POINT TO NEXT BUFFER SLOT
02A7 C29A02            JNZ     RCVCHR  ;128 CHARS RECEIVED?
02AA 51                MOV     D,C     ;CHECKSUM TO D
02AB 0601              MVI     B,1     ;SHORT WAIT
02AD CDE403            CALL    RECV    ;GET CHECKSUM
02B0 DA3102            JC      RCVSERR ;NO CHAR - ERROR
02B3 BA                CMP     D       ;RECEIVED = COMPUTED?
02B4 C23102            JNZ     RCVSERR ;NOT SAME - ERROR
02B7 3AEC04            LDA     RCVSNO  ;GET RECEIVED SECTOR NUM
02BA 47                MOV     B,A     ;INTO B FOR COMPARE
02BB 3AED04            LDA     SECTNO  ;GET LAST SECTOR NUMBER
02BE B8                CMP     B       ;SAME?
02BF CAC802            JZ      RECVACK ;REPEAT - IGNORE
02C2 3C                INR     A       ;INCREMENT LAST
02C3 B8                CMP     B       ;SAME AS RECEIVED?

02C4 C2D402            JNZ     ABORT   ;NO - ERROR
02C7 C9                RET             ;RETURN TO MAIN LOOP
                ;
02C8 CDCE02    RECVACK CALL    SENDACK ;HANDSHAKE
02CB C31802            JMP     RCVSECT ;GET NEXT SECTOR
                ;
02CE 3E06      SENDACK MVI     A,ACK
                ;
02D0 CD9004    SEND    CALL    WRMDEM  ;SEND CHAR TO MODEM
02D3 C9                RET
                ;
02D4 312B05    ABORT   LXI     SP,STCK ;RESTORE CP/M STACK POINTER
                ;
02D7 0601      ABORTL  MVI     B,1     ;SHORT WAIT
02D9 CDE403            CALL    RECV    ;GET CHARACTER
02DC D2D702            JNC     ABORTL  ;LOOP UNTIL INPUT STOPS
02DF 3E18              MVI     A,CAN   ;CANCEL CHARACTER
02E1 CDD002            CALL    SEND    ;SEND TO MODEM
                ;
02E4 0601      ABORTW  MVI     B,1     ;SHORT WAIT
02E6 CDE403            CALL    RECV    ;GET CHARACTER
02E9 D2E402            JNC     ABORTW  ;LOOP UNTIL INPUT STOPS
02EC 3E20              MVI     A,' '   ;GET A BLANK
02EE CDD002            CALL    SEND    ;SEND TO MODEM
02F1 CD4504            CALL    ERXIT   ;LEAVE (ERROR)
02F4 4D424F4F54        DB      'MBOOT PROGRAM CANCELLED$'
                ;
030C 3AED04    INCRSNO LDA     SECTNO  ;GET CALCULATED SECTOR NUM
030F 3C                INR     A       ;INCREMENT IT
0310 32ED04            STA     SECTNO  ;PUT IT BACK
0313 C9                RET
                ;
0314 115C00    ERASFIL LXI     D,FCB   ;FCB ADDRESS
0317 0E11              MVI     C,17    ;SEARCH FOR FIRST CODE
0319 CD0500            CALL    BDOS
031C 3C                INR     A       ;255 MEANS NO MATCH
031D C8                RZ              ;NOT FOUND IS OK
031E CD3604            CALL    ILPRT   ;PRINT MSG
0321 2B2B46494C        DB      '++FILE EXISTS, TYPE Y TO ERASE:',0
0341 CD2A04            CALL    KEYIN   ;GET KEYBOARD CHAR
0344 F5                PUSH    PSW     ;SAVE A COPY
0345 CD1204            CALL    TYPE    ;ECHO IT
0348 CD0B04            CALL    CRLF    ;CAR RET LINE FEED
034B F1                POP     PSW     ;GET CHAR BACK
034C E65F              ANI     5FH     ;REMOVE PARITY AND CASE
034E FE59              CPI     'Y'     ;Y = ERASE IT
```

(Continued on page 74)

Listing One

```
0350 C24B04             JNZ     MXIT    ;NOT Y - EXIT PROGRAM
0353 115C00             LXI     D,FCB   ;FCB ADDRESS
0356 0E13               MVI     C,19    ;DELETE FILE CODE
0358 CD0500             CALL    BDOS
                ;
035B 115C00     MAKEFIL LXI     D,FCB   ;FCB ADDRESS
035E 0E16               MVI     C,22    ;MAKE FILE CODE
0360 CD0500             CALL    BDOS
0363 3C                 INR     A       ;A=255 IF ERROR
0364 C0                 RNZ             ;RETURN - OK
0365 CD4504             CALL    ERXIT   ;LEAVE (ERROR)
0368 2B2B455252         DB      '++ERROR - CAN''T MAKE FILE',CR,LF


0383 4449524543         DB      'DIRECTORY MUST BE FULL$'
                ;
039A 115C00     CLOSFIL LXI     D,FCB   ;FCB ADDRESS
039D 0E10               MVI     C,16    ;FILE CLOSE CODE
039F CD0500             CALL    BDOS
03A2 3C                 INR     A       ;A=255 IF ERROR
03A3 C0                 RNZ     ;NOT ZERO IS OK
03A4 CD4504             CALL    ERXIT   ;LEAVE (ERROR)
03A7 2B2B43414E         DB      '++CAN''T CLOSE FILE$'
                ;
03BA =          WRSECT  EQU     $       ;WRITE SECTOR
03BA 115C00             LXI     D,FCB   ;FCB ADDRESS
03BD 0E15               MVI     C,21    ;WRITE SECTOR CODE
03BF CD0500             CALL    BDOS
03C2 B7                 ORA     A       ;SET FLAGS
03C3 C2C703             JNZ     WRERR
03C6 C9                 RET
                ;
03C7 CD3604     WRERR   CALL    ILPRT
03CA 2B2B455252         DB      '++ERROR WRITING FILE',CR,LF,0
03E1 C3D402             JMP     ABORT
                ;
                ;       READ ONE CHARACTER FROM MODEM
                ;
03E4 D5         RECV    PUSH    D
03E5 CDB504     MWTI    CALL    RDMDEM2 ;READ MODEM CHAR
03E8 D2F203             JNC     MCHAR   ;NO CARRY=CHAR RCVD
03EB 05                 DCR     B       ;DECREMENT WAIT CTR
03EC C2E503             JNZ     MWTI    ;WAIT SOME MORE (IN 6510)
03EF D1                 POP     D
03F0 37                 STC             ;INDICATE ERROR
03F1 C9                 RET
                ;
03F2 D1         MCHAR   POP     D
03F3 F5                 PUSH    PSW     ;SAVE A COPY OF CHAR
03F4 81                 ADD     C       ;ADD TO CHECKSUM
03F5 4F                 MOV     C,A     ;PUT CKSUM BACK IN A
03F6 F1                 POP     PSW     ;GET CHAR BACK
03F7 B7                 ORA     A       ;CLEAR CARRY
03F8 C9                 RET
                ;
03F9 2A0100     INITADR LHLD    BASE+1  ;GET XFER BASE ADDRESS
03FC 110300             LXI     D,3     ;GET OFFSET
03FF 19                 DAD     D       ;ADD OFFSET FOR STAT
0400 222304             SHLD    VSTAT+1 ;PUT IN CALL
0403 19                 DAD     D       ;ADD OFFSET TO KEYIN
0404 222E04             SHLD    VKEYIN+1
0407 19                 DAD     D       ;ADD OFFSET TO TYPE
0408 221804             SHLD    VTYPE+1
                ;
```


(Continued on page 76)

MBOOT and MODEM7 (Listing Continued, text begins on page 62)

Listing One

```
                ;
040B 3E0D       CRLF    MVI     A,CR    ;CARRAIGE RET
040D CD1204             CALL    TYPE
0410 3E0A               MVI     A,LF    ;LINE FEED
                ;
0412 F5         TYPE    PUSH    PSW     ;TYPE A CHARACTER
0413 C5                 PUSH    B
0414 D5                 PUSH    D

0415 E5                 PUSH    H
0416 4F                 MOV     C,A
0417 CD0000     VTYPE   CALL    $-$     ;CALL BDOS
041A E1                 POP     H
041B D1                 POP     D
041C C1                 POP     B
041D F1                 POP     PSW
041E C9                 RET
                ;
041F C5         STAT    PUSH    B       ;CONSOLE STATUS
0420 D5                 PUSH    D
0421 E5                 PUSH    H
0422 CD0000     VSTAT   CALL    $-$     ;CALL BDOS
0425 E1                 POP     H
0426 D1                 POP     D
0427 C1                 POP     B
0428 B7                 ORA     A
0429 C9                 RET
                ;
042A C5         KEYIN   PUSH    B       ;READ A KEY
042B D5                 PUSH    D
042C E5                 PUSH    H
042D CD0000     VKEYIN  CALL    $-$     ;CALL BDOS
0430 E1                 POP     H
0431 D1                 POP     D
0432 C1                 POP     B
0433 E67F               ANI     7FH
0435 C9                 RET
                ;
0436 E3         ILPRT   XTHL            ;TYPE A LINE
                ;
0437 7E         ILPLP   MOV     A,M     ;GET CHARACTER
0438 B7                 ORA     A       ;SET FLAGS
0439 CA4304             JZ      ILPRET  ;ZERO MEANS DONE
043C CD1204             CALL    TYPE    ;TYPE CHARACTER
043F 23                 INX     H       ;NEXT MEM LOC
0440 C33704             JMP     ILPLP   ;NEXT CHARACTER
                ;
0443 E3         ILPRET  XTHL            ;MODIFIED RETURN TO STACK
0444 C9                 RET
                ;
0445 D1         ERXIT   POP     D       ;EXIT (NOT ALWAYS ERROR)
0446 0E09               MVI     C,9     ;PRINT EXIT MESSAGE
0448 CD0500             CALL    BDOS
                ;
044B CD3604     MXIT    CALL    ILPRT
044E 0D0A444F4E         DB      CR,LF,'DON''T FORGET TO DISCONNECT MODEM'
0470 0D0A00             DB      CR,LF,0
                ;
0473 CD8704     EXIT    CALL    CLMDEM  ;CLOSE RS-232 FILE
0476 CD0B04             CALL    CRLF    ;PRINT A FINAL CRLF
0479 2A2B05             LHLD    STCK    ;CP/M STACK POINTER
047C F9                 SPHL            ;BACK INTO SP
047D C9                 RET             ;BACK TO CP/M
                ;
                ;       6510 SETUP ROUTINES
                ;
047E 3E05       OPMDEM  MVI     A,5     ;FILE OPEN CMD
```

(Continued on page 78)

MBOOT and MODEM7 (Listing Continued, text begins on page 62)

Listing One

```
0480 320306            STA     MDFUNC  ;STORE FOR 6510
0483 CDD804            CALL    GO6510  ;TURN ON 6510
0486 C9                RET
                ;
0487 3E04       CLMDEM MVI     A,4     ;CLOSE FILE CODE
0489 320306            STA     MDFUNC  ;STORE FOR 6510
048C CDD804            CALL    GO6510  ;TURN ON 6510
048F C9                RET
                ;
0490 F5         WRMDEM PUSH    PSW     ;SAVE A COPY OF CHAR
0491 320406            STA     MDCHAR  ;SAVE FOR 6510
0494 3E03              MVI     A,3     ;WRITE MODEM CODE
0496 320306            STA     MDFUNC  ;STORE FOR 6510
0499 CDD804            CALL    GO6510  ;START 6510
049C F1                POP     PSW     ;GET CHAR BACK
049D C9                RET
                ;
049E 3E01       RDMDEM MVI     A,1     ;READ FUNCTION
04A0 320306            STA     MDFUNC  ;STORE FOR 6510
04A3 CDD804            CALL    GO6510  ;START 6510
04A6 3A0506            LDA     MDREC   ;SEE IF CHAR RCVD
04A9 FE00              CPI     0       ;0 - NONE RECEIVED
04AB CAB304            JZ      RDMNC   ;GO SET CARRY
04AE 3A0406            LDA     MDCHAR  ;GET CHARACTER
04B1 B7                ORA     A       ;SHOW CHAR RCVD
04B2 C9                RET
04B3 37         RDMNC  STC             ;SET - NO CHAR
04B4 C9                RET
                ;
04B5 3E02       RDMDEM2 MVI    A,2     ;READ WITH WAIT FUNCTION
04B7 320306            STA     MDFUNC  ;STORE FOR 6510
04BA CDD804            CALL    GO6510  ;START 6510
04BD 3A0506            LDA     MDREC   ;SEE IF CHAR RCVD
04C0 FE00              CPI     0       ;0 - NONE RECEIVED
04C2 CACA04            JZ      RDMNC2  ;GO SET CARRY
04C5 3A0406            LDA     MDCHAR  ;GET MODEM CHAR
04C8 B7                ORA     A       ;SHOW CHAR RCVD
04C9 C9                RET
04CA 37         RDMNC2 STC             ;CARRY - NO CHAR
04CB C9                RET
                ;
04CC 3E06       KYBD   MVI     A,6     ;READ KEYBOARD FUNCTION
04CE 320306            STA     MDFUNC  ;STORE FOR 6510
04D1 CDD804            CALL    GO6510  ;TURN ON 6510
04D4 3A0406            LDA     MDCHAR  ;GET CHARACTER
04D7 C9                RET
                ;
04D8 E5         GO6510 PUSH    H       ;SAVE HL
04D9 210016            LXI     H,MD65  ;GET 6510 RTNE ADDR
04DC 2206F9            SHLD    BIOSAD  ;STORE FOR BIOS65
04DF 3E09              MVI     A,9     ;BIOS CODE TO CALL SUBPROG
04E1 3200F9            STA     BIOSFN  ;STORE FOR 6510
04E4 3E01              MVI     A,1     ;1 TURNS ON 6510
04E6 3200CE            STA     ON6510  ;TURN IT ON
04E9 00                NOP             ;REQD FOR HARDWARE
04EA E1                POP     H       ;RESTORE HL
04EB C9                RET
                ;
04EC 00         RCVSNO DB      0
04ED 00         SECTNO DB      0
04EE 00         ERRCT  DB      0
04EF                   DS      60
052B            STCK   DS      2
                ;
052D                   END
```

End Listing One

Listing Two

```
CPMMD65.TX
LINE# LOC   CODE        LINE

00001  0000             ;*****************************************
00002  0000             ;
00003  0000             ;       MODEM OR RS-232 HANDLER
00004  0000             ;       FOR USE WITH CP/M MBOOT
00005  0000             ;
00006  0000             ;       W.G. PIOTROWSKI
00007  0000             ;
00008  0000             ;*****************************************
00009  0000             ;
00010  0000             ;       EQUATES
00011  0000             ;
00012  0000             FILE    =128            ;FILE NUMBER
00013  0000             ;
00014  0000             SCNKEY  =$FF9F          ;KERNAL ROUTINES
00015  0000             SETLFS  =$FFBA          ;   "       "
00016  0000             SETNAM  =$FFBD          ;   "       "
00017  0000             OPEN    =$FFC0          ;   "       "
00018  0000             CLOSE   =$FFC3          ;   "       "
00019  0000             CHKIN   =$FFC6          ;   "       "
00020  0000             CHKOUT  =$FFC9          ;   "       "
00021  0000             CLRCHN  =$FFCC          ;   "       "
00022  0000             CHROUT  =$FFD2          ;   "       "
00023  0000             GETIN   =$FFE4          ;   "       "
00024  0000             ;
00025  0000             RIBUF   =$F7            ;RS-232 INPUT BUF ADD
00026  0000             ROBUF   =$F9            ;RS-232 OUTPUT BUF ADD
00027  0000             RIDBS   =$29B           ;RS-232 BUF END PTR
00028  0000             RIDBE   =$29C           ;RS-232 BUF STRT PTR
00029  0000             ENABL   =$2A1           ;RS-232 ACTIVE
00030  0000             ;
00031  0000             XLATE   =$2100          ;C64 TO ASCII TABLE
00032  0000             BUFIN   =$2200          ;RS-232 INPUT BUFFER
00033  0000             BUFOUT  =$2300          ;RS-232 OUTPUT BUFFER
00034  0000             ;
00035  0000                     *=$1600
00036  1600             ;
00037  1600  4C 06 16   MODEM   JMP START
00038  1603  00         FUNC    .BYTE 0         ;FUNCTION CODE
00039  1604  00         CHAR    .BYTE 0         ;CHARACTER IN/OUT
00040  1605  00         CHREC   .BYTE 0         ;RS-232 CHAR RCVD FLAG
00041  1606             ;
00042  1606  AD 03 16   START   LDA FUNC         ;GET FUNCTION
00043  1609  18                 CLC             ;CLEAR FOR ADD
00044  160A  6D 03 16           ADC FUNC        ;*2 FOR TABLE ACCESS
00045  160D  AA                 TAX
00046  160E  BD 6B 17           LDA ADRTBL-2,X  ;GET LOW ADDRESS
00047  1611  8D 1B 16           STA JMPSUB+1    ;PUT IN JMP
00048  1614  BD 6C 17           LDA ADRTBL-1,X  ;GET HI ADDRESS
00049  1617  8D 1C 16           STA JMPSUB+2    ;PUT IN JMP
00050  161A  4C 00 00   JMPSUB  JMP 0           ;JMP TO SUBPROGRAM
00052  161D             ;
00053  161D             ;   CLOSE RS-232 FILE AND CLEAN UP
00054  161D             ;
00055  161D  A9 80       CLOSIT  LDA #FILE       ;FILE NUMBER
00056  161F  20 C3 FF           JSR CLOSE       ;KERNAL
00057  1622  20 57 16           JSR CLRKBF      ;CLEAR KEYBOARD BUFFER
00058  1625  60                 RTS
00059  1626             ;
00060  1626             ;   OPEN RS-232 FILE AND DO SETUP
00061  1626             ;
00062  1626  20 1D 16   OPENIT  JSR CLOSIT       ;CLOSE-IN CASE
00063  1629  A9 80              LDA #FILE       ;FILE NUMBER
00064  162B  A2 02              LDX #2          ;RS-232 DEVICE
00065  162D  A0 FF              LDY #$FF        ;NO COMMAND
00066  162F  20 BA FF           JSR SETLFS      ;CALL KERNAL
```

(Continued on next page)

```
00067  1632  A9 02          LDA #2          ;TWO CHAR NAME
00068  1634  A2 7A          LDX #<SET232    ;LO ADDRESS
00069  1636  A0 17          LDY #>SET232    ;HI ADDRESS
00070  1638  20 BD FF       JSR SETNAM      ;KERNAL
00071  163B  20 C0 FF       JSR OPEN        ;KERNAL
00072  163E  A2 00          LDX #<BUFIN     ;RS232 INBUF LO ADDR
00073  1640  A0 22          LDY #>BUFIN     ;  HI ADDR
00074  1642  86 F7          STX RIBUF       ;MOVE IT
00075  1644  84 F8          STY RIBUF+1
00076  1646  A2 00          LDX #<BUFOUT    ;RS232 OUTBUF LO ADDR
00077  1648  A0 23          LDY #>BUFOUT    ;  HI ADDR
00078  164A  86 F9          STX ROBUF       ;MOVE IT
00079  164C  84 FA          STY ROBUF+1
00080  164E  20 0E 17       JSR BLDXTB      ;BUILD XLATE TABLE
00081  1651  A9 00          LDA #0
00082  1653  8D 7E 17       STA LSTCHR      ;SHOW LAST CHAR AS A NULL
00083  1656  60             RTS
00084  1657                ;
00085  1657                ;   EMPTY KEYBOARD BUFFER
00086  1657                ;
00087  1657  20 9F FF    CLRKBF JSR SCNKEY     ;KEY PRESSED?
00088  165A  20 E4 FF       JSR GETIN       ;GET A CHARACTER
00089  165D  C9 00          CMP #0          ;IS IT A NULL
00090  165F  D0 F6          BNE CLRKBF      ;NOT A NULL - GET NEXT
00091  1661  60             RTS
00092  1662                ;
00093  1662                ;   CHARACTER OUTPUT TO RS-232
00094  1662                ;
00095  1662  A2 80       OUTPUT LDX #FILE      ;GET FILE NUM
00096  1664  20 C9 FF       JSR CHKOUT      ;OPEN FOR OUTPUT
00097  1667  90 06          BCC OTOK        ;CARRY SET IS ERROR
00098  1669  20 26 16       JSR OPENIT      ;MUST BE CLOSED
00099  166C  4C 62 16       JMP OUTPUT      ;TRY AGAIN
00100  166F  AD 04 16    OTOK   LDA CHAR       ;GET CHARACTER AGAIN
00101  1672  20 D2 FF       JSR CHROUT      ;KERNAL
00102  1675  AD A1 02    WAITOT LDA ENABL      ;GET STATUS
00103  1678  29 01          AND #1          ;STILL RUNNING BIT
00104  167A  D0 F9          BNE WAITOT      ;HANG UNTIL DONE
00105  167C  20 CC FF       JSR CLRCHN      ;CLEAR CHANNEL
00106  167F  60             RTS
00107  1680                ;
00108  1680                ;   CHARACTER INPUT FROM RS-232
00109  1680                ;
00110  1680  A2 80       INPUT  LDX #FILE      ;FILE NUMBER
00111  1682  20 C6 FF       JSR CHKIN       ;OPEN FOR INPUT
00112  1685  90 06          BCC INOK        ;CARRY SET IS ERROR
00113  1687  20 26 16       JSR OPENIT      ;MUST BE CLOSED
00114  168A  4C 80 16       JMP INPUT       ; TRY AGAIN
00115  168D  A9 00       INOK   LDA #0         ;ZERO FOR CLEAR
00116  168F  8D 05 16       STA CHREC       ;SHOW NO CHAR RCVD
00117  1692  AD A1 02       LDA ENABL       ;STATUS BYTE
00118  1695  29 02          AND #2          ;RECEIVING BIT
00119  1697  D0 10          BNE WAIT        ;RUNNING - WAIT
00120  1699  AD 9B 02       LDA RIDBS       ;GET BUF START PTR
00121  169C  CD 9C 02       CMP RIDBE       ;COMPARE TO END
00122  169F  F0 1D          BEQ INEX        ;NOTHING IN BUFFER
00123  16A1  A9 01          LDA #1          ;ONE FOR SET
00124  16A3  8D 05 16       STA CHREC       ;SHOW CHAR RECEIVED
00125  16A6  4C B8 16       JMP INGET       ;GET THE CHAR
00126  16A9  AD A1 02    WAIT   LDA ENABL      ;STATUS BYTE
00127  16AC  29 02          AND #2          ;RECEIVING BIT
00128  16AE  F0 08          BEQ INGET       ;NOT RUNNING - EXIT
00129  16B0  A9 01          LDA #1          ;ONE FOR SET
00130  16B2  8D 05 16       STA CHREC       ;SHOW CHAR RECEIVED
00131  16B5  4C A9 16       JMP WAIT        ;WAIT UNTIL IN
00132  16B8  20 E4 FF    INGET  JSR GETIN      ;GET CHARACTER
00133  16BB  8D 04 16       STA CHAR        ;STORE FOR CP/M
```

(Continued on page 82)

```
00134  16BE  20 CC FF     INEX   JSR CLRCHN
00135  16C1  60                   RTS
00136  16C2               ;
00137  16C2               ;    CHARACTER INPUT FROM RS-232 WITH WAIT LOOP
00138  16C2               ;
00139  16C2  A9 14        INPUT2 LDA #20          ;OUTER LOOP LIMIT
00140  16C4  8D 7D 17           STA OTCTR        ;PUT IN LOOP CONTROL LOC
00141  16C7  A9 00              LDA #0           ;INNER LOOP COUNTS AROUND
00142  16C9  8D 7C 17           STA INCTR        ;PUT IN LOOP CONTROL LOC
00143  16CC  20 80 16     IN2LP  JSR INPUT        ;SEE IF THERE'S A CHARACTER
00144  16CF  AD 05 16           LDA CHREC        ;GET RECEIVED CHAR
00145  16D2  D0 0A              BNE IN2EX        ;0 - NONE RECEIVED
00146  16D4  CE 7C 17           DEC INCTR        ;COUNT DOWN INNER LOOP
00147  16D7  D0 F3              BNE IN2LP        ;WAIT
00148  16D9  CE 7D 17           DEC OTCTR        ;COUNT DOWN OUTER LOOP
00149  16DC  D0 EE              BNE IN2LP        ;WAIT SOME MORE
00150  16DE  60           IN2EX  RTS
00151  16DF               ;
00152  16DF               ;    INPUT KEYBOARD CHARACTER AND DEBOUNCE
00153  16DF               ;
00154  16DF  20 9F FF     KBDCHR JSR SCNKEY       ;SCAN THE KEYBOARD
00155  16E2  20 E4 FF           JSR GETIN        ;GET THE CHARACTER
00156  16E5  CD 7E 17           CMP LSTCHR       ;SAME AS LAST CHAR?
00157  16E8  D0 05              BNE KBNCHR       ;NO - NEW CHARACTER
00158  16EA  A9 00              LDA #0           ;SAME - NEED A NULL
00159  16EC  4C F2 16           JMP KBDBNC       ;GO TO DEBOUNCE LOOP
00160  16EF  8D 7E 17     KBNCHR STA LSTCHR       ;NEW CHAR - SAVE IT
00161  16F2  A0 07        KBDBNC LDY #7           ;DEBOUNCE OUTER LOOP
00162  16F4  A2 00              LDX #0           ;INNER LOOP INIT
00163  16F6  C9 00              CMP #0           ;REPEATING CHAR LO LIM
00164  16F8  F0 06              BEQ KBDDLY       ;NULL IS OK
00165  16FA  C9 21              CMP #33          ;SPACE IS HI LIM
00166  16FC  B0 02              BCS KBDDLY       ;33 OR MORE IS OK
00167  16FE  A0 32              LDY #50          ;LONG DELAY FOR RPTG CHAR
00168  1700  CA           KBDDLY DEX              ;DELAY LOOP
00169  1701  D0 FD              BNE KBDDLY       ;INNER LOOP
00170  1703  88                 DEY
00171  1704  D0 FA              BNE KBDDLY       ;OUTER LOOP
00172  1706  AA                 TAX              ;CHARACTER TO A
00173  1707  BD 00 21           LDA XLATE,X      ;GET ASCII VALUE
00174  170A  8D 04 16           STA CHAR         ;STORE FOR Z-80
00175  170D  60                 RTS
00176  170E               ;
00177  170E               ;    BUILD ASCII TRANSLATION TABLE
00178  170E               ;
00179  170E  A2 00        BLDXTB LDX #0           ;LOOP TO CLEAR TABLE
00180  1710  A9 00              LDA #0
00181  1712  9D 00 21     BLDX1  STA XLATE,X
00182  1715  E8                 INX
00183  1716  D0 FA              BNE BLDX1        ;LOOP
00184  1718               ;
00185  1718  A2 00              LDX #0           ;START INDEX
00186  171A  A0 41              LDY #65          ;STOP INDEX+1
00187  171C  A9 00              LDA #0           ;NULL
00188  171E  20 5D 17           JSR MVINTB
00189  1721  A2 41              LDX #65          ;LOWER CASE
00190  1723  A0 5B              LDY #91
00191  1725  A9 61              LDA #97
00192  1727  20 5D 17           JSR MVINTB
00193  172A  A2 5B              LDX #91          ;MISC CHARACTERS
00194  172C  A0 60              LDY #96
00195  172E  A9 5B              LDA #91
00196  1730  20 5D 17           JSR MVINTB
00197  1733  A2 C1              LDX #193         ;UPPER CASE
00198  1735  A0 DB              LDY #219
00199  1737  A9 41              LDA #65
```

(Continued on page 84)

Listing Two

```
00200  1739  20 5D 17          JSR MVINTB
00201  173C  A9 00             LDA #0
00202  173E  8D 1C 21          STA XLATE+28
00203  1741  8D 1D 21          STA XLATE+29
00204  1744  8D 1E 21          STA XLATE+30
00205  1747  8D 1F 21          STA XLATE+31
00206  174A  A9 08             LDA #8          ;BACKSPACE
00207  174C  8D 14 21          STA XLATE+20    ;INST DEL
00208  174F  8D 94 21          STA XLATE+148   ;INST DEL UC
00209  1752  A9 0D             LDA #13         ;CR
00210  1754  8D 8D 21          STA XLATE+141   ;RETURN UC
00211  1757  A9 20             LDA #32         ;SPACE
00212  1759  8D A0 21          STA XLATE+160   ;SPACE UC
00213  175C  60                RTS
00214  175D                 ;
00215  175D  8C 79 17     MVINTB STY MXSTOP      ;SAVE STOP VAL IN MEMORY
00216  1760  9D 00 21     MVINT1 STA XLATE,X     ;PUT ASCII EQUIV IN TABLE
00217  1763  18                CLC              ;CLEAR FOR ADD
00218  1764  69 01             ADC #1           ;NEXT CHAR CODE
00219  1766  E8                INX              ;NEXT TABLE SLOT
00220  1767  EC 79 17          CPX MXSTOP       ;COMPARE TO STOP VALUE
00221  176A  D0 F4             BNE MVINT1       ;LOOP
00222  176C  60                RTS
00223  176D                 ;
00224  176D                 ;    DATA
00225  176D                 ;
00226  176D  80 16        ADRTBL .WORD INPUT     ;SUBPROG XFER TABLE
00227  176F  C2 16               .WORD INPUT2    ;  IN ORDER OF REQ CODE
00228  1771  62 16               .WORD OUTPUT
00229  1773  1D 16               .WORD CLOSIT
00230  1775  26 16               .WORD OPENIT
00231  1777  DF 16               .WORD KBDCHR
00232  1779  00           MXSTOP .BYTE 0
00233  177A  06           SET232 .BYTE 6,0       ;RS232 PARAMS
00233  177B  00
00234  177C  00           INCTR  .BYTE 0         ;LOOP CTR - INPUT2
00235  177D  00           OTCTR  .BYTE 0         ;LOOP CTR - INPUT2
00236  177E  00           LSTCHR .BYTE 0         ;LAST KEYBOARD CHAR
00237  177F                      .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 ADRTBL   176D    BLDX1    1712    BLDXTB   170E    BUFIN    2200
 BUFOUT   2300    CHAR     1604    CHKIN    FFC6    CHKOUT   FFC9
 CHREC    1605    CHROUT   FFD2    CLOSE    FFC3    CLOSIT   161D
 CLRCHN   FFCC    CLRKBF   1657    ENABL    02A1    FILE     0080
 FUNC     1603    GETIN    FFE4    IN2EX    16DE    IN2LP    16CC
 INCTR    177C    INEX     16BE    INGET    16B8    INOK     168D
 INPUT    1680    INPUT2   16C2    JMPSUB   161A    KBDBNC   16F2
 KBDCHR   16DF    KBDDLY   1700    KBNCHR   16EF    LSTCHR   177E
 MODEM    1600    MVINT1   1760    MVINTB   175D    MXSTOP   1779
 OPEN     FFC0    OPENIT   1626    OTCTR    177D    OTOK     166F
 OUTPUT   1662    RIBUF    00F7    RIDBE    029C    RIDBS    029B
 ROBUF    00F9    SCNKEY   FF9F    SET232   177A    SETLFS   FFBA
 SETNAM   FFBD    START    1606    WAIT     16A9    WAITOT   1675
 XLATE    2100

END OF ASSEMBLY
```

End Listing

Unstructured Forth Programming

An Introduction

by Richard Wilton

One of the features of Forth is that it is a "structured" language. This means that Forth programs are written in variously sized chunks that fit together like a Chinese box puzzle. This is in contrast to "unstructured" Fortran or BASIC programs, which are written in variously sized chunks that fit together like the noodles on a plate of spaghetti. This is what makes "structured" programming languages such as Forth so much superior to "unstructured" languages such as Fortran.

Nowadays, Fortran has acquired roughly the same status among recent computer science graduates as Middle English or Aramaic. Nevertheless, there are still some of us around who not only remember what Fortran is, but have actually written programs in it. In fact, there are still a few people who think that "Forth" is just an abbreviation for IBM's Fortran 4 (H) compiler.

Let's face it: despite the universal scorn heaped upon it by "modern" structured programming disciples, good old Fortran has its strong points. In fact, in certain situations, the obligatory structured nature of Forth makes elegant programming impossible. What Forth programmer doesn't long for a simple, unstructured, unconditional GOTO every once in a while? Who doesn't wish that the nice, straightforward arithmetic IF, which can be expressed so concisely in Fortran or in two or three machine instructions, were somehow built into Forth? For that matter, wouldn't it be nice every once in a while to be able to fall back on that old Fortran standby, the computed GOTO?

Of course it would! It is the purpose of this little article to fill in the gaps in the pristine structure of Forth, to repair the glaring omissions we've just mentioned, and to restore a little unstructured sanity to the deeply nested, block-structured world of the dedicated Forth hacker. Note: all examples are written in Forth-83. FIG-Forth and Forth-79 users will have to adjust the "tick" and ROLL references accordingly.

"What Forth programmer doesn't long for a simple, unstructured, unconditional GOTO every once in awhile?"

The GOTO Statement

```
: GOTO ( cfa --- ) R> DROP
    EXECUTE ;
```

This little gem of a definition expects the code field address of a Forth definition on the stack. You'll note, however, that control passes unconditionally to that definition. This eliminates the need for leaving status flags, return codes, and other assorted garbage on the data stack. Using GOTO also makes it easy to extract yourself from those messy BEGIN-WHILE-REPEAT's and IF-ELSE-THEN's you've nested ten deep.

The Arithmetic IF Statement

```
: AIF ( cfa_A cfa_B cfa_C n --- )
    ? DUP 0=
    IF      ROT
    ELSE  0>
          IF    ROT ROT
          THEN
```

Richard Wilton, Laboratory Microsystems, P.O. Box 10430, Marina Del Rey, CA 90295.

```
    THEN 2DROP GOTO ;
```

The arithmetic IF statement is obvious in concept: transfer control to point A if n is negative, to point B if it's zero, and to point C if it's positive. This is the sort of thing you do all the time in assembly language:

```
    OR   AX,AX
    JS   POINT_A
    JZ   POINT_B
POINT_ C:
```

But just try to express that succinctly in Forth:

```
DUP 0< IF ['] POINT_A
       ELSE 0 = IF ['] POINT_B
            ELSE ['] POINT_C
            THEN
       THEN
EXECUTE
```

Pretty clumsy, right? Now here's the elegant solution, which is concise and straightforward because it ignores the dogma of block-structuring:

```
: EXAMPLE ( n --- )
          ['] POINT_A
          ['] POINT_B
          ['] POINT_C
          3 ROLL AIF ;
```

The Computed GOTO Statement

As a final example, we implement Fortran's familiar

```
GOTO (x1,x2,x3, . . . ,xn),i
```

a control statement so elegant that it was copied almost verbatim by the creators of BASIC. Making it work in Forth requires setting up a table of code field addresses in advance. Then you simply index the table and GOTO the right address.

```
: CGOTO ( cfa_table n --- ) 2* +
    @ GOTO ;
```

The utility of using CGOTO, rather than a viper's nest of IF-ELSE-THEN's or CASE statements, is obvious to any competent programmer and is not worth belaboring here.

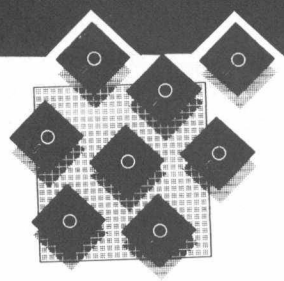
The more astute reader might observe at this point, "But what about statement numbers?" Well, folks, if you think about it for a moment, you'll realize that statement numbers make good sense in Forth. For one thing, numbered statememts would make it easy to do "source-directed" editing, thus eliminating the need for those clumsy, space-wasting screen files. Also, silly philosophical debates about the "proper" names for Forth definitions would be unnecessary if Forth definitions were numbered instead of named.

By now it should be obvious how much can be gained by writing unstructured Forth programs. No doubt, further improvements could be obtained by incorporating elements of PL/1 or even COBOL into Forth.

Of course, such improvements must be left as an exercise for the reader. We're too busy adapting the syntax of Forth to fit onto 80-column punched cards.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**



APL*PLUS/PC, Version 3.1; TOOLS, Volume 1; FINANCIAL & STATISTICAL LIBRARY, Version 1.0

Company: STSC, Inc., 2115 East Jefferson St., Rockville, MD 20852

Computer: 192K IBM PC (or compatible) with one or more disk drives; TOOLS requires 256K and MSDOS Version 2.0 or later

Price: $595 for APL*PLUS/PC; $295 for TOOLS; $275 for LIBRARY

Circle Reader Service No. 129

Reviewed by Stefan H. Unger

I spent several months wondering which advanced language I would get for my PC. At the time there was a lot of press about C, but in the back of my mind I really wanted to get APL. The best APL was reputedly from STSC, Inc., but I felt it was a little too expensive for my home use.

I had programmed in APL since about 1972, mostly statistical routines that I needed in my work in computer-aided drug design. In fact, I had even published some APL functions, but I was certainly not what you would call a hard-core convert to APL. This language simply offered me—as a chemist and not as a programmer—the greatest freedom to be creative without worrying about the stultifying detail required by most other languages. At work I originally had used a STSC APL time-sharing service, switched to another APL time-sharing service for several years, then ported everything to our inhouse IBM 3081 VS APL 4.0 system. However, recently I have been busy with computer graphics and quantum/molecular mechanics and not statistics (i.e., APL) at all.

Having been exposed to the ease and elegance of APL, I found that learning BASIC for my PC was more self-flagellation than enlightenment. It was slow, awkward, and, well, silly (Isn't it silly to have to dimension variables? After all, computers should do that kind of stuff for you. . . .).

Turbo Pascal—with its easier user interface—was not yet getting much press, and the language didn't seem to offer any particular advantages over Fortran or BASIC in general approach. So I bought a C compiler, knowing C might come in handy at work. At least it would run faster. Unfortunately, it was a disaster. I couldn't accept the amount of time it took to compile and link after each minor debugging step, and, worst of all, I couldn't really make much sense of the I/O function calls . . . it all seemed so needlessly complex.

At this low point in my experience with C, I was given the opportunity of reviewing STSC's APL*PLUS/PC system for *DDJ*. The irony of this is that *DDJ* is such a staunch advocate of C (which means that I have now enraged most of my audience). Of course, I'm not proposing that APL and C are equivalent or even on the same level. For example, some implementations of APL have been done in C (e.g., Dyalog APL), but not vice versa. (A recent article by D. Saunders, "Unix, C and APL," *Unix/World*, 1[6] ,59, 1984, in fact argues that the two languages are naturally complementary.) But for the average PC applications developer, is APL worth a serious look? Read on!

First of all, I think learning well-fined APL symbols is less ridiculous than trying to match braces in C or remembering the dozens of function calls with their nonstandard syntax and location. The only real problem with APL is the character set and some of the difficulties that it causes in graphics/printer interfaces. Of course, C "gets closer to the machine," but STSC APL*PLUS/PC allows interfaces to outside assembly code, DOS, DOS files, interrupts, peeking/poking, and so on. It is also much easier to learn, debug, and use, being what Saunders calls a "very high-level language." From descriptions of the user interface to Turbo Pascal (full-screen editing, pointers between source and object code, fast compilations), I would say that APL is at least as good as—if not considerably better than—Turbo Pascal in this regard.

The System

The APL*PLUS/PC system is supplied on two diskettes, and a special APL character ROM replaces the one on your graphics card (the IBM version of APL uses software-generated characters and requires both a color monitor and 8087). Special ROMs are available for IBM, AT&T, Compaq, Columbia, Eagle, Hyperion and Televideo microcomuters. Corona is supported, but without graphics, Seequa is supported without communications and Wang is supported without graphics and sound. The IBM AT, other MSDOS computers and the DEC Rainbow are scheduled for support. (Although all of these machines are "supported", there might be some tradeoffs in performance. Not all graphics boards are supported, either; check carefully.)

Only the least useful characters of the original character set have been replaced by the special APL characters, so the system has virtually no impact on ordinary word processing, spreadsheets, and so on. The system requires 192K and PC or MSDOS with one or more disk drives. Documentation in is four IBM-style loose-leaf binders with sections on installation, user's guide, introductory tutorial, formatting, files, programmer's manual, system functions, and a comprehensive index. Color-coded APL keyboard labels are supplied—the tackiest part of the

package. The Gilman and Rose textbook classic *APL: An Interactive Approach* and a quarterly newsletter are also provided. You also get on the STSC mailing list, which will definitely keep your mailbox full.

Other sources of information are available for additional cost: two volumes of *Collected Whizbangs* by Roy Sykes, Jr. (a fantastic collection of advanced tricks) and a volume entitled *APL in Practice: What You Need to Know to Install and Use Successful APL Systems and Major Applications*, edited by Rose and Schick (to assuage any remaining doubt that APL is not taken seriously by the business world). The Rose and Schick book is directed toward mainframe APL uses, but the overview is good and a number of chapters contain some generally useful advice, particularly in the third part of the book: "The Core of APL."

STSC has also announced the first collection of application development TOOLS ($295, requires 256K, PC or MSDOS at 2.0 or later and APL*PLUS/PC at 3.0 or later), which provides communications, screen management, report and output formatting, disk and file management, software development tools, and two games.

Additionally, STSC has developed a Financial and Statistical Library ($275) for the PC, containing many primitive functions that can be incorporated into application programs; this means no window dressing on the output, although "dressed" versions are also available for some of the functions. The package is overpriced for what you get. For example, my favorite correlation coefficient program is faster than the one supplied in STATISTI; not all of the terms are defined in the manual (e.g., Durbin-Watson Statistic); there doesn't appear to be any ANOVA, a common feature of most statistical packages (at least not in the index nor any of the obvious places); and we get the program FEDTAX79, my favorite . . . it *is* 1984, isn't it? The list goes on.

A new addition to STSC's product line is Pocket APL, which is a simplified and more affordable version of APL (only $95). It requires 128K RAM but uses a soft character set (instead of the special ROM) for color systems and the keyword APL system for monochrome. Communications, graphics, full files and workspaces, and the full-screen editor are not supported, nor is there an interface to DOS. However, it *does* supply on-line calculator mode, full-screen cursor control, on-line help, a file system, ⎕*FMT* formatting, error trapping features, and over 50 system functions. STSC apparently believes that Pocket APL will be the Turbo of the APL world.

STSC has also solved *the* major problem for applications developers who want to use APL, namely the requirement of a full APL interpreter. Because the cost of the STSC system is at least $595, this used to be a major drawback. Now, a Run Time System is available for a license fee of from $50 to $100, depending upon quantity and possibly your negotiating skills. The Run Time System allows applications packages that buffer the user from APL; the user has no APL directly available. Neither calculator mode nor definition-editing-display of functions is available. In fact, the APL character ROM is not required, and the keyboard stays in text mode. The full-screen editor can be used only to edit variables. Applications developers are offered a software publishing program with royalties of "5 to 15 percent."

Generally, all of the documentation is very well done with good quality paper, typesetting, editing, and so on. Of course, nothing is perfect. The Gilman and Rose textbook unfortunately is weakest on the most difficult aspects of APL (a consistent theme, as you shall see), and there are a number of minor editing problems. The front cover proclaims "Includes APL2 and APL for PC's," but this is not really explained—in a forward, for example. In fact, I didn't know what APL2 was and eagerly awaited the revelation inside. Alas, the best I could do was to figure out that it was an IBM product. The index didn't help either. There were only a few references to "APL for PC's" inside the text, despite the 1983 publication date, and a reference to "see inside the back cover for details" in the preface . . . very funny! It's blank.

Two serious deficiencies in the package are the lack of quick reference cards and the need for more detailed tutorial material on graphics, memory, and machine language features—the most difficult aspects of the extended language. This makes it tough on the neophyte or convert. A workspace demo on graphics was so confusing that I couldn't make *it do anything* useful. Documentation for the interrupt handler ⎕*INT* gives only a few examples, a stern warning about "improperly constructed arguments," and a reference to the section on ⎕*CALL*. Documentation for the latter system function is more extensive but still difficult to follow.

To whet the microsystem user's appetite, I should mention that STSC has announced two interesting developments: the maxi/mini APL*PLUS system will be available under Unix, which means speed, nested arrays, and a good multiuser environment, and an APL "compiler" for IBM mainframe VM/CMS and MVS/TSO will be available, promising a three-fold enhancement in speed. (There are quotes around compiler because you still must run the compiled functions under the standard APL environment.) STSC is the most active, broadest-based APL vendor in the world, which I hope speaks well for its commitment toward APL for the PC.

Finally, even as I write this review of Version 3.1, Version 4.0 is being announced. Included is documentation on all workspaces, new stickers, and a placard with the APL character set. Speedups for)*LOAD*,)*COPY*, ⎕*DEF*, and ⎕*SS* (substring search), exponentiation, membership, and inner product are implemented (some up to 10 times faster). A soft character set for use on an IBM color monitor with IBM color graphics card eliminates the special APL ROM—at the considerable sacrifice of off-screen scrolling and full-screen editor. Several new APL keyboard facilities, new enhancements to the full-screen editor, graphics, and some new language features (e.g., enhanced validity checking of certain primitives) are implemented. The workspaces have all been "reviewed and revised." Communications using the smart terminal mode have been improved to allow, for example, the host to execute a line in the PC. An upgrade for existing users will be about $90.

Features

STSC APL*PLUS/PC is a nearly com-

plete version of standard APL, lacking only some of the newer extensions such as nested arrays and some of the extensions of operators found in the IBM mainframe APL2 or STSC's Unix-based system for maxis/minis. I will review a large number of the existing enhancements.

A keyword mode is available through a function key toggle; in this mode, APL symbols are replaced by English words. An on-line HELP facility can be customized for use in applications. User-definable function keys are available through a system function.

Workspace size expands to available memory. On my 256K system a *)CLEAR* workspace is 112,336 bytes, but memory can be partitioned for a non-APL functionality under DOS; for example, TOOLS supplies a RAM disk program (although I find the AST Superdrive much easier to use). Maximum object size is 65,536 bytes, including overhead. In a clear workspace, this translates to about 8,190 floating-point and 32,761 integer numbers. Floating-point numbers are eight bytes (17 significant digits in range +− 1.797 . . . . E308); both integer and Boolean are two bytes (range +− 32767). No bit packers need apply!

An optional 8087 math coprocessor is automatically detected by the system and can be toggled on/off with a □*POKE*. It significantly speeds up exponential, transcendental, and other functions. See the section on benchmarks.

High-resolution black-and-white or color graphics and sound are supported with system functions. Use of the graphics primitives is nontrivial, dangerous in certain circumstances (misuse can blitz your graphics board!), and lacks a good tutorial description. The only documentation is in the system function outline and a brief section in the user's guide. Using the session parameter E=64 to reduce "snow" will cause a significant slowdown in the speed of color graphics; without this setting, color graphics literally flash on the screen, but with the setting, it takes about 3 seconds to paint a full screen. The sound primitive □*SOUND* is extremely easy to use, and a fascinating music-generating program in one of the workspaces produces pleasing little ditties. Simple windows and screen scrolling control are available; the TOOLS I package offers further enhancements such as menu processing, maintenance, screen input facility, and forms maintenance.

Communications facilities for uploading and downloading from mainframe or other computers comprise several system functions, plus a number of programs in one workspace as illustrative examples. Additional 3278 and IRMA facilities are in TOOLS I. One of STSC's ads shows an 11-line program that sorts a DOS file, plots the results as a histogram, calculates mean and variance, uses a full-screen editor to create a memo combining histogram, statistics, and descriptive test, issues a DOS command to the PC, dials a host computer, and ships off the memo . . . not bad for 11 lines. A built-in terminal emulator can be toggled on/off with a function key so that executing programs on the host or PC can continue executing. The simulated terminal mode is the extended Datamedia 1520, but some customization is possible.

Both full-screen and del editors are available. The full-screen editor is nicely done, but on my computer and graphics card (both Columbia), I must operate with the session parameter E=64 to reduce the small band of dashes that flashes randomly when any key is hit. This is supposed to disable the full-screen editor, but in fact it still works. The only degradation in performance is that the editor does not exit in a way to restore the original screen: line numbers and a reverse video EDITIN are left and they do not scroll off screen. The full-screen editor works fine if I am willing to tolerate snow for each key press; alternately, I could define a function key to clear the screen following exit from the full-screen editor. These are relatively minor inconveniences, but you may feel that for $595 you should get all features of the "supported" hardware.

TOOLS I contains a large collection of special functions for maintaining other functions and workspaces, transferring a workspace to a file and back, and documenting workspaces. Most of these are of interest only to the professional applications developer.

APL*PLUS/PC allows interfaces to non-APL programs such as assembler or DOS and allows □*PEEK*- and □*POKE*ing. The only documentation is in the reference manual and is difficult to follow; reportedly STSC has improved this in Version 4.0.

Both a logical set of APL file system commands (compared to the awkward IBM shared variables) and an interface to "native" DOS files are provided. Up to 20 files can be tied at one time. There is complete forward compatibility with APL files, but Versions 2.0 or 2.6 will not work on Version 3.1 files.

A set of powerful formatting, error trapping, and string searching primitives is supplied. The □*FMT*, adapted from the mainframe APL*PLUS, is particularly well suited to business report formatting. Some additional functions in a workspace help you set up tables, and even more functions are available in TOOLS I, including functions that can direct output to screen, APL file, DOS file, or printer and provide pagination, page breaks, and so on.

APL*PLUS/PC has 135 system functions and over 200 utility programs. About 26 of these are also system commands of the *)xxx* type long familiar to APL users, but most of the system functions are usable within functions. In Version 3.1, *)SAVE*, *)LOAD*, and *)COPY* can be painfully slow, with lots of thrashing about by the disk drives; *)LOAD* and *)COPY* have been speeded up in Version 4.0. An alternative is to use a RAM disk—for safety, instead of using *)OFF*, you can write a small *SIGNOFF* function that will copy files/workspaces to permanent storage and exit via □*SA OFF*.

Enhancements

Many of the enhancements to APL, especially adaptation to the PC environment, come in the form of special system-level functions, variables, and constants. Because they are system-level features, they are always available in every workspace. These are largely, but not entirely, unique to each vendor, and any APL program that incorporates any of the nonstandard system features will not be compatible with other products. However, because STSC provides the Run Time System as a total environment for applications, this is of little concern. It would be a problem only for an individual trying to port applications between a mainframe (even between

some STSC products) and APL*PLUS/PC or between friends with different systems.

The 135 system features in the APL 2*PLUS/PC system provide information about the session, active workspace, and objects in the workspace; retrieve objects or entire workspaces from disk; assist in debugging; communicate with other devices; manipulate screen (cursor, window, graphics, and so on) and sound effects; and observe and manipulate the operating environment.

System variables contain information about the environment, such as comparison tolerance, index origin, printing precision, and random link, about latent expressions, HELP filenames, what action to take if execution stops for immediate input, and memory segment for peek/poke. Other system variables are for cursor position, keyword setting, printing width, and screen window control.

System constants comprise a dozen constants that control the character set (atomic vector) and generate bell, backspace, delete, escape, form/line feed, new line feed, null character, and so on.

Table I (below) gives an indication of the depth of support.

Benchmarks

An excellent comparison of STSC APL*PLUS/PC (Version 2.6) and IBM APL (Version 1.00) appeared in the March 1984 issue of *Byte*. The only weakness of these comparisons was that the exact timing algorithm was not stated. I have used the same benchmarks but have replaced J. Bensimon's large chess problem with the *DDJ* Savage floating point benchmark and have used the documented *TIMER* function supplied by STSC in the workspace *CIN0* (Listings One – Four, page 98). This *TIMER* corrects for all overhead.

All benchmarks were run first on my Columbia MPC 1600-1 without an 8087. To include comparisons with the earlier version of STSC APL*PLUS/PC used in the *Byte* article (Version 2.6) and to obtain 8087 data, I enlisted the unsuspecting help of the STSC small systems hotline staff who happily performed tests No. 1 – 19 on both Versions 2.6 and 3.1, both with and without the 8087 (a *POKE* turns the 8087 off, and the machine acts as if it were not there in all respects).

The comparable STSC benchmarks were all uniformly faster than mine; this precipitated a number of phone calls to both Columbia and STSC and between the two vendors. We could not discover any obvious reason for this discrepancy, outside of a difference in the way the benchmarks were run: the STSC benchmarks were run using a master program to feed the arguments to *TIMER*, while my benchmarks were originally performed manually. The STSC benchmarks for No. 5 were performed on the wrong variable, so I have omitted No. 5, but the STSC benchmark No. 2 was still about half the value that I obtained on the Columbia.

I had originally decided to report only ratios for the STSC data, but I later decided to check the performance on an IBM myself. With the cooperation of two colleagues at work, I was able to boot my exact workspace on an IBM XT with 640K RAM and to perform all benchmarks in the same manner as on my Columbia at home. The XT without 8087 gave much closer agreement to the Columbia values, including a value of about 9 msec for No. 2.

The disagreement with STSC's values still bugged me, so I coded the master program and reran all benchmarks on my Columbia. The discrepancies were still there: I then copied all variables and functions into a *)CLEAR* workspace and reran all benchmarks. Lo and behold, the value of 4.5 flashed up for No. 2, in close agreement with

Files: 21 for APL and 13 for DOS
Character constants: 9
Debugging tools: 8
Detached I/O: 4
Disk: 4
DOS: 1
Exception handling: 5
Execution: 6
Function Definition: 11
General Information: 12
Graphics/Screen: 21
Help: 3
Input/Output: 10
Keyboard: 6
Latent Expression: 3
Machine Language: 3 (call/interrupt/symbol table pointer)
Memory: 3 (peek/poke/seg)
Object: 9
Session: 4
State Indicator: 6
Workspace/Object Manipulation: 12
Workspace: 9
Other: 1 substring searching, 1 delay execution

Table I
List of System Features by Type

NEWS newsletter are a financial system and a DBMS for the broadcasting industry.

R.W. Butterworth makes some interesting points—albeit directed mainly toward mainframe systems—in his article "When APL is Inappropriate" (in *APL in Practice*, page 43). The main criterion is who or what reads the program most often or importantly during its full life cycle. Low-level languages (e.g., assembly code or even C) favor machine readability, while very high-level languages favor people readability. Therefore, general support utilities, high-volume processing, or very complex calculations ("number crunching" scientific applications) are not good for APL because they are too machine intensive. On-line real-time applications involving complex tasks are also not good: portability is somewhat limited if the code involves lots of enhancements, but this is true of most high-level languages. Hybrid solutions, however, offer many benefits; APL*PLUS/PC offers interfaces to machine code and communications.

When is development on micros using APL warranted? Any project that involves a lot of *ad hoc* analysis is a good candidate for APL. However, with the Run Time System, all possible *ad hoc* variants must be predetermined, and either menu choices or predefined function names are required to determine which one will be used. This is probably not a limitation. If the environment cannot be predefined, you have only two choices. Either your APL application using the Run Time System must have a command simulation processor to allow the user to enter equations, possibly with the APL keyword system, or you should suggest the purchase of the full APL system. You can then supply a more "standard" set of functions and perhaps some training in APL.

Probably the most useful application of the APL Run Time System would be situations where the package needs to be customized quickly or frequently in a manner that cannot be achieved by menu choices of options, input of formulae, and so on, or if your business consists of lots of small custom jobs. If these concepts are correct, then I find the pricing of the Run Time System to be out of whack because it seems to pre-

STSC's values; all other values improved significantly in agreement.

An inexplicable bug apparently had crept into the original workspace and affected timing, not accuracy, in a manner not due to pending functions, limited □*WA*, or any other obvious problem; I have shipped a copy of the "good" and "bad" workspaces off to STSC for examination. Benchmarks in APL are not as reproducible as they may be in other languages because a lot of bookkeeping goes on "behind closed doors." The state of the workspace depends to some extent on what has preceded it. Therefore, I now store a clean workspace for benchmarking and)*LOAD* it fresh for each run.

First, let's look at the performance of Version 2.6 (the *Byte* review version) compared with the current Version 3.1 without an 8087. Curiously, some of the simple functions have gotten slightly worse (about 10% worse for No. 1 plus reduction, No. 3 maximum reduction, No. 6 indexing, No. 8 take, and No. 10 transposition); however, much more substantial improvements were realized in No. 14 matrix division (66%), No. 15 Fibonacci series (33%), and No. 17 division (14%). Comparing the two versions with an 8087 shows improvements in the newer version of 69% for No. 1 plus reduction and 87% for No. 14 matrix division, with smaller improvements in Nos. 2, 13, 15, 16, 17, and 19. There are 10 – 15% decrements in Nos. 3, 6, and 10.

Use of the 8087 math coprocessor gives improvements of 100 – 3600% in some benchmarks (Version 3.1; Nos. 1, 4, 13, 14, 16, 17, 18 and 19), particularly the logarithm-exponential and transcendental benchmarks.

A comparison of the Columbia MPC 1600-1 benchmarks with those for the IBM XT, both without 8087, shows the Columbia (256K with □*WA* of 95K) running fairly consistently at about 96% of the speed of the XT (640K with □*WA* of 469K). There are insignificant differences between the VP and MPC (Cols. E+H) and between the XT and PC (Cols. C+F and D+G), showing that □*WA* is not an important variable.

The well-known Sieve of Eratosthenes (No. 20) involves a lot of branching and showed little enhancement using the 8087, as might be expected. The Savage floating-point benchmark (No. 21) can be coded easily and elegantly in APL as shown in Listing Three. The first line prints a time-stamp for benchmarking, initializes two variables, and precomputes the branchpoints for the iteration. The second line evaluates the expression and tests for iteration number. The last line prints a second time-stamp. Without an 8087 coprocessor (which significantly enhances exactly the primitives used in this calculation), the speed was about equal to double-precision BASIC, 855 seconds on the Columbia and 833 seconds on the XT, but there was a low error of 1E-9. With the 8087, the time was only 198 seconds, which is not bad for an interpreted language. This compares favorably with several compiled languages (e.g., Supersoft Fortran on the PC), and is slower but more accurate than the only other APL listed, IBM APL (with 8087 implied), but I have not seen the coding used.

Users usually base their choices between upgrading to newer versions or between different vendors more on features than on small differences in "speed" benchmarks. However, STSC has clearly succeeded (intentionally or not) in reducing any glaring weaknesses that the *Byte* comparison might have showcased, particularly the laggard performance of matrix division on the older Version 2.6.

Finally, some minor quibbles: I have used the *Byte* benchmarks for consistency, although some of these (Nos. 14 and 19) involve two primitives when a new variable might be more appropriate. The PRIMES benchmark was from the *Byte* article; the variables should have been localized in the header.

Development Package?

What kinds of applications would you develop in APL? Surprisingly, the main revenue stream of the commercial APL systems is business and not scientific applications. For example, APL did not come to our IBM 3081 mainframe because the scientists or statisticians wanted it; it came because a tie-wearing manager type wanted *ADRS*, which just happens to be written in APL. Two examples of applications from the recent STSC PLUS*

clude low-volume projects. (Perhaps a better pricing scheme for STSC would be a higher initial purchase price with possibly a small royalty fee.)

Programming in APL*PLUS/PC

To illustrate some of the fun and problems involved in using APL*PLUS/PC, I decided to relate my experiences in converting a BASIC program (Listing Five, page 99) that I had written into the APL*PLUS/PC system (Listing Six, page 101 and Figure page 100). My Columbia came with Perfect Filer, one of the weakest members of the otherwise quite good Perfect series. Perfect Filer does not have math capabilities, so to do a billing system for my wife's small private practice, I used the Filer simply to store the raw data. The bills then were written to a native DOS file (a:newaku.mss). The BASIC program read through this file line by line (or record by record) and looked for certain keywords/symbols. It then totaled current month, added balance forward, and subtracted any payments. The total was written in the appropriate spot, and if the balance forward was greater than the payments received, a message was written that payment in full is required each month; the revised bill for the first client was written out (a:new0ak.mss) then on to the next client.

Although processing the file line by line is probably the most natural way for a BASIC programmer to attack this problem (and was the approach I originally took before I had APL*PLUS/PC), an APL programmer knows that you really want to process all records simultaneously. The only problem is to differentiate between the clients. The APL program took about six hours compared to about twenty hours for the original BASIC program. One of the nicest features of APL*PLUS/PC is the full-screen editor and scrolling buffer. These allow you to try many algorithms easily; for example, you can use calculator mode to run-modify-run-modify and so on by editing on the screen. Then, at some point, you can write a function and further modify it using the full-screen editor)*EDIT*.

A native DOS file is simply a long string of bytes in APL (8501 in this example), so it is easy to search the string for all occurrences of the keywords/symbols (variable ALL gives the positions). Originally I did this with a series of □*SS* substring searches. The actual numbers were then a fixed number of bytes away from these locations (line [6] of *PROCESS*). A function (*CLEAN*) was used to clean up any stray characters, to convert the normal minus to an APL negative, and to insert 000 in blank "numeric" fields. The characters then were converted to numbers. The resulting vector CM was a list of all session charges in order.

How to sum by client? There are usually several ways to do almost anything in APL. Much of what follows could have been avoided by the use of extended operators in APL2 or even a loop with the take primitive. However, the method that I found using APL*PLUS/PC is more typical. Line [7] of *PROCESS* reshaped the vector into a matrix that had number columns = number of clients (4 here) and number rows = total number of sessions (12; matrix A). Each column contained the original vector of session fees. What was then needed was to discover a mask of 0,1's that could be used to multiply this matrix so that the columns (clients) could be summed to give the current month's total charges for each client.

Because of the layout of the bill, the location of the balance forwards always occurred right after the current month's charges. Therefore, a matrix formed by finding the location of all keywords/symbols less than the balance forwards (matrix B) would identify each client—the first client would have all columns filled, the second n-1 columns, and so on to the last client, which would have only the last column filled. The last step would be to keep only the first 1 for each client; this was accomplished using the "less than scan" (matrix C). The final expression gives the sums *CSUMS*.

The program actually ran rather slowly. This was obviously due to the multiple string searches using □*SS* (reportedly speeded up in Version 4.0). It would be nicer and faster to search the main string only once, at least for the new sessions if not for everything. This proved possible by using the membership function (see *FINDALL*) to find all occurrences of the symbols i, g, A, and W (for individual, group, administrative, and workshop). This would include any words that happened to have these letters present, but the actual location would have a blank on either side. Therefore, all you had to do was examine the locations to the left and right, test for blanks, and do a logical *AND* (line [5] of *PROCESS*). The resulting algorithm was twice as fast as using the multiple □*SS*'s.

The APL version contains about the same number of lines—excluding comments and function headers—but runs almost exactly three times faster! Further optimization is undoubtedly possible.

Having the APL*PLUS/PC system in the first place would have made it unnecessary to improve Perfect Filer because the proper data base could have been set up easily. However, the example illustrates many of the features of programming in APL*PLUS/PC: it is fast, fun, well-adapted to the PC environment, and encourages elegant global solutions. Of course, what may be elegant for me may be pedestrian for another APL user.

APL was designed as a heuristic tool before it became a computer language. I find describing the conceptualization in English much more difficult than the symbolic description in APL shown in Listing Six. Unless you have some experience with APL, you may find the opposite to be true. Those with some APL under their belt tend to program in a twilight zone between the English verbalization and the vector/matrix concepts of APL. In many instances, you *know* that there is a trick; you just have to remember it, probably by fiddling on screen. In this case, I knew there was a scan function that would do it—line [7]—I just couldn't remember which one.

Menu a la TOOLS

Having received the TOOLS package just as I was completing the major portion of this review, I have not had a lot of time to fully evaluate all of its features. I have spent most of my time with the menu and screen design sections.

The benchmarks table (Table II, page 95) was produced using functions in the distribution workspace *FORMAT* and *OUTPUT* workspace from TOOLS. Due to scanty documentation, most of the minor difficulties involved function *COLNAMES* from *FORMAT*, which required a bit of fussing to work

		A	B	C	D	E	F	G	H
Computer		PC	PC	PC	PC	VP	XT	XT	MPC
APL Version		2.6	2.6	3.1	3.1	3.2	3.1	3.1	3.1
8087?		8087		8087			8087		
⎕WA		410K	410K	302K	302K	87K	480K	480K	95K
RAM		512	512	512	512	512	640	640	256
1. PLUS REDUCTION	Z←+/VI	104	158	32	169	174	32	169	175
2. LOGICAL REDUCTION	Z←∨/VL	5	6	4	4	5	5	4	5
3. MAXIMUM REDUCTION	Z←⌈/[1]MI	30	30	35	34	35	35	35	36
4. EXPONENTIATION	Z←VI*.1	1697	9571	1704	9249	9426	1704	9250	9473
5. ABSOLUTE VALUE	Z←|VR					130	65	120	130
6. INDEXING	Z←VR[VI[⍳20]]	22	23	26	26	26	25	25	26
7. SORTING	Z←VI[⍒VI]	118	118	119	119	125	116	116	122
8. TAKE	Z←¯2 1↑MR	29	29	29	30	31	29	30	31
9. MEMBERSHIP	Z←VI∈VI	149	149	154	153	154	154	153	156
10. TRANSPOSITION	Z←2 1⍉MC	64	64	70	70	71	71	71	72
11. OUTER PRODUCT, CHARACTER	Z←VC∘.=VC	131	130	137	137	143	136	136	143
12. OUTER PRODUCT, INTEGER	Z←(⍳50)∘.+⍳50	445	445	440	441	465	449	444	467
13. INNER PRODUCT, REAL	Z←VR∘.+VR	346	553	280	575	591	280	575	594
14. MATRIX DIVISION	Z←MR⌹10↑VR	1486	2217	196	755	778	196	755	782
15. FIBONACCI SERIES	Z←FBNCC (LISTING 1)	3832	3945	2487	2687	2801	2435	2638	2781
16. MULTIPLICATION	Z←VR×3.14	146	478	112	490	507	113	489	509
17. DIVISION	Z←VR÷3.14	152	733	118	632	652	118	632	657
18. LOGARITHM	Z←⍟VR	131	5085	131	4872	4970	131	4872	4994
19. SINE	Z←1○VR×.1	484	6060	453	6103	6224	444	6094	6243
20. SIEVE OF ERATOSTHENES	(LISTING 2) SEC.x10						1634	1666	1755
21. SAVAGE FLOATING PT.	(LISTING 3) SEC.						198	833	855

---

1) Times are in milliseconds per execution for Benchmarks #1-19 and seconds for #20-21. #20 is reported for 10 executions, as per convention.

2) TIMER (APL*PLUS/PC Workspace CINO, see Listing 4) was with 100 repetitions and with 'R←0' for NULLEXP, except for #15 where 'R←NONFN' was used. NONFN is nil-adic and simply assigns R←0. Benchmarks #20 and #21, since they are full functions and are relatively slow, were timed with ⎕AI[2], as shown in the Listings 2-3.

3) The PC and VP had monochrome monitors, the XT had both monochrome and color monitors and boards and the MPC had color monitor. All tests were done in monochrome mode. The PC,VP and XT used DOS 2.1, the MPC 2.0.

4) The variables, as in the Byte March 1984 review, are:

```
MI←10 10⍴VI←(500⍴0 1 0 0 1)/⍳500
VL←1 0 1 1 0 0 0 1
MR←10 10⍴VR←VI←0.1
MC←26 26⍴VC←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Table II

properly. Use of the functions in *OUTPUT* was relatively straightforward.

To test menu and screen functions from TOOLS, I decided to adapt some recombinant DNA programs to the PC using these features. A menu is divided into title, text, choice, and footnote sections and allows multiple choices with validity checking; it has one write-in area, time-out features, and full color/monochrome decorations. An enormous number of possible options are available. A screen is similar but more general; for example, multiple write-ins can be made. Some run-of-the-mill screens can be set up very simply: *'MENU MENU' CHOICE FROM '/AU CHAMBERTIN/PEARL RIVER SEAFOOD/CHUCK E. CHEESE'*.

In setting up the main DNA menu, I decided it would be useful to have the name of the registered owner appear somewhere. It was not at all obvious how to enter current information such as date, time, or user into the menu. The screen is first designed and then "compiled" for fast display (not a true compilation, rather an interpretation into more primative APL functions). Several of the key functions are locked and the terse error messages are not very helpful. The documentation is weak on how to create certain effects; some of these can be gleaned only from the single example in the workspace (for example, how to test for color boards and change attributes appropriately or how to center text). After a few calls to the hotline, I found that the only way to enter current information is to use some special editing functions to replace defined fields in the "compiled" menu.

Once the menu is "compiled," it flashes instantly on the screen—unless you are using E=64 at sign-on, in which case it takes several seconds to paint the screen. Despite the shortcomings of the documentation, once you get the hang of it, you can design some very nice menus.

I also had several problems with the screen functions and documentation. In the step-by-step discussion of how to design a sample screen, a critical "press enter" was omitted following the naming of the first field (*EXT*). No other exiting gambit works. The documentation is also confusing concerning the use of the function *SCFKEYSOFF* with *SCCLOSE*; you must not use these together. Instead, if you want to set function keys, you must follow a more complicated procedure that is outlined in the documentation, but the incompatibility is not made clear.

SCINIT calls the undocumented *SCCURSORSET*, which is *⎕IO* sensitive (=0 works), a function similar in purpose to *WHITEWASH* in the *MENU* workspace for color/mono conversions. If you want to re-use a screen (as I did), you must blank out any filled-in fields that you do not want to carry forward before you store it using *SCOUTPUT ' '*. After finally working out all of these details, I had a screen that looked nice, only lit up the write-in field when a function key was pressed and waited for proper input; unfortunately, even when compiled the screen took about three seconds to appear, much slower than a menu. The *SCREEN* workspace was the least satisfactory of the TOOL features that I was able to test.

You Get What You Pay For

STSC APL*PLUS/PC is not a cheap package, but it's worth every penny you spend on it. The claim is that the package is a development tool . . . and it is. It's possible to develop serious packages or to use the system in conjunction with LANs or mainframes to do serious work (for example, work done in one of our statistics groups). Certain low-volume applications may not have a low price tag (because of the $50 – $100 Run Time System license fee), but this may change.

I find the most curious aspect of APL to be its marketing by STSC, IBM, and others. STSC seems to be following IBM's lead . . . or non-lead. Although STSC runs occasional full-page ads in the computer rags, there is a minimum of second sourcing. For example, I have seen STSC APL*PLUS/PC offered from only two alternate vendors. One is a well-known programmers shop (at about a 16% discount), and the other is a vendor specializing in 8087 chips and related software. Of course, IBM APL is never advertised in the usual computer rags and is never second sourced (I believe). Apparently a lot of work gets done at IBM in APL, but the marketing mavens believe in benign neglect for this product. Why STSC is not more vigorous in promoting its PC version will probably remain a mystery. The new Pocket APL may be an attempt to popularize their excellent products; if so, time will tell.

DDJ

(More reviews on page 102)

Software Reviews (Text begins on page 88)

Listing One

```
    ▽ Z←FBNCC
[1]   Z← 1 1
[2]  L:→(100>ρZ←Z,+/¯2↑Z)/L
    ▽

    ▽ R←NONFN
[1]   R←0
    ▽
```

End Listing One

Listing Two

```
    ▽ PRIMES
[1]   ⎕AI[2] ◊ I←1 ◊ F←(8191⍴Y),E
[2]  Y:F[I+P×⍳L(8191-I)÷P←1+I+I]←N
[3]  N:→F[I←I+1]
[4]  E:(⍕Y+.=F),' PRIMES' ◊ ⎕AI[2]
    ▽
```

End Listing Two

Listing Three

```
    ▽ A←FPBENCH;N;R
[1]   ⎕AI[2] ◊ A←N←1 ◊ R←(2499⍴L),E
[2]  L:A←1+3o¯3o⋆⍟(A×A)⋆0.5 ◊ →R[N←N+1]
[3]  E:⎕AI[2]
    ▽
```

End Listing Three

Listing Four

```
    ▽ ⍙⍙R←⍙⍙N TIMER ⍙⍙E;⍙⍙T1;⍙⍙T2;⍙⍙T3;⍙⍙T4
[1]  ⍝ Result ⍙⍙R is the time in milliseconds to execute ⍙⍙E ⍙⍙N times.
[2]  ⍝ NULLEXP is the APL expression whose time is to be subtracted from
[3]  ⍝ the timing.  It is typically 'S←0' if ⍙⍙E begins with 'S←'.
[4]   ⍙⍙T2←(⍙⍙N×1+⍴NULLEXP)⍴NULLEXP,'◊' ◊ ⍙⍙T4←(⍙⍙N×1+⍴⍙⍙E)⍴⍙⍙E,'◊'
[5]  ⍝ ⎕WA on next lines forces a garbage collection.
[6]   ⍙⍙R←⎕WA ◊ ⍙⍙T1←⎕TS ◊ ⍎⍙⍙T2 ◊ ⍙⍙T2←⎕TS
[7]   ⍙⍙R←⎕WA ◊ ⍙⍙T3←⎕TS ◊ ⍎⍙⍙T4 ◊ ⍙⍙T4←⎕TS
[8]   ⍙⍙T1← 60 60 60 1000 ⊥¯4↑⍙⍙T1
[9]   ⍙⍙T2← 60 60 60 1000 ⊥¯4↑⍙⍙T2
[10]  ⍙⍙T3← 60 60 60 1000 ⊥¯4↑⍙⍙T3
[11]  ⍙⍙T4← 60 60 60 1000 ⊥¯4↑⍙⍙T4
[12]  ⍙⍙R←(⍙⍙T4-⍙⍙T3)-⍙⍙T2-⍙⍙T1
    ▽
```

End Listing Four

Listing Five

```
90 PRINT TIME$: SUM=0!:LS$=""
100 OPEN "I",#1,"a:newaku.mss"
105 OPEN "O",#2,"a:newOak.mss"
200 LINE INPUT#1,X$:IF EOF(1) THEN 9000 ELSE 201
201 IF X$=CHR$(13) THEN PRINT#2,X$:GOTO 200
204 A=INSTR(X$,": $"):IF A THEN 1000 ELSE PRINT #2,X$
205 REM:find name
206 N=INSTR(X$,".R."):IF N GOTO 209 ELSE 290
209 L=LEN(X$)-(N+4):RN$=RIGHT$(X$,L)
211 L=1+LEN(RN$)-INSTR(RN$,CHR$(77)):NA$=RIGHT$(RN$,L):GOTO 200
290 FP=INSTR(X$,"Payment"):IF FP GOTO 315 ELSE FI=INSTR(X$,"/8")
300 IF FI GOTO 301 ELSE 310
301 R$=RIGHT$(X$,LEN(X$)-27):L$=MID$(X$,8,5)+STR$(VAL(R$))
302 SUM=SUM+VAL(R$):LS$=LS$+L$+" ":GOTO 200
```

(Continued on next page)

 (Listing Continued, text begins on page 88)

Listing Five

```
310 A=INSTR(X$,"+"):IF A THEN 311 ELSE 315
311 BA=VAL(MID$(X$,A+2,5)):SUM=SUM+BA:GOTO 200
315 A=INSTR(X$,"-"):IF A THEN 316 ELSE 200
316 PA=VAL(MID$(X$,A+2,6)):SUM=SUM-PA
317 IF (BA-PA)(=0! GOTO 200 ELSE PRINT#2,"***PAYMENT IN FULL IS REQUIRED EACH MO
NTH!***"
350 GOTO 200
1000 REM write total now due to invoice
1005 Y$=STRING$(48,32)+"Total Now Due: $"
1010 PRINT#2,Y$,SUM
1050 SUM=0!:LS$=""
1099 REM next bill
1100 GOTO 200
9000 CLOSE: PRINT TIME$
```

End Listing Five

```
      ρR
8501
      BF
1286 3413 5510 7672
      PR
1345 3471 5569 7728
      TND
1534 3658 5756 7913
      ALL
769 838 894 2908 2977 5018 5074 7107 7176 7232 7301 7357 7426 7482
      CM
25 25 25 35 30 50 65 45 45 45 45 65 65 45
      CSUMS
75 65 115 355
      CBF
290 50 ¯49 0
      CPR
25 50 15 0
      CTND
340 65 51 355
```

Matrix A

```
⍉((ρBF),ρCM)ρCM
25 25 25 25
25 25 25 25
25 25 25 25
35 35 35 35
30 30 30 30
50 50 50 50
65 65 65 65
45 45 45 45
45 45 45 45
45 45 45 45
45 45 45 45
65 65 65 65
```

Matrix B

```
ALL∘.<BF
1 1 1 1
1 1 1 1
1 1 1 1
0 1 1 1
0 1 1 1
0 0 1 1
0 0 1 1
0 0 0 1
0 0 0 1
0 0 0 1
0 0 0 1
0 0 0 1
0 0 0 1
0 0 0 1
```

Matrix C

```
<\(ALL∘.<BF)
1 0 0 0
1 0 0 0
1 0 0 0
0 1 0 0
0 1 0 0
0 0 1 0
0 0 1 0
0 0 0 1
0 0 0 1
0 0 0 1
0 0 0 1
0 0 0 1
0 0 0 1
0 0 0 1

+⌿(<\(ALL°.<BF))×⍉((ρBF),ρCM)ρCM
75 65 115 355
```

Figure

Listing Six

```
      PROCESS NAME;R;BF;PR;TND;ALL;A;CM;CSUMS;CBF;CPR;CTND;NP;MES
[1]   ⎕AI[2]
[2]   NAME ⎕NTIE ¯1 ◊ R←⎕NREAD ¯1 82 ,(⎕NSIZE ¯1),0 ⍝ TIE FILES AND READ INTO WS
[3]   BF←R FIND '+)' ◊ PR←R FIND '-)'
[4]   TND←R FIND ': $'
[5]   ALL←((R[A+1]=' ')^R[A-1]=' ')/A←R FINDALL 'igAW'
[6]   CM←⍉,(CLEAN R[ALL•.+ 28 29 30]),' ' ⍝ CLEAN REMOVES JUNK CONVERTS ¯←→-
[7]   CSUMS←+⌿((\(ALL•.(BF))×⍳((⍴BF),⍴CM)⍴CM ⍝ SUM SESSIONS BY CLIENT
[8]   CBF←⍉,(CLEAN R[BF•.+1+⍳3]),' ' ◊ CPR←⍉,(CLEAN R[PR•.+1+⍳3]),' '
[9]   CTND←CSUMS+CBF-CPR
[10]  →(~∨/NP←0)CPR-CBF)/L ◊ MES←NP/PR ◊ R[MES•.+5+⍳45]←'***PAYMENT IN FULL IS REQUIRED EACH MONTH!***'
[11] L:R[TND•.+2+⍳4]←(2⍴(⍴TND))⍴'M(->)I4' ⎕FMT CTND
[12]  NAME[4+2×+/NAME∈':']←'0' ◊ NAME ⎕NTIE ¯2 ◊ 0 ⎕NRESIZE ¯2 ◊ R ⎕NAPPEND ¯2
[13]  ⎕NUNTIE ⎕NNUMS
[14]  'DONE!' ◊ ⎕AI[2]
    ∇

    ∇ Z←R FIND F
[1]   ⍝SUBSTRING SEARCH OF R FOR F, GIVES LOCATION
[2]   Z←(R ⎕SS F)/⍳⍴R
    ∇

    ∇ Z←R FINDALL S
[1]   ⍝FINDS ALL OCCURRANCES OF S IN R
[2]   Z←(R∈S)/⍳⍴R
    ∇


    ∇ Z←CLEAN V;Q
[1]   ⍝REMOVES ANY STRAY CHARACTERS AND CONVERTS NEG. TO MINUS
[2]   Q←,V ◊ Z←Q[(,~Q∈' 0123456789+-¯.')/⍳⍴Q]←' '
[3]   Q←(⍴V)⍴MINUS Q
[4]   Q[(∧/Q∈' ')/⍳1↑⍴V;]←'0' ◊ Z←Q
    ∇

    ∇ Z←MINUS M;N;RM;NM
[1]   ⍝ CONVERTS ¯TO- OR -TO¯;NO EFFECT IF ¯- NOT PRESENT
[2]   N←⍴RM←,M
[3]   →(0(+/NM←,M∈'-')/L ◊ →(0(+/NM←,M∈'¯')/LL ◊ Z←M ◊ →0
[4]  L:RM[NM/⍳N]←'¯' ◊ →E
[5]  LL:RM[NM/⍳N]←'-'⍝ DONT USE 'M(-)...' ⎕FMT CAUSE
[6]  E:Z←(⍴M)⍴RM ⍝ DONT KNOW FORMAT AHEAD OF TIME
    ∇
```

End Listings

VSI – Virtual Screen Interface, Version 2.09

Company: Amber Systems, Inc., 1171 S. Saratoga-Sunnyvale Rd., San Jose, CA 95129

Computer: IBM PC and direct compatibles

Circle Reader Service No. 131

Reviewed by Ronald G.Parsons

VSI is a programmer's tool kit for building applications that manipulate the PC's screen to take advantage of overlapping windows and features such as color and borders. The authors of the program liken VSI for screens to a file system for disks. Using VSI, application programmers need not be concerned with the details of the physical screen, just as they are not concerned with the tracks and sectors of the disks they are writing on. VSI greatly facilitates the now common process of showing a menu on a pop-up window, prompting the user to make a selection with a pointing device such as a mouse or keyboard, erasing the menu, and prompting the user through the rest of the selection.

Windows can be created, moved, enlarged and reduced, placed behind or in front of other windows, and closed with simple calls to the interface. Text can be placed into the windows, with automatic wrapping at the boundaries of the window, at the programmer's option. Similarly, when the input cursor reaches the edge of the window, the window will automatically scroll to keep the cursor in the window. The box around a window can be of any color, character, and attribute; this allows easy creation of "attention" windows for urgent messages. Windows are given a priority: windows of higher priority appear in front of those of lower priority. Best of all, the programmer does not

need to bother with the details of all this screen action. Writing a program to do all this is very simple using VSI and produces a fast window action.

VSI supports both color and monochrome cards on the IBM PC, but not graphics. The VSI interface is in two *levels*: a level 0 interface to assembly language and a level 1 interface to high-level languages—Lattice C in the copy reviewed. Both interfaces provide similar functions because the level 1 interface is built on the level 0 interface. The C interface is especially easy to use: functions with parameters carry out the instructions on the screen.

Additional functions available include an exit reset function, several clear functions, a copy function, functions to read and write to a window, and many others.

Up to 255 different screens may be defined with sizes from one-by-one to 255 rows or columns. VSI can manage up to 65K of screen space. ANSI escape sequences may be used, and VSI will run under DOS 1.1 or 2.*x*. Diagnostic trace capabilities are provided for debugging. A VSI exerciser (VSIX) program that interprets commands from a file can be used to demo the system or for help in understanding the VSI. Mice can be used as input devices to VSIX if they place cursor characters into the keyboard buffer.

The VSI Distribution Kit

As indicated earlier, the version of VSI reviewed included an assembly language interface and a C interface configured for Lattice C V1.03. Other interfaces are available for Pascal (IBM or Microsoft), Compiled BASIC (IBM), Fortran (IBM), PL/I (DRI), and COBOL (Realia); these cost $199 each. All necessary object modules are included to assemble/compile and link a program. In addition, source modules are supplied for the level 1 C interface, for assembly language routines to interface the C code, and for buffer assignment. The source code for VSIX is also included. This program contains many good hints for writing VSI code. An assembly language interface for making VSI a resident program allows the VSI interface to be accessed through a software interrupt.

A PC-size manual is provided, along with two reference cards for the level 0 and 1 interfaces. The documentation is clear and concise.

A Test of Portability

As a test of the portability of the VSI interface, I converted the system to use the Computer Innovations (CI) C86 C V2.1 compiler rather than the Lattice C V2.0 compiler used in other parts of this review. The C interface is supplied in a version tested with the Lattice C V1.03 compiler. Both the level 1 C interface code and the C-based exerciser are supplied in source form and were recompiled and linked with Lattice C V2.0 using the small model. No problems were encountered. Then the C interface code was compiled using the small model of CI C86. Since the only linkage between the C code and the rest of the interface is through one function, which passes the function parameters to the VSI system (this function is implemented in assembly language), the assembly language interface had to be reassembled. Because the assembly language interface is similar in Lattice C and CI C86, only the group and segment names in the supplied source needed to be changed to conform to their respective conventions. Once these two steps were done, my test programs were compiled and linked with CI C86, and they worked as before except for differences in the two C's I/O code. No changes were required in the VSI code. The steps were simple and took little time to complete. Thus, the VSI system appears to be quite portable among C compilers.

Conclusion

If you will be writing a windowing application, seriously consider VSI. It is fast, easy to use, and full of capabilities, and it seems to be bug free.

The Fancy Font System, Version 2.0

Company: SoftCraft, Inc., 222 State Street, Madison, WI 53703 (606) 257-3300

Price: $180

Computer: CP/M, Apple CP/M, IBM PC, Osborne, Kaypro, Victor 900, Epson QX-10 computers and Epson MX,FX, RX, Gemini 10X, IBM Graphics Printer, Riteman Inforunner, TI 855/850, or C. Itoh and NEC8023 (IBM only) printers

Reviewed by David D. Clark

The Fancy Font System is a personal typesetting system for creating nicely printed documents using a personal computer and a dot matrix printer. It uses the high-resolution graphics capabilities of popular printers to produce high quality printing. You can mix different font styles, sizes, and special symbols in the same document. You can use the fonts supplied or create and customize your own alphabets and special symbols.

The Fancy Font System can no longer be considered a new product; you have surely seen the advertisements by now. I've had a copy for over a year. I originally bought version 1.7 of the system, and frankly it was a pain to use. It did produce beautiful print but only after a lot of work. The purpose of this article is to review the updated system. The new version 2.0 is a real improvement. It is much easier to use, has lots of additional options and controls, includes improved documentation, and is available on more computers and printers. If you have an Epson FX-80 printer, the news is even better: it produces even higher quality print.

System Contents

What you get depends on the computer and printer you use. I have a CP/M 2.2-based system with an Epson FX-80 printer. My update from version 1.7 consisted of a new user manual, a quick reference card, and two disks containing the font printing, editing, and creating programs as well as a number of fonts and the Hershey character data base. Also included were files to automatically create some additional fonts that would not fit on the disks.

Besides the standard fonts supplied with the system, SoftCraft is building a font library from which it will sell additional fonts. They are actively soliciting users to submit any fonts that they may have developed themselves. The company also puts out a periodic newsletter informing users of new developments and tips on using the system.

Documentation

The user manual and quick reference card are excellent examples of some of the capabilities of the system: they were printed with Fancy Font. They are very pleasant to the eye, besides containing well-written documentation for the system.

The manual starts with an introduction followed by three large sections,

one for each of the three programs, and a number of appendices. The documentation follows the same format for each program. Each section starts with an introduction outlining the purpose and use of the program. After the generalities are taken care of, the manual describes the commands in detail, devoting one or more pages to each. The detailed descriptions consist of usage instructions, examples, and notes. Depending on the command, there may also be sections on default values, legal values, errors, and suggestions.

The appendices contain a glossary, font descriptions and samples, hints for using Fancy Font with word processors, a summary of error messages, a description of the distribution files, a print-out of the Hershey data base, a description of data file formats, the ASCII character set, and a parameter and command summary. The quick reference card is keyed to the manual by page number; if the card doesn't give you all the information you want, it is simple to look it up.

Pfont

Pfont, the printing program, is the program you will probably use most often. Before using Pfont, you must use your word processor or text editor to create a file containing embedded Pfont formatting commands. Then you start Pfont and set up the printing parameters.

There are 27 embedded formatting commands, usually with several variations available for each one. They control things like font selection, horizontal and vertical movement, underlining, justification, word wrap, indentation, paging, centering, and so on. In general, you use the formatting commands the same way you would in a powerful word processor. Pfont is more advanced than a typical word processor in some ways but more primitive in others. For example, how many word processors can you tell to switch from an 8-point font to a 40-point font or to move backwards up the page 2.63 cm before printing the next line? On the other hand, it's pretty tough to create a five-line page header with Pfont.

Once you have prepared the text file containing the formatting commands, Pfont is executed to print the file. Before printing begins, you have the opportunity to enter a number of printing parameters. About 30 printing parameters control things like print quality (and speed), number of copies, line width, margins, first and last page to be printed, pause after each page, page length, and so on. You are required to enter only the names of the files to be printed and the names of the fonts to be used during printing; all of the other parameters have reasonable default values. The parameters can be entered interactively or retrieved from a parameter file.

Because of the relatively heavy use Pfont will receive relative to the other two programs, it has been constructed with a powerful user interface. You can obtain help with individual parameters and review the settings of all parameters.

Once printing has started, another group of commands becomes active. These commands will immediately stop printing, skip to the next page, skip to the next file, pause after the current page, or allow the user to change the draft mode.

Efont

Efont is the font editing program. It can be used to alter existing fonts or to create new ones from scratch. Of the three programs, this one is probably the most difficult to use—not because of inherent problems in the program but because the mechanics involved in editing a font can become quite involved.

The fundamentals of the process involve loading a font, creating expanded versions of some characters in a group of ASCII text files, editing those files, replacing elements of a font with the edited version, and saving the altered font. It is also possible to create characters from scratch by creating an ASCII text file of the proper type and using the "replace" command for an otherwise empty font file. Additional commands print characters, alter font information, move a character set up or down (to create super- and subscripts, for example), modify the margins for a set of characters, and so on.

When a particular character or range of characters is to be edited, the program will create a series of ASCII text files, one for each character, that contain a line of information about the character (the left and right margins and a value related to the height of the character) followed by the actual character represented as asterisks. One asterisk in the text file represents one pin strike in the finished character. To alter a character, you must exit the Efont program and use a text editor to add, delete, or move asterisks around and to make appropriate changes to the information line. When the changes are complete, the Efont program is entered again to read the altered text file and create an internal character representation from it.

It is sometimes difficult to get a good feel for how the finished character will look: the aspect ratio of the text file is different than the aspect ratio in the printed character. By aspect ratio I mean the ratio of horizontal to vertical elements in a unit square. For example, when Pfont is used to print a character on my printer, there are up to 240 horizontal positions by 216 vertical positions per square inch; 240/216 yields an aspect ratio of about 1.11. However, on my terminal that same square inch can have about nine horizontal elements and four vertical elements, giving an aspect ratio of 2.25. Because of this difference in aspect ratios, the character in its text file representation always looks taller and thinner than it will look when printed.

In practice, it is not difficult to make simple changes to only a few characters. An example in the user manual runs through the process of adding a tilde to an *n*. It is a fairly simple process and well explained. As your editing needs become more complex, it will be necessary to learn some technical aspects of the data representations and conventions used for character fonts. Most users, however, will probably never need to use this program.

Cfont

Cfont is used in conjunction with the Hershey character data base to create new fonts and special symbols. The Hershey data base was created by Alan V. Hershey for the National Bureau of Standards. Besides various styles and sizes of Roman alphabets, it contains old English styles, Greek alphabets of different sizes, and even the Cyrillic (Russian) alphabet. There are also a number of special symbols and graphics from mathematics, music, and who knows where else.

Creating a font from the Hershey data base is well described in the manual and really very easy. It consists of creating a mapping between the number of a character in the data base and a corresponding ASCII number. Then you supply Cfont with scale factors that determine the size of the font created and a baseline—almost always a seven for technical reasons explained in the manual. The process can be performed interactively, or you can create a file of mappings. Examples of both methods are provided. My disk came with map files and a submit file to automatically create three sizes each of a script font and an old English font.

General Impressions

With any complicated program, you can usually expect some problems. I found one, but I'm not sure about it. When I updated my system from version 1.7 to 2.0, a font conversion utility was provided to create fonts from my old fonts that would be compatible with the new version. The copy of the program I got didn't work correctly when parameters were supplied from the command line. I bashed a couple of files this way before switching to the interactive mode. Then things worked fine. I don't know if the problem was with the program or just with my copy, but since it worked correctly in the interactive mode, I didn't bother to get a new copy from SoftCraft.

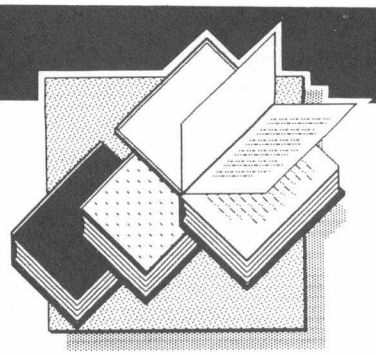
The biggest drawback to Fancy Font is speed. Although it prints beautifully, it prints slowly. This isn't really a problem with the program, though; it's a limitation of the printer. The program uses the graphics capabilities of the printer, driving it as fast as the printer can run. In the highest quality print mode on an FX-80 printer, the program makes at least six passes over a line of text, usually more, depending on the size of the letters. It can take quite a while to print out a page. Versions configured to the Toshiba 1350 or Epson LQ-1500 can print in one pass according to the manual and, I presume, would be faster. (Although the Toshiba and LQ-1500 printers are not mentioned in the advertising, they are discussed in the user manual. You should probably call SoftCraft to find out if versions for those printers are available.)

Another difficulty on my system is the size of the font files. When Pfont prints a file, it loads as many fonts as it can into main memory and reads pieces of the rest from disk as it needs them. My 64K CP/M system usually has room to keep only one font in memory. My disks, however, are fast enough to just about keep up with the printer when Pfont is reading font information from the disk. On systems with slower disks, I have seen the print speed decrease dramatically. For printing rough drafts, the selection can be changed so that all of the information in a font file need not be read in. On IBM systems with larger memories or on systems that use a printer with lower resolution than the FX-80, this may not be a problem.

This program produces very nice print, but in general it is not quite as good as that produced by a daisy-wheel printer. On my printer, the horizontal resolution is 240 dpi (dots per inch) with a vertical resolution of 216 dpi. Although human eyes have a resolution of over 1000 dpi under favorable circumstances, ink on paper usually cannot be resolved that finely. On relatively rough paper, such as that used in computer printers as opposed to that used to print *Scientific American*, the highest resolution appears to be about 400 – 500 dpi. Fancy Font approaches this resolution but can't quite achieve it. This is most noticeable on lightweight diagonal strokes. On vertical and horizontal strokes, the printed image is perfect, as you might expect; the individual pin strikes cannot be discerned. One method to overcome the resolution limits of the printer is to print the text larger than desired then reduce the image. The user manual, which was created this way, looks good. The justified, proportionally spaced output of the program looks better to my eye than that produced by most word processors with daisy-wheel printers. Using Fancy Font is also considerably more fun.

In summary, the program produces good output and works reliably. Except for the glitch with the font conversion utility, I have never had a problem with the system. The new additions made to version 2.0 make the system considerably more flexible and easy to use.

DDJ



The C Programming Tutor

by Leon A. Wortman and Thomas O. Sidebottom
Published by Robert J. Brady Co.
274 pages, paperback
Reviewed by Ian Ashdown

This is most decidedly not your average book on the C programming language. While the authors follow the usual format of explaining the features of the language with examples, it is the examples that set this book apart from all others.

Kernighan & Ritchie in *The C Programming Language* (Prentice-Hall) may have set the stage for *The C Programming Tutor* through their own examples. These included a fully functional word frequency utility using binary trees and a very elegant demonstration of recursion. Wortman and Sidebottom, however, have taken the concept of instructive examples one giant step further by presenting nontrivial and useful programs to demonstrate C. They explain each feature of the language in the context of an overall program rather than as an isolated example.

To be more specific, these programs include (in order of appearance and complexity) a simple calculator, a histogram generation utility, a file encryption program, a printer configuration utility, three text readability analyzers, a word frequency analyzer, a C program cross-reference listing utility, and a chart generator for C program function calls. By the time the authors get around to presenting the function call chart generator, they have interspersed the text with some very complex C code. It is all explained in detail, however, so that you can understand and learn from the listings as you key them into your computer.

The book comprises two parts: "Tutorial" and "Useful Programs." The tutorial takes you by the hand through the usual programming examples to demonstrate the various features of the language—but with a firm emphasis on solving problems. Each example is stated first as a problem with the solution blocked out in English-language statements. The following text then shows how you can implement these statements in C. Very much in the spirit of C, the authors use the resultant functions later for the solution of more complex problems. As more interesting features of C are discussed, many of the functions are progressively rewritten to include them.

I should say here and now that I do not recommend *The C Programming Tutor* as a first and only guide to learning the C language. Learning how to program in C is not particularly easy, and the fast pace of this book is likely to leave the novice confused and frustrated. Learn C from a book like Jack Purdum's *C Programming Guide* (Que Corp.) then use this book to polish your skills.

Since this book teaches C by way of programming examples, I also do not recommend it as a reference guide to the language. While most of C is described in detail, the information is subordinate to the programs. Unless you read the book from cover to cover, you will have a hard time trying to find specific information about some of C's features. As an example, looking up "#define" tells you only that it is a compiler directive used for defining constants. You have to know already about its macro capabilities to realize that the index entry for "code macros – expanding the macro" is an important related topic.

One nice touch is the ongoing discussion of various C compilers, including both Unix and microcomputer implementations. While C is nominally "the" transportable compiler, every compiler writer seems to have his or her own ideas about compatibility. The authors frequently warn you that your particular compiler, especially microcomputer versions, may not support the feature currently under discussion.

The chapter on strings, whimsically named "Things Called Strings," presents a bonus: complete implementations of strlen, strcpy, strcat, strcmp, and index. A recent trend among the companies distributing C compilers is to sell the source code to the standard library functions at additional cost, if at all. This is most unfortunate, for this code is an invaluable resource when it comes to learning the language and writing variations of these functions. While there are many more functions in the standard library, the code presented is nonetheless instructive.

"Things Called Strings" also presents the source code for a pattern-matching utility that finds a string in an ASCII file. While having nowhere near the power and flexibility of the Unix command grep, it is again instructive as a program example. With a bit of programming initiative, you could expand it into something quite useful.

The next chapter, entitled "Paths And Pointers," illustrates the interesting approach the authors have taken to this often confusing topic. *The C Programming Tutor* likens pointers to paths that lead to various places. Between this analogy and the illustrations, the mystery of pointers is clearly explained.

The tutorial part of the book ends with the histogram generation utility, version 10. Part Two, "Useful Programs," is a bit of a misnomer, for it covers such essential C language features as the preprocessor's macro capabilities (the preprocessor itself is never discussed), memory allocation, structures (in two pages!), initialization, typedefs, and buffering. Other features such as separate compilation

of modules, storage classes, pointers to functions, external functions, unions, and bit fields are ignored altogether.

If you know anything at all about C, you may think from the last statement that *The C Programming Tutor* is somewhat lacking as a tutorial on the language. The authors fully admit this—they did not intend for their book to cover every aspect of C. The purpose of *The C Programming Tutor* is to teach the reader how to program effectively in C, not to teach the language *per se*.

Five major programs are presented, ranging from EPSET, a simple utility program for configuring software-programmable dot matrix printers, to CALLS, which prints a chart of how functions in a C program call one another. (To be candid, CALLS is the reason I bought this book. It has proven useful as a debugging and documentation tool, well worth the price of the book in itself.)

Three versions of a text analysis program are given. One is an extended version of the Unix command wc (word count) that counts the number of characters, words, lines, and sentences in a file. The second version applies a definitive "readability" formula from Rudolf Flesch's *The Art of Readable Writing* (Harper & Row) that uses the count of words, average sentence length, syllable count, names, and personal pronouns to determine the ease of reading and human interest content of a body of text.

The third version uses the same formula but optimizes its performance by substituting a hash search method for the linear method used in the second version. The discussion on hashing is very instructive and produces some interesting code that can be adapted for use in many other programs.

The next program is the word frequency analyzer WFREQ. While of some interest, this program is eclipsed by Kernighan & Ritchie's version in *The C Programming Language*. The inherent sorting of words performed by their recursive transversal of a binary tree is far more interesting than the hash table entry and shell sort methods used here. (With the power of recursion available in C, Wortman and Sidebottom did not bother to discuss Quicksort . . . . Shame!)

XREF is a C program cross-reference listing utility that reads a file of C source code, adds line numbers, and makes an alphabetically sorted list of all the names (tokens) used in the file, along with a record of the line numbers associated with each name. Apart from its obvious usefulness in debugging and documenting other C programs, XREF affords the authors a chance to discuss such topics as linked lists and compound data structures. The code itself is quite complex but well documented. As with the text analysis programs, some of the developed functions are useful in their own right.

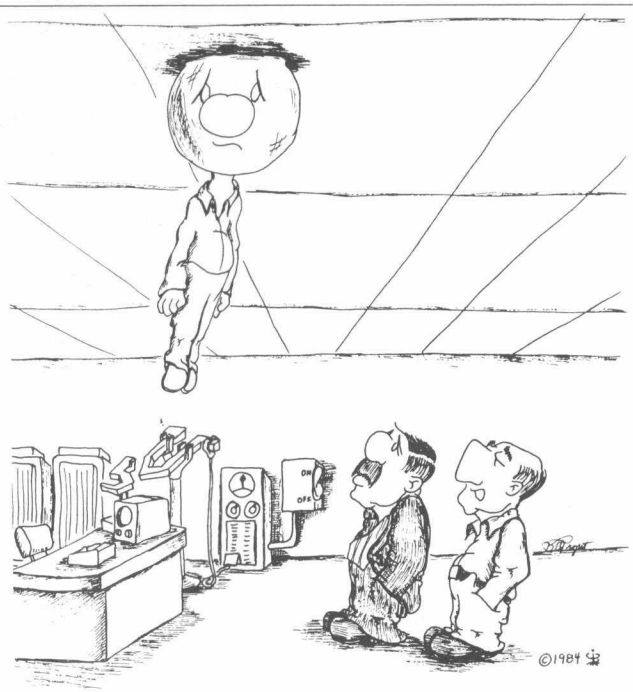
Finally, there is CALLS. As stated earlier, this program is the reason I bought *The C Programming Tutor*. In operation, CALLS scans a C source code file looking for functions. When it finds one, it scans the body of the function for calls made to other functions. When it has done this for all functions, a chart is printed showing the functions in their order of occurrence. Function calls are indicated by indenting the names, recursive functions are identified, and repetitively called functions have their first occurrences referenced by line number. A key advantage of the call chart output is that it is a graphic display of the program structure.

The book finishes with four appendices. Appendix A discusses the various C compilers available (as of July 1983) for CP/M-80, CP/M-86, and MSDOS. The information presented is interesting but was already out of date by the time the book was published. Only one thing remains constant in the world of microcomputer C compilers: they will all undergo continual improvement until they are fully compatible with their Unix brethren.

The remaining three appendices discuss the C header files ctype.h and math.h. The source code is given for ctype.h, which defines the character classification and conversion functions (isascii, iscntrl, toascii, etc.) of C's standard library. The source code is also given for math.h, but the transcendental functions are declared as "external." The authors simply note that if a compiler supports float and double data types but does not provide mathematical functions, an experienced programmer can write them in C and declare them using math.h.

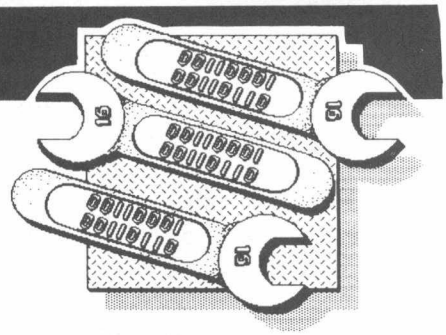
Conclusion? If you program in C, buy this book for CALLS and XREF alone. The rest of the book will be of varying usefulness and interest to different people, but most will learn and profit from reading it.

DDJ



HE WAS WORKING ON A NEW CONCEPT IN BUBBLE MEMORY WHEN THINGS GOT OUT OF HAND.

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

PCDOS version 3.0

Along with the recently announced PC/AT computer, IBM announced release 3.0 of PCDOS. Together with a bad case of bloat (version 3.0 occupies some 48 kilobytes of memory on a PC/AT with hard disk), the new release of PCDOS included a number of additional or modified function calls compared to version 2. Since the DOS Technical Manual is still a very scarce commodity, I will provide an overview of the significant changes in this month's column.

One of the most interesting features of the release of PCDOS 3.0 is the virtual disappearance of its originator, Microsoft, into the background noise. In fact, the word 'Microsoft' doesn't appear at all in sign-on messages or the various DOS manuals, and can be found only in very small print on the diskette label. Although IBM never discusses its future product plans, still, if you were in charge of the Entry Systems Division and you saw Bill Gates on television advertising the Macintosh, what would you be motivated to do? More on this subject later, but reflect on the fact that TopView is a high-performance, windowing environment that completely hides PCDOS and has been earmarked as an IBM "Strategic Product," and you may get an idea which way the wind is blowing.

But back to the new software. The Critical Error Handler interrupt (Int 24H) has been slightly modified, with the addition of some new status information passed in AH and a new DOS response upon return (the application can ask for the offending system call to be "failed" as well as just ignore, retry, or terminate the program). The documentation on how to field the critical error interrupt, which was rather sketchy before, is still somewhat cryptic, but has been expanded to the point of being comprehensible.

Interrupt 2FH is new, and is essentially an interface between application software and the print spooler. A user program can add files to the print queue, cancel files, or examine the spooler's list of waiting files.

Under the general heading of Interrupt 21H (DOS function calls), the following three functions were modified:

Function 38H (Get or Set Country Information) was radically expanded: first, with the ability to set a country code as well as to interrogate it; next, with the capacity for more than 255 country codes; and last, with much more information passed back in the data block regarding country specific delimiters and formats.

Function 3DH (Open File) was enhanced to support both multi-tasking and networking. When a file is opened, the application can specify whether it will be "inherited" by a child process running in the same network node, and whether it can be "shared" (i.e. independently opened) by another task running in the same or another network node.

Function 44H (I/O Device Control) has two new capabilities: to interrogate whether a particular block device has removable media, and to set the number of retries and the delay between retries for "file sharing conflicts."

There are also five completely new documented function calls for Interrupt 21H:

Function 59H (Get Extended Error) can be called by an application after an error code is returned by some other function to get more detailed information about the failure, the class of failure (temporary, system internal, hardware, etc.), and the recommended response for the application (retry, delay and retry, abort, etc.)

Function 5AH (Create Temporary File) is passed a path string by the application, generates a unique filename string, creates the file in the specified subdirectory, and returns a complete path and file specification string. The file can then be opened by the application and used in any way desired. The file is not deleted automatically when the application terminates.

Function 5BH (Create File) is exactly like the DOS 2.0 function 3CH (Create File), except that it will fail if the file already exists. Function 3CH, on the other hand, truncates a previously existing file by the same name to zero length, then returns a success code.

Function 5CH (Lock/Unlock File Access) provides a general mechanism for a task to gain exclusive access to a region of a file, even if the file is being shared with other tasks.

Function 62H (Get Program Segment Prefix Address) is self-explanatory. This will be handy for writers of EXE-type programs.

With release 3.0 of PCDOS, it is indeed clear that this operating system is going to continue to get more and more complex and powerful, even if it (or its descendent) is eventually completely submerged in TopView or some other IBM-proprietary shell. I suggest that the following guidelines for "well-behaved" applications will make the programmer's life easier in the brave new world of multi-tasking and networking that is nigh upon us:

• Use the Extended File functions

(3CH-42H) to open, close, read, and write files, rather than the old "FCB" type calls which were cloned from CP/M. The extended functions are more powerful, have better error reporting, and support the hierarchical file structure and file sharing.

• Use the Modify Allocated Memory Blocks call to release any memory that is not required by the application program. In the same vein, be sure to examine the program segment prefix to find the amount of memory allocated to your application, and do not access memory outside those bounds (even if you can establish that more memory is physically present).

• Don't access the interrupt vectors directly; use the DOS function calls 25H and 35H to set and get the contents of interrupt vectors respectively. This has implications for TopView as well as for the 80286 in the PC/AT, which can support multiple tables of interrupt vectors in protected mode.

• When your program gets control, use function 30H to get the DOS function number. If your program is running under DOS 3.x, interrogate the extended error codes (function 59H) when a DOS function fails to get more detailed information and the suggested action. In any case, employ the critical error handler interrupt capabilities to make your program more forgiving of hardware errors.

• Use the Exec call to spawn other tasks or load overlays; don't try to set up Program Segment Prefixes yourself.

• Write directly to the video buffer if you must, but perform all mode changes through calls to the ROM BIOS. Cause all video code to access a variable that contains the buffer segment address. This will make modification of a program for operation under TopView (where a "shadow buffer" is assigned) much simpler.

• If writing to a standard device or a file, send strings rather than single characters whenever possible. Take advantage of the buffering and fast transfer logic written into the DOS drivers. Conversely, don't waste your program's resources on internal blocking and deblocking of records or characters when the DOS is already imposing multiple layers of buffering between you and the device.

• Before exiting the application, be careful to close all file control blocks and/or handles. If regions of a file have been locked, unlock them before closing the file.

• Terminate your program with a return code via function 4CH of Interrupt 21H, rather than the previously accepted mechanism of function 0 or Int 20H. The return code can be interrogated by the invoking process or by the batch subcommands IF and ERRORLEVEL.

• If writing a resident driver, use Interrupt 21H Function 31H to terminate and stay resident, rather than the previously approved Interrupt 27H. This allows for return information to be passed, and also for resident programs larger than 64 kilobytes.

Graphics Routines for the IBM PC

Bruce A. Smith writes: "Enclosed is an 8088 assembly listing of a high performance line drawing and point plotting routine for the IBM PC or PCjr. These routines were written for utilization in application programs that I am devel-

oping in 'C'. I wish to encourage the integration of graphics in software and I thought these two fast 'nuts and bolts' plotting routines were worth sharing with your readers.

"Listing One (page 110) contains the two procedures to plot a point or draw a line on the medium resolution screen of the PC. The line drawing routine uses self-modifying code. They are both written to OR the color bits onto the screen. The line drawing routine is approximately three times faster than the one listed in Morgan's *Bluebook of Assembly Routines for the IBM PC & XT*. It is also three times faster than the line drawing routine used in the ROM Basic on the PCjr. The point plotting routine is a little more difficult to compare, because the overhead of the call and the computation for the next point become significant. In performance tests, it plotted points half again as fast as the routine in Morgan's book, and in spite of the overhead, it performed four times faster than the ROM BIOS point plotting routines.

"Listing Two (page 122) is a 'C' program which illustrates use of these routines from a high level language." Interested readers can contact Bruce Smith at 305 E. Edgewood Blvd., Apt. 5, Lansing, MI 48910.

The line drawing routine sent by Mr. Smith employs Bresenham's Algorithm, which is particularly well suited to microcomputer graphics routines because it has a simple inner loop and uses only integer arithmetic. It was first published by J. E. Bresenham in the article "Algorithm for Computer Control of Digital Plotters," *IBM Systems Journal*, 4 (1) 1965, pages 25 – 30.

Those of you delving into graphics on any microcomputer will find the following two books to be invaluable:

Fundamentals of Interactive Computer Graphics, by J. D. Foley and A. Van Dam, Addison-Wesley, 1982. This book is the standard by which all future graphics texts will be measured. Incidentally, it contains a nice explanation of the Bresenham algorithm on pages 433 – 436. on pages 433 – 436.

Applied Concepts in Microcomputer Graphics, by Bruce A. Artwick (author of the IBM PC Flight Simulator program), Prentice Hall, 1984. No-nonsense, practical advice from a guy who has done it all.

DDJ

16-Bit (Text begins on page 108)

Listing One

```
        PAGE    255,132
;---------------------------------------------------------------
;
;   Plot Point or Draw Line on IBM Color Graphics Adaptor
;
;   "orline.asm"
;   "orpt.asm"
;
;   written by Bruce A. Smith  7/24/84
;
;---------------------------------------------------------------
;
DGROUP  GROUP   DATA
DATA    SEGMENT WORD PUBLIC 'DATA'
        ASSUME  DS:DGROUP
        PUBLIC  COLOR,X1,X2,Y1,Y2
;
; the order and type of declaration is important here
;
y2      dw      0
x2      dw      0
y1      dw      0
x1      dw      0
color   db      0
;
; color = color * 4 + 1, color 0 == mask
;
ctableh db      03Fh,0CFh,0F3h,0FCh
        db      040h,010h,004h,001h
        db      080h,020h,008h,002h
        db      0C0h,030h,00Ch,003h
;
fakedw  = 1000h
;
DATA    ENDS
;
;---------------------------------------------------------------
;
```

(Continued on page 112)

16-Bit (Listing Continued, text begins on page 108)

Listing One

```
PGROUP  GROUP   PROG
PROG    SEGMENT BYTE PUBLIC 'PROG'
        PUBLIC  ORLINE,ORPT
        ASSUME  CS:PGROUP
;
;-----------------------------------------------------------
;
;   orline.asm
;
;-----------------------------------------------------------
;
; ROUTINE TO OR A LINE ONTO MEDIUM RESOLUTION SCREEN
;
; uses Bresenham's algorithm
;
orline  proc    near
        push    bp              ; save calling bp
                                ; only reg needed to save for 'C'
        push    ds              ; save ds
; ------------------------------
; get x & y values
        mov     si,OFFSET y2    ; addr y2
        lodsw                   ; ax = y2
        xchg    ax,dx           ; dx = y2
        lodsw                   ; ax = x2
        xchg    ax,di           ; di = x2
        lodsw                   ; ax = y1
        xchg    ax,cx           ; cx = y1
        lodsw                   ; ax = x1
        mov     bx,si           ; bx = addr color
        xchg    ax,si           ; si = x1
;
        cmp     si,di           ; cmp x1,x2
        jle     swapxy          ; skip if (x1<=x2)
        xchg    cx,dx           ; (x1>x2): swap y1,y2
        xchg    si,di           ; swap x1,x2
swapxy:
;
;    ..............................................
;    |       H           |          L            |
; ----------------------------------------------
; ax |                   |                       |
; ----------------------------------------------
; bx |              addr color                   |
; ----------------------------------------------
; cx |          0        |       y1              |  if ( x1 > x2 )
; ----------------------------------------------  swap (x1,y1) with
; dx |          0        |       y2              |      (x2,y2)
; ----------------------------------------------  ie.
; si |                  x1                       |    xchg cx,dx
; ----------------------------------------------    xchg si,di
; di |                  x2                       |
; ----------------------------------------------
; bp |                                           |
; ----------------------------------------------
;
;
; ch = deldy = (y1>y2) ? -80 : 80
; dx = |y2-y1|
        sub     dx,cx           ; y2-y1
        mov     al,80           ; deldy = 80
        jge     ydown           ; skip if (y1<=y2)
        neg     dx              ; |y2-y1|
        neg     al              ; deldy = -1
ydown:
        sub     di,si           ; x2-x1
; di = |x2-x1|
;
```

(Continued on page 115)

16-Bit (Listing Continued, text begins on page 108)

Listing One

```
;    ..........................................
;    |       H          |          L          |
; ---------------------------------------------   al=80
; ax |                  |   (y1>y2)? -80: 80  |   dx-cx = y2-y1
; ---------------------------------------------   if neg (y1>y2)
; bx |              addr color                |     neg dx =|y2-y1|
; ---------------------------------------------     al=-80
; cx |          0       |       y1            |
; ---------------------------------------------   deldy = al
; dx |          0       |    absdy = |y2-y1|   |
; ---------------------------------------------
; si |                  x1                    |
; ---------------------------------------------   di-si
; di |          absdx = x2-x1                 |   = |x2-x1|
; ---------------------------------------------
; bp |                                        |
; ---------------------------------------------
;
;
        cmp     di,dx           ; absdx,absdy
        lahf
        jnl     minmax          ; skip if (absdx>=absdy)
        xchg    di,dx
minmax:
; dx = dmin
; di = dmax
;
;    ..........................................
;    |       H          |          L          |  if (absdx < absdy)
; ---------------------------------------------  cmp di,dx  lahf
; ax | flags absdx,absdy|       deldy          |  jnl -
; ---------------------------------------------    swap(absdx,absdy)
; bx |              addr color                |    xchg di,dx
; ---------------------------------------------  -:
; cx |          0       |       y1            |
; ---------------------------------------------
; dx |          0       |       dmin          |
; ---------------------------------------------
; si |                  x1                    |
; ---------------------------------------------
; di |                  dmax                  |
; ---------------------------------------------
; bp |                                        |
; ---------------------------------------------
;
;
        xchg    ax,bp           ; bp=flags(absdx,absdy) & deldy
;
; ROUTINE TO FIND INITIAL Y-ADDR, X-ADDR, AND ROTATED COLOR
;
; multiply y-coord by bytes per row and adjust for even/odd lines
        ror     cl,1            ; adjust odd/even
        mov     ax,cx           ; ax = cx = adj y-coord
        and     al,7Fh          ; page mask
        sal     cx,1            ; times 2
        sal     cx,1            ; times 4
        add     cx,ax           ; y-coord times 5
        sal     cx,1            ; times 10
        sal     cx,1            ; times 20
        sal     cx,1            ; times 40
        sal     cx,1            ; times 80
;
;    ..........................................
;    |       H          |          L          |
; ---------------------------------------------
; ax |          0       |                     |
; ---------------------------------------------
```

(Continued on next page)

Listing One

```
; bx |       addr color = addr ctableh-1        |
; ---------------------------------------------
; cx |                  y-addr                  |
; ---------------------------------------------
; dx |          0       |       dmin            |
; ---------------------------------------------
; si |                  x1                      |
; ---------------------------------------------
; di |                  dmax                    |
; ---------------------------------------------
; bp | flags absdx,absdy|       deldy            |
; ---------------------------------------------
;
;
; compute the rotated mask and color
; bx = ctableh-1 = addr color
        mov     al,3            ; pixel position mask
        and     ax,si           ; just the bit count into the index
        add     al,[bx]         ; pixel position + color (* 4 + 1)
        xlat                    ; look up the masks al=[al+bx]
;
        mov     bx,0B800H       ; disp seg base addr
        mov     ds,bx           ; ds = display base addr
;
;  al = rotated color
;  cx = y-addr offset
;  ds = display addr
;
;    .........................................
;    |       H          |          L          |
; ---------------------------------------------
; ax |          0       |    rotated color    |
; ---------------------------------------------
; bx |                  |                     |
; ---------------------------------------------
; cx |                  y-addr                  |
; ---------------------------------------------
; dx |          0       |       dmin            |
; ---------------------------------------------
; si |                  x1                      |
; ---------------------------------------------
; di |                  dmax                    |
; ---------------------------------------------
; bp | flags absdx,absdy|       deldy            |
; ---------------------------------------------
;
;
        sal     dx,1            ; dx = delse = dmin * 2
        xchg    cx,di           ; cx = dmax, di = y-addr
        xchg    ax,dx           ; ax=delse, dx=rotated color
        xchg    ax,bp           ; ax=flags(absdx,absdy), bp=delse
;
;    .........................................
;    |       H          |          L          |
; ---------------------------------------------
; ax | flags absdx,absdy|       deldy           |
; ---------------------------------------------
; bx |                  |                     |
; ---------------------------------------------
; cx |                  dmax                   |
; ---------------------------------------------
; dx |          0       |    rotated color    |
; ---------------------------------------------
; si |                  x1                     |
; ---------------------------------------------
; di |                  y-addr                 |
; ---------------------------------------------
; bp |            dmin * 2 = delse             |
; ---------------------------------------------
```

```
;
;
        sahf                    ; cmp absdx,absdy
        pushf
;
        cbw                     ; ax = deldy
        mov     cs:delsy,ax     ; save deldy
        mov     cs:deldy,ax
        mov     cs:deldy2,ax
        mov     bx,di           ; bx = y-addr
        or      ax,ax
        js      negdeldy        ; if deldy<0 jmp
;
        test    bh,20H          ; is page bit set?
        jz      toggle          ; skip to toggle page
;
        add     bx,ax           ; add deldy to y-addr
        jmp     toggle          ; skip to toggle page
negdeldy:
        test    bh,20H          ; is page bit set?
        jnz     toggle          ; skip to toggle page
;
        add     bx,ax           ; add deldy to y-addr
toggle:
        xor     bh,20H          ; flip page bit
; bx = next y-addr
;
;    ..............................................
;    |       H          |          L             |
; ---------------------------------------------
; ax |                  |                        |
; ---------------------------------------------
; bx |             y-addr (next)                 |
; ---------------------------------------------
; cx |                 dmax                      |
; ---------------------------------------------
; dx |          0       |    rotated color       |
; ---------------------------------------------
; si |                 xl                        |
; ---------------------------------------------
; di |                 y-addr                    |
; ---------------------------------------------
; bp |           dmin * 2 = delse                |
; ---------------------------------------------
;
;
        xchg    ax,bp           ; ax = dmin * 2 = delse
        mov     cs:delse,ax     ; save delse
        mov     cs:delse2,ax    ; save delse
        sub     ax,cx           ; ax = dmin * 2 - dmax = d
        mov     bp,ax           ; bp = d = error term
        sub     ax,cx           ; ax = dmin * 2 - dmax * 2
        mov     cs:delde,ax     ; save delde
        mov     cs:delde2,ax    ; save delde
;
        xchg    ax,bx           ; ax = next y-addr
;
; figure x-coord address
        mov     bx,si           ; get x-coordinate
        sar     bx,l            ; divide
        sar     bx,l            ; by 4
;  bx = x-addr offset
;
;    ..............................................
;    |       H          |          L             |
; ---------------------------------------------
; ax |                 y-addr  (next)            |
; ---------------------------------------------
; bx |                 x-addr                    |
; ---------------------------------------------
; cx |                 loop count                |
; ---------------------------------------------
```

(Continued on next page)

Listing One

```
; dx |                  |    rotated color      |
; ------------------------------------------------
; si |                  x value                  |
; ------------------------------------------------
; di |                  y-addr                   |
; ------------------------------------------------
; bp |                  error term               |
; ------------------------------------------------
;
;
        popf                    ; cmp absdx,absdy
        jns     delsx2          ; if (absdx>=absdy) goto delsx2
;
; ------------------------------------------------
; delsx = 0  (absdx < absdy)
;
        or      [bx][di],dl     ; or disp with color (plot point)
        jcxz    lineexit        ; quit when cx=0
        or      bp,bp           ; set bp flags
        jge     diagonal        ; if bp>=0 jmp
;
; case for straight move
straight:
        xchg    ax,di           ; every other for page adj
delsy   = $+1
        add     ax,fakedw       ; ++y
;
        or      [bx][di],dl     ; or disp with color (plot point)
        dec     cx              ; --loop counter
        jz      lineexit        ; quit when cx=0
;
delse   = $+2
        add     bp,fakedw       ; update error term
        js      straight        ; if bp<0 goto straight
;
; case for diagonal move
diagonal:
        inc     si              ; ++x value
        mov     bx,si           ; bx = x value
        sar     bx,1
        sar     bx,1            ; bx = x addr offset
;
        ror     dl,1            ; adjust color position
        ror     dl,1
;
        xchg    ax,di           ; every other for page adj
deldy   = $+1
        add     ax,fakedw       ; ++y
;
        or      [bx][di],dl     ; or disp with color (plot point)
        dec     cx              ; --loop counter
        jz      lineexit        ; quit when cx=0
;
delde   = $+2
        add     bp,fakedw       ; update error term
        js      straight        ; if bp<0 goto straight
        jmp     diagonal        ; if bp>=0 goto diagonal
;
; ------------------------------------------------
delsx2:
; delsx = 1  (absdx >= absdy)
;
        or      [bx][di],dl     ; or disp with color (plot point)
        jcxz    lineexit        ; quit when cx=0
        or      bp,bp           ; set bp flags
        jge     diagonal2       ; if bp>=0 jmp
;
```

(Continued on page 120)

Listing One

```
; case for straight move
straight2:
        inc     si              ; ++x value
        mov     bx,si           ; bx = x value
        sar     bx,1
        sar     bx,1            ; bx = x addr offset
;
        ror     dl,1            ; adjust color position
        ror     dl,1
;
        or      [bx][di],dl     ; or disp with color (plot point)
        dec     cx              ; --loop counter
        jz      lineexit        ; quit when cx=0
;
delse2  = $+2
        add     bp,fakedw       ; update error term
        js      straight2       ; if bp<0 goto straight
;
; case for diagonal move
diagonal2:
        inc     si              ; ++x value
        mov     bx,si           ; bx = x value
        sar     bx,1
        sar     bx,1            ; bx = x addr offset
;
        ror     dl,1            ; adjust color position
        ror     dl,1
;
        xchg    ax,di           ; every other for page adj
deldy2  = $+1
        add     ax,fakedw       ; ++y
;
        or      [bx][di],dl     ; or disp with color (plot point)
        dec     cx              ; --loop counter
        jz      lineexit        ; quit when cx=0
;
delde2  = $+2
        add     bp,fakedw       ; update error term
        js      straight2       ; if bp<0 goto straight
        jmp     diagonal2       ; if bp>=0 goto diagonal
;
; ------------------------------------------
lineexit:
        pop     ds              ; restore ds
        pop     bp              ; restore calling bp
        ret
;
orline  endp
;
;-------------------------------------------------------------
;
;   orpt.asm
;
;-------------------------------------------------------------
;
; ROUTINE TO OR A POINT ONTO MEDIUM RES COLOR SCREEN
;
orpt    proc    near
;
; get initial values for x and y
        mov     si,OFFSET yl    ; addr yl
        lodsw                   ; ax = yl
;
; multiply y-coord by bytes per row and adjust for even/odd lines
        ror     al,1            ; adjust odd/even
;
        mov     dx,0B87FH       ; disp addr and page mask
```

(Continued on page 122)

 (Listing Continued, text begins on page 108)

Listing One

```
        and     dl,al           ; mask page bit, disp + y.coord
        sal     ax,1            ; times 32
        sal     ax,1            ; times 64
        add     dx,ax           ; addr disp seg + y-coord times 5 (80)
;
; compute x-coord address offset
        lodsw                   ; ax = x1
        mov     di,ax           ; get x-coordinate
        sar     di,1            ; divide
        sar     di,1            ; by 4
;
; compute the rotated mask and color
        and     al,3            ; just the bit count into the index
        add     al,[si]         ; pixel position + color (* 4 + 1)
        mov     bx,si           ; bx = ctableh - 1
        mov     si,ds           ; save ds
        xlat                    ; look up the masks al=[al+bx]
;
        mov     ds,dx           ; set seg to disp + y-addr
        or      [di],al         ; or the byte with the color
;
        mov     ds,si           ; restore ds
        ret
;
orpt    endp
;
;-------------------------------------------------------------
;
PROG    ENDS
        END
```

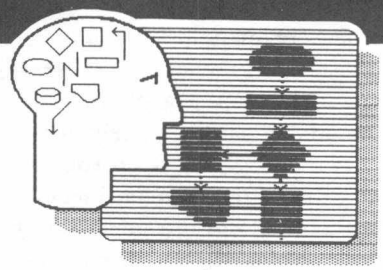
End Listing One

Listing Two

```
/*
This is an example written in 'C' using the line drawing routine.
-- Bruce Smith.
*/

extern int x1,y1,x2,y2;
extern char color;

main()
{
int incr=3;
pcvsvm(4);  /* color med res graphics */

color=(2<<2)+1;

x1=160; y1=100; x2=0; y2=0;

for (; x2<319; x2+=incr) orline();   /* draw line */
x2=319;

for (; y2<199; y2+=incr) orline();   /* draw line */
y2=199;

for (; x2>0; x2-=incr) orline();   /* draw line */
x2=0;

for (; y2>0; y2-=incr) orline();   /* draw line */

cget();     /* wait for keypress */

pcvsvm(2);  /* 80 col b & w */
}
```

End Listing Two

THE SOFTWARE DESIGNER



by Michael Swaine

Tom Evslin and his company, Solutions Inc., have written seven programs for the Macintosh so far, including Dow Jones Straight Talk and Spreadsheet Link, Rags to Riches Payable and Receivables, and learning games for Que. Eighty percent of the company's development work is software for the Mac. As the first person in his company to do Mac development, Evslin has some opinions about what it's like to develop software for the machine named after a raincoat.

Evslin: As everyone has been saying, the Mac presents a bigger hurdle to software development than any past microcomputer. The vocabulary of capabilities represented by those utilities in ROM is powerful, but the Mac gives you no easy way to start using them. Traditionally, when you were going to write a program for an Apple computer, you started by getting into BASIC and writing a Hello program. Well, there's no way to do that on the Mac. There is no immediately obvious way to write to the screen, no obvious way to grab keyboard input. This was intentional; it forces you to get into and use that excellent interface. But at first, it's discouraging. But then there follows a period of euphoria. Each programmer on our team has gone through this depression, when there appear to be random events going on. Then the euphoria comes as he begins to learn how to make pulldown menus and scrollbars work.

DDJ: I'm sure that's a nice feeling, but what's the practical implication?

Evslin: Overall, you become a more productive software developer. You just wouldn't build that kind of user interface into every product if you had to do it from scratch every time. The Mac software environment puts that interface into every product for you.

DDJ: What problems do you run into in porting software into the Mac environment?

Evslin: You don't port software to the Mac. Or we don't. Our products are all new code. We look at the old code; we have it to look at as we work; but what we write is all new code. You can't just translate software from another environment to the Mac; when you do, the result is a piece of junk.

DDJ: What kind of support do developers get from Apple in learning to work in the environment, learning to develop software for the machine? Is Mac college helpful?

Evslin: Apple runs a Mac college but until recently there was no Mac high school. There was no way to get there from here. Now there are a few people providing that level of training, but most people have had to get there by the school of hard knocks. Mark Ursino (formerly of Microsoft) is one of these teaching at the lower level; Mark has taught a Mac high school at Apple. That was helpful.

DDJ: The user documentation for the Mac is attractive and well organized, but I was a bit surprised when I first saw the pile of photocopied pages that make up the documentation a developer gets. Isn't that documentation a handicap?

Evslin: The lack of high-level, structured documentation has been a problem. Apple has given programmers one level of documentation, with no way to get an overview. You read the window management section and you will probably learn more than you would ever want to know about window management in the Mac environment, but there's no way to know, short of exhaustive search, if what you want to know about a particular point in window management is in the documentation.

DDJ: What about support?

Evslin: There are bugs, but few of them, considering all that is in the ROM. And there is developer support. Not every programmer gets all the access to the developers that he wants when he wants it, but contrast this with IBM, where developers are just inaccessible. Right now, you can actually talk to the people who were involved in the development of the Mac software. You can get at the actual developers.

DDJ: Has it been a hindrance that you have to work on the Lisa?

Evslin: Having to do development on another machine is a barrier. It's not so bad for a company like ours, but it is difficult for somebody working in his basement to have to go out and buy another machine to develop software for the Mac.

DDJ: Do you think that the 512K Mac is going to make things easier for software developers?

Evslin: The 512K Mac will not be as great a boon to programmers as everyone thinks. Memory has been overrated as an aid to the developer. You don't have to build a user interface or do those other things the ROM routines do for you, so you have more of the RAM to use anyway. 128K is not terribly limiting. What's confusing is that the Mac memory management is hard to get used to. That's really where the problems lie, not in the amount of memory. The larger machine may in fact mask bugs more easily than the 128K machine. With the 128K, the heap gets reorganized fairly often. It's easy to write routines that depend on something being relocatable when in fact it's not. Then the heap gets reorganized and things get moved and you see an erratic, nonrepeatable bug. With 512K you may never see the bug crop up in beta test.

I was the first in our company to do development on the Mac and I saw all the others going through the learning process. Nowhere did anybody have as much trouble as in memory management. Where handles get allocated,

heap management. Hardest-to-track bugs.

DDJ: The predictions are that Jack Tramiel at Atari and everyone else will soon be bringing out Maclike machines, and the predictions seem likely to come true, more or less. What do you think we can expect from such machines?

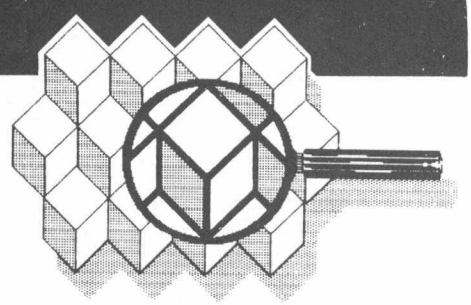
Evslin: Those who are going for the low-cost market will opt for maximum compatibility with the Mac; those selling functionality will go for merely using the conceptual interface and will try to differentiate themselves in terms of functions. Both high and low levels of compatibility will emerge.

The Mac itself is poised to penetrate two new markets: people who would never have bought a computer before and the professional market. The former, for sure, as more software is developed that is fun. We will see not so much a fight with IBM for market share as an expansion of the market. Professionals, I'm not so convinced about, but as the so-called power software comes out, the opportunity will exist.

[*After we spoke, Apple leaked its current marketing plans: the Christmas "test drive a Mac" deal and the plan to go after corporate middle managers, leaning on the new Lotus product for the Mac, the planned Mac local-area network, a laser printer, and a telephone integrated into the machine. CEO John Sculley claimed that Apple has a two-year window for getting Macs on corporate desktops and said that the plan was not one that puts Apple on a collision course with IBM.*]

DDJ

OF INTEREST



by R. P. Sutherland

Subliminal Software

New Life Institute has released software for the IBM PC and compatibles designed to reprogram the user's habits, behaviors, and personal beliefs. The program entitled **Subliminal Suggestions and Self Hypnosis Programs *for Your Computer!*** flashes user-chosen messages as a super-high-speed subliminal background on the screen. The subliminal messages can be used to reprogram the user's subconscious. The package includes separate programs for self-hypnosis and deep relaxation. The accompanying documentation contains a warning about possible misuses of subliminal technology. The "Subliminal Suggestions and Self Hypnosis" package retails for $75.00. The programs are written in C with several assembly language routines. Contact New Life Institute, P.O. Box 2390, Santa Cruz, CA 95063 (408) 429-1122. **Reader Service No. 109.**

Babel

The Babel that towers above Silicon Valley casts a shadow around the globe. A high-tech lingua franca could speed aspects of the microcomputer revolution, but diversity promotes creativity. If, by joining forces, the United States and Japan could produce the man-like machine that Turing predicted by the 21st century, one can't help but wonder if the present confusion is God's attempt to frustrate our progress.

I attended Gary Kildall's PC Faire presentation in San Francisco last year, in which he was supposed to prophesy the direction of the microcomputer industry over the next three years. Instead, he traced the development of microprocessors, memory, and input/output devices during the last decade and limited any predictions to one year. Within a year, he said, programmers may assume that the standard business computer will have 3 megabytes of RAM, a CPU in the 80286 category or a 32-bit processor, and, thanks to laser disk technology, enough storage to contain the *Encyclopaedia Britannica*. All of this speed and power means that we are going to be able to get a lot more work out of the microcomputer. How? Kildall had the solution: a copy of Concurrent DOS, still in the shrink-wrap, which he offered to sell because, he said, he needed plane fare home. Concurrent DOS, like TopView, is a multitasking, windowed program to enhance MSDOS. Microsoft Windows, a similar product, has been delayed until June.

Kildall seemed most enthusiastic about **Virtual Device Interface** (VDI) technology. By raising the functionality of all devices to a common level, VDI allows one to create programs that are not at the mercy of technological leapfrogs. The VDI presents a general interface through which high-level programs can communicate with a wide variety of hardware. IBM, Digital Research Institute, and now Ashton-Tate are licensing GSS-Drivers from Graphic Software Systems. GSS-Drivers is an implementation of the American National Standards Institute's proposed VDI programming standard, joined with a library of device drivers, which permits input and output device independence. Graphic Software Systems is located at 25117 South West Parkway, Wilsonville, OR 97070. Contact: William Merchant (503) 682-1606.

Vertex Systems has announced **Apple-Turnover**, a board that gives IBM PCs the ability to read and write Apple-Dos 3.3 and Apple CP/M disks. Vertex Systems also has XenoCopy, a system of programs to copy files between disks of different computers. I wanted to compare it to microVersal, a similar product that we have been using with mixed results, but Vertex was unable to part with a review copy. Apple-Turnover is priced at $279.50 and XenoCopy by Fred Cisin is $99.50 or $149.50 for XenoCopy Plus, which permits writing to and formatting other disks. Vertex Systems, Inc., 6022 W. Pico Blvd., Los Angeles, CA 90035 (213) 938-0857. **Reader Service No. 105.**

CompuPro users may now run PCDOS at CompuPro speed. Computer House announces **PCPRO** version 2.4. PCPRO is IBM's PCDOS version 2.1 modified to operate on CompuPro letter series systems. The package includes INTERDOS, a CP/M to MSDOS transfer utility, and COFIGIO, an interactive I/O configurative utility. PCPRO: PCDOS for CompuPro costs $395.00 from Computer House, 722 B Street, San Raphael, CA 94901 (415) 453-0865. **Reader Service No. 107.**

Hardware

Logitech has applied infrared technology to the mouse and produced **the first cordless mouse** as a result of custom development work for Metaphor Computer Systems. Logitech Inc. is located at 805 Veterans Blvd., Redwood City, CA 94063 (415) 365-9852. **Reader Service No. 111.**

CompuPro has introduced a high-speed, low-power static RAM board compatible with both 8- and 16-bit processors. The **RAM 23** provides up to 128K of static RAM and operates at up to 12 MHz with 8086/8088/80286 CPUs. The RAM 23 has a suggested retail price of $400.00 for the 64K version and $775.00 for the 128K version. Contact Jeff Swartz, CompuPro, 3506 Breakwater Court, Hayward, CA

94545 (415) 786-0909. **Reader Service No. 115.**

Morrow has begun shipping a 10-pound battery-operated portable computer that operates under MSDOS and uses standard 5¼-inch floppy-disk drives. A 640K dual drive version is available for $3,695.00. Standard features include: clock, built-in 300 baud modem, calculator, and word processor. The CPU is a CMOS 80C86. The **Pivot** is available from Morrow, 600 McCormick, San Leandro, CA 94577. **Reader Service No. 117.**

Miscellany

Musical Software

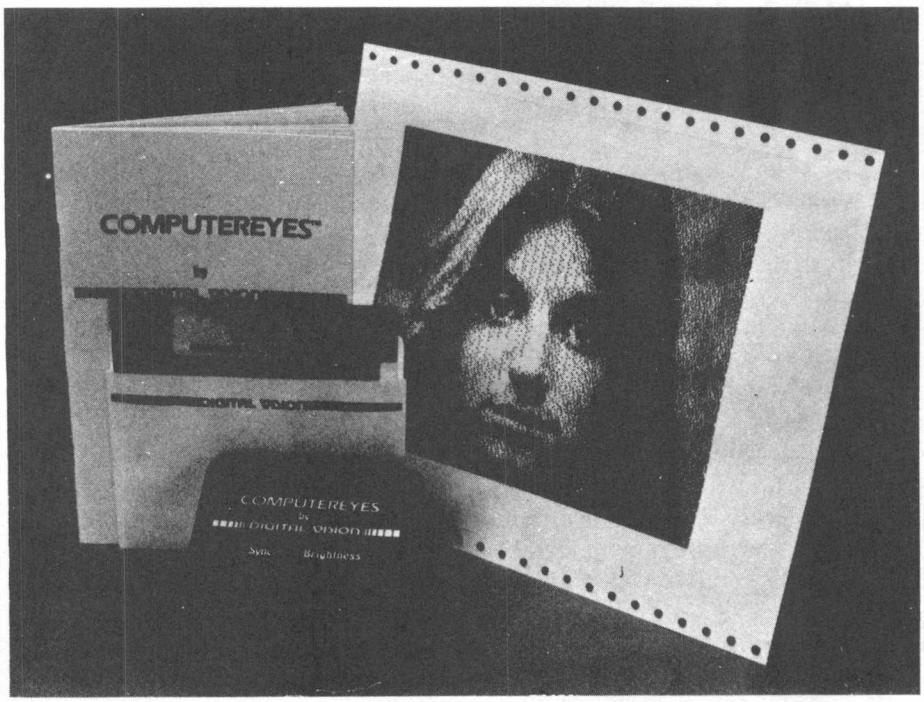
Tune Smith/PC is the first software for musical composition for the IBM PC. Price: $49.95 from Blackhawk Data Corporation, 307 N. Michigan Avenue, Chicago, IL 60601 (312) 236-8473. **Reader Service No. 119.**

Disk Preventive Maintenance

Disk P.M. from Digital Pathways, Inc., performs preventive maintenance on hard and floppy disks for IBM PCs and compatibles. Disk P.M. diagnoses problems, automatically condenses hard or floppy disks, restores order, rebuilds damaged directories, recovers damaged files, locks out faulty areas, and copies system information to disks that refuse to boot. Retail price is $49.95 from Digital Pathways, Inc., 1060 East Meadow Circle, Palo Alto, CA 94303. **Reader Service No.121.**

Public Domain Help Function

This software package provides full **on-screen help** for all PCDOS version 2.0 and 2.1 commands. The actual help text may be altered by the user. Copies are available for $10.00 from Chris Bailey, P.O. Box 332, Peterborough, Ontario, Canada, K9J 6Z3. **Reader Service No. 125.**



COMPUTEREYES is a low-cost video acquisition system for the Commodore 64 series. The product includes machine language image capture routines and image save-to-disk capability. COMPUTEREYES is the Commodore version of Digital Vision's video acquisition system for the Apple II series. The complete system, including video camera, is priced at $349.95 from Digital Vision, Inc., 14 Oak Street Suite 2, Needham, MA 02192 (617) 444-9040. **Reader Service No. 113.**

TRS-80 Screen Dump

A high-resolution **screen dump utility for TRS-80** Models I, II, III, and 4 allows users to dump the contents of a video screen, text and graphics, to an Epson or Gemini printer. The price is $19.95, available from Softbyte Computing, Box 217, Wallingford, CT 06492 (203) 239-6923. **Reader Service No. 123.**

Tools for Turbo Pascal

Programming Tool Kit by Paragon Courseware is a set of utilities for Turbo Pascal programmers. The kit includes a Window Package, a Function Evaluating Package, and a System Information Package. The Graphics Package has several procedures that draw various shapes such as triangles, parallelograms, other polygons, circles, and ellipses. The Tool Kit retails for $49.95 from Paragon Courseware, 4954 Sun Valley Road, Del Mar, CA 92014 (619) 481-1477. **Reader Service No. 127.**

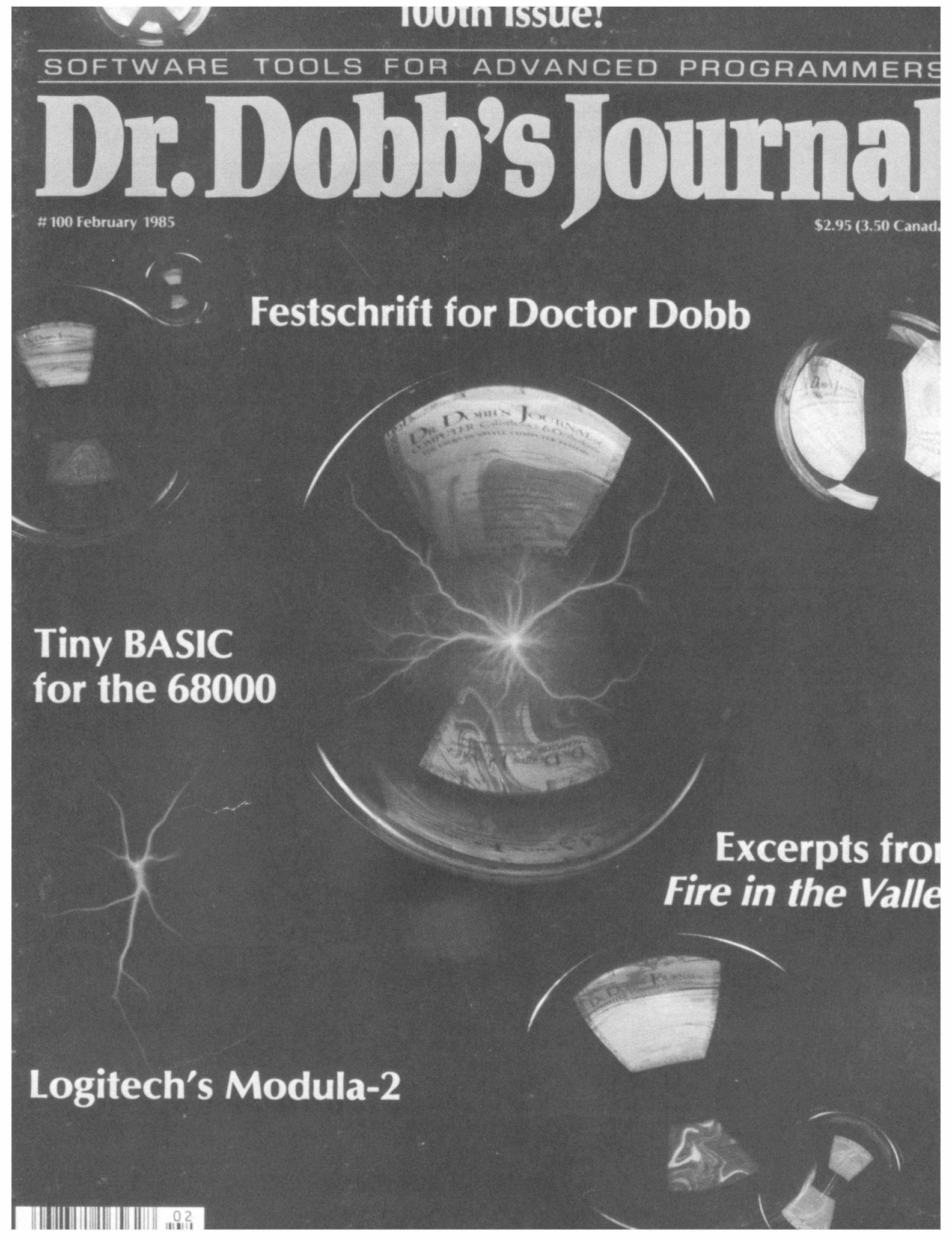
Forth Authors

The Forth Interest Group announces the formation of its **Author Recognition Program**, which offers free FIG membership to the author of any Forth-related article published in a periodical. FIG hot line: (415) 962-8653.

The **1985 Rochester Forth Conference** will be held June 12 – 15 at the University of Rochester, New York. The focus of the Conference will be on Software Engineering and Software Management. Papers on the following topics are welcomed: 1) Software Engineering and Software Management Practices; 2) Forth Applications; and 3) Forth Technology. Submit a 200-word abstract by March 30 to Lawrence P. Forsely, Laboratory for Laser Energetics, 250 East River Road, Rochester, NY 14623 (716) 235-0168.

DDJ

Readcr Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 198.**


100th Issue!
SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal
100 February 1985
$2.95 (3.50 Canada
Festschrift for Doctor Dobb
Tiny BASIC
for the 68000
Excerpts fro
Fire in the Valle
Logitech's Modula-2

Dr. Dobb's Journal

Editorial

Editor-in-Chief *Michael Swaine*
Managing Editor *Randy Sutherland*
Assistant Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*
Editorial Intern *Mark Johnson*

Production

Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover *Tom Upton*

Advertising

Advertising Director *Stephen Friedman*
Advertising Sales *Walter Andrzejewski, Shawn Horst, Beth Dudas*
Advertising Coordinators *Alison Milne, Lisa Boudreau*

Circulation

Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.

Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

February 1985
Volume 10, Issue 2

CONTENTS

In This Issue

For this 100th issue of *DDJ,* Mike Swaine interviewed the men whose names became the Doctor's: Dennis Allison and Bob Albrecht. Dennis is in "Software Designer" and Bob is in "Festschrift." To meet Bob and Dennis is to understand why the "temporary" project they started almost a decade ago is still gathering momentum. Bob and Dennis are visionary, and powerful. Suzanne Rodriguez' contribution to the "Festchrift" provides an interesting glimpse of the way things were at *People's Computer Company* under Bob and Dennis' leadership. Excerpts from *Fire in the Valley* provide a wider context for what was happening. We couldn't resist dropping in Gordon Brandly's "Tiny BASIC for the 68000" to link our roots with the state of the art.

The inclusion of Charles Burton's "An Enhanced ADFGVX Cipher System" reflects a philosophy that has continued from an early *People's Computer Company* poster: "Use computers for people, not against them." The power to encrypt one's communications, once the domain of high government and the military, has now been placed in the hands of the people. Since 1976, we have enjoyed taking what is secret, hidden, expensive, protected, or otherwise inaccessible, and placing it in the public domain.

This Months Cover

This months cover was created by Tom Upton. The photograph shows bubbles with sparks reflecting and old *DDJ* cover.

Next Month

The March issue will announce the winner(s) of the AI programming competition. The issue is shaping up to be quite Prolog oriented. We would really like to work up a public domain Prolog for micros. Contributions? In the "Realizable Fantasies" column next month you will read of a proposed project to write a complete public domain Unix, called GNU (acronym for "GNU's Not Unix"). In the same vein, the author of the grep.c program that we published in October 1984, Allen Holub, will pick up where Anthony Skjellum left off and become our new C columnist. We are pleased to announce that the C column will now run every month.

This Month's Referees

Dave Cortesi, Resident Intern

John Taber, IBM

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

Festschrift for Doctor Dobb — 26
Contributions by Suzanne Rodriguez, Tom Pittman and Bob Albrecht
Some old friends flame about the past, present and future in honor of the Doctor's 100th issue **(Reader Ballot No. 192)**

Fire in the Valley — 32
by Michael Swaine and Paul Freiberger
The Homebrew Computer Club, Processor Technology, the S100 bus meetings **(Reader Ballot No. 193)**

Tiny BASIC for the 68000 — 42
by Gordon Brandly
Li Chen Wang's original Palo Alto Tiny BASIC, now in 68000 code, with installation instructions and a bulletin board number that carries the code **(Reader Ballot No. 194)**

An Enhanced ADFGVX Cipher System — 48
by C.E. Burton
An enhanced version of the ADFGVX cipher system, written in C, wth a brief history of the original German system **(Reader Ballot No. 195)**

More dBASE Tips and Techniques — 71
by Gene Head
A program FLIPIT that changes the command file extension .CMD to .PRG and vice versa **(Reader Ballot No. 199)**

COLUMNS

Dr. Dobb's Clinic — 16
by D.E. Cortesi
Throughput from various perspectives **(Reader Ballot No. 190)**

C/Unix Programmer's Notebook — 22
by Anthony Skjellum
C style, comments on printf() **(Reader Ballot No. 191)**

Software Designer — 118
by Michael Swaine
Running light in 1985 **(Reader Ballot No. 196)**

CP/M Exchange — 96
by Bob Blum
Buffered disk I/O **(Reader Ballot No. 197)**

DEPARTMENTS

Editorial 6

Letters 8

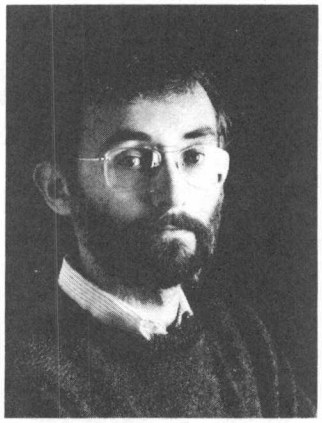
Reviews 74 Logitech's Modula-2

Books 120

Of Interest 124 **(Reader Ballot No. 198)**

Advertiser Index 128

EDITORIAL



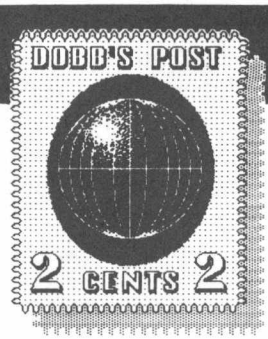
Before they put me on the payroll here, I was a loyal reader of *DDJ*. This is my personal thank-you to some of the people who created *DDJ* over the past hundred issues. I've left out many people through oversight, but here are well over a hundred.

Thank you,

Phyllis Adams, Bob Albrecht, Dennis Allison, Robin Allison, Walter Andrzejewski, Julie Anton, Jean Aring, John Arnold,
Rick Bakalinsky, Stephanie Barber, Jeannie Barroga, Sam Bassett, Anatta Blackmarr, Beatrice Blatteis, Robert Blum, Christine Botelho, Lisa Boudreau, Kathleen Boyd, Sally Brenton, Bill Bruneau,
Ron Cain, Dave Caulkins, Ward Christensen, Maureen Christine, Peter Clark, Dave Cortesi, Gavin Cullen, Carole Cullenbine,
Leah Dansby, Jim Day, Frank DeRose, Shelley Rae Doeden, Steve Dompier, Beth Dudas, Ray Duncan,
Gordon Eubanks,
Paula Fairchild, Fred Fehlau, Lee Felsenstein, Eugene Fisher, Laird Foshay, Stephen Friedman,
Mike Gabrielson, Bill Gale, Gary Gaugler, Mag Glick, Greg Gordon, H.T. Gordon, F.J. Greeb, Richard Grigonis, Ed Gueble,
Aleeca Hamilton, Craig Harper, David C. Harris, Kim Harris, Hank Harrison, Gene Head, Michael Heatherman, Matthew Heiler, Jim Hendrix, Andy Hertzfeld, Nancy Heubach, Alice Hinton, Susan Hinton, Tita Hinton, Willard Holden, Allen Holub, Shawn Horst, Alana Hunter,
Meredith Ittner,
Bob Jacobsen, John S. James, Mark Johnson, Do-While Jones,
Laura Kares, Maria Kent, Gary Kildall, Art Kleiner, Polly Koch, Cynthia Kosina, Craig LaGrow, Jane Laidley, Carl Landeau, Fritzi Lareau, J.L. Lawrence, Michael Madaj, Rosehips Malloy, Linda Marohn, Cindy Martin, Tom Martin, Terri Marty, W.D. Maurer, Dwight McCabe, Al McCahon, Dennis McGhie, Jane McKean, Lorraine McLaughlin, Mary Jo McPhee, Laura Mendell, Ann Merchenberger, Doug Millison, Alison Milne, Ann Miya, Michelle Moonstone, Carl Moser, Erik Mueller, Andrea Nasher,
Theresa O'Rourke, Marlin Ouverson,
Sahnta Pannuti, Ronald Parsons, Janet Payne, Detta Penna, Giuliana Peter, Tom Pittman, Terri Pond, Beverly Pyle,
Bill Ragsdale, Jef Raskin, Ed Ream, Pete Roberts, Beverly Robinson, Suzanne Rodriguez, Betsy Roeth, Gloria Romanoff, Dan Rosset, Barbara Ruzgerian, Barbara Rymsa,
Jonathan Sachs, Kent Safford, Carol Sevilla, Jane Sharninghouse, Anthony Skjellum, Maureen Snee, John Starkweather, Susan Strange, Robert Suding, Randy Sutherland,
Ari Taylor, Frank Trujillo, Marianne Tryens, Margaret Tsai,
Lichen Wang, Jim Warren, Judith Wasserman, Sara Werry, Ann West, Clifford West, Charles Wetherell, Dick Whipple, Michael Wiesenberg, Renny Wiggins, Tom Williams, Steve Willoughby, Marvin Wizenread, Donna Lee Wood, Steve Wozniak,
Greg Yob
and Michael Zick.

Michael Swaine

Michael Swaine



Minimax Exchange Algorithm

Dear Sir:
I was very interested in the Minimax algorithm presented by Steven A. Ruzinsky in *DDJ* No. 93 (July 1984). He presented several approximations that were better than the generally published series. Seeing this, I was inspired to program a form of the exchange algorithm and compare my results with his. Given the work that obviously went into his approach, I was surprised that even his results could be improved upon. In particular, for *sin (pi*x/2)* approximated by a 5-term series, Ruzinsky gave a maximum relative error of 5.31748E-09. My exchange algorithm gave a maximum relative error of 5.31399E-09. While this difference is so small as to be almost trivial, I believe it points up a logical error in Ruzinsky's approach.

To understand this error one must first realize that for an L_∞ approximation with N parameters, the approximation is guaranteed to reach its maximum error at least N + 1 times. Ruzinsky's algorithm tries to find these N + 1 or more places by giving ever-increasing weights to predetermined points near the actual maximum error positions, but, of course, these points are virtually never in the right positions for a true L_∞ approximation.

The exchange method, on the other hand, is directed toward finding these N + 1 points and building an approximation around them. When it works, it works best, but it appears to be far more finicky than Ruzinsky's algorithm.

In fact, Ruzinsky actually underestimates the maximum error of his approximations because the maximum error of his approximations does not occur at his lattice points, but somewhere in between.

In light of all this I propose the following modification of Ruzinsky's algorithm: after the Speedup step, which halves the number of lattice points, add new lattice points midway between each of the remaining points. This has the effect of increasing the lattice density near the maximum error and, I expect, will give the correct L_∞ approximation.

A few technical notes: GW BASIC on the Eagle PC that I used, like most Microsoft BASIC interpreters, does not support double precision transcendentals, so my first order of business was implementing *sin(p*x/2)* in double precision. This is not too bad if you don't care much about speed. Also the coefficients I got for Figure 8, page 94, are:

A(0) = 1.570796318447697
A(1) = −0.6459637105998668
A(2) = 0.0796896789479709
A(3) = −4.67376661266352E−03
A(4) = 1.514851308552791E−04

I also tried *arctan(x)* and again found a small improvement on Ruzinsky's result.

Finally, I would be interested in how Mr. Ruzinsky linearized his various functions. Perhaps the contest he mentioned (for a free one-year subscription to *DDJ*) could be ended.

Sincerely,
Allen E. Tracht
12469 Cedar Rd., #11
Cleveland Heights, OH
44106

Forth Standards

Dear Editor:
As a Forth vendor, I must take issue with your September 1984 article by Ray Duncan and Martin Tracy, "The FVG Standard Floating-Point Extension." I would like to point out that the document discussed in the article was based on the input of about eight vendors (including the authors). Most of these vendors do not produce floating-point applications systems. In fact, those of us who use the pre-existing Mountain View Press Floating-Point Standard were quite shocked to see this material in print. My own company has a mature floating-point system with utilities, matrix math, and FFTs, which has been available for over two years and was purchased by Martin Tracy of Micromotion about a year ago. This new "standard" bears little resemblance to the working standard that has come out of the experience of several years of real applications programming with floating-point Forth by those at ForthKit, Redshift Limited, and Mountain View Press.

In a recent editorial [October 1984] you commented that the FVG Standard showed that Forth was maturing and was no longer a language in flux. This conclusion is certainly inaccurate.

Micromotion and Creative Solutions (among others) were invited over a year ago to participate in a formal standards effort sponsored by Mountain View Press. They declined to participate. Through a great deal of work, those companies that did contribute developed a solid, mature standard. The programming systems based on our standard have received a great deal of use in laboratories and schools around the world; we consider them to have passed the test of time.

Something also needs to be said about the use of separate floating-point stacks. The FVG document claims that applications adhering to the names and functions of the "standard" will be transportable among the various floating-point implementations. Unfortunately, this is not the case. The paragraph on floating-point stacks says that the document does not concern itself with the issue of separate stacks. The

document states that on systems with a separate floating-point stack the source or destination of a real number argument is the floating-point stack. These statements seem to ignore the fact that programs written for a separate stack will not transport to a system that uses the parameter stack for floating-point numbers and vice versa.

A separate floating-point stack is necessary for efficient coding of complex algorithms. I used a parameter stack-based system for a year before switching to a separate floating-point stack in 1981, and I was amazed to see all my code simplify. But converting from one system to the other is a tedious process and involves rethinking and rewriting all the code—this can hardly be called transportable! The Mountain View Press Standard specifies a separate stack to aid transportability, although it is our feeling that floating-point applications cannot be absolutely portable, due to the differences in hardware and software number packages.

I feel that Ray Duncan has performed a disservice to the Forth community by publishing this confusing document and further clouding the floating-point standards issue.

Sincerely,
Charlie Springer
Redshift Limited
417 Forest Ave.
Palo Alto, CA 94301

Forth Compiler Changes

Dear Editor,
In my article "A Forth Native-Code Cross Compiler for the MC68000" (September 84) there are a couple of errors that should be corrected. I have also done some additional work that should make the compiler more portable.

On page 71 (second column, first paragraph) the sentence:

> Each time 2* is called, the code implementing it is *sorted* in the host dictionary, and the dictionary pointer is updated.

should read:

> Each time 2* is called, the code implementing it is *stored* in the host dictionary, and the dictionary pointer is updated.

On page 75 under "Final Comments" the sentence:

> This word should reset all of the compiler variables in screen 9 to their original *style* and generate an appropriate header to permit the operating system to load and execute the module.

should read:

> This word should reset all of the compiler variables in screen 9 to their original *values* and generate an appropriate header to permit the operating system to load and execute the module.

I have used the compiler on a Forth-83 system (F83 model by Laxen and Perry) with very few changes. The following is a list of the changes that need to be made:

1. Change the variable initialization in screens 9 and 10 to conform to the Forth-83 standard.
2. Change the definition of the word HIGH-BYTE on screen 12 to the following.
   : HIGH-BYTE FLIP ; (FLIP is not Forth-83 standard, for F83 only.)
3. Change the word <BUILDS on screens 14 and 17 to CREATE or use the definition of <BUILDS given in screen 44 line 10.
4. Delete the word HERE from the definition of BYTES on screen 21.
5. Delete the word ;S from screens 24, 33, and 43. (Not all systems support this word and it is not required.)
6. Change the phrase A BYTES to 0A BYTES in the following places (the word A is used in the F83 model to select the alternate screen): screen 35 line 8, screen 36 line 2, screen 37 line 6, and screen 42 line 7.
7. Change all of the uses of the word ENDIF to THEN or use the definition of ENDIF given in screen 44 line 2.

The following is a better implementation of the word BYTES that appears in screen 21. This word now saves the current base and always uses HEX notation. Note that this word now uses the Forth-83 standard definition of WORD, so older systems will have to add the word HERE following WORD just as in the original.

```
:BYTES
    BASE @ >R HEX
    0 DO BL WORD NUMBER DROP
        C, LOOP
    R> BASE ! ;
```

The following is one possible method of implementing the word M68OUT in screen 18. This version saves the output code in an unused portion of memory. This block of memory can then be saved on disk or transferred in some other manner to the 68000 target machine. If you happen to be using the compiler on a 68000 system, you could execute the code directly from the host system.

```
M68K DEFINITIONS

( Pointer to current location in output
    buffer area of memory )
VARIABLE $M68OUT

( Store byte in output buffer area of
    memory )
: M68OUT ( b -- )
   $M68OUT @ C! 1 $M68OUT +! ;
```

The following code should be modified according to your needs and should be loaded after the rest of the cross compiler code.

```
( Start of output buffer space )
FORTH DECIMAL
HERE 10000 +
M68K DEFINITIONS
CONSTANT ($M68OUT)

( Initialize output buffer pointer )
($M68OUT) $M68OUT FORTH !
```

That sums up the changes I have made to the compiler. I am using the 68000 version of F83, which has Michael Perry's assembler built in. The assembler works just fine in conjunction with my cross compiler so people who want an assembler built into the compiler should take a look at the September 1983 issue of *Dr. Dobb's Journal.*

Sincerely,
Raymond L. Buvel
Box 3071

Moscow, ID 83843

Debugging Tool

To the Editors of *Dr. Dobb's Journal*,
A few weeks ago, I stumbled onto a tool that has greatly improved my debugging time. I primarily use Turbo Pascal and Forth in software development, and when a program crashes, I spend much of my time figuring out where it crashed, and why. I have evolved the code in the accompanying source listing [see Listing, page 12] to solve these problems. The necessary code to make an error trapping route is included.

The basic data structure is a stack of strings called *place*. The stack pointer is named *place_ptr*, and the stack depth is called *place_depth*, which is a constant.

To initialize the stack, call the procedure *starting*. At every level of invocation, such as the beginning of a procedure, function, or loop, call *entering* with the name of the place you are tracking. Upon exiting the procedure, function, or loop, invoke *leaving*. At the end of the section of the program you are tracking, call *ending*.

Let's consider the error handler. This procedure is passed a string of any acceptable length by its calling procedure, which is assumed to be an error message. After displaying the error message, which may be optionally printed on a printer, the place stack can be dumped either to the screen, the printer, or both. This indicates to any desired level of precision where in the program things went wrong and the invocation path as to how the program got there. It is then possible to type control-C and abort the program.

There are two innate errors that these procedures can trigger: stack underflow and stack overflow. The first problem is caused by invoking *leaving* more places than *entering*. This problem is detected by the procedure *leaving*, and the place stack as is will provide no useful information, because the place stack has been depleted prematurely. I find that the most common cause for this is placing *entering before* the loop starts and *leaving in* the loop. For example,

```
begin
   entering('loop_test');
   for i:=1 to 10 do
      begin
         do_some_stuff;
         leaving('loop_test');
      end;
end;
```

is an example of this kind of mistake. The *place* array still has the name *loop_test* in its first cell. By setting the *place_ptr* to *place_depth*, the dumping of the place stack will provide vital clues as to where the underflow occurred.

The second problem, stack overflow, is essentially the reverse situation: there were more invocations of *entering* than *leaving*. This problem can be detected two different ways, either while attempting to use *entering* or when *ending* has been invoked. In either case, the place stack dump will provide useful insights into what has been going on. When *ending* is encountered, the place stack should be empty; it if isn't, then there have been more invocations of *entering* than *leaving*, which indicates some fundamental failure of structuring in the program or in the placement of the *entering* and *leaving* invocations.

Earle Jennings
Director of R&D
Incremental Computing Corp.
P.O. Box 2875
Santa Clara, CA 95055

DDJ

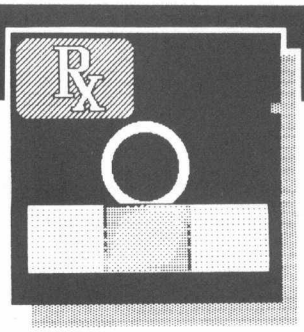
(Listing begins on next page)

```
{*************************************************
**
**       Diagnostic tools in TURBO PASCAL
**         by Earle Jennings 10/21/84
**
*************************************************}
CONST
  name_length=24;  { length of names stored on the place stack}
  place_depth=50;  { the depth of the place stack }
TYPE
  name=string[name_length];
    { name is a string of name_length chars max}
  anystring=string[255];
VAR
  place_ptr:integer;
  place:array[0 .. place_depth] of name;
    { This is the place stack that provides our travel log}
{*************************************************
**  right_pad( Its_name,name_length )
*************************************************}
function right_pad(Its_name:anystring;name_length:integer ):anystring;
VAR i:integer;
  temp:anystring;
begin
  temp:=Its_name;
  if length(Its_name)<name_length then
  for i:=length(Its_name)+1 to name_length do temp:=concat(temp,' ');
  right_pad:=temp;
end; { of right_pad}
{*************************************************
**  dump_place_stack(target:text);       10/20/84
*************************************************}
procedure dump_place_stack(VAR target:text);
VAR i:integer;
begin
  writeln(target,'our present location is:');
  if place_ptr>-1 then
    for i:=place_ptr downto 0 do
    begin
      write(target,right_pad(place[i],name_length),',');
      if ( ( place_ptr-i) mod 3 ) = 0 then writeln(target);
    end;
end;  { of dump_place_stack }
{*************************************************
**  dump_places                          10/20/84
*************************************************}
procedure dump_places;
VAR it:integer;
begin
  writeln('dump place stack to: 1-crt, 2-lp, 3-both, else skip');
  readln(it);
  case it of
    1: dump_place_stack(con);
    2: dump_place_stack(lst);
    3: begin
        dump_place_stack(con);
        dump_place_stack(lst);
      end;
  else
  end;
```

```
end;  { of dump_places }
{*************************************************
**  error(message)                          11/3/84
*************************************************}
procedure error(message:anystring);
VAR y_or_n:char;
begin
  writeln('*****************************************');
  writeln(message);
  writeln('Do you wish error message output to printer?');
  readln(y_or_n);
  y_or_n:=upcase(y_or_n);
  if y_or_n = 'Y' then writeln(lst,message);
  dump_places;
  writeln('--->>>>  hit Ctrl-C to kill program, else return to continue');
  readln(y_or_n);
end;  { of error }
{*************************************************
**  enterring(it)                           11/3/84
*************************************************}
procedure enterring(it:name);
begin
  if place_ptr<place_depth then
  begin
    place_ptr:=place_ptr+1;
    place[place_ptr]:=it;
  end
  else error('place stack overflow upon enterring "'+it+'"');
end;  { of enterring }
{*************************************************
**  leaving                                 11/3/84
*************************************************}
procedure leaving;
begin
  if place_ptr<0 then
  begin
    place_ptr:=place_depth;
    error('place stack underflow');
  end
  else place_ptr:=place_ptr-1;
end;  { of leaving}
{*************************************************
**  starting                                10/20/84
*************************************************}
procedure starting;
begin
  place_ptr:=-1;
  enterring('main');
end;  { of starting }
{*************************************************
**  ending                                  11/3/84
*************************************************}
procedure ending;

begin
  leaving;
  if place_ptr<>-1 then
  begin
    error('place stack imbalance');
  end;
end;  { of ending }
```

End Listing

DR. DOBB'S CLINIC



D.E. Cortesi

A Random Choice

The generators of pseudo-random numbers are algorithms that lie on the boundary between computer science and recreational mathematics. They're as little tainted by real-world considerations as we could want, but they are endlessly measurable, improvable, discussable.

These thoughts were inspired by a recent issue of *PC World*, in which one Gary Andrew reported an unusually simple algorithm for generating random integers. Andrew gives credit to Drs. Wilson Talley and Nicholas Metropolis for the design of the algorithm. He claims it stands up well to a variety of tests, a claim we've not attempted to verify.

Andrew gives the algorithm in BASIC; we recoded it in C by way of understanding it. Our interpretation appears in the Listing (page 20). It looks to us as if a version that produced 16-bit numbers would be most effectively coded in assembly language. Then some tricks could be played with carry flags, etc., to avoid having to carry long-integer intermediate results. Anyone who codes one up, or who does any statistical tests of this generator, be sure to write.

Case-name	Machine	CPU	DOS	Drives and other factors
Cage	NorthStar	Z80 4MHZ	CP/M 2.2a	5-inch
Castle-1	NorthStar	Z80 4MHZ	N*DOS	5-inch
Castle-2	NorthStar	Z80 4MHZ	CP/M 2.2a	5-inch
Chamberlin-1	MTU-140	6502 1MHZ	CODOS	8-inch, 5K buffer
Chamberlin-2	MTU-140	6502 1MHZ	CODOS	8-inch, 32K buffer
Cortesi-1	S-100	Z80 4MHZ	CP/M 3.0	8-inch thinline
Cortesi-2	TRS-80 4P	Z80 4MHZ	CP/M 3.0	5-inch thinline
Cortesi-3	Apple-ALS	Z80 6MHZ	CP/M 3.0	Apple 5-inch
Floyd-1	PC	8088	PCDOS 2.0	5-inch, verify off
Floyd-2	PC	8088	PCDOS 2.0	5-inch, verify on
Floyd-3	PC	8088	PCDOS 2.0	5-inch to Mdisk
Floyd-4	PC-XT	8088	PCDOS 2.0	5-inch, verify off
Floyd-5	PC-XT	8088	PCDOS 2.0	5-inch, verify on
Floyd-6	PC-XT	8088	PCDOS 2.0	5-inch to Mdisk
Gunn-1	NorthStar	Z80 4mhz	CP/M 2.2	5-inch Tandon 100-2
Gunn-2	NorthStar	Z80 4mhz	CP/M 3.0	5-inch Tandon 100-2
Gunn-3	Zilog MCZ-1	Z80 2.5mhz	CP/M 2.2	8-inch Shugart 800-2
Hoffman-1	S-100	Z80 4MHZ	CP/M 2.2	8-inch single-density
Hoffman-2	S-100	Z80 4MHZ	CP/M 2.2	8-inch double-density
Hole-1	CompuPro	8088 8MHZ	MP/M 8-16	8-inch
Hole-2	CompuPro	8088 8MHZ	MP/M 8-16	21MB hard disk
Hole-3	CompuPro	8088 8MHZ	MP/M 8-16	Mdrive-H
Johnson	Morrow MD-3	Z80 4MHZ	CP/M 2.2	5-inch
Prince	n.a.	Z80 2MHZ	CP/M 2.2	8-inch single density
Sabin-1	Altos 5	Z80 4MHZ	CP/M 2.2	n.a.
Sabin-2	TRS-80 III	Z80 2MHZ	CP/M 2.2	5-inch
Schemm-1	Digital Grp	n.a.	CP/M 2.2?	8-inch
Schemm-2	Digital Grp	n.a.	Disk-MON	8-inch
Schemm-3	IMS	n.a.	CP/M 2.2?	8-inch
Schemm-4	NorthStar	Z80 4MHZ	CP/M 2.2	5-inch
Schemm-Z1	Z-110	8086	ZDOS	5-inch
Schemm-Z2	Z-110	8086	CP/M-86	5-inch
Swearingen	Big Board	Z80 4MHZ	CP/M 2.2	8-inch

Table 1
The cases—reporters and configurations—analyzed in the throughput study.

Skies Still Falling, Ho-Hum

A couple of readers had comments on our November review of *Computers In Crisis*. Charles Martin of Durham, NC, comments: "The same issue—ohmigod what will happen to our Julian-date routines on 1 Jan 2000—was covered rather completely (read: thrashed half to death) in *ComputerWorld* in about 1975. That someone would write a book about it now is surprising; that Petrocelli would publish it only proves you don't have to be smart to be an editor . . . ."

Terry Jackson of Lombard, IL, with tongue firmly implanted in cheek, wrote: "The issue of date storage should be addressed promptly." Why? Because "it is well known to anyone associated with large DP operations that 16 years is not enough for them to solve this problem." But he has an answer: "The obvious solution is to get the whole world to agree to write the year in hexadecimal. That defers the crisis until 19FF (2155 old style), thus allowing time to rewrite the software, and *maybe* enough time to debug."

Throughputting

In the July Clinic we wrote about the idea of "throughput" and suggested that the throughput of a system's fundamental file-copy utility would be a useful capsule measurement of the performance of that configuration of software and hardware. We measured the time it took CP/M Plus and PIP to copy files of different sizes on three different hardware configurations. The numbers seemed to fit a linear model pretty well.

A dozen readers took the time to measure their own systems' copy times and report them. Their names, and the

essential details of their hardware and software configurations, are listed in Table 1 (page 16). Most cases describe CP/M systems, but we received a few reports on NorthStar DOS, CODOS (what?), and PCDOS. No measurements of 68K-based systems, Apple DOSses, Triss-DOSses, Commodore 64s, or Ataris were reported. Where is everybody?

Table 2 (below right) lists the copy times reported with each case in Table 1. As shown, the numbers appear to be precise to the nearest tenth of a second, but this was *not* the case. Most times were taken with hand-operated watches, and so are accurate to the nearest 0.2 second at best (it is not difficult to get repeatable times to this accuracy using a hand-operated watch, but it is difficult to be any better). Some were reported only to the nearest sejod and were entered with a fraction digit of zero. And there may be a few transcription errors, since some numbers were submitted in minuscule handwriting on reader service cards!

Furthermore, we can't be sure that the measurements were taken in a consistent way. They were supposed to reflect the time to copy a file from one disk to another, exclusive of any time needed to load a copy program (the built-in COPY command of MSDOS; multiple-copy mode of CP/M PIP). There was an unstated (but obvious) assumption that all buffered data should be forced to disk in the measured time. We suspect that some sort of buffering phenomenon is affecting the more remarkable ZDOS and PCDOS measurements.

Throughput Models

Several readers made thoughtful comments on the numbers. Bruce Johnson of Lakewood, CO, had this to say: "The throughput you showed in the table is really a combination of two factors. One is the speed at which PIP transfers data—which I would call throughput—and the other is a theoretical minimum time to transfer a zero-length file, which I would call the setup time. The setup time corresponds to the Y-intercept of each line on the graph, while the throughput corresponds to the reciprocal of the slope of the line. I would suggest as a measure of throughput for linear utility programs:

$$T = 60/(T64 - T4)$$

where T = throughput, T64 = seconds to transfer a 64K file, and T4 = seconds to transfer a 4K file. This formula will yield a useful measure of throughput over the range of file sizes of most interest with only two measurements."

Johnson's ingenious formula does just what he claims; it produces numbers that are in fair agreement with a more elaborate model to be discussed below.

William Hole, of Barrington, RI, made the same point, that the timings we published and those he took on his own system were a good fit to a linear model with a fixed startup time. He put his numbers through a linear regression test to check the model and got good agreement.

CP/M Measurements

Because Bill Hole seemed to know his way around the fundamental tools of statistics, we dumped the entire contents of Table 2 on him and asked him to test them for conformity to the following model:

$$T = s + e * \text{floor}(n/16) + n / t$$

where T is the reported time in seconds, s is a setup time, e is the time required by CP/M for opening the subsequent "extent" of a file, n is the number of kilobytes that are transferred, and t is the number of kilobytes that can be transferred in one second, that is, the actual "throughput" of the system.

He performed this analysis for us (and many thanks for the time it must have taken) but the results are somewhat equivocal. Table 3a (page 18) reports the analysis for the CP/M-compatible systems. The first group of three columns presents the computed results for the setup time. The estimat-

	1k	2k	4k	8k	16k	24k	32k	48k	64k	128k
Cage	3.8	4.8	6.6	10.5	18.8	-.-	36.2	53.2	72.0	151.1
Castle-1	5.5	5.8	7.9	11.5	20.0	-.-	40.0	-.-	78.0	
Castle-2	6.0	6.4	7.2	9.0	16.6	-.-	29.4	-.-	56.7	
Chamberlin-1	2.4	2.2	3.1	3.8	4.7	6.4	7.6	10.6	13.6	23.4
Chamberlin-2	2.5	2.5	2.9	3.6	4.6	5.6	6.5	6.5	10.4	18.5
Cortesi-1	3.4	3.4	3.4	4.8	5.8	8.2	9.6	13.2	18.0	
Cortesi-2	3.2	4.0	5.0	7.2	11.0	17.4	20.6	30.0	39.5	
Cortesi-3	4.0	4.2	5.0	6.4	9.2	13.8	17.4	23.6	32.5	
Floyd-1	2.3	2.3	2.8	3.4	4.1	5.1	5.8	7.3	12.5	
Floyd-2	3.5	3.5	4.2	4.8	5.9	7.4	8.4	10.6	17.5	
Floyd-3	1.3	1.3	1.7	2.0	2.4	3.1	3.3	4.1	6.3	
Floyd-4	4.3	4.6	4.7	5.2	5.3	5.3	5.4	6.2	7.5	
Floyd-5	4.8	4.9	5.0	5.6	5.7	5.9	6.4	7.3	8.7	
Floyd-6	3.1	3.5	3.5	3.6	3.9	3.9	4.0	4.5	5.4	
Gunn-1	2.6	2.6	2.9	3.8	5.6	7.6	9.2	14.2	18.6	
Gunn-2	3.5	3.6	4.1	4.7	6.7	8.0	9.7	14.1	17.9	
Gunn-3	2.2	2.4	3.0	4.2	6.0	8.9	11.2	17.3	22.6	
Hoffman-1	4.0	5.0	6.0	9.0	16.0	25.0	31.0	42.0	55.0	
Hoffman-2	-.-	2.5	3.5	4.0	6.0	10.0	13.0	16.5	22.0	
Hole-1	2.9	3.1	3.6	3.9	5.5	7.7	9.8	13.5	17.0	33.5
Hole-2	2.1	2.2	2.4	3.5	6.4	6.9	8.7	13.3	16.4	30.8
Hole-3	0.6	0.6	0.7	0.9	1.3	1.7	2.1	2.9	3.8	7.0
Johnson	3.6	3.7	4.7	6.0	9.6	12.2	17.1	23.5	31.7	
Prince	-.-	2.4	2.6	3.4	5.6	-.-	11.5	-.-	22.7	
Sabin-1	2.5	3.5	5.0	7.0	12.5	17.0	23.0	48.0	44.0	
Sabin-2	2.5	3.0	4.5	6.0	9.0	11.5	17.0	22.5	31.0	
Schemm-1	-.-	4.9	5.2	5.9	8.5	-.-	13.8	-.-	24.2	
Schemm-2	-.-	2.6	2.6	3.4	3.7	-.-	5.8	-.-	-.-	
Schemm-3	-.-	7.6	8.0	9.7	13.8	-.-	23.3	-.-	42.8	
Schemm-4	-.-	6.2	7.2	7.3	9.4	-.-	14.8	-.-	22.9	
Schemm-Z1	-.-	1.1	0.9	1.1	1.3	-.-	1.6	-.-	2.9	
Schemm-Z2	-.-	3.1	3.1	3.2	4.5	-.-	6.8	-.-	12.0	
Swearingen	3.4	3.6	4.2	5.8	8.2	-.-	15.9	20.9	28.3	

Table 2

Reported times, in seconds, to transfer files of different sizes. These data were input to the analysis performed by W. T. Hole. The two italicized entries were mistranscriptions (see text).

ed setup time appears in the first column. The second column (headed "s/e") gives the standard error in seconds; that is, the estimated setup time, plus or minus this value, should encompass 68% of the observations.

The third column, headed "prob," is a confidence rating. It is the probability that the setup value should equal zero. If it exceeds 0.05, the setup time is probably not a reasonable part of the model. The eye is drawn at once to the third line, case Sabin-1, the only case that appears not to conform to a model that includes a fixed setup time. Why is this? Quite possibly because, as indicated in Table 2 by an italicized entry, one of Joe Sabin's numbers was mistranscribed (the 48K copy time should be 34.0 seconds). There wasn't time to rerun the analysis, so we left this case in place as an interesting exception. The fact that the only case not to fit the model contained bad data seems to strengthen the case for the model.

Case	--- Startup ----			--- Extent -----			--- Throughput -----		
	sec.	s/e	prob	sec.	s/e	prob	kb/sec	s/e	prob
Cage	3.13	0.92	0.014	5.17	1.58	0.017	1.15	0.14	0.0001
Hoffman-1	3.32	0.75	0.004	-.07	1.73	0.967	1.21	0.14	0.0001
Sabin-1	1.26	3.23	0.710	-.72	7.40	0.924	1.27	1.10	0.0747
Castle-2	4.67	0.55	0.001	2.36	1.23	0.126	1.42	0.14	0.0003
Cortesi-2	3.06	0.24	0.000	1.50	0.54	0.033	1.99	0.11	0.0001
Schemm-3	6.36	0.24	0.000	2.47	0.48	0.014	2.20	0.13	0.0004
Sabin-2	2.08	0.49	0.005	-.22	1.13	0.848	2.21	0.31	0.0002
Johnson	2.76	0.38	0.000	0.43	0.88	0.638	2.36	0.27	0.0001
Cortesi-3	3.52	0.29	0.000	1.86	0.67	0.032	2.79	0.28	0.0001
Swearingen	2.93	0.34	0.000	0.93	0.77	0.280	2.86	0.35	0.0003
Hoffman-2	2.17	0.55	0.011	0.82	1.12	0.493	3.64	0.96	0.0048
Hole-2	1.52	0.26	0.000	1.72	0.47	0.165	3.73	0.41	0.0001
Schemm-1	4.14	0.22	0.000	1.00	0.44	0.106	3.74	0.35	0.0013
Gunn-3	1.88	0.19	0.000	1.41	0.44	0.019	3.90	0.37	0.0001
Prince	1.70	0.18	0.002	1.82	0.36	0.015	4.10	0.34	0.0010
Schemm-4	5.81	0.46	0.001	0.59	0.91	0.559	4.12	0.99	0.0139
Gunn-2	3.14	0.22	0.000	0.52	0.50	0.339	4.93	0.69	0.0002
Gunn-1	2.16	0.21	0.000	1.22	0.49	0.047	5.10	0.73	0.0002
Hole-1	2.68	0.17	0.000	1.19	0.31	0.006	5.75	0.63	0.0001
Cortesi-1	3.08	0.21	0.000	1.35	0.49	0.033	6.11	1.07	0.0005
Schemm-Z2	2.70	0.20	0.000	0.94	0.39	0.096	9.96	2.50	0.0157
Hole-3	0.51	0.01	0.000	0.04	0.03	0.258	20.61	0.80	0.0001

Table 3a
Analysis of the CP/M (and MP/M) cases, ranked by increasing throughput.

Case	--- Startup ----			--- Extent -----			--- Throughput -----		
	sec.	s/e	prob	sec.	s/e	prob	kb/sec	s/e	prob
Castle-1	3.96	0.32	0.000	3.24	0.72	0.010	0.99	0.03	0.0001
Chamberlin-1	2.17	0.21	0.000	-.28	0.37	0.473	5.45	0.71	0.0001
Chamberlin-2	2.50	0.44	0.000	0.42	0.78	0.607	10.03	7.92	0.0580
Schemm-2	2.43	0.21	0.007	0.64	0.62	0.409	11.80	4.38	0.0660

Table 3b
Analysis of 8-bit DOS's other than CP/M. These cases were a poor fit to the hypothesized model.

Case	--- Startup ----			--- Extent -----			--- Throughput -----		
	sec.	s/e	prob	sec.	s/e	prob	kb/sec	s/e	prob
Floyd-2	3.44	0.67	0.002	1.80	1.54	0.286	9.00	19.73	0.1959
Floyd-1	2.27	0.54	0.005	1.26	1.23	0.345	12.26	36.28	0.2291
Floyd-3	1.35	0.20	0.000	0.34	0.48	0.500	18.24	13.81	0.0594
Floyd-5	4.79	0.15	0.000	0.06	0.35	0.853	18.72	9.05	0.0222
Floyd-4	4.47	0.20	0.000	0.01	0.47	0.974	24.50	32.31	0.1303
Floyd-6	3.28	0.13	0.000	0.01	0.30	0.956	35.08	38.44	0.1043
Schemm-Z1	0.97	0.12	0.004	0.33	0.23	0.252	72.99	515.24	0.3383

Table 3c
Analysis of the few reported PC-, MS-, or Z-DOS cases. Not only the extent, but the linear model is rejected by these numbers.

The second group of three columns in Table 3a represents the estimate for factor "e," the time to open a new 16K file extent. Again the first column is an estimate of the time, the second is the standard error—read it as "plus or minus seconds"—and the third a confidence rating. You'll immediately notice that the second line exhibits a remarkably unconfident probability of 0.967. A 16K extent overhead isn't apparent in the numbers for this case (which were transcribed correctly). That's odd, because (as we can see in Table 1) this case involved CP/M single-density 8-inch disks. Other cases that disagree with the hypothesized extent factor are easier to explain, e.g., because they use a 32K extent or (in the case of the M-drive) because the extent overhead is actually near zero.

The final group of three columns presents the estimated throughput, with the same interpretation as before. A poor confidence rating here (one greater than 0.05) indicates that these data do not fit a linear model. There is one case in Table 3a for which this is definitely true and two others that are suspect.

But by and large, the linear model of throughput fits very well, so the throughput value (on which Table 3a is sorted) is probably a fairly good estimate for these CP/M-based systems. It shows a general trend of increasing speed from 5-inch drives to 8-inch to M-drives, with Hole's CompuPro M-drive the fastest. Schemm's Z-100 (8-bit CP/M, but with I/O handled by a 16-bit BIOS) produced a remarkable time for a 5-inch disk drive. We suspect this measurement was affected by some sort of internal buffering—notice that it conforms less well to the linear model than most.

Non-CP/M Throughput

Table 3b (page 18) contains the analy-

sis of the cases that involved 8-bit systems other than CP/M. Mostly these deny an "extent" hypothesis—not unreasonably, since that's a CP/M peculiarity—and half of them appear to be nonlinear. One of these is due to a transcription error: we miscopied Hal Chamberlin's 48K time as 6.5 seconds when it should have been 8.3 seconds. With that change, case Chamberlin-2 should be much closer to linearity, but time didn't permit rerunning the analysis.

Table 3c (page 18) details the analysis for the MSDOS (PCDOS, ZDOS) cases. We expected that an MSDOS copy command would be just as linear as a CP/M PIP operation. That appears not to be so, as *none* of these cases conforms to the linear model. That they don't support the "extent" factor is no surprise; there are no "extents" in MSDOS. But that they don't appear linear is most puzzling, as is the remarkable speed they suggest. (After all, no amount of software cleverness will make a 5-inch drive rotate or seek any more quickly, so why should a PC be so *very* much quicker than an 8-bit system with equivalent drives?) Can a reader suggest what might be going on? Is there a measurement procedure for MSDOS that will produce linear numbers?

The Johnson Model

We present Table 4 (below) as a final view of the data. It shows all the cases that support a linear model, ranked by what we might call their "Johnson Number"—the figure of merit produced by Bruce Johnson's formula discussed earlier. There is a fairly close correspondence, so Johnson's rule of thumb is useful *when the data is linear.* But since it presumes linearity, it should only be applied where linearity is known to exist.

DDJ

(Listing begins on page 20)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

Case	4k Time	64k Time	Johnson Number	Linear Model
Castle-1	7.9	78.0	0.86	0.99
Cage	6.6	72.0	0.92	1.15
Castle-2	7.2	56.7	1.21	1.42
Hoffman-1	6.0	55.0	1.22	1.21
Schemm-3	8.0	42.8	1.72	2.20
Cortesi-2	5.0	39.5	1.74	1.99
Cortesi-3	5.0	32.5	2.18	2.79
Johnson	4.7	31.7	2.22	2.36
Sabin-2	4.5	31.0	2.26	2.21
Swearingen	4.2	28.3	2.49	2.86
Prince	2.6	22.7	2.99	4.10
Gunn-3	3.0	22.6	3.06	3.90
Schemm-1	5.2	24.2	3.16	3.74
Hoffman-2	3.5	22.0	3.24	3.64
Gunn-1	2.9	18.6	3.82	5.10
Schemm-4	7.2	22.9	3.82	4.12
Cortesi-1	3.4	18.0	4.11	6.11
Hole-2	2.4	16.4	4.29	3.73
Gunn-2	4.1	17.9	4.35	4.93
Hole-1	3.6	17.0	4.48	5.75
Chamberlin-1	3.1	13.6	5.71	5.45
Schemm-Z2	3.1	12.0	6.74	9.96
Hole-3	0.7	3.8	19.35	20.61

Table 4
The cases that matched a linear model, ranked by Bruce Johnson's rule-of-thumb formula.

Dr. Dobb's Clinic Listing (Text begins on page 16)

```
/*
        Random integer algorithm reported by Gary Andrew to
        the "*.*" column of PC World, December 1984, with
        credit to Drs. Wilson Talley and Nicholas Metropolis.
        Recoded to C by DEC.

        This version produces 15-bit integers; for 16-bit
        unsigned integers longs must be used for intermediate
        computations.
*/
#define BITS 15 /* how many bits in numbers */
#define MAX0 1 << BITS
#define MAX (MAX0) - 1

#define private static
#define cardinal unsigned /* is this Wirth-while? */

private cardinal seedA = 32749, seedB = 32633;

int rseed(a,b)
cardinal a,b;
{ /* seed the generator, returning first number of sequence */

        a &= MAX;       /* reducing negative signed ints, */
        b &= MAX;       /* ..etc., to values <= MAX */

        a *= 2; if (a>MAX) a -= MAX;
        b *= 2; if (b>MAX) b -= MAX;

        seedA = a; seedB = b;
        return (b);
}

private cardinal rand0()
{ /* cycle the generator, return next number of sequence */

        register cardinal c;

        c = seedA + seedB;
        if (c>MAX) c -= MAX0;
        c *= 2; if (c>MAX) c -= MAX;
        seedA = seedB; seedB = c;
        return (c);
}

cardinal randi(n)
cardinal n;
{ /* return a random number in 1..n */

        register cardinal c;

        c = rand0();
        return ( 1+ (c % n) );
}
```

End Listing

C/UNIX PROGRAMMER'S NOTEBOOK



by A. Skjellum

Before beginning this month's column, I want to express my congratulations to *DDJ* on its 100th issue. (My first *DDJ* article appeared a little more than five years ago.) I hope that readers will continue to make their excellent contributions to keep the magazine going during the next few years; I include both members of the "old guard" and new readers in that wish.

This month, I include more of the reader responses received over the past few months—plus some ideas about variable formats in printf( ) functions.

A C Style Reference

Last year, we had a long-running discussion on C style. Several references were cited at that time. Since then, I've run into another paper on the subject. While the formatting is different from what I prefer, I think the paper is well worth reader attention. The article, called "A C Style Sheet," was written by Martin Minow of Digital Equipment Corp. The article lists an address, so perhaps this is the best way to acquire a copy:

Martin Minow
Digital Equipment Corp.
146 Main St., MLO 3-3/U8
Maynard, MA 01754

It includes a list of references from which the work has been abstracted.

Wish List

John M. Gamble of Batavia, Ohio, writes:

"I am very glad you printed this column [August 1984] (I ranked it number one for the month) because it inspired me to jot down my own wish list. It is only one wish long, but it is something I have longed for for some time."

Mr. Gamble's wish is for a way to control more precisely the sizes of intrinsic data types in C. For example, integers can have different sizes on different machines. He continues:

"The size of the various types (int, char, long, etc.) vary too much from machine to machine. However, I don't think that forcing a size to a type (a la Plum Hall) is the answer (besides, I hate the baby words Plum uses to define them). Instead, I think that another 'storage class specifier' (see *The C Programming Language*, p. 192) like typedef could be defined. I'm going to call it 'sizetype.' Sizetype would be used just like typedef, but what is declared is the size of the storage class. For example, we could define a type small this way:

```
sizetype u8 small;
```

The letter 'u' is optional and stands for 'unsigned.' Thus, a variable of type small is unsigned and 8 bits long. Another example would be:

```
sizetype 16 hexsize;
```

which would guarantee that variables of type hexsize are 16 bits long."

I am very enthusiastic about Mr. Gamble's suggestion. It would enhance portability of C code. Anyone who has moved between microcomputers and minis is aware that significant problems can occur when moving from machines with signed or unsigned characters. Integer length differences are also annoying. He adds:

"Use of the sizetype declaration would guarantee portability between machines (which currently is a heck of a problem). It also means that only one more reserved word is added, instead of the many which would be needed to define a type for every conceivable integer length. If the sizeof operator could be altered to return fractions, we could use sizetype to define bit lengths that are not multiples of eight. On the other hand, I notice that the sizeof operation on a bit-field structure is not defined in *The C Programming Language*, so maybe it is just up to the person who writes the compiler."

Another August Response

Readers may have followed the somewhat fervent comments made by Gerald Evenden in the August 1984 column. He has responded to my remarks, and these remarks deserve further commentary. To summarize the previous material: Mr. Evenden was extremely displeased by my comments about C because he felt that they could convey the wrong impressions to the uninitiated. Furthermore, he felt it unfair for me to discuss C I/O libraries without strongly emphasizing that C and its libraries are completely separate concepts (which they indeed are). He now writes:

"In regard to your response to my letter in the August issue . . . I feel I must expand upon some of my earlier points and make some additional comments. In addition, please excuse the excesses of a middle-age curmudgeon. Scars acquired in numerous battles of the computer wars tend to create a knee-jerk reaction when I sense some potentially deviant and dangerous thought processes. I wanted to emphasize that the compiler and the support library are two very distinct entities and we must be careful to maintain the distinction. When I talk of C, I am referring to the compiler . . . when I talk about the C library, I am referring to what I feel is currently a very vague and poorly defined item."

The original concept of a C library is defined in *The C Programming Language*. Other libraries are available on Unix systems, but these vary from installation to installation and from version to version. I agree with Mr. Even-

den that the words "C library" are currently vague. In my opinion, we should standardize functions that are not inherently Unix-only features. For example, we should include:

(1) "block" input-output (read, write, and relatives)
(2) stream input-output functions (e.g., fopen, fwrite, fputc)
(3) memory allocation functions (calloc, malloc, . . .)
(4) setjmp/longjmp procedures
(5) alarm (but not signal, which is Unix dependent)
(6) exit
(7) scanf, printf, and relatives

Furthermore, we should include the Unix math library, because these are fundamental routines (e.g., sin( ), exp( )). Readers may wish to formulate an exact list for exclusion and inclusion. If there is sufficient response, we could propose a standard for the C library in a future column.

Mr. Evenden continues:

"This sensitivity to the compiler-library problem is caused by having to deal with compilers where too many features that should have been relegated to the support library were included as part of the compiler [i.e., language definition]. For example, Fortran gives us READ, WRITE, and a few other input-output support operations, which must be treated by the compiler as special operations since the syntax does not match normal external module calls (of course, external modules are involved, but they are transparent to the programmer). When specialized input-output is required, which can't be handled by these statements, all sorts of contortions are done by the programmer to get around these restrictions . . . typically these gyrations are specific to the host system. In addition, many manufacturers will compound a bad situation by supplying a compiler with supplementary functions to provide access to unique features of their system. Goodbye transportability!"

Mr. Evenden is one hundred percent correct. Fortran's implicit connection of input-output functions to the language is a terrible failing. Pascal also suffers from this malady, even though it is a newer structured language. Evenden continues with the following remarks:

"In the case of C, the compiler writer doesn't have to go out of his way to handle special input-output syntax, and the programmer utilizing the typical C library can go to basically three levels of input-output to handle his problem:

(1) basic block 'read' – 'write'
(2) buffered (stream) input-output with the getc( ), putc( ) functions
(3) Fortran-like scanf( ) and printf( ) operations

This is an excellent example of building-block code: the read-write level is the lowest level and is the only place where we have to deal with the host machine's operating system; each successive level uses the previous level's entries. The applications programmer can thus choose the starting level best suited to his job and add the remaining tiers of code to perform his task."

Given this buildup of the C library, Mr. Evenden returns to the reasons for his original objections:

"One of the principal fears I have is that if we get to talking about the C compiler and a standard library in one breath, we will find some well-meaning ignoramus developing a C compiler with built-in input-output functions (or, for that matter, other 'special' features). In this situation, our input-output is engraved in stone, and we will be forced back into the same situation involved in Fortran coding. With C, we can individually or collectively trash the input-output part of the library in favor of some new software and still preserve the compiler itself. In addition, the old software is still good as long as we maintain a working copy of the old library. We often cannot do this if the compiler has been rewritten. I would much rather try to transport a program where a few specialized routines had to be rewritten than . . . deal with compiler variations."

To summarize the principal points Mr. Evenden makes (and with which I agree):

(1) Computer languages like C are superior because they segregate their library from the language definition.
(2) Because of (1), the language offers greater maintainability, even through revisions of the libraries, because we can retain old libraries more readily than whole compiler environments.
(3) Specialized applications can completely ignore the standard software library without any loss of power.
(4) When informing/teaching people about C, we must emphasize this unique feature to ensure that it is retained in future incarnations of the language. We hope future languages will also be constructed in this way.

To summarize his point, Evenden states:

"The problem of C libraries and what is a 'standard' C function is not yet resolved and needs further discussion. Tight binding of C and Unix is unfortunate, and we need to dissociate the two if we are to encourage non-Unix use of C and transportable C software. An important part of this unbinding is specifying a viable C library which can be installed without ambiguity and omission on a wide range of operating systems."

This is similar to what I mentioned above. I encourage a reader effort along these lines.

Tongue Biting

In his previous letter, Mr. Evenden indicated that he usually "bit his tongue" instead of complaining about C. I suspect that he is truly concerned that dissent about C could lead to its demise as an important language. He addresses this point as follows:

"My 'tongue biting' *remark related* to criticizing the C language was, of course, mostly rhetoric. But most of my criticisms are minor except for one: evaluation of all floating-point data elements in double precision. When two type float values are joined by a binary operator, I fail to see any reason why we should pay the costly runtime premium of double-precision evaluation! Of course, conversion of float function arguments to double is equally strange and pointless. I suspect that floating-point arithmetic was one of the last features added to the language and got shortchanged in the final stages of development of C. If the people at Bell Labs have any excuse for this peculiar handling of floating point by C, I'd sure like to hear it! Hopefully, some . . . readers may also have comments."

Mr. Evenden has hit on an important point. Not only is the conversion of float to double expensive, programming can get messy when dealing with pointers and arrays. For example, a program that needs a large number of floats (in an array) is difficult to use

with a function that expects pointers to double. I see no reason for using arrays of double-precision numbers for many applications. Yet, to simplify programming logic, memory must often be wasted (by a factor of two) for floating-point arrays. On systems like the 8086/8087, the actual cost for single/double-precision operations is the same, but conversions and other effects are still cause for grief.

Evenden continues:

"I have given some thought to what would be required to make C's floating point behave in a more traditional manner and have come to the conclusion that upward compatibility of a new compiler might not be possible as far as floating-point syntax is involved. Obviously, the 'standard' library routines will have to be changed ('printf' will need '%E' and '%e') . . . functions returning floating-point values will have to be in two precisions (sqrt( ) and dsqrt( ), for example)."

Variable Formats in printf( )

I'd like to change gears and consider an aspect of printf( ) format strings. Consistent with the previous discussion, I remind readers that this is a discussion relevant to the C library and not the compiler.

A typical printf( ) call might look as follows:

```
float number;
...
printf("%7.3e\n",number);
```

In certain cases, we might wish to vary the format dynamically. One way to accomplish this is shown in the Figure (page 25).

The format string in the sprintf( ) presented in the figure is "%%%sf." The first two percent signs place a single output percent in fmt_string. The "%s" causes the string format (containing "7.3") to go into fmt_string. Finally, the letter "f" is interpreted literally and is sent to fmt_string. The point of giving this example is to show how cumbersome the operation can be. Now I want to pose a simple solution to the problem.

To allow variable formats as part of a single printf( ), we need a way to indicate an indirection. Then printf( ) could use the current member of its argument list as a source for the format. This is illustrated as follows:

```
float number;
...
printf("%&s\n","7.3f",number);
```

The indirect format is "&s," which tells printf( ) to take the first argument as a string and print it into the format string before proceeding. Thus, the ultimate format string is "%7.3f." Another possibility would be:

```
float number;
float format = 7.3;
...
printf("%&f\n",format,number);
```

which demonstrates the range of choices allowed by indirect formats. To print an ampersand, we would need the following sequence:

```
printf("%&&");
```

Why would these be useful? Primarily, they allow programs to readily adapt to data variations. This could allow greater user selectability or, if extended to scanf( ), greater ability to read and write "foreign" data files.

Conclusions and Comments

In this column, I have presented more reader feedback and some brief comments about printf( ).

The "C/Unix Programmer's Notebook" was started in September 1983. I think that I have achieved a lot in writing the column during this last year. Unfortunately, other responsibilities make it impossible for me to continue. Therefore, I wish to thank those readers who have responded for their assistance, ideas, and criticisms. I also hope the readership will provide the new columnist with the same level of enthusiastic support that I have received. Intentionally, I have left few loose ends in my discussion. This will allow the next columnist to develop areas of discussion without too much loss of continuity. With a new columnist comes a new perspective, and I hope that more Unix coverage will be possible. Certainly, I don't expect that C will be excluded for the benefit of Unix but rather that a balance will be struck. Perhaps someday we will have separate "Notebooks" for each subject.

Thank you. I look forward to enjoying the next one hundred issues of *DDJ*, and I hope you enjoy them too.

An Addendum: The Future of C

Concerning future upgrades/modifications of C, Evenden writes:

"Some people criticize C as being a 'Spartan' language, but I maintain that this Spartan attribute is its principal and strongest feature . . . [C] is a real programmer's language providing an excellent tool for doing everything from real-time processing, to writing other compilers, to sophisticated scientific applications. If we ever make changes to C, we will have to be very careful to maintain this strong feature of the language."

I leave this quotation as my parting remark.

DDJ

```
float number;
...
char fmt_string[100];                /* make format string here*/
char format[10];                     /* format contained here */
strcpy(format,"7.3");                /* copy a specific format */
...
sprintf(fmt_string,"%%%sf",format); /* make format string */
printf(fmt_string,number);           /* print number in format */
```

Figure

Festschrift for Doctor Dobb

Contributions by Suzanne Rodriguez, Tom Pittman and Bob Albrecht in celebration of the 100th issue of *DDJ*

Suzanne Rodriguez was editor of DDJ *from January 1979 to September 1980. It was Suzanne who introduced the funky colored border (designed by Betsy Roeth and Aleeca Harrison) and Suzanne who published Ron Cain's original Small-C compiler.*

Tom Pittman wrote to DDJ *in March 1976 to announce his version of Tiny Basic for the Motorola 6800 microprocessor. Unlike the* DDJ *version, Tom's Basic was not free ("software is my living," he said). The price: $5.*

Bob Albrecht is the "OB" of Dr. Dobb. After launching (with Dennis Allison) DDJ, *he went off to launch new projects. He is presently involved in a corporation dedicated to computers and small children, fantasy role-playing games, and tennis. His contribution here launches a new* DDJ *department by Bob and Mike Swaine: Realizable Fantasies.*

Who is Dr. Dobb? Can you hack on a Mac? And who are the hackers of tomorrow?

The Bob and Dennis Show
by Suzanne Rodriguez

In the past year I've read more than one book or article discussing the founding of *Dr. Dobb's Journal*, and chances are that you have, too. It all sounds fairly straightforward on the surface: Bob Albrecht asked Dennis Allison to write a 2K BASIC that could be published in the newspaper Albrecht had founded, *People's Computer Company*. Allison complied. The rest is history.

What's not history—at least not yet—is how incredibly prolific and creative Albrecht and Allison are. Together or individually, they've founded many successful organizations, and even a few publications, got them going and then gone on to pursue new ventures. Allison's ventures tend to take place in the business and academic worlds; Albrecht's almost exclusively with children and computer learning.

What is also not history is the long list of people whose lives have been influenced, at a formative time, by one or the other—or both—of this duo. If you've been following the routine rules and regulations of the world, doing things the way you've heard they're supposed to be done, your first interaction with Allison and/or Albrecht can leave you somewhat dumbfounded. Prolonged interaction either gets you running with a vengeance back toward the world of rules, or it changes you.

Take me, for instance. I first met Allison and Albrecht in late August of 1978, when they interviewed me to become the third editor of *Dr. Dobb's Journal*. (Jim Warren, *DDJ*'s first editor, had gone on to found both the *Computer Faire* and the newspaper that would, in time, become *InfoWorld*. Warren's successor, Tom Williams, was soon leaving *DDJ* to pursue other endeavors.) At the time I was young and inexperienced enough to be impressed with my own credentials, which were really rather slender: I had published a number of magazine articles; I had been a technical writer at Zilog, writing their first piece of documentation; and I was working on my Master's in Journalism at Stanford. The idea of being editor of *DDJ* was very exciting: I could combine what I'd

be learning in school with the technical aspects of the magazine.

At least that's what I thought. I imagined that I'd be hired for a straightforward, rational, logical job, and that I'd be working closely with rational and logical people, applying well thought-out publishing principles in a logical and rational manner. Boy, did I have a thing or two to learn!

First there was the interview with Allison. He greeted me in his antique-and-book-laden home and proceeded to question me very thoroughly about my recent trip to Greece. We sipped retsina and talked for an hour about Ios and Samos, ouzo and feta, sleeping on beaches and hitching rides on donkeys. Finally he turned to me and said: "Well, Suzie, as far as I'm concerned, you've got the job."

I was very puzzled. We hadn't mentioned the word *computer*, and the topic of *Dr. Dobb's Journal* had not come up at all. "But," I stammered, "don't you want to hear about my background? I mean, I've worked with . . ."

He dismissed my words with an impatient wave of his hand, a gesture which I would come to learn was characteristic. "No, no! I think *DDJ* will be in good hands. I make these decisions based on character, and it doesn't seem that too much will get by you. You might find it difficult working with Albrecht, though—he can be trying—but I think you can handle him."

I went both dazedly and apprehensively off to my interview with Albrecht. He had asked me to join him in a "piano bar safari." I had no idea what that was, but I was soon to find out. A *true* piano bar safari consists of going from one piano bar to another, drinking beer and singing in each. At our initial meeting, however, we only hit one piano bar. I managed to find him in the dark interior, and we spent the next few hours (when he wasn't singing) talking about fantasy role playing, Greek dancing and tennis. Once more, computers and *Dr. Dobb's* were never discussed. When it was time for me to go, Albrecht said, almost as if in afterthought: "Well, if Dennis says you're ok, that's good enough for me."

This time I didn't even bother to start flaunting my qualifications. "Fine," I said, "I'm looking forward to working with such a talented bunch."

"Oh, you'll learn a lot, there's no doubt about that," said Albrecht. "You might have trouble with Allison, though—he can be difficult—but I think you can handle him!"

This was all the beginning of something great—my two years at *DDJ*. Every day I would sit in my classes and learn about *the way it's supposed to be* in the publishing world. And then I would dash to *People's Computer Company* and experience *the way it really is*. Or at least the way it was in that particular place and at that particular time.

While the day-to-day decision making for *DDJ* was in my hands, larger decisions—for instance, should we go to glossy paper—were decided by the Board. This occasioned marvelous and stormy battles, the best of which were with Bob Albrecht. In the end he would always give in and tell me he'd wanted it all along. "I just like a good fight," he would say. This would usually be a day in which I had taken a mid-term in something like "Human Resources and Intra-Company Dynamics". If I'd used my real-life experiences with Albrecht as illustrative examples, I would have flunked the course.

Dennis Allison, on the other hand, never fought; he just told you exactly what he thought. "I think that's a fantastic idea" made you feel just great. "I

think that's the dumbest idea I ever heard" plunged you into the depths. I kept telling Dennis that what I had learned in my seminar on how to motivate others taught me that he was going about things all wrong. In response, he said that was the dumbest thing he'd ever heard.

There were the even stormier meetings of the Board of Trustees where Allison would bait Albrecht, or Albrecht would bait Allison, and one or the other would storm out, only to return 5 minutes later with a bottle of champagne. We would all toast to friendship, and I would silently ponder the term paper I had just finished for a course in the law school: "Obligations and Duties of Trustees in a California Non-Profit Corporation." What would the professor think of the scene I'd just witnessed?

I devoted most of my attention to my studies and to my duties at *Dr. Dobb's*—there wasn't time for much else—but I always managed to keep aware of what Allison and Albrecht were up to. (All of you who know them are probably saying, "How could she *help* it?") They were always trying to tell me that I'd never learn from a book what I needed to know about life and my work, and about getting along with people. For two years I argued with them, together and individually, telling them I just didn't understand the haphazard, unplanned, chaotic way they went about doing things. They kept telling me that I didn't know it yet, but I was just as crazy as they were.

I suppose they were right. In the end when I'd finished my schoolwork and learned everything I could at *Dr. Dobb' Journal*, I turned down each of the highly-paid job offers that came my way. I moved to Sausalito to starve and write a novel. There are a couple of characters in it who remind me of Allison and Albrecht; they're quite vivid, extremely bohemian and very strong. With the exception of the female protagonist, they rather steal the show.

Mac the Hack
by Tom Pittman

I recently attended a "Hackers Conference," where one of the subjects of discussion was the definition of *hacker*. There was no consensus. Why?

When Apple introduced the Macintosh, I got excited about it just like everybody else. This was the computer I really wanted! At the Hackers Conference, over half the people were into the Mac; none was willing to admit to spending his time on IBM. Why?

About the time Jim Warren sold the *West Coast Computer Faire*, most of us had noticed that the fun was gone. Where were "the good old days?" Steven Levy's book *Hackers* chronicles "the last hacker." The *Homebrew Computer Club* just isn't what it used to be. Why?

When I started to write this, I thought I had answers to these questions, a sort of "unified field theory" of computerism. I'm not so sure any more, but at least I can share some ideas.

A hacker is an artist, and computer artistry is not distinguished from other art forms except in the medium chosen.

Just as paintings range from great (e.g. Rembrandt, Dali) to professional (painted houses and cars) to spray-can grafitti, so computer artisty ranges from the sublime (e.g. MacPaint, TEX) to the professional (1-2-3 and MSDOS) to the so-called 414s. Unlike painting, sculpture, music, and the like, few people can really appreciate the artistry in a computer product. Perhaps that will change.

It is the nature of great art that it compresses a great deal of design, intellectual sweat and individual personality into the created object. There are no great paintings painted by a committee of artists. Curiously, it seems to matter little what tools the artist had, or where his starting point was, provided that his accomplishment from that point exceeds what the rest of us could have done. Ansel Adams had color and motion available, but he chose to limit himself to black and white stills—and what art he created! A virtuoso on a violin produces art; the same sound from a Moog is ho-hum.

There is another facet to computing that deserves exploration: the urge to produce results that are impossible. There is a feeling of accomplishment in climbing the north face of Halfdome for the first time, in breaking the ("impossible") four-minute mile or the sound barrier, in producing a tune from a computer with only 256 bytes of RAM. This same feeling can accompany feats that are impossible only because some authority said "you can't" (or must not). Thus, the urge to create finds its expression in the hacker who cracks the protection scheme in commercial software, or breaks into a "secure" computer installation with his modem.

Macintosh is a computer that says, "you can't." It is promoted as a computer that is easy to use, but the box is sealed with screws you can't reach, the software (in ROM) can't be changed, and the development tools can't be obtained. Although I don't think that is why computer people buy it, still after you get over the realization that it does not do all they seem to claim for it, you are left with a lot of "impossible" things to challenge the aspiring intellect, a lot of opportunity to show off your freedom from such oppressive forces.

The computer hacker, like any real human being, wants recognition. I write programs for a living. I don't know if that qualifies me as a hacker, but it is more than just a job: it's *fun*. But I write my best stuff for other people, or at least to help me do things for others. I never seem to get around to doing things just for me.

To get recognition requires that the program (or hardware, or whatever) be noticeably outstanding. It must be head-and-shoulders above the competition, so that it is *clear* how good it is. If you start with limited tools you can do an outstanding job, but who can tell? Start with the best tools (would you believe Mac? Next year, anyway), and then your Herculean effort will be seen for what it is. The result: recognition, praise, glory.

Computing is a drug. Some people take aspirin to cure a headache, so they can get on with what they are trying to do. Others pop pills just for the immediate effects. Drugs have a way of growing on you. Once you have free-based Unix, it's pretty hard to go back to snorting CP/M. Mac gives a high something like Unix—it would be way up in the stratosphere if it had better software tools, like Unix already has—but Mac costs a lot less than Unix: it's affordable.

Computing is a religion. Centuries ago, in the 1940's, the founders of this religion (Turing, Von Neumann, etc.)

brought salvation to the human race, entrusting the Holy Truth to a small band of devoted disciples. As these saints spread the Gospel, great churches grew up (IBM, Univac) with an ecclesiastical hierarchy (we call them "priests" even today). A few anabaptists (the hackers at MIT) nipped at their heels, but the Reformation happened in Silicon Valley, led by Martin "8080" Luther; the doctrine of universal priesthood (all believers are priests) was once again gospel.

Not for long. Slowly the encrustations of institutional religion again took over. Scientific Enlightenment (bean counters in three-piece suits) replaced the faith of the Reformation. But there is hope! John Wesley and Billy Graham sparked revival in Christendom; Macintosh is doing the same for Computerdom. First Jimmy, then Ronnie; now Woz for President!

The urge to create, the delirious feeling of power, the longing for praise: these are universal human drives, dating back to the first rejection of authority in the Garden of Eden. The first vaporware announcement was, "You shall not surely die, but you shall be like gods." Macintosh, is thy name Serpent?

Tiny Hackers: a Realizable Fantasy by Bob Albrecht (with Mike Swaine)

Ob returns.

I first used the expression "realizable fantasies" in the *PCC* newsletter back before there was a *DDJ*. Starting next issue I'll be back in the pages of this magazine doing a column called "Realizable Fantasies" with Mike Swaine because we both think that people doing things with computers today need a kick in the imagination. The idea of the column will be to open discussion each month on some project that any fool can see is a good idea, but that, for whatever reason, nobody is carrying out. A public-domain Unix, an open-architecture Macclone, things like that.

This Festschrift paper can be considered the pilot for the series, because I have such a realizable fantasy. It has a name. I call it Tiny Hackers, and like any good fantasy, it wears a different face every time I look at it. Its origin goes back to before *DDJ*, back to the early sixties when I started teaching kids to play with computers. It recently got rekindled by listening to Brian Harvey, at the Hackers' Conference you're hearing so much about in this issue, talking about passing on knowledge to a new generation of hackers.

I have been working with kids and computers since 1962. I wasn't really having a good time in life until I began teaching kids about computers. I now work with tiny kids three to six years old, helping them to play with computers, although most of them are not, strictly speaking, hackers.

My corporation, *Dragonquest*, works with the East Menlo Park Boys' Club. Boys' Clubs are an ideal place for hackers who want to bring wonderment to kids. They are open to girls as well, and there are also Girls' Clubs. If kids are going to do things with computers, they're going to do so at home or at places like the clubs, not in the schools, which are the last place you're going to see any hacking. The stuff they're doing in schools is dreadful.

Around 1981 we started Computer Kids, of which Tim Finger is the director. With Computer Kids, we're bringing such things as programs that teach touch typing into the clubs. Once somebody gets comfortable with using the machine to store and print words, it's only a short step to seeing the possibility of putting your thoughts into words and printing many copies and distributing them; and that's a newsletter. We'd like to explore the possibility of kids in the Black community getting into self-publishing and using the newsletter to find out what's going on around them.

I want to bring tools to these kids, powerful tools. Future tools. Important tools. Once they have those tools, some of them are going to become the next generation of hackers; they will be the kids who redefine the word hacker.

What kind of tools should they have? Is there a Tiny Hackers' machine? Cheap, powerful, designed so small children can use it effectively and artfully?

If you dig through back issues of *Rainbow*, a magazine for users of the Radio Shack Color Computer, you can find a wealth of tips for turning the Color Computer into a system with surprising programming power, like a cheap 128K upgrade. I priced out a system: $750 will give you a Color Computer with O/S-9, a Unix-like operating system. There is a Basic compiler available, a Pascal compiler and a C compiler. That's one possibility for a Tiny Hackers' machine. [*Radio Shack may in fact bring out a Color Computer in 1985 with 128K RAM, an RGB monitor and O/S-9 in ROM. —Ed.*]

But there are other possibilities. Now is the time for somebody to have the courage to produce a home computer with Logo built in. With sprites. Kids don't give a damn about turtle graphics, but they love sprites. And a touch pad and a keyboard with keys in alphabetical order so the kids can find the letters. The Qwerty layout is about as easy to use as if the letters were distributed randomly. Why put barriers like that in kids' way?

It would be great to see kids in the clubs building Lee Felsenstein's proposed Hackers' Macs from kits.

So that's the realizable fantasy: Tiny Hackers. It's a confluence of ideas that could split off into many streams: projects, columns, newsletters. One of the forgotten motivations behind the original Tiny BASIC was that it be a language for tiny *people*. Maybe we still need a tiny language. If you are interested in the next generation of hackers, if you think that we need home machines with sprite chips or that computers for kids should use random number generators that return integer values, consider *DDJ* a forum for bringing the issues forward.

I am starting a small, personal newsletter called *Dragonsmoke*, and I suspect that some of these issues will also come up there. The focus will be on whatever *Dragonquest* is up to, including the work at the Boys' Clubs. If you'd like to be on the mailing list, send an SASE to PO Box 310, Menlo Park CA 94026 (that's the original 1976 *DDJ* PO box).

But the thing I'd most like to see is for some of *DDJ*'s readers to do something good for a Boys' Club or Girls' Club. Dig out those old Sols and Imsais. Better yet, give time. **DDJ**



FIRE IN THE

by Paul Freiberger and
Michael Swaine

Strange things happen when hackers get organized.

There was a strong feeling [at the Homebrew Club] that we were subversives. We were subverting the way the giant corporations had run things. We were upsetting the establishment, forcing our mores into the industry. I was amazed that we could continue to meet without people arriving with bayonets to arrest the lot of us.

Keith Britton

The Homebrew Computer Club

Felsenstein and Marsh weren't there during Homebrew's gestation. Early in 1975, a number of San Francisco Bay Area counterculture information exchanges existed for people interested in computers. Community Memory was one. There was also PCC [People's Computer Company] and the PCC spin-off, the Community Computer Center. In addition, peace activist Fred Moore was running a non-computerized information network out of the Whole Earth Truck Store in Menlo Park, matching people with common interests about anything, not just computers. Moore became interested in computers when he realized he needed a machine and a base of operations, and he talked to Bob Albrecht at PCC about both. Soon, Moore was teaching children about computers and learning about them himself.

At the same time, Albrecht had been looking for someone to write certain assembly language programs and found Gordon French, a mechanical engineer and computer hobbyist, who then supported himself building slot car motors.

When the Altair story appeared in *Popular Electronics*, the need for a more direct information exchange became clear. The PCC people took the Altair seriously from the beginning. Keith Britton, a demolition consultant and PCC's treasurer, thought its arrival foretold the eventual demise of the computer priesthood. "All of us were champing at the bit

---

Michael Swaine, Editor-in-Chief of DDJ
Paul Freiberger, West Coast Editor of Popular Computing
From Fire in the Valley: The Making of the Personal Computer. *Copyright 2/3 1984 by McGraw-Hill, Inc. Used with the permission of Osborne/McGraw-Hill.*

VALLEY



Photo 1
Gordon French, one of the founding members, addressing a meeting of the Homebrew Computer Club in 1979

to get an Altair," French recalls. So Fred Moore pulled out his list of the computer curious, the revolutionaries, the techies, and the educational innovators, and sent out the call. "Are you building your own computer? Terminal? TV Typewriter? I/O device? Or some other digital black box? Or are you buying time on a time-sharing service?" Moore's flyer asked. "If so, you might like to come to a gathering of people with like-minded interests. Exchange information, swap ideas, talk shop, help work on a project, whatever." The announcement tentatively called the group the Amateur Computer Users Group or Homebrew Computer Club, and it met on March 5, 1975, in Gordon French's garage.

Felsenstein read about the upcoming meeting and intended not to miss it. He collared Bob Marsh and they drove in Felsenstein's pickup truck through the rain across the Bay Bridge to the peninsula that stretches from San Francisco in the north to Silicon Valley in the south. Gordon French lived in suburban Menlo Park, a town jogging distance from Stanford and right on the edge of Silicon Valley.

At the first meeting, Steve Dompier reported on his visit to Albuquerque. MITS had shipped 1500 Altairs and expected to ship 1100 more that month. The company was staggering under the weight of the orders and couldn't begin to fill them, Dompier said. Bob Albrecht displayed the Altair that PCC had received that week—PCC was just behind Harry Garland and Roger Melen, over at Stanford, on MITS's list—and passed out the latest issue of PCC.

Dompier, like Marsh and Felsenstein, had driven down from Berkeley, but most of the 32 or so attendees were from the San Francisco Peninsula. Albrecht and Gordon French, who chaired the meeting, and Fred Moore, who took notes for a newsletter, and Bob Reiling, who soon took over that newsletter, all lived in Menlo Park. Many other people had come from farther south—from deeper into Silicon Valley: Mountain View, Sunnyvale, Cupertino, San Jose—people like Allen Baum, Steve Wozniak, and Tom Pittman, who described himself as a microcomputer consultant, perhaps the first in the world.

As the meeting concluded, one Homebrewer held up an Intel 8008 chip, and asked who could use it, and gave it away. Many people present that night sensed the opportunity in this Homebrew spirit and in Dompier's words. One of

them was Bob Marsh. Marsh went immediately to see Gary Ingram about forming a business enterprise. I've got a garage, he said. It seemed enough. They decided to call themselves Processor Technology, or Proc Tech. Marsh designed three plug-in circuit boards for the Altair: two I/O boards and a memory board. They looked good, he thought. He devised a flyer announcing Proc Tech's products, made hundreds of copies of it on a campus photocopying machine, and took 300 of them to distribute at the third meeting.

By this time the club was flourishing. Fred Moore was exchanging newsletters with Hal Singer, who put out the *Micro-8 Newsletter* in Southern California and had formed a Micro-8 club shortly after Homebrew started. Other publications were passed around. PCC and Hal Chamberlain's *The Computer Hobbyist* attracted special interest. A Denver organization identifying itself as a provider of support for Micro-8 and TV Typewriter hobbyists and calling itself The Digital Group offered subscriptions to its newsletter. The movement was getting hard to keep up with. Intel, with its 4004, 8008, and 8080 chips, and at least 15 other semiconductor manufacturers had introduced microprocessors into the market, and the newly formed club tried to keep its members informed about them all.

The third Homebrew meeting drew several hundred people, too many for Gordon French's garage. It was moved to the Coleman mansion, a Victorian building serving as a schoolhouse. There Marsh gave a brief talk, explaining that he was selling memory and I/O boards for the Altair. He hoped to present Proc Tech as a serious company, not just the fancy of an unemployed electronics engineer with access to a copying machine. He offered a 20% discount for cash prepayment. To his disappointment, no one came to talk to him at or after the meeting.

The next week the first order arrived. Garland and Melen were seeking Processor Technology's cheapest advertised product. Their request was written on the stationery of their new company, Cromemco. They sent no check, just a purchase order requesting 30 days net credit, hardly what Marsh had expected. He had made Proc Tech look like a real company, all right.

After the Cromemco order, many others followed, most enclosing cash. Ingram fronted $360 for an advertisement in *Byte*. With the cash streaming in, Marsh and Ingram could afford to advertise in *Popular Electronics,* and they did, spending $1000 for a one-sixth-page ad. They incorporated, and Ingram was made president. As corporate headquarters and factory, they had half of an 1100-square-foot garage, no products, no schematics for proposed products, no supplies, no employees, and thousands of dollars in cash orders. It was beginning to appear that they had some work ahead of them.

Meanwhile, Lee Felsenstein was getting more involved with Homebrew. He took over from Gordon French as the master of ceremonies—he refused to think of himself as a chairman. The meetings were now held in the auditorium at the Stanford Linear Accelerator Center. Over the years, Felsenstein became intimately associated with the club and fostered its anarchic structure. The group had no official membership, no dues, and was open to everyone. Its newsletter, offered free after a nudge from Felsenstein, became a pointer to information sources and a link between hobbyists. As group toastmaster, Felsenstein performed with a curious kind of populist showmanship. As one attendee, Chris Espinosa, said, "People call him the Johnny Carson of Homebrew, but he's more than that. He kept order, he kept things moving, he made it *fun* to go to the meetings. There were 750 people in that room at one time, and he worked it like a rock

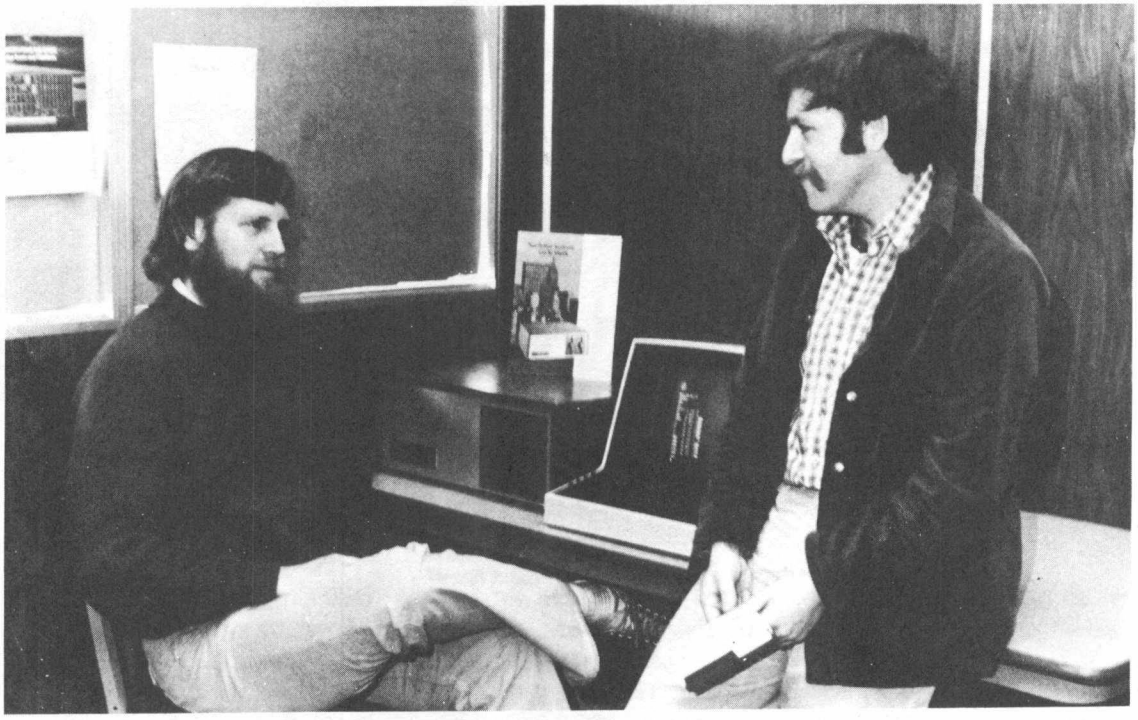


Photo 2
Chuck Grant and Mark Greenberg, founders of Kentucky Fried Computers and North Star Computers, with a NorthStar Horizon

concert. It's hard to describe, but to see him work a crowd like a Baptist preacher. . . . He was great."

The meetings didn't follow Robert's *Rules of Order* with Felsenstein running them: he gave them their own special structure. First came a mapping session, during which Felsenstein recognized people who briefly proffered their interests, questions, rumors, or plans. Felsenstein sometimes had quick answers to their questions or witty comments on their plans. A formal presentation followed, generally of someone's latest invention. Finally, there was the Random Access session, in which everyone scrambled around the auditorium to meet those they felt had interests in common with them. It worked brilliantly, and numerous companies were formed. A remarkable amount of information was exchanged at those meetings, and much information had to be exchanged; they were all in unfamiliar territory.

About this time, a San Francisco branch of Homebrew started. It held its first meeting at the Lawrence Hall of Science in Berkeley. Although it was called the San Francisco branch, Berkeley was a logical place for it to meet. Universities were becoming hotbeds of self-taught microcomputer expertise. Professors with grant money now found it cost-effective to buy minicomputers rather than buy time on the university mainframe computer, which was invariably out of date and overworked. DEC was selling PDP-8 and PDP-11 minicomputers to professors as fast as it could build them. They were particularly popular in psychology labs, where they were used for experimenting on human subjects, automating rat and pigeon labs, and analyzing data. The invasion of the psych lab by minicomputers created a new kind of expert: one who might know something about psychological research, but who was more clearly a hacker and a computer nut—someone to figure out how to run the computer and make it do what professors, who were generally ignorant about the machine, wanted.

Howard Fulmer was such a person. Fulmer worked in the Psychology Department at UC Berkeley running PDP-11s, selecting minicomputers for professors to buy, building interfaces, and programming experiments. In early 1975, one of Fulmer's professors bought an Altair, and Fulmer learned to use it. Soon after, Fulmer left his job to devote more time to microcomputers.

He was not alone: the Altair raided the University of California at Berkeley. George Morrow, a graduate student in math, worked at the university's Center for Research in Management Science with two other students, Chuck Grant and Mark Greenberg. They were trying to develop a language to use with a microprocessor in computer-controlled research.

Morrow, Grant, and Greenberg found that they worked well together. All three were perfectionists, although in different ways. Morrow, thin, prematurely balding, and with a twinkle in his eye and an irrepressible wit, seemed always to be enjoying himself, perhaps especially when he was hard at work. Grant and Greenberg appeared cut from a darker cloth. They were more businesslike. Although Grant and Greenberg often attended Homebrew meetings and profited from the free, open exchange of information, they never considered themselves part of the hobbyist community. Technically, the three formed a good team. Morrow knew hardware, Grant preferred software, and Greenberg was at home with either.

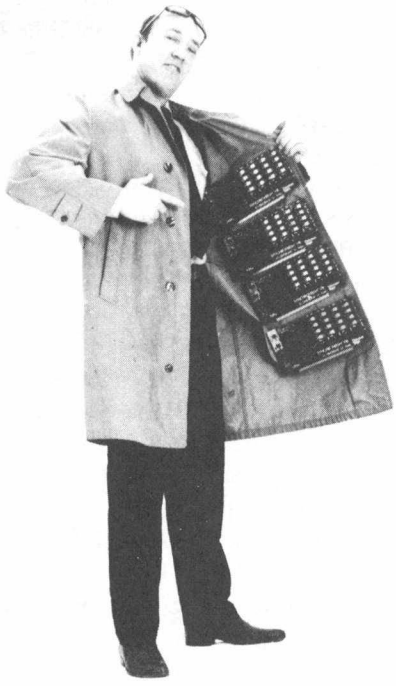


Photo 3
George Morrow in a pose for an early advertisement

The trio considered making boards for the Altair or even a kit computer of their own. They knew that they were a good design team, but they also knew they lacked sophistication in marketing. So Morrow sought the advice of Bill Godbout. Middle-aged, blunt, opinionated, with a paunch that he joked about and an airplane that he flew stunts in, Godbout was the electronics distributor whom Bob Marsh had tried to interest in his walnut digital clock when he and Felsenstein had first worked in the garage. Morrow told Godbout about their plans.

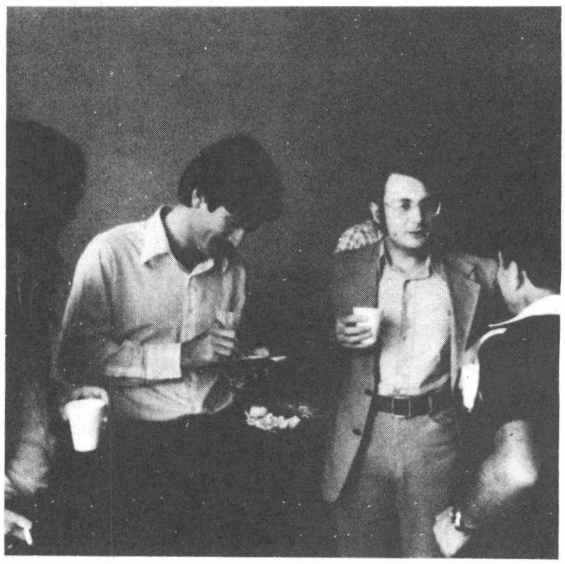


Photo 4
Steve Dompier, Bob Marsh, and Lee Felsenstein at a meeting of Processor Technology dealers in 1979.

Godbout was then selling chips and minicomputer memory boards by mail, and Morrow asked if he intended to sell Altair memory boards. Godbout scoffed. He wouldn't so dignify the product, he said. Morrow wondered if he might be interested in distributing a good computer, the creation of a top-notch design team. "You guys?" Godbout asked. He looked Morrow over. Godbout believed he was good at sizing people up, and Morrow looked all right. They agreed to split profits down the middle and shook on it. No written contract, Godbout said. Written contracts were a sign of mistrust and an invention of lawyers, and if there was anybody Godbout didn't trust, it was a lawyer.

By this time a motley group of engineers and revolutionaries had assembled in Silicon Valley in the infancy of a billion-dollar industry: irascible Bill Godbout, who suspected lawyers; ex-*Berkeley Barb* technical editor and current Homebrew toastmaster Lee Felsenstein; Bob Albrecht, who left a high-paying career to teach children about computers, who smoked cigars, and called himself "The Dragon"; Bob Marsh, testing his own abilities, turning his love for electronics into a garage corporation; and Keith Britton, who saw himself and the other Homebrewers as pivotal in "an equivalent of the industrial revolution but profoundly more important to the human race." A surprising number of them held political views that would have shocked the local Rotary Club, and almost all had no love for IBM and the computer establishment. But they and others like them were pulling off the most startling entrepreneurial achievement in recent times. And much of the action took place at Homebrew.

The Homebrew Computer Club was not merely the spawning ground of many Silicon Valley microcomputer companies. It was also the intellectual nutrient in which they first swam. Presidents of competing companies and chief engineers would gather there to argue design philosophy and announce new products. Statements made at Homebrew changed the directions of corporations. Homebrew was a respected critic of microcomputer products. The Homebrewers were sharp, and could spot shoddy merchandise and items that were difficult to maintain. They blew the whistle on faulty equipment and meted out praise for solid engineering and convivial technologies. Homebrewers soon developed the power to make or break new companies. In part due to Lee Felsenstein, Homebrew encouraged the conviction that computers should be used for and not against people. Homebrew thrived in a kind of joyous anarchy, but the club was also an important step in the development of a multi-billion dollar industry. Processor Technology was one of the children of Homebrew.

The first part of the meeting we were involved in open combat with Intel. Intel was out to torpedo any standardization effort on the S-100 bus.

George Morrow

Home Rule

A continuing fear in the developing microcomputing industry was that "the big boys" would come in and spoil all the fun. Sometimes "the big boys" meant IBM and the other mainframe computer and minicomputer companies; sometimes it meant Texas Instruments, Commodore, and the other electronics companies that had waged Pyrrhic price-cutting wars in the calculator industry; especially it meant Texas Instruments, known for its ruthless price-cutting. Lee Felsenstein summarized the dread of the hobbyist entrepreneurs: "Anyone but TI!" Intel and some of the other semiconductor companies, although well situated to produce microcomputers from their own chips, had expressed reluctance to do anything that could be construed as competing with their own customers, and the hobby-born microcomputer companies had developed enough clout by this time to be taken seriously as semiconductor customers.

In December of 1976, Commodore International, an electronics firm with a lot of market muscle, leaked information to *Electronic Engineering Times* about a new product. Commodore, the story went, was ready to release a machine very much like a low-cost Sol. Proc Tech was just shipping the first Sols and Marsh was thinking about the company's next product, a new version of the Sol with an integrated keyboard and 64K of memory, that would be cheap at $1000. It was, unfortunately, essentially the Commodore machine.

Convinced that Commodore actually had the computer on the launch pad and that Proc Tech could never compete with it, and even more worried by news that National Semiconductor was also planning a microcomputer, Marsh scrapped the project. The laws of battle in the semiconductor wars five years earlier had been to cut prices to the baseline and push the technology madly, even under threat of corporate extinction. Proc Tech couldn't compete with National, especially in mortal combat. But in fact the Commodore machine would not appear for some time, and the National computer never materialized at all.

Many new hobbyist-born companies were starting to manufacture microcomputer products, but most of these were turning out boards for the Altair or IMSAI, and practically all were small companies, start-ups like Proc Tech.

Howard Fulmer began such a firm in his Oakland basement. After reading an editorial by Ed Roberts in Dave Bunnell's *Computer Notes* that attacked the compatible board companies as "parasites," he considered calling his own company Symbiotic Engineering to emphasize his conception of the proper relationship between MITS's products and his own. But a group called the Symbionese Liberation Army was making a name for itself then, and he wanted to avoid confusion. He called his company Parasitic Engineering.

George Morrow and Howard Fulmer were both designing Altair-compatible products, and in the spring of 1977 decided to build a computer together. Morrow would supply Fulmer the boards he had designed at a cheap price, and Fulmer would devise the remainder of the computer. Fulmer called it the Equinox 100. It was a solid design, for they had listened to the ideas of Bob Mullen, one of the founders of Diablo Systems, a Silicon Valley disk-drive manufacturer, and of Bill Godbout about improving the S-100 bus.

The timing of the machine's release was unfortunate though. The Equinox was an 8080 machine, and Technical Design Labs in New Jersey, Garland and Melen's Cromemco, and The Digital Group in Denver were all known to be designing computers based on the new and apparently better Z80 chip. Cromemco had already produced a Z80

central processor board, and hobbyists were dropping it into the IMSAI chassis to create a mongrel Z80 machine.

Marsh wondered if Proc Tech shouldn't do a Z80 machine as well. But it seemed irrational to dump a successful design to achieve a marginal improvement in performance. Besides, the processor mattered much less than the software, he believed. The software made the computer work, and that would distinguish one machine from another.

Proc Tech called two programmers, Jerry Kirk and Paul Greenfield of MicroTech in Sunnyvale, who had produced high-level language compilers for minicomputers. They were hired to create a set of programmer's tools, programs that would make it easier to write, edit, and debug other programs on the Sol. Ingram developed their work into Software Package One.

Ownership of software was an inflammatory issue in the Valley and elsewhere. Proc Tech was aggressively pro-piracy, and its hobbyist founders swapped program tapes at Homebrew meetings along with everyone else. Gordon French, who after helping to start Homebrew had become Proc Tech's General Factotum (his official title), argued for an open system, that is, free dissemination of software code and internal workings to everyone. He wanted outside programmers and peripheral manufacturers to be able to create compatible products and expand the market. At that time, Ed Roberts and the entire mainframe and minicomputer industry held the opposite view. But the hobbyists were bringing their own values into their industry. An open architecture, the publicly known physical design of the machine, was one emerging ideal. An open operating system was another. But at Proc Tech the idea of an open operating system was frowned upon. Marsh and Ingram wanted a proprietary operating system.

In fact, Proc Tech had its own disk operating system very early on. The company bought PTDOS from its author, 19-year-old Bill Levy, who developed it at the Lawrence Hall of Science at Berkeley. Levy modeled PTDOS after Unix, a mainframe/minicomputer operating system in use at UC Berkeley. Marsh thought PTDOS much better than CP/M, but PTDOS was slow to reach the market because of "the drive fiasco."

Disk drives posed an alluring challenge in 1976, when the Sol was released. They existed and were used heavily in mainframe and minicomputers, but to mount a disk drive on a microcomputer was prohibitively expensive. Drives typically cost $3500. So Marsh was very intrigued when George Comstock, Bob Mullen's partner at Diablo Systems, declared at Homebrew one night that he wanted to develop a disk drive for microcomputers. Comstock thought that a drive, complete with a controller board and software, could be sold for about $1000.

But Diablo was not then involved in the growing microcomputer industry, and Comstock felt a disk-drive system would only flail around without close consultation with microcomputer companies. He later proposed a joint effort to Marsh. Diablo would develop the drives, the physical mechanisms that read and write information from and to disks, and Processor Technology would write the software and develop an S-100 board to control the drives. He also proposed that Proc Tech could market the board on its own.

Disk drives were so clearly destined to belong in any serious microcomputer system that engineers were already vying to develop a low-cost drive system with software and a controller board. Shugart's 5¼-inch drives seemed attractive, but they had one drawback. IBM had been using 8-inch drives and had established certain standards for the devices. There were no standards for small disk drives and no guarantees that disks written on one brand of machine would be readable on another. North Star had selected the Shugart drive and sold it for under $800. Using an idea of Eugene Fisher's of Lawrence Livermore Labs, both Morrow and San Francisco engineer Ben Cooper had begun developing relatively low-cost 8-inch disk drives. Cooper had perhaps the first commercial 8-inch disk controller for microcomputers. Morrow, shortly thereafter, had the first one available for the $1000 price Comstock was aiming for, and he negotiated with Digital Research and Microsoft for an operating system (CP/M) and BASIC to distribute free with the drive system. Both Morrow and Cooper continued to develop significant disk products, and Cooper created the first hard disk controller for microcomputers.

But at Proc Tech, the disk drive plans were crumbling. Diablo encountered trouble with the drives and dropped the project, leaving Proc Tech so far into development of the controller that it had to continue. Marsh and Ingram raised the price of the system to $1700 and substituted a more expensive drive offered by Perscii. The price was too high, and Proc Tech's drives didn't always work. Customers could find better deals from Cooper, Morrow, and North Star.

Despite such problems, Proc Tech seemed to be thriving. The executives were recycling their profits into the company. (Lee Felsenstein was investing his in the Community Memory project.) The Proc Tech staff in Emeryville now numbered 85, not counting non-employee/consultant Felsenstein, and headquarters was growing crowded. That year, 1977, Proc Tech moved south to the bedroom community of Pleasanton. The new offices boasted a spacious executive suite with large windows looking out over the valley.

But there was competition. As 1977 came to an end, Proc Tech found itself in a more serious industry. The trading of information, the shirt-sleeve management, the flashes of idealism, and the lack of detailed planning that had characterized the industry from the start still existed. The chief users and the designers and company presidents were still hobbyists at heart, and most of the world knew nothing of the revolution that was afoot. But new companies were emerging like mushrooms overnight. Among the computer and computer-related companies at the end of 1977 were Apple (which some insiders thought had great promise), Exidy, IMSAI, Digital Microsystems, Alpha Micro Systems, Commodore, Midwest Scientific, GNAT, Southwest Technical Products, MITS, Technical Design Labs, Vector Graphic, Ithaca Audio, Heathkit, Cromemco, MOS Technology, RCA, TEI, Ohio Scientific, The Digital Group, Micromation, Polymorphic Systems, Parasitic Engineering, Godbout Engineering, Radio Shack, Dynabyte, North Star, Morrow's Microstuff, and, of course, Processor Technology.

Many companies were located in the Bay Area and were associated with the Homebrew Club. The club had become large, and by 1977 tended to assemble in fairly predictable

groups. In front, performing, was Lee Felsenstein. Bob Marsh and the Proc Tech group were usually assembled along one wall. Steve Wozniak and the boys from Apple and the other 6502 processor fans sat in the back. Jim Warren of *Dr. Dobb's* sat on the aisle three seats from the back, stage left, ready to stand during the mapping session and do his Core Dump, an extemporaneous outpouring of all the news and rumors he had heard. The front row always had Gordon *French*, who maintained the software library, and Bob Reiling, who wrote the newsletter.

In December 1977, Reiling wrote, "The development of special-interest groups has probably been the biggest change during the past year. At the beginning of the year the 6800 group was holding regular meetings. At the end of 1977, the groups include not only the 6800 group, but also the F8 Users, North Star Users Group, Sol Users Society, and PET Users." At that time, the Homebrew attendees (the club did not have members) included key people from Apple, Cromemco, Commodore, Computer Faire, *Dr. Dobb's,* Itty Bitty Computers, M&R Enterprises, Mountain Hardware, IBEX, Mullen Computer Boards, North Star, PCC, Proc Tech, and the Bay Area computer stores. The most significant of these then was Proc Tech. Marsh had, to some extent, realized his dream. The company seemed to be doing very well.

And in December, Reiling could report optimistically, "The IEEE now has a standards group to sort out the various hardware and software standards." That blithe statement subsumed a wrangling struggle and a remarkable achievement which brought new legitimacy to the industry. The sorting out had been no simple matter.

Bob Stewart was a consultant in optics and electronics and a member of the Institute for Electrical and Electronics Engineers (IEEE). He had bought an Altair and had become frustrated with it. At a meeting at Diablo Valley College to discuss the S-100 bus he met some company presidents: Harry Garland of Cromemco, Howard Fulmer of Parasitic Engineering, Ben Cooper of Micromation, and George Morrow of what he was then calling Thinkertoys. *Byte*'s Carl Helmers was also there. The idea was to cure the obvious problems of the bus and to establish common standards, so that one company's board would work with another's. Garland explained the virtues of his and Melen's shielded bus, but Morrow thought he had a better approach. No immediate agreement was forthcoming. Stewart suggested creating an official IEEE standard for the bus. With the group's encouragement, he petitioned the IEEE to form a microcomputer standards subcommittee of the computer standards committee. The petition succeeded, and the group became official.

Roberts was invited to participate in the microcomputer standards subcommittee, but declined to send a representative or even to respond directly. He did say in print that he felt MITS had the sole right to define the bus. The subcommittee ignored him. At first, the meetings involved contention with Intel, which fought standardization. Morrow got the impression that Intel wanted no standards unless Intel was setting them. But when the subcommittee decided to formulate standards whether Intel liked them or not, Intel acquiesced. This was outrageous cheek. A bunch of hobbyists turned entrepreneurs had simply ignored the biggest microcomputer company of that time and had faced the leading chip manufacturer and not been struck by lightning.

In spite of its solidarity, the subcommittee had no guarantee that it could really create standards. The subcommittee had 15 assertive, opinionated people disputing an issue about which they held legitimate and conceivably irresolvable differences. Each of the members had a product that would be incompatible with anything likely to be proposed. As the meetings went on, Roger Melen came in for Cromemco. Alpha Micro was represented. Elwood Douglas appeared for Proc Tech and judged the standard against the memory board he was designing. George Millard spoke for North Star. Someone arrived from IMSAI to read its formal position, which resembled Ed Roberts'. The subcommittee ignored that position too. Most of its members had written IMSAI off as a place where training in *est* mattered more than training in engineering.

At times the subcommittee members weren't too fond of each other. They argued for hours, with no one yielding an inch. They would then return to their companies and discuss how to compromise their own designs to achieve a standard. At the next meeting, they would find themselves closer to agreement. Little by little, these creative, independent people subordinated their egos and short-term economic gains for the good of the entire microcomputer field.

The committee was attempting guerrilla design. In mainframes and minicomputers, the bus was always whatever the bus designer said it was. Although the IEEE suggested subtle variations in tolerance during the process of formalizing the company bus into a standard, independent committees did not assemble to redesign the whole bus. Timing parameters and other features were dictated by the companies. IBM and DEC worked this way. In a way their method was certainly easier than communal design. But the S-100 committee members dug into the Roberts bus, figured out how it worked, and were scrapping it in favor of a new, independent bus open to all. This was a populist revolt against the tyranny of the big company, with MITS hoisted as a poor but adequate symbol of the big company. The revolution was succeeding.

DDJ

Tiny BASIC for the 68000

by Gordon Brandly

Dr. Dobb's Journal *was created by Dennis Allison and Bob Albrecht in 1975 as a vehicle for getting public-domain versions of the BASIC programming language into the hands of computer enthusiasts. Except for Gates and Allen's BASIC for the MITS* Altair, *there was no high-level language available for the new microcomputers in 1975, only arcane assembly language, so Allison and Albrecht, interested in giving kids and tinkerers and as many people as possible access to the technology, set about developing a microcomputer BASIC and publishing reports on its development in their newsletter, PCC. Since at that time 4K of memory was a lot, they made their BASIC* Tiny. *BASIC itself is ubiquitous now and still lacking any practical standard definition; no longer Tiny, it is now Better or Professional or True.* DDJ *publishes little BASIC code now, but over the years* DDJ *has continued to publish programming tools, including small versions of the C and Ada languages.*

But memory is *cheap today; does it make any sense to write about Tiny Basic for the Motorola 68000 microprocessor? The answer, we think, is yes. Tiny BASIC was never intended to be a development language, but it* is *a programming language, and it* is *tiny, and "cheap" is a relative term. The question is, how many people are interested in getting into the 68000 as cheaply as possible? For them, this.—Editor.*

This 3K interpreter beats Applesoft on the Sieve benchmark.

Tiny BASIC was more popular than its inventors expected and they soon found that they needed a separate newsletter just to publish Tiny BASIC developments. Dick Whipple and John Arnold of Tyler, Texas, wrote a 2.9K Tiny BASIC Extended, which was published in octal in the first issue of Dr. Dobb's Journal of Tiny *BASIC* Calisthenics and Orthodontia (Running Light without Overbyte), *in January, 1976. Other Tiny BASICs followed.*

Well, a lot followed, in fact. Virtually the whole personal computer revolution to date: the demise of the early microcomputer companies (MITS, Imsai, Proc Tech, Sphere), Apple's rise and Osborne's rise and fall, the entries of Tandy and IBM, the irony of Atari, expensive advertising and cheap memory.

Gordon Brandly, R. R. 2, Fort Sask., AB, Canada, T8L 2N8.

Remember the good old days? When the 8080 microprocessor reigned supreme, 8K of memory cost an arm and a leg, ah yes . . . Well, the years went by, microcomputers got bigger, software grew more sophisticated, and prices went up. This is just fine, of course, if you can afford the higher prices. The less fortunate among us, however, must build or buy smaller 16-bit "educational" systems. This is fine too—if you don't mind having hardly any software.

This is just the sort of situation that gave rise to *Dr. Dobb's Journal* in the "good old days." The solution back then was to publish a tiny BASIC interpreter that could be adapted to just about any 8080 microcomputer around. This solution worked fabulously and gave many a hobby computer its first taste of useful software. Well, if the solution worked once, why not again? I therefore decided to produce a tiny BA-

SIC interpreter for the relatively small 68000 systems such as the Motorola Education Computer Board, the EMS 68000 board, and so on.

To produce this BASIC, I took one of the most successful 8080 tiny BASICs, Li Chen Wang's Palo Alto Tiny BASIC (*Dr. Dobb's Journal*, May 1976), and translated it into 68000 code. I then added a few features and optimized the code a little, producing a surprisingly usable interpreter.

First, I'll describe the differences between my interpreter, Palo Alto Tiny BASIC, and the ubiquitous Microsoft BASICs. I then will describe how you can install this software on your 68000 system. Finally, I'll give my evaluation of the interpreter's present performance and how it can be improved.

Features

Those who know the original Palo Alto Tiny BASIC (or the Sherry Brothers' version on CP/M User's Group Volume 11) will find this interpreter very similar. I have made two or three changes to the interpreter's syntax to bring it closer to the *de facto* Microsoft standard. The colon is used instead of the semicolon to separate multiple statements on a line. The inequality operator (#) has been changed to the more standard < > . I also added the PEEK, POKE, CALL, BYE, LOAD, and SAVE commands, which are described later.

Those of you used to a bigger BASIC, such as the various Microsoft interpreters, will find that this version works almost the same within its limitations. Following are some excerpts from Li Chen Wang's original documentation, mixed with descriptions of my extensions.

The Language

Numbers

In this version of Tiny BASIC, all numbers are 32-bit integers and must be in the range 2,147,483,647 to −2,147,483,648. I decided to use 32 bits so that the PEEK and POKE commands could access the entire address range of the 68000. This slows down arithmetic operations somewhat, but sticking to 16 bits would have produced unnecessary complications.

Variables

There are 26 variables, denoted by the letters A through Z, and a single array @(I). The dimension of this array (i.e., the range of value of the index I) is set automatically to make use of all the memory space that is left unused by the program (i.e., 0 through SIZE/4, see the SIZE function below). All variables and array elements are four bytes long.

Functions

There are four functions:

(1) ABS(X) gives the absolute value of X.
(2) RND(X) gives a random number between 1 and X (inclusive).
(3) SIZE gives the number of bytes left unused by the program.
(4) PEEK(X) gives the value of the byte at memory location X.

Commands

The LET command

```
LET A=234-5*6, A=A/2,
    X=A-100, @(X+9)=A-1
```

will set the variable A to the value of the expression 234−5*6 (or 204), set the variable A (again) to the value of the expression A/2 (or 102), set the variable X to the value of the expression A−100 (or 2), and then set the variable @(11) to 101 (where 11 is the value of the expression X+9 and 101 is the value of the expression A−1).

The PRINT command

```
PRINT A*3+1,
    "abc 123 !@#", 'cba '
```

will print the value of the expression A*3+1 (or 307), the string of characters abc 123 !@# and the string cba, and then a CR-LF (carriage return and line feed). Note that you can use either single or double quotes to quote strings, but pairs must match. If a comma appears at the end of the PRINT command, the final CR-LF will not be printed. Note also that commas separate adjacent items (most other BASICs use the semicolon to perform this function).

```
PRINT A, B, #3, C, D, E, #10, F, G
```

will print the values of A and B in 11 spaces; the values of C, D, and E in 3 spaces; and the values of F and G in 10 spaces. The value will be printed in full even if there aren't enough spaces specified for it.

```
PRINT 'abc',_,'xxx'
```

will print the string abc, a CR without

a LF, and then the string xxx (over the abc), followed by a CR-LF.

The INPUT command

```
INPUT A, B
```

will cause Tiny BASIC to print A: and wait to read in an expression from the console. The variable A will be set to the value of this expression. Then B: will be printed and variable B set to the value of the next expression entered. Note that you can enter complete expressions as well as numbers. This enables an interesting trick: you can set the variable Y to an unusual value (e.g., 9999) and use it to get the answer to a yes-or-no question, such as:

```
10 Y=9999
   : INPUT 'Are you sleepy?'A
   : IF A=Y GOTO 100
```

The user can answer the question with the expression Y, which puts the numeric value of Y into the A variable.

```
INPUT 'What is the weight'A,
   "and size"B
```

is the same as the first INPUT example except that the prompt A: is replaced by "What is the weight:" and the prompt B: is replaced by "and size:". Again, you can use both single and double quotes as long as they match.

```
INPUT A, 'string',_,
   "another string", B
```

with the strings and the _, has the same effect as in PRINT.

The POKE command

```
POKE 4000+X,Y
```

puts the value produced by expression Y into the byte memory location specified by the expression 4000+X.

The CALL command

```
CALL x
```

will call a machine language subroutine at the address specified by the expression x. All of the CPU's registers except the stack pointer can be used in the subroutine.

The BYE command will return control to the resident monitor program or operating system.

The SAVE command will save your BASIC program on the storage device you provide. Details on installing this device are given in the source code. As set up for the Educational Computer Board, this command will send the program out to the host computer in an easily stored text form. This isn't, however, human-readable program text because the line numbers are represented in hexadecimal.

The LOAD command will delete the program in memory and load in a program from your storage device.

Stopping Program Execution

You can stop the execution of the program or listing of the program by pressing the control-C key on the console. Additionally, you can pause in a program listing by pressing control-S and then pressing any key to continue.

Abbreviations and Blanks

You may use blanks freely within a program except that numbers, command keywords, and function names cannot have embedded blanks.

You may abbreviate all command keywords and function names, following each by a period. For instance, P., PR., PRI., and PRIN. all stand for PRINT. You may also omit the word "LET" in LET commands. The shortest abbreviations for all the keywords are given in the Table (page 46).

Note that, in some cases, the same abbreviation applies to different keywords. The interpreter is "smart" enough to identify the correct keyword for a particular situation. For instance, if the abbreviation P. appears at the beginning of a line, it can only mean PRINT. In a statement like A=P.(8), the P. only makes sense if it stands for PEEK.

Error Reports

There are only three error conditions in Tiny BASIC. The line containing the error is printed out with a question mark inserted at the point where the error is detected.

(1) "What?" indicates an error in a statement's syntax:

```
What?
260 LET A=B+3,
   C=(3+4?. X=4
```

(2) "How?" means that the statement in question is syntactically correct, but for some reason the command can't be carried out:

```
How?
310 LET A=B*C?+2
   where B*C is larger than
      2147483647
How?
380 GOTO 412?
   where line 412 does not exist
```

(3) "Sorry." means that the interpreter understands the statement and knows how to do it but lacks sufficient memory to accomplish the task.

Error Corrections

If you notice an error in your entry before you press RETURN, you can delete characters with the backspace (control-H) key or delete the entire line with control-X. To delete an existing program line, just type the line number and press RETURN.

Installation

Now, how do you get this wonderful piece of software running on your computer? Very easily, if you have a setup similar to mine. Installation on other systems should also be fairly easy if you have access to a 68000 assembler of some kind.

My setup is a Motorola MEX-68KECB Educational Computer Board (ECB) connected between my terminal and my CP/M system. The source code was assembled with the Quelo version 1.9 public domain 68000 cross-assembler for CP/M. (By the way, if you use this assembler, you will get 36 "trim16 address" errors, which is normal.) Tiny BASIC is then loaded into the ECB and executed at the cold start address of hex 900.

BASIC programs are saved and loaded by setting up an appropriate CP/M command before using SAVE or LOAD. For example (user input is underlined):

```
After a program is written,
   exit to the monitor:
   > BYE
Enter transparent mode:
   TUTOR 1.x> TM
Issue a PIP command to the CP/M
host:
   A> PIP PROGRAM.BAS=CON:
Exit transparent mode and do a
BASIC warm start:
   TUTOR 1.x> GO 904
Do the actual save:
   SAVE
```

The warm start is an entry point into the interpreter that will preserve any program you may have already entered.

Program LOADs are done similarly, except instead of PIP you must run a small program that will wait to receive a carriage return before sending the program to the ECB. Here is a sample program in Microsoft BASIC:

```
10 INPUT "Program to send?";F$
20 OPEN "I",1,F$
30 INPUT "Now exit Transparent
   Mode and do a LOAD.";Z$
40 WHILE NOT EOF(1):LINE IN-
   PUT #1,A$:PRINT A$:WEND
```

Admittedly, this way of LOADing and SAVEing is a fairly complex procedure, but it allows you to save your programs on disk while keeping the interpreter itself small. If your ECB isn't connected to another computer, you probably could change the AUXIN and AUXOUT subroutines to use the cassette interface. (I haven't tried it, though, so *caveat emptor!*)

For other 68000 systems, you will have to modify only the OUTC, INC, AUXOUT, AUXIN, and BYEBYE routines at the end of the interpreter program. In addition, you must put the address of the first unavailable memory location above BASIC into the location ENDMEM. BASIC programs are SAVEd in a form that can be stored as ASCII text and read back quickly by the 68000; if your storage device can't handle the present format or if you would like the program saved in a human-readable form, you need modify only the SAVE and LOAD subroutines.

One warning: I wrote the DIRECT and EXEC routines assuming that the interpreter itself would be somewhere in the first 64K of memory ($0 to $FFFF). If you move it above 64K, you will have to modify the EXEC routine and check the rest of the code carefully to make sure the addressing modes are correct.

Evaluation

I am quite pleased with how the interpreter turned out. Even though I added extra error checking, lower-case conversion, and more commands and extended the variable size to 32 bits, the whole thing still fits inside 3K of memory. I ran the Sieve of Eratosthenes benchmark program on this interpreter and on the Sherry Brother's CP/M tiny BASIC with the following results:

68000 at 4 MHz	Z80 at 4 MHz
2670 seconds	3000 seconds

Although I adjusted the results for the usual 10 iterations of the basic algorithm, I actually ran the program only for one iteration to keep running times within a practical limit. This tiny BASIC may not be a speed demon, but it does beat Applesoft and PET BASIC at running the Sieve benchmark. I should add that I compressed the Sieve program listing to the maximum for speed considerations; I normally use more spaces and some comments so that I can figure out later what the program was supposed to do!

Of course, many improvements can be made given more available memory. My Educational Computer Board has 32K of memory, so I probably will add such things as more variables, strings, and keyword tokenization. The last is a method used by most BASIC interpreters to compress keywords such as LET and PRINT into single bytes. This would greatly speed up the interpreter while using less memory to store the BASIC program.

Availability

By the time you read this, the interpreter source code and some example programs should be available on a couple of the RCP/M bulletin board systems in my area:

Meadowlark RCP/M (403) 484-5981

Edmonton RCP/M (403) 454-6093

The Edmonton RCP/M accepts both 300 and 1200 baud, but the Meadowlark system allows access to its CP/M area only at 1200 baud. Both systems run 24 hours a day.

The interpreter source code is known as TBI68K.AQM, which is a "squeezed" text file. If you don't have a MODEM7 type program and a way to unsqueeze this file, you can use these systems' LIST command to list out the source code while you capture it with a telecommunications program.

A short documentation file, TBI68K.DQC, and some sample programs, TBIPROGS.LBR, are also available. The latter is a CP/M library file, which contains several programs. You can list the library's contents with the LDIR command and extract individual programs using either the systems' XMODEM or LTYPE commands. The Quelo cross-assembler is also sometimes available on these systems under the names A68K.COM and A68K.DOC.

Although I'd prefer that you obtain the source code from one of the above sources, for $20 I can provide the code in the following forms: 8-inch CP/M SSSD diskette, 5-inch Osborne or Apple CP/M diskettes, or a paper listing.

If you find any bugs in the interpreter or have any questions, please write to me or contact me on the above RCP/M system.

DDJ

```
A.=ABS
C.=CALL
F.=FOR
GOS.=GOSUB
G.=GOTO
IF=IF
I.=INPUT
L.=LIST
LO.=LOAD
N.=NEW
N.=NEXT
P.=PEEK
PO.=POKE
P.=PRINT
REM=REMARK
R.=RETURN
R.=RND
R.=RUN
S.=SAVE
S.=SIZE
S.=STEP
S.=STOP
TO=TO
no keyword = LET
```

Table

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 194.**

An Enhanced ADFGVX Cipher System

by C. E. Burton

Cryptography is too important to be left to the NSA.

In another article, I described a public key system based on the RSA algorithm.[1] The major disadvantage of that algorithm was that it was not very fast, either in key generation or in encryption/decryption. However, if you need the security, you probably should use the RSA cryptography system or the DES cryptosystem.

Most information, however, has a time value associated with it. Thus, it is not necessary to use a RSA or a DES cryptography system on all messages. For example, if a military operation were going to be mounted within a month, we could encrypt messages relating to the operation using any cryptographic system that we can show to be unbreakable over that period of time (plus some "slop" for weather, delays, etc.).

Private key systems are fast and can be difficult to break. The cryptography system described in this article is based on a method developed by the Germans during World War I. David Kahn, in his book *The Code Breakers* (MacMillan, 1967), indicated that this field cipher system was probably the toughest then developed.[2] The original cipher was termed the ADFGX system, and it took Painvin, a French cryptoanalyst, over a month to decode the first batch of messages that the Germans encrypted.

The ADFGX system, however, has the same problem that the DES system or any other private key system has: the key must be known to both the encryptor and the decryptor. Muller-Schloer has suggested one way around this problem.[3] Having some way to send the key with the message would eliminate the logistics of key distribution. The question becomes: How can we do that without giving everything away? The answer is really simple! We encrypt the message with a private key system. Next we encrypt the private key, using a public key system, and append the encrypted form of the private key to the message.

We now have the best of both worlds. The private key system can encrypt/decrypt a message rapidly (in hardware or in software), and we can even change the key for every message. Because the private key is short, it does not take long to encrypt/decrypt it using a public key system.

This article describes an enhanced version of the ADFGX cipher system: the cipher, the character substitution method, the use of a key or password to do an irregular columnar transposition, a run-length data compression method, and some of the housekeeping functions performed in the encryption/decryption process. The substitution method contains a "card"-shuffling algorithm and a random number generator. The language used to write these routines is C.

Before we get into the details of my implementation, we should look at the original German ADFGX cipher system to get a feel for where we will be heading.

The ADFGX Cipher

The ADFGX cipher was named after the letters of the Morse code used in the transmission of the encrypted messages. Be-

---

Charles E. Burton, 13284 W. Utah Circle, Denver, CO 80228.

cause these particular characters are quite distinctive, the Germans felt that, even if noise partially garbled the message, these characters could not be confused. For those of you unfamiliar with Morse code, it represents the alphabet by a series of dots and dashes. The telegraph system sent and received messages translated into Morse code, and after Marconi developed the radio, Morse code was used over the air waves. The Morse code representation of the cipher letters is:

A .–
D –..
F ..–.
G ––.
X –..–

With these letters, the Germans could represent a scrambled alphabet in a 5 × 5 square matrix (leaving out one letter, say, Q). The substitution matrix might have looked like this:

	A	D	F	G	X
A	O	D	C	M	S
D	U	Y	X	L	H
F	T	A	N	W	Z
G	E	J	G	I	P
X	F	V	B	R	K

The next item required was a key, which is used to do a columnar transposition. We assign each letter in the key a column number. Then we sort the key alphabetically by character to get the column order for transmission. Let us choose a key: DOG MEAT. Squeezing out the blanks and alphabetically sorting the letters of the key, we get ADEGMOT. The sorted column order is 6/1/5/3/4/2/7:

D	O	G	M	E	A	T
1	2	3	4	5	6	7

becomes

A	D	E	G	M	O	T
6	1	5	3	4	2	7

Now, we need a message to see how all of this works. Let's use the message, "Meet me at the zoo at ten." After removing all of the spaces and punctuation, we have:

MEETMEATTHEZOOATTEN

Next we take each letter, in turn, and pass it through the ADFGX substitution table to get the row and column address (in the form of a row/column pair using the letters ADFGX). For example, the letter M would become AG (row A, column G):

M	E	E	T	M	E	A	T	T	H	E
AG	GA	GA	FA	AG	GA	FD	FA	FA	DX	GA

Z	O	O	A	T	T	E	N
FX	AA	AA	FD	FA	FA	GA	FF

We place these letter pairs in a table that has columns headed by the key letters (DOG MEAT). The key transposition table is loaded a row at a time:

D/1	O/2	G/3	M/4	E/5	A/6	T/7
A	G	G	A	G	A	F
A	A	G	G	A	F	D
F	A	F	A	D	X	G
A	F	X	A	A	A	A
F	D	F	A	F	A	G
A	F	F				

Message
M E E T
M E A
T T H E
Z O O
A T T E
N

Finally, we pull the columns out and rearrange them as rows in the previously determined order (6/1/5/3/4/2/7):

6	/	1	/	5	/	3	/
AFXAA		/AAFAFA		/GADAF		/GGFXFF	/

4	/	2	/	7
GAAA		/GAAFDF		/FDGAG

The message is now encrypted and ready for transmission (without the "/"s). Note that, although the message length has doubled, sending the encrypted message in this fashion gave the Germans benefits that outweighed this disadvantage: the distinctive nature of the letters transmitted. Certainly they could have run the transposed message back through the substitution table to get a message of length equal to the original:

AF	XA	AA	AF	AF	AG	AD	AF	GG	FX
C	F	O	C	C	M	D	C	I	Z

FF	AG	AA	AG	AA	FD	FF	DG	AG
N	M	O	M	O	A	N	L	M

But this would mean using the whole alphabet rather than five letters and increase the chance of error introduction at two additional stages (one at the sending end and one at the receiving end). Remember, there were no computers during World War I. On the other hand, had the Germans taken this extra step, in all likelihood it would have taken much longer to break the cipher.

The decryption process follows the above steps in the reverse order: the received letters are put into the key table in their original columns; character pairs are read out from the rows; and these pairs are used in the substitution table to get the original message. Note that the receiver must know two pieces of information to decipher the message: the key and the substitution table.

The letter V was added to the original ADFGX formulation to allow all 26 letters and 10 digits to be used; the substitution table produced by the ADFGVX cryptosystem was a 6 × 6 matrix. Again the 36 characters were put into the substitution matrix in a random order. The same procedure to encrypt/decrypt was used.

Enhanced ADFGVX

I have enhanced the basic ADFGVX system in several areas. First, because it is useful to be able to encrypt/decrypt text, as well as data, programs or anything else, we must be able to deal with byte-wide data. A substitution table that can handle characters from 0 to 255 (00 to FF hex) must be a 16 × 16 matrix.

Second, because the "messages" will be longer than a few "words," one columnar transposition will not do; we must block the "message" into something more suitable. Each block will be passed through the algorithm for encryption and decryption. Because each letter of the message will be expanded to two letters by the substitution step, a transposition table with an even number of characters is desirable (but not mandatory). To make the transposition more difficult to figure out, the table must not end on an even square, such as 25 or 36. That way the messages are not blocked into a form that provides information about the ciphering method. Foster has shown that the use of incomplete or irregular columnar transposition makes the deciphering task much harder.[4]

Third, some simple data compression, especially for text, would reduce storage capacity and transmission time. We also will do the backward substitution operation at the end so that the message size does not double—we have a computer that should not make the mistakes that a human might.

A couple of additions could make this implementation even more difficult to attack and break. First, make the incomplete columnar transposition variable. For example, the previous key transposition table had seven columns. Say that we need between 4 and 10 rows in the transposition table before transposing. We could fill the transposition table with an even number of characters from the substitution operation so that 28 (7 × 4) to 70 (7 × 10) of these characters are loaded into the transposition table before each column transposition operation. For instance, choose the repeating character count sequence: 46, 62, 30, 54, and 38. On the first pass, we load the first 23 (46 / 2) row/column pairs into the transposition matrix; on the next pass, we load the next 31 row/column pairs; and so on, cycling through the five character count sequences. The main problem with this variability option is that it involves another piece of information that both the sending and receiving parties must know.

The second addition, which involves a double irregular columnar transposition, again requires that another piece of information be available to both parties. To apply this option, we do everything as usual, using one key to do the first transposition, then we use a second key to form a second transposition table (just as we did with the first key); we fill the second transposition table with the output of the first transposition table. Next we perform the second transposition, and finally we do the backward substitution to get the encrypted message. The technique of double transposition has been used as a military cipher system ("U.S. Army Transposition") because of the difficulty in deciphering the result. Foster indicated that he knew of no method to crack the double transposition technique by hand or by microcomputer.

Figures 1 and 2 (page 51) present the structure charts for the encryption and decryption processes, respectively. Rather than have each of the boxes represent a separate routine, I have combined many of these routines together. Those boxes with an asterisk (*) above them indicate separate modules; their names are given below the box. Those boxes without the special notation appear in their parent box. The looping arrow at the bottom of a box indicates repetition, and the diamond indicates a decision (the child may or may not be called, depending on the outcome of the decision).

We now are ready to discuss the various procedures. Even though I designed the software in a top-down fashion, I will introduce the modules in a somewhat bottom-up fashion.

Substitution Matrix

We must generate a 16 × 16 matrix (256-entry table) to get our character substitutions. Rather than use characters (e.g., ADFGVX or equivalent) to represent the rows or columns, we will use the row and column numbers (0 to 15). We must fill the matrix positions in a random order. A simple card-shuffling routine can be used to do this. We also must overcome one minor problem: how can we find the "card" value, given its position in the "deck," and how can we find the position in the "deck" of a "card" of a given value. We need these answers to perform both the forward and backward substitution procedures.

The routines that handle the matrix filling are shuffle( ) and ran( ). shuffle( ) takes a linear array (having 256 character positions) and fills the entries in linear order. Then it passes through the "deck" and swaps the current "card" with another card in a random position as determined by a call to ran( ).

Once the "deck" has been shuffled, it is scanned to find the position of each "card" within it. The result is two linear arrays:

position[I] – holds the position of card I within the deck
value[J] – holds the value of the Jth card in the deck

Given position[ ], we can find the row and column position of a character (or perform the forward substitution) by:

row = position[char] / 16
column = position[char] % 16
(i.e., position[char] mod 16)

Given value[ ], we can find the character occupied by a row and column (or perform the backward substitution) by:

char = value[16 * row + column]

Note that we are using a linear array to represent a matrix.

ran(MAX) returns a random integer between 0 and (MAX−1). We can alter the random number generation sequence by changing the initial value of IY to be greater than 0 and less than 2796203. One possibility might be to declare IY as a global variable and to ask for a seed value in the main program. Remember, however, both the sender and the receiver *must* have the same starting seed to produce identical substitution tables. For the same reason, the substitution table must be generated before any other use is made of ran( ).

Irregular Columnar Transposition

From the ADFGX example, we found that the transposition involved several steps. First, given a password or key, we must sort the password on a character-by-character basis. This job is performed by pwsort( ). In pwsort( ), I have limited the length of the password to values greater than 4 and less than 11. If the password is less than 5, the routine gives an error message and exits to the operating system. If the password is greater than 10, the password is truncated to the first 10 characters. You can alter the routine to use other password lengths. Once the password is validated, we sort

the characters individually using a simple bubble sort.

The array pwcolumnorder[ ] holds the sorted order of the transposition columns. This sort arranges the columns in descending (largest to smallest) order; i.e., DOGMEAT becomes TOMGEDA. Variations could arrange the sorted columns in ascending order, inside-out, outside-in, and so on, as long as the sender and receiver agree. pwcolumnorder[0] provides the first column and pwcolumnorder[pwlen−1] gives the last column to extract.

Second, we must determine the size of the transposition table. This is done in the main( ) routine. I have picked a length to ensure the columns are irregular by using:

```
blklen = pwlen * pwlen + pwlen / 2
blklen + = odd(blklen) ? 1 : 0
```

(i.e., if blklen is odd then increment blklen else leave it alone)

This makes the transposition matrix (pwlen + 1) rows and pwlen columns, with the last row incomplete. Note that I have made the number of entries even so that row/column pairs are not broken.

Finally, we must fill the transposition matrix and perform the transposition. For encryption, we must enter the information from the forward substitution matrix into the transposition matrix by rows and in the substitution row/column order. Once the transposition matrix is filled, the routine transpose( ) is called to pull out the columns in sorted order. These new row/column pairs are passed through the backward substitution and then saved as the encrypted message.

For decryption, we must enter the information from the forward substitution matrix into the transposition matrix by columns and in the sorted column order. Once the transposition matrix is full, the dual routine untranspose( ) is called to pull out the rows in their original order. These new row/column pairs are run through a backward substitution and saved as the plain text message.

Actually, the transposition matrix is filled in a linear order, and the transpose( ) and the untranspose( ) routines take care of obtaining the proper ordering of the rows and columns. The transposition matrix-filling operation produces the message blocking that I mentioned previously.

One minor problem with this operation is the handling of messages that are not an even multiple of blklen (the usual case). A simple solution is to pad the end of the message with random characters. However, to make sure that the decryption returns the original message without the padding, we must arrange for the length of the original plain text message to be found and saved at the beginning of the encrypted file. I have done this by using a union called filelen. It is used identically to an "equivalence" in Fortran. Because the C library routine putc( ) handles only bytes, I built the union to hold a long integer (the file length) and an equivalent-sized character array. This construction allowed me to load/save the file length from/to a file in a byte-at-a-time mode.

Data Compression/Expansion

A useful program is available on most CP/M-based bulletin board systems. Called SQ/USQ (squeezer), it allows files to be compressed and expanded so that you can save disk space. Richard Greenlaw wrote the software, which uses two types of compression: run length compression and Huffman coding. The achievable compression ranges from 3% to 70%, depending on the makeup of the file. I have borrowed and modified the run length compression routine to perform my data compression. The expansion routine that I used is modeled after the compression routine. (Although Greenlaw's USQ program has a more efficient expansion routine, I chose to do my expansion differently.)

What makes these compression/expansion routines interesting is that they are good examples of state machine routines. Figures 3a and 3b (below) show the state diagrams for the compress( ) and expand( ) routines, respectively. These routines show how the switch-case construct can be used to advantage in representing the states. In the figures, the bubbles represent the states and the arrows represent the state transition conditions. Similar directed graphs are used in electrical engineering to represent logic circuit operations.

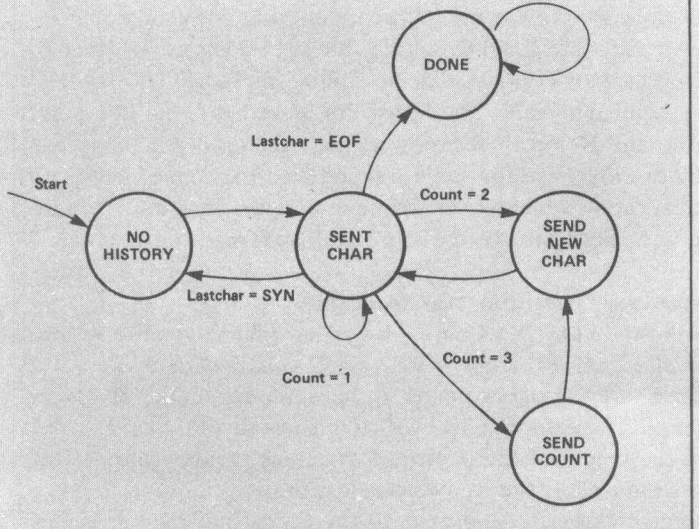


Figure 3a
COMPRESS( ) State Diagram

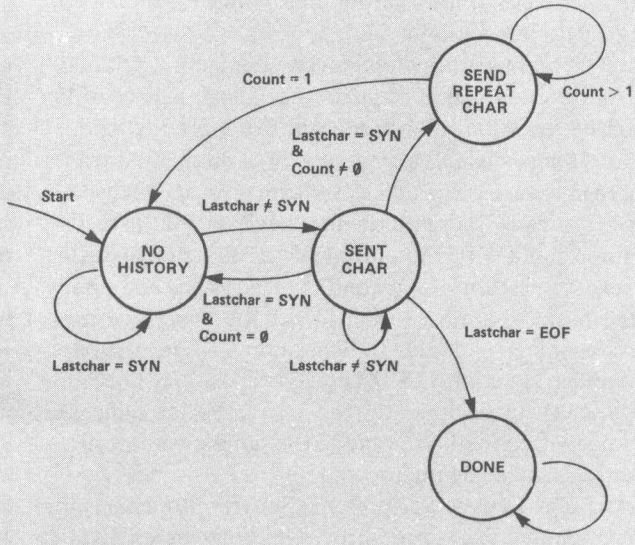


Figure 3b
EXPAND( ) State Diagram

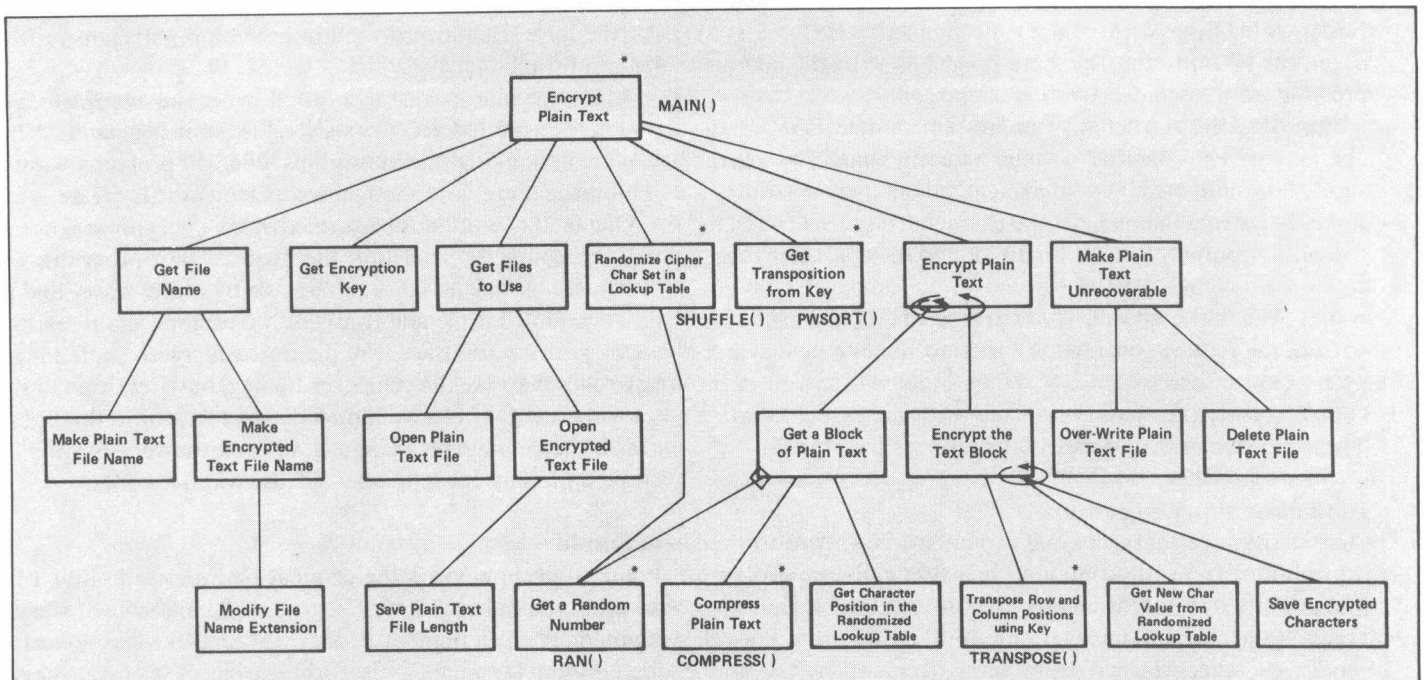


Figure 1
Encrypt Structure Chart

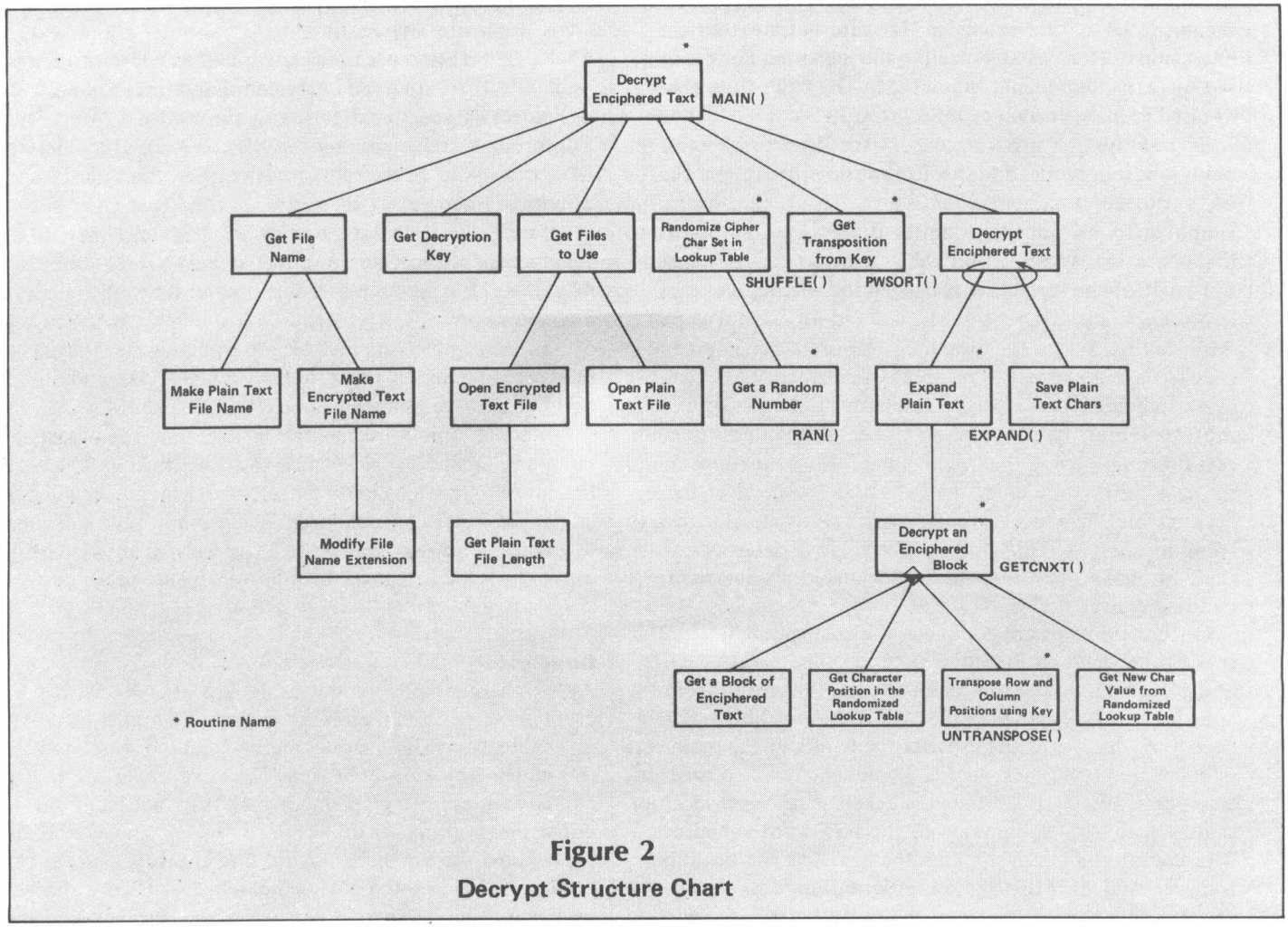


Figure 2
Decrypt Structure Chart

Software dealing with control mechanisms also uses state diagrams. Comparing the state diagrams with the routines provides an easy understanding of the code.

Basically, the run length compression routine looks at the character stream. If three or more adjacent characters are the same, the routine counts the like characters and substitutes a three-byte string that incudes the character, a special character indicating compression has occurred, and the character repeat count; i.e., <char> SYN <count>. To handle the circumstance where the special character occurs in the character stream, the routine generates the special character and a character count of zero; i.e., SYN <0>. Finally, if an end of file (EOF) occurs, the routine responds with an EOF thereafter. The expansion operation essentially reverses the process.

Housekeeping Operations

One of the nice features of C is that it allows you to read information from the command line that calls the program into execution. I have used this capability in both the encryption and the decryption programs. With either program, you specify the execution program name, followed by the plain text filename, followed by the password or key. Most of the early housekeeping involves picking up the filename and password.

Once the plain text filename is retrieved, the encrypted filename is generated from the plain text filename by modifying the filename extension. If the plain text filename extension has at least one character, a Y is substituted for the second letter of the extension. If there is no extension, an extension of .YYY is appended to the filename. For example, if the plain text filename is MESSAGE.TXT, then the encrypted filename becomes MESSAGE.TYT.

The encryption program checks for the existence of the plain text source file, and the decryption program checks for the existence of the encrypted source file. If the source file cannot be found and opened for reading, the programs exit to the operating system after generating an error message. Similarly, if the target file is opened for writing and a problem occurs, a message is generated and an exit takes place. After taking care of the files, the password is retrieved and is checked using pwsort( ), as already described.

As indicated previously, the length of the plain text file must be found. The encryption process determines the plain text file length using the fseek( ) and ftell( ) functions found in the C library. It saves the length at the head of the encrypted file. The decryption process reads the file length from the encrypted file and uses the value to determine when the end of the plain text file has occurred, exclusive of the padding characters.

Once these steps are complete, the encryption/decryption process proceeds as already explained. The arrays BUF1[ ] and BUF2[ ] hold the row/column pairs that result from the substitution process. These arrays contain a block of message, the length of which is twice the length of the plain text or encrypted text block. BUF1[ ] holds the forward substitution pairs; BUF2[ ] contains the result of the transposition that is ready for submission to the backward substitution. The encryption program uses these arrays in the main( ) routine, and the decryption program uses them in the getcnxt( ) routine. Note that these arrays are large enough for the largest password used in the columnar transposition (plus a little bit).

After the encryption/decryption process is complete, you no longer need to keep the source file, so it is erased. When you are debugging these programs, you will probably want to inhibit the unlink( ) function that performs this file deletion.

One final operation is performed in the encryption program prior to source file deletion. Because certain programs can "unerase" or look at a disk on a sector-by-sector basis, and we do not want to leave any traces of the original plain text file, we must overwrite the entire plain text file with something. I have chosen to use the character F6 hex, but any byte character will do. Again you will probably want to inhibit this operation while debugging. (There is no need to overwrite the encrypted file, since it is "gibberish" anyway.)

Example

Figure 4a (page 55) is the original plain text file that I encrypted and subsequently decrypted. I have shown the encrypted text file (Figure 4b, page 55) and the decrypted text file (Figure 4c, page 56) in both hex and ASCII representations; this is essentially the same representation used by the D command under DDT. Note that the encrypted file's first four bytes provide the length of the original plain text file. It is after that point (from the fifth byte to the end of the file) that the encrypted text of the original plain text file occurs. The plain text file has several repeated character sequences to demonstrate data compression.

To duplicate my results, let's call the plain text file ADFGVX.TXT and use the password GERMANS. To encrypt ADFGVX.TXT, you type in the command line:

```
ENCRYPT adfgvx.txt germans
```

This produces the encrypted file ADFGVX.TYT and destroys ADFGVX.TXT. To decrypt the encrypted file, you type the command line:

```
DECRYPT adfgvx.txt germans
```

This produces the decrypted file ADFGVX.TXT and erases ADFGVX.TYT. You enter the command lines at the operating system prompt; e.g., at the >A under CP/M. You must compile, assemble, and link the programs so that the encryption program has a name of ENCRYPT.COM and the decryption program has the name DECRYPT.COM.

Note that this implementation does not remove spaces, punctuation, and so on. Because the substitution table covers the entire byte-wide character set, what you put in will come out at the other end unaltered. You do not have to squeeze together then break up the text, as we had to do with the original German cipher. In addition, you can encrypt/decrypt any file.

Conclusion

Although this cipher is not as secure as a RSA or a DES cryptosystem, its character set scrambling makes it indistinguishable from these other methods. According to Kahn, this cipher is not easily broken. The advantages are that the cipher is simple to implement and fast in execution. The major disadvantage is that both the sender and the receiver must know the key or password and the substitution table ordering. As I pointed out, you can overcome this difficulty by using a public key system to encrypt the key and table

First of all, let's generate a string of repeated characters:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
```

These sequences will be converted by the data compression routine to the following three character representation:

```
<x> SYN <52> = 78 16 34  (in hex)
<Y> SYN <52> = 59 16 34  (in hex)
```

Notice that we are not restricted to upper case letters and digits only. This implementation does not have to remove spaces and punctuation. Thus, all ASCII characters (0 to 127) can be used, as well as the byte values from 128 to 255.

The encrypted file will not have a one-to-one correspondence between these characters and those produced by the encryption process, i.e. letter frequency analysis will not help. This result is due to the transposition, which may replace an "a" in one case by a "Y" and in another case by a "*", etc.

Figure 4a

Plain Text to be encrypted. File Name is ADFGVX.TXT.

```
80 04 00 00 56 FA B0 CD  B7 AC 8D EB 32 6C 67 59   ....V.......2lgY
F7 9E 20 9E 4B A0 8D 26  20 25 B0 75 5C 83 6F 60   .. .K..& %.u\.o`
09 AC 28 46 E1 32 26 B9  FE 0B 82 5C 5C 6D 0B 20   ..(F.2&....\\m.
9E 20 0B 2C 7C 65 92 9E  3E F9 16 F6 0B 3D 54 54   . .,|e..>....=TT
B4 79 D7 E8 9D E6 C2 2B  77 27 20 41 5E 2C AE 7A   .y.....+w' A^,.z
C2 F2 0A 65 F7 A7 C2 31  31 B3 D2 20 54 F2 AC 8D   ...e...11.. T...
A8 53 89 F7 F3 0D 54 C6  CD 33 31 0C 31 25 8D 8D   .S....T..31.1%..
37 FB 53 31 CF 48 20 00  65 18 A0 31 57 83 DF 0B   7.S1.H .e..1W...
81 20 AA 20 7A FB AB BD  0B FA 5C A0 55 62 3E 68   . . z.....\.Ub>h
20 65 90 82 C2 FB 5C 56  44 0B 35 C4 20 52 89 31    e....\VD.5. R.1
9E 65 6E 65 46 4A 00 35  57 57 EB ED 5F CF FF 6D   .eneFJ.5WW.._...m
20 F7 F7 0B 8D 5D BD 87  87 B1 25 B4 91 42 AC 5F    ....]....%..B._
BA 35 E2 2C 86 15 0B 0B  FF BD FB 62 E4 57 8D 2C   .5.,.......b.W.,
8D E8 8D 9E 5E 6B CA 04  C2 34 F7 25 D2 42 6D 55   ....^k...4.%.BmU
AC 91 E8 3D 62 38 08 0D  31 57 00 33 0E 79 92 8D   ...=b8..1W.3.y..
29 C1 6D B7 E6 FB C2 AC  90 FB 00 53 67 90 12 CD   ).m........Sg...
0E 04 58 D2 0B 20 FF 4F  CA 36 20 CF F7 25 04 AC   ..X.. .O.6 ..%..
6D 39 BB 91 86 90 FB 38  79 20 31 57 20 C2 09 79   m9.....8y 1W ..y
18 BD 81 40 25 31 C2 52  65 A8 D2 9E AA 53 65 C6   ...@%1.Re....Se.
5F 90 71 F7 77 0D 8E 57  AC 33 5C 6D 57 F7 40 05   _.q.w..W.3\mW.@.
20 84 FB 71 BD 41 9E ED  65 FB 20 9E BD 9E 62 40    ..q.A..e. ...b@
7F AB 1B FF 35 F7 49 BD  0B AE ED 6D 3C 62 65 AA   ~...5.I....m<be.
20 3E 20 CD C2 FB 38 E4  44 0B 9D DD 20 31 8D FB    > ...8.D... 1..
0B 6C AA 48 35 C7 93 23  25 25 09 C9 CB 63 CD B7   .l.H5..#%%...c..
CD 8B A0 8D 70 69 09 82  0E 3C 23 A5 6F 20 ED ED   ....pi...<#.o ..
20 DF 0B 32 0B 9E 31 FF  74 35 FB 3F 20 61 8D C7    ..2..1.t5.? a..
0C 6D 9E 31 E2 D2 58 6C  8D 43 FB FB 8D 16 32 40   .m.1..Xl.C....2@
8D 31 65 61 C2 5C 85 C7  70 29 FA 20 90 BD CF 8D   .1ea.\..p). ....
2C B0 C6 0D 8D 65 69 25  E2 44 EA 63 20 8B 0F 31   ,....ei%.D.c ..1
20 A2 A6 0B 48 A4 0E 8B  C2 41 20 AC A8 7A C6 21    ...H....A ..z.!
57 9D 57 52 84 69 6D 80  0B 20 20 63 75 8D 87 BD   W.WR.im..  cu...
F8 49 09 0B C6 20 20 31  BD D2 C9 44 68 A0 20 CD   .I...  1...Dh. .
55 CE 5C 31 C7 31 CF 20  92 1C F7 F8 20 40 75 D3   U.\1.1. .... @u.
7A 20 20 5C 65 C4 00 A0  3C F7 20 65 20 EC 75 26   z  \e...<. e .u&
44 6E 0C 31 57 18 86 17  25 20 FB 9D 62 8D 6F 0D   Dn.1W...% ..b.o.
62 20 5A C2 74 2B 53 AC  20 3F 17 A5 25 A0 F2 62   b Z.t+S. ?..%..b
34 0B 97 15 62 F7 20 9C  00 20 9D C6 20 CC 00 CD   4...b. .. .. ...
56 65 E4 E7 6E 40 7A 4A  0C FF 65 04 68 FA 2B 1C   Ve..n@zJ..e.h.+.
75 6F 57 57 73 00 E0 FB  5C CD FA 1F C6 18 8D 0E   uoWWs...\.......
F7 87 87 0C 20 65 9D 9F  C2 E0 31 2B 20 20 F8 1A   .... e....1+  ..
```

Figure 4b

Encrypted Text using the Plain Text in Figure 4a. The first four bytes (80 04 00 00) define the Plain Text file length (00 00 04 80) in bytes. Usage: ENCRYPT ADFGVX.TXT GERMANS

```
20 20 20 20 20 20 20 20  46 69 72 73 74 20 6F 66
20 61 6C 6C 2C 20 6C 65  74 27 73 20 67 65 6E 65
72 61 74 65 20 61 20 73  74 72 69 6E 67 20 6F 66
20 72 65 70 65 61 74 65  64 20 63 68 61 72 61 63
74 65 72 73 3A 0D 0A 0D  0A 20 20 20 20 20 20 20
20 20 20 20 20 20 20 78  78 78 78 78 78 78 78 78
78 78 78 78 78 78 78 78  78 78 78 78 78 78 78 78
78 78 78 78 78 78 78 78  78 78 78 78 78 78 78 78
78 78 78 78 78 78 78 78  78 78 78 0D 0A 20 20 20
20 20 20 20 20 20 20 20  20 20 20 59 59 59 59 59
59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 59
59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 59
59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 0D
0A 0D 0A 20 20 20 20 20  20 20 20 20 54 68 65 73
65 20 20 73 65 71 75 65  6E 63 65 73 20 20 77 69
6C 6C 20 62 65 20 63 6F  6E 76 65 72 74 65 64 20
62 79 20 74 68 65 20 20  64 61 74 61 20 20 63 6F
6D 70 72 65 73 73 69 6F  6E 20 0D 0A 20 20 20 20
20 20 20 20 20 72 6F 75  74 69 6E 65 20 74 6F 20
74 68 65 20 66 6F 6C 6C  6F 77 69 6E 67 20 74 68
72 65 65 20 63 68 61 72  61 63 74 65 72 20 72 65
70 72 65 73 65 6E 74 61  74 69 6F 6E 3A 0D 0A 0D
0A 20 20 20 20 20 20 20  20 20 20 20 20 20 20 3C
78 3E 20 53 59 4E 20 3C  35 32 3E 20 3D 20 37 38
20 31 36 20 33 34 20 20  28 69 6E 20 68 65 78 29
0D 0A 20 20 20 20 20 20  20 20 20 20 20 20 20 20
3C 59 3E 20 53 59 4E 20  3C 35 32 3E 20 3D 20 35
39 20 31 36 20 33 34 20  20 28 69 6E 20 68 65 78
29 0D 0A 0D 0A 20 20 20  20 20 20 20 20 20 20 4E 6F
74 69 63 65 20 20 74 68  61 74 20 77 65 20 61 72
65 20 6E 6F 74 20 72 65  73 74 72 69 63 74 65 64
20 74 6F 20 75 70 70 65  72 20 63 61 73 65 20 6C
65 74 74 65 72 73 20 20  61 6E 64 20 0D 0A 20 20
20 20 20 20 20 20 64 69  67 69 74 73 20 20 6F 6E
6C 79 2E 20 20 20 54 68  69 73 20 69 6D 70 6C 65
6D 65 6E 74 61 74 69 6F  6E 20 64 6F 65 73 20 6E
6F 74 20 68 61 76 65 20  20 74 6F 20 20 72 65 6D
6F 76 65 20 0D 0A 20 20  20 20 20 20 20 20 20 20
73 70 61 63 65 73 20 20  61 6E 64 20 70 75 6E 63
74 75 61 74 69 6F 6E 2E  20 20 20 54 68 75 73 2C
20 20 61 6C 6C 20 41 53  43 49 49 20 63 68 61 72
61 63 74 65 72 73 20 28  30 20 20 74 6F 20 0D 0A
20 20 20 20 20 20 20 20  20 31 32 37 29 20 63 61
6E 20 62 65 20 75 73 65  64 2C 20 61 73 20 77 65
6C 6C 20 61 73 20 74 68  65 20 62 79 74 65 20 76
61 6C 75 65 73 20 66 72  6F 6D 20 31 32 38 20 74
6F 20 32 35 35 2E 0D 0A  0D 0A 20 20 20 20 20 20
20 20 20 54 68 65 20 20  65 6E 63 72 79 70 74 65
64 20 66 69 6C 65 20 77  69 6C 6C 20 6E 6F 74 20
68 61 76 65 20 61 20 6F  6E 65 2D 74 6F 2D 6F 6E
65 20 63 6F 72 72 65 73  70 6F 6E 64 65 6E 63 65
20 0D 0A 20 20 20 20 20  20 20 20 20 62 65 74 77
65 65 6E 20 74 68 65 73  65 20 63 68 61 72 61 63
74 65 72 73 20 61 6E 64  20 74 68 6F 73 65 20 70
72 6F 64 75 63 65 64 20  62 79 20 74 68 65 20 65
6E 63 72 79 70 74 69 6F  6E 20 0D 0A 20 20 20 20
20 20 20 20 70 72 6F 63  65 73 73 2C 20 20 69 2E
65 2E 20 6C 65 74 74 65  72 20 66 72 65 71 75 65
6E 63 79 20 61 6E 61 6C  79 73 69 73 20 77 69 6C
6C 20 6E 6F 74 20 68 65  6C 70 2E 20 20 54 68 69
73 20 0D 0A 20 20 20 20  20 20 20 20 20 20 72 65
73 75 6C 74 20 69 73 20  64 75 65 20 74 6F 20 74
68 65 20 74 72 61 6E 73  70 6F 73 69 74 69 6F 6E
2C 20 20 77 68 69 63 68  20 6D 61 79 20 72 65 70
6C 61 63 65 20 61 6E 20  22 61 22 20 0D 0A 20 20
20 20 20 20 20 20 69 6E  20 6F 6E 65 20 63 61 73
65 20 62 79 20 61 20 20  22 59 22 20 61 6E 64 20
69 6E 20 61 6E 6F 74 68  65 72 20 63 61 73 65 20
62 79 20 61 20 22 2A 22  2C 20 65 74 63 2E 0D 0A
1A
```

```
        First of
 all, let's gene
rate a string of
 repeated charac
ters:....
        xxxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxx..
            YYYYY
YYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYY.
...         Thes
e  sequences  wi
ll be converted
by the  data  co
mpression ..
     routine to
the following th
ree character re
presentation:...
.           <
x> SYN <52> = 78
 16 34  (in hex)
..
<Y> SYN <52> = 5
9 16 34  (in hex
)....          No
tice  that we ar
e not restricted
 to upper case l
etters  and ..
       digits  o
nly.   This impl
ementation does
not have  to  re
move ..
spaces  and punc
tuation.   Thus,
  all ASCII char
acters (0  to ..
         127) ca
n be used, as we
ll as the byte v
alues from 128 t
o 255.....
   The  encrypte
d file will not
have a one-to-on
e correspondence
 ..         betw
een these charac
ters and those p
roduced by the e
ncryption ..
     process,  i
.e. letter frequ
ency analysis wi
ll not help.  Th
is ..          re
sult is due to t
he transposition
,  which may rep
lace an "a" ..
       in one ca
se by a "Y" and
in another case
by a "*", etc...
.
```

Figure 4c

Decrypted Text using the Encrypted Text in Figure 4b.
Usage: DECRYPT ADFGVX.TXT GERMANS

then appending this information to the encrypted message.

In everyday use, this cipher is probably adequate. You can make it more secure by using the additions that I have suggested. Remember, however, that it all depends on the time value of the information that you are securing. If you want to be sure that unauthorized persons cannot decipher your information for decades, you should use a RSA or a DES cryptosystem. For shorter periods, this enhanced ADFGVX cipher should be effective: until the discovery of the DES and RSA methods, it was claimed to be one of the "best" ciphers around.

References

[1] C. E. Burton, "RSA: A Public Key Cryptography System, Parts 1 and 2," *DDJ*, March and April 1984.

[2] D. Kahn, *The Code Breakers*, Macmillan Book Co., 1967.

[3] C. Muller-Schloer, "A Microprocessor-based Cryptoprocessor," *IEEE MICRO*, vol. 3, no. 5, October 1983, pp. 5-15.

[4] C. C. Foster, *Cryptoanalysis for Microcomputers*, Hayden Book Co., 1982.

DDJ

(Listings begin on next page)

I would like to thank John K. Taber, the reviewer, for suggesting three improvements to this cipher system: a better random number generator, a more thorough data compression technique, and an alternate padding method.

Mr. Taber first pointed out that my pseudo-random number generator has a fairly small period (about 2,796,203) as pseudo-random number generators go. The short period makes possible a concerted attack on the cipher. Because there are 256! ways to fill the 16 × 16 array, and I am using only a small fraction of them (based on the period), he suggested a random number with a period of 2 ** 64, or possibly 2 ** 128, as being safe for the forseeable future. Note, however, that the ran( ) function is a stand-alone module and can be replaced by a random number generator of the user's choice, as long as the interface remains unchanged. The reader might be interested in knowing that Knuth has devoted an entire chapter to random numbers, their generation, and their testing in his book on seminumerical algorithms.

Other ways around this problem might be to use truly random physical phenomena to generate the random numbers. For example, the Allies during World War II used the X-System: a mercury vapor rectifier tube that generated wideband thermal noise. The output was sampled into six levels of equal probabilities at a 20 msec rate using nonuniform quantization. These random numbers, recorded onto plastic disks similar to phonograph records, were used to scramble voice messages. (See D. Kahn, "Cryptology and the Origins of Spread Spectrum," *IEEE Spectrum*, vol. 21, no. 9, September 1984, p. 74.)

But remember, you have to contend with the problem of key distribution: sending these random numbers, as well as the irrregular transposition key, to the receiver. The value of the message and its life time will dictate the extremes to which you must go to protect your data.

Mr. Taber next suggested that better data compression schemes exist. Because I used this cipher in particular to encrypt data files that consisted of numerous repeated character strings, the run length code suited my purposes. As I pointed out in the article, Mr. Greenlaw has an excellent, public domain data compression package (SQ/USQ). Readers interested in Huffman compression should obtain a copy of Mr. Greenlaw's source code from your local RCP/M bulletin board system. You could certainly use it to enhance my compression scheme.

One reason that I did not use the Huffman algorithm was because you have to scan the text twice to obtain the compression. You use the first scan to obtain the frequency information and to assign the compression codes. The second scan compresses the data using the compression codes and saves the result on disk. To uncompress the data, you must retain the character/compression code substitution pairs (or encrypt the compression codes along with the compressed data). If the codes are stored in a known position, they could be used to break the cipher. On the other hand, because these Huffman characters are variable length codes, they might make an attack on the cipher more difficult. Anyway, I did not have the need to go to such lengths.

Finally, Mr. Taber suggested an alternate padding method. He recommended the use of an array at least as long as the padding needed, say, 256 bytes. The pad array is filled with some known pattern (byte by byte). Then each byte of the clear text is read and exclusive or'ed with the next element of the pad array in a circular fashion; i.e., when the 256th byte is used, the process starts over with the first byte. When padding is required, the pad bytes are drawn from the pad array. This method, or another comparable method, avoids the weakness of having a known range of bytes or known bytes in the final block.

I do not believe that most people will have to make any of these modifications; the cipher (as written) should be safe enough for most uses. For example, MSDOS/PCDOS, unlike MP/M-II and CP/M-3, does not allow for making files read-only. Thus, users could use this cipher to encrypt those sensitive personnel files contained on a PC/XT that is used by several employees.

With respect to irregular transposition, Mr. Taber pointed out that C. A. Deavours, editor of *Cryptologia*, has written a BASIC program to help determine the cipher "breakpoints" (i.e., the possible tops and bottoms for given irregular transposition dimensions). The program uses an "odds in favor" approach from statistics to measure when the right breakpoints are determined. Deavours claims that he can break the cipher within five to ten minutes using this technique—starting from scratch. (Deavours' article describing the technique, along with the program, appears in *Cryptologia*, vol. 5, no. 4, October 1981, pp. 247-251.) However, when you combine the irregular transposition cipher with the enhanced ADFGVX cipher, the job of breaking the code becomes orders of magnitude more difficult!

ADFGVX Cipher System (Text begins on page 48)

Listing One

```
/*##########################################################################
###                                                                      ###
###        Copyright 1983, Charles E. Burton, Denver, Colorado           ###
###                                                                      ###
### All rights reserved.  Permission granted to use this software for    ###
###   personal, non-commercial purposes only.                            ###
###                                                                      ###
##########################################################################*/

/*
 * PROGRAM NAME:  SHUFFLE.C
 * PURPOSE:  Card shuffling routine --
 *                Shuffles a "deck" of cards.  The number of cards is NOCARDS.
 *           Random number routine --
 *                Generates a random interger
 *
 * LANGUAGE:  C
 * AUTHOR:  CEB
 * USAGE:  SHUFFLE(POSITION,VALUE)
 *              POSITION [I] -- returns as the position of Card #I in the Deck.
 *              VALUE [J] -- returns as the value of the Jth Card in the Deck.
 *         RAN(MAX)
 *              MAX -- integer specifying the maximum random integer that can
 *                     be returned.
 * ARRAYS USED:  POSITION[NOCARDS],VALUE[NOCARDS]
 * EXTERNALS:
 * UPDATE HISTORY:  INITIAL RELEASE -- 11/25/83 CEB
 */

/* Card shuffling routine */

shuffle(position,value)
     char position[]; /* POSITION [NOCARDS] on entry */
     char value[]; /* VALUE [NOCARDS] on entry */

     {
     char temp;
     int i,j;

     for (i=0; i < NOCARDS; i++) /* initialize CARDS values */
          value [i] = i;
     for (i=0; i < NOCARDS; i++) /* shuffle */
          {
          j = ran(NOCARDS); /* swap index */
          temp = value [i]; /* swap CARDS values */
          value [i] = value [j];
          value [j] = temp;
          }
     for (i=0; i < NOCARDS; i++) /* get positions of cards */
          position [ value [i] ] = i;
     }

/*************************************************************************/

/*
   Random number generator --
   adapted from the FORTRAN version
   in "Software Manual for the Elementary Functions"
   by W.J. Cody, Jr and William Waite.
 */

ran(max)
     int max;

     {
     static long int iy = 100001;

     iy *= 125;
     iy -= (iy/2796203) * 2796203;
     return ((int) (max * iy / 2796203));
     }
```

End Listing One

(Listing Two begins on page 60)

ADFGVX Encryption (Listing Continued, text begins on page 48)

Listing Two

```
/*##########################################################################
###                                                                      ###
###        Copyright 1983, Charles E. Burton, Denver, Colorado           ###
###                                                                      ###
### All rights reserved.  Permission granted to use this software for    ###
###   personal, non-commercial purposes only.                            ###
###                                                                      ###
##########################################################################*/

/*
 * PROGRAM NAME:  PWSORT.C
 * PURPOSE:  Password sorting routine --
 *           Sorts password in largest to smallest character ordeer.  Used to
 *           specify the column order to pull out columns in TRANSPOSE() and
 *           UNTRANSPOSE().
 *
 * LANGUAGE:  C
 * AUTHOR:  CEB
 * USAGE:  PWSORT(PASSWORD)
 *              PASSWORD -- character array containing the encryption/
 *                          decryption key.
 *
 * ARRAYS USED:  PASSWORD[]
 * EXTERNALS:
 * UPDATE HISTORY:  INITIAL RELEASE -- 11/25/83 CEB
 */

#define min(x,y) ((x) < (y) ? (x) : (y))

pwsort(password)
     char password[];

     {
     int i,j,temp;

     pwlen = strlen(password); /* get length of PASSWORD string */
     if (pwlen < MINPWLEN) /* PASSWORD too short */
          {
          printf("\n*** Password:  %s is too short ( < 5 characters ) ***\n",
                  password);
          exit(0);
          }
     pwlen = min(pwlen,MAXPWLEN); /* truncate PASSWORD if necessary */
     for (i = 0; i < pwlen; i++) /* initialize Password Column Order */
          pwcolumnorder[i] = i;
     for (i = 0; i < pwlen - 1; i++) /* sort Password by Column Order */
          for (j = i + 1; j < pwlen; j++)
               if (password[pwcolumnorder[i]] < password[pwcolumnorder[j]])
                                   /* largest to smallest character order */
                    {
                    temp = pwcolumnorder[i];
                    pwcolumnorder[i] = pwcolumnorder[j];
                    pwcolumnorder[j] = temp;
                    }
     }
```

End Listing Two

Listing Three

```
/*##########################################################################
###                                                                      ###
###        Copyright 1983, Charles E. Burton, Denver, Colorado           ###
###                                                                      ###
### All rights reserved.  Permission granted to use this software for    ###
###   personal, non-commercial purposes only.                            ###
###                                                                      ###
##########################################################################*/

/*
 * PROGRAM NAME:  ENCRYPT.C
 * PURPOSE:  Encryption using the ADFGVX Cipher --
 *           re. C.C. Foster, "Cryptoanalysis for Microcomputers," Hayden Book
 *           Co. (Rochelle Park, NJ), p. 222.
```

```
 *
 *
 * LANGUAGE:  C
 * AUTHOR:  CEB
 * USAGE:  ENCRYPT <filename> <password>
 *              <filename> -- File Name to be encrypted
 *              <password> -- 5 to 10 character key to be used for encryption
 *
 * ARRAYS USED:  FILENAME[20],ENCFNAME[20],PASSWORD[11],BUF1[110],BUF2[110],
 *               POSITION[NOCARDS],VALUE[NOCARDS],PWCOLUMNORDER[MAXPWLEN]
 * EXTERNALS:  PWSORT(), SHUFFLE(), RAN()
 * UPDATE HISTORY:  INITIAL RELEASE -- 11/25/83 CEB
 */

#include "stdio.h"

#define NOCARDS 256

#define NO 0
#define YES 1

main(argc,argv)
     char *argv[];
     int argc;

     {
     static char filename[20],encfname[20],password[11],*str;
     static char c,buf1[110],buf2[110],position[NOCARDS],value[NOCARDS];
     int i,fnlen,pwlen,blklen,idx,
         done = NO;
     long int ftell(),dummy;
     union
          {
          long int xlong;
          char xbyte[sizeof(dummy)];
          } filelen;
     FILE *fopen(),*fpin,*fpout;

  if (argc != 3)
       {
       printf("\nUsage:  ENCRYPT <filename> <password>\n");
       exit(0);
       }
  for (i=0, idx=sizeof(filename), str=argv[1];  /* get file name to use */
       *str != '\0' && i < 14; i++, str++)
       {
       filename[i]=encfname[i]=*str;  /* get file name */
       if (*str == '.')  /* start of <ext> found? */
            idx=i+1;  /* get index to start of <ext> */
       }
  filename[i]=encfname[i]='\0';  /* terminate file name */
  fnlen=i;  /* get length of file name */
  switch (fnlen-idx)  /* based on length of <ext> */
       {
       case 1:
            /* only one character in <ext> */
            encfname[idx+2]='\0';  /* move EOS over one character */
            fnlen++;  /* account for added character */
       case 2:
       case 3:
            /* two or three characters in <ext> */
            encfname[idx+1]='Y';  /* make 2nd character of <ext> a 'Y' */
            break;
       default:
            if (idx == fnlen)  /* no <ext>, but '.' exists */
                 strcpy(encfname+fnlen,"YYY");  /* add <ext> */
            else if (idx == sizeof(filename))  /* no '.' and no <ext>? */
                 strcpy(encfname+(fnlen++),".YYY");  /* add <ext> */
            else  /* invalid file name */
                 {
                 printf("\n*** Bad <filename>:  %s ***\n",filename);
                 exit(0);
                 }
            fnlen += 3;  /* account for added "YYY" */
            break;
       }
  for (i=0, str=argv[2]; *str != '\0' && i < sizeof(password)-1; i++)
       password[i]=*(str++);  /* get Password */
  password[i]='\0';  /* terminate Password */
  pwlen=i;  /* get length of Password */
```

(Continued on next page)

ADFGVX Cipher System (Listing Continued, text begins on page 48)

Listing Three

```
blklen=pwlen*pwlen+pwlen/2;  /* get length of block to read/write for
                                file I/O to get irregular columns for
                                password */
if (blklen != 2*(blklen/2))  /* BLKLEN odd? */
     blklen++;  /* make it even */
if ((fpin=fopen(filename,"r")) == NULL)  /* cannot open input file? */
     {
     printf("\n*** Cannot open %s ***\n",filename);
     exit(0);
     }
if ((fpout=fopen(encfname,"w")) == NULL)  /* cannot open output file? */
     {
     printf("\n*** Cannot open %s ***\n",encfname);
     fclose(fpin);  /* close input file */
     exit(0);
     }
fseek(fpin,0L,2);  /* find end of file */
filelen.xlong=ftell(fpin);  /* get length of plain text file */
fclose(fpin);  /* close the input file */
fpin=fopen(filename,"r");  /* re-open the input file */
for (i=0; i < sizeof(filelen.xlong); i++)
     putc(filelen.xbyte[i],fpout);  /* save file length at beginning of
                                       output file */
shuffle(position,value);  /* randomize cipher character set */
pwsort(password);  /* get transposition for row/column pairs */
do  /* encrypt plain text file */
     {
     for (i=0; i < blklen; i++)  /* read a block of plain text */
          {
          if ((c=compress(fpin)) == EOF)  /* Compress text, EOF found? */
               {
               done=YES;  /* indicate last pass of plain text */
               if (i)  /* at least one value in BUF1 */
                    for ( ; i < blklen; i++)  /* fill in rest of BUF1 */
                         {
                         c=ran(NOCARDS);  /* generate a random char */
                         buf1[i++]=position[c]/16;
                                        /* char row position */
                         buf1[i]=position[c]%16;
                                        /* char column position */
                         }
               break;
               }
          /* fill BUF1 */
          buf1[i++]=position[c]/16;  /* char row position */
          buf1[i]=position[c]%16;  /* char column position */
          }
     if (i) /* BUF1 full? */
          {
          transpose(buf1,buf2,blklen);  /* encrypt using Password */
          for (i=0; i < blklen; i += 2)  /* write a block of encrypted
                                            text */
               putc(value[16*buf2[i]+buf2[i+1]],fpout);
          }
     }
while (done == NO);
fclose(fpin);  /* close files */
fclose(fpout);
if ((fpout=fopen(filename,"r+")) == NULL)  /* cannot open output file? */
     {
     printf("\n*** Cannot open %s ***\n",filename);
     exit(0);
     }
     else  /* overwrite all records so that they cannot be recovered */
          {
          while (filelen.xlong--)  /* Null out plain text file */
               putc(0xF6,fpout);  /* use formatting data character to null */
          fclose(fpout);  /* close the file */
          }
     unlink(filename);  /* erase plain text file */
     printf("\nEncryption completed\n");
     }
```

(Continued on page 64)

ADFGVX Cipher System

(Listing Continued, text begins on page 48)

Listing Three

```
/*************************************************************************/

/* Multiple Character Compression.  Encodes repeated characters. The Stream is
   byte for byte pass-through except that SYN is encoded as SYN/0 and repeated
   byte values are encoded as Byte/SYN/Count, where Count >='3.  This routine
   is  a state machine representation and is modeled after the GETNCR()
   function in SQ.C (squeezer) by Richard Greenlaw.
*/

#define NOHISTORY 0       /* do not consider previous input */
#define SENTCHAR  1       /* LASTCHAR sent, no lookahead yet */
#define SENDNEWC  2       /* NEWCHAR sent, previous sequence done */
#define SENDCNT   3       /* NEWCHAR sent, SYN sent, send COUNT next */

#define SYN 0X16          /* duplicate character indicator */

compress(stream)
     FILE *stream;

     {
     static char state = NOHISTORY; /* states */
     static int count; /* Count of consecutive identical characters */
     static int lastchar,nextchar;

     switch (state)
          {
          case NOHISTORY:
               /* no relevant history */
               state = SENTCHAR;
               return (lastchar = getc(stream));
               break;
          case SENTCHAR:
               /* LASTCHAR is sent, need to lookahead */
               switch (lastchar)
                    {
                    case SYN:
                         /* actual SYN character found */
                         state = NOHISTORY;
                         return (0);
                         break;
                    case EOF:
                         /* end of file found */
                         return (lastchar);
                         break;

          default:
               /* any other character found */
               for (count=1; (nextchar = getc(stream)) == lastchar &&
                    count < 255; count++) /* count like characters */
                    ;
               switch (count)
                    {
                    case 1:
                         /* one character found */
                         return (lastchar = nextchar);
                         break;
                    case 2:
                         /* two characters found */
                         state = SENDNEWC;
                         return (lastchar);
                         break;
                    default:
                         /* three or more characters found */
                         state = SENDCNT;
                         return (SYN);
                         break;
                    }
               break;
          }
 case SENDNEWC:
      /* previous sequence complete, send NEWCHAR */
      state = SENTCHAR;
      return (lastchar = nextchar);
      break;
 case SENDCNT:
```

```
            /* sent SYN for repeat sequence, send COUNT */
            state = SENDNEWC;
            return (count);
            break;
      default:
            printf("\n*** Bad STATE in COMPRESS() ***\n");
            exit(0);
            break;
      }
}
/*************************************************************************/

/* Transpose via Columns based on Password.  This routine takes a string and
   transposes the characters to column order based on a password. The password
   length is the width of the table and the password has been sorted to
   specify the column order to pull out the columns. */

#define MAXPWLEN 10
#define MINPWLEN  5

static int pwlen,pwcolumnorder[MAXPWLEN];

#include "PWSORT.C"

transpose(from,to,len)
      char from[],to[];
      int len;

      {
      int i,j,k;

      i = 0;
      for (j = 0; j < pwlen; j++) /* transpose in column order */
            for (k = pwcolumnorder[j]; k < len; k += pwlen)
                                                /* transpose characters */
                  to[i++] = from[k];
      }

#include "SHUFFLE.C"
```

End Listing Three

Listing Four

```
/*#########################################################################
###                                                                     ###
###         Copyright 1983, Charles E. Burton, Denver, Colorado         ###
###                                                                     ###
### All rights reserved.  Permission granted to use this software for   ###
###    personal, non-commercial purposes only.                          ###
###                                                                     ###
#########################################################################*/

/*
 * PROGRAM NAME:  DECRYPT.C
 * PURPOSE:  Decryption using the ADFGVX Cipher --
 *           re. C.C. Foster, "Cryptoanalysis for Microcomputers," Hayden Book
 *           Co. (Rochelle Park, NJ), p. 222.
 *
 *
 * LANGUAGE:  C
 * AUTHOR:  CEB
 * USAGE:  DECRYPT <filename> <password>
 *              <filename> -- File Name to be decrypted
 *              <password> -- 5 to 10 character key to be used for decryption
 *
 * ARRAYS USED:  FILENAME[20],ENCFNAME[20],PASSWORD[11],BUF1[110],BUF2[110],
 *               POSITION[NOCARDS],VALUE[NOCARDS],PWCOLUMNORDER[MAXPWLEN]
 * EXTERNALS:  PWSORT(), SHUFFLE()
 * UPDATE HISTORY:  INITIAL RELEASE -- 11/25/83 CEB
 */

#include "stdio.h"

#define NOCARDS 256

#define NO 0
#define YES 1
```

(Continued on next page)

ADFGVX Cipher System

(Listing Continued, text begins on page 48)

Listing Four

```
static char password[11],position[NOCARDS],value[NOCARDS];
static int blklen;

main(argc,argv)
     char *argv[];
     int argc;

     {
     static char filename[20],decfname[20],*str,c;
     int i,fnlen,pwlen,idx;
     long int dummy;
     union
          {
          long int xlong;
          char xbyte[sizeof(dummy)];
          } filelen;
     FILE *fopen(),*fpin,*fpout;

     if (argc != 3)
          {
          printf("\nUsage:  DECRYPT <filename> <password>\n");
          exit(0);
          }
     for (i=0, idx=sizeof(filename), str=argv[1];  /* get file name to use */
          *str != '\0' && i < 14; i++, str++)
          {
          filename[i]=decfname[i]=*str;  /* get file name */
          if (*str == '.')  /* start of <ext> found? */
               idx=i+1;  /* get index to start of <ext> */
          }
     filename[i]=decfname[i]='\0';  /* terminate file name */
     fnlen=i;  /* get length of file name */
     switch (fnlen-idx)  /* based on length of <ext> */
          {
          case 1:
               /* only one character in <ext> */
               decfname[idx+2]='\0';  /* move EOS over one character */
               fnlen++;  /* account for added character */
          case 2:
          case 3:
               /* two or three characters in <ext> */
               decfname[idx+1]='Y';  /* make 2nd character of <ext> a 'Y' */
               break;
          default:
               if (idx == fnlen)  /* no <ext>, but '.' exists */
                    strcpy(decfname+fnlen,"YYY");  /* add <ext> */
               else if (idx == sizeof(filename))  /* no '.' and no <ext>? */
                    strcpy(decfname+(fnlen++),".YYY");  /* add <ext> */
               else  /* invalid file name */
                    {
                    printf("\n*** Bad <filename>:  %s ***\n",filename);
                    exit(0);
                    }
               fnlen += 3;  /* account for added "YYY" */
               break;
          }
     for (i=0, str=argv[2]; *str != '\0' && i < sizeof(password)-1; i++)
          password[i]=*(str++);  /* get Password */
     password[i]='\0';  /* terminate Password */
     pwlen=i;  /* get length of Password */
     blklen=pwlen*pwlen+pwlen/2;  /* get length of block to read/write for file
                                     I/O to get irregular columns for both
                                     passwords */
     if (blklen != 2*(blklen/2))  /* BLKLEN odd? */
          blklen++;  /* make it even */
     if ((fpin=fopen(decfname,"r")) == NULL)  /* cannot open input file? */
          {
          printf("\n*** Cannot open %s ***\n",decfname);
          exit(0);
          }
      if ((fpout=fopen(filename,"w")) == NULL)  /* cannot open output file? */
           {
           printf("\n*** Cannot open %s ***\n",filename);
```

(Continued on page 68)

Listing Four

```
          fclose(fpin);  /* close input file */
          exit(0);
          }
     for (i=0; i < sizeof(filelen.xlong); i++)
          filelen.xbyte[i]=getc(fpin);  /* get original file length */
     shuffle(position,value);  /* randomize cipher character set */
     pwsort(password);  /* get transposition for row/column pairs */
     while (((c=expand(fpin)) != EOF) && (filelen.xlong--))
                                        /* decrypt & expand the file */
          putc(c,fpout);  /* write plain text */
     fclose(fpin);  /* close files */
     fclose(fpout);
     printf("\nDecryption completed\n");
     }

/*************************************************************************/

/* Multiple Character Expansion.  Decodes compressed characters. The Stream
   is byte for byte pass-through except that SYN/0 is decoded as SYN and
   Byte/SYN/Count is decoded as Byte repeated Count times.  This routine
   is a state machine representation and is modeled after the GETNCR()
   function in SQ.C (squeezer) by Richard Greenlaw.
*/

#define NOHISTORY 0       /* do not consider previous input */
#define SENTCHAR  1       /* LASTCHAR sent, no lookahead yet */
#define SENDRPT   2       /* LASTCHAR sent, SYN found, found COUNT, repeat
                             LASTCHAR */

#define SYN 0X16          /* duplicate character indicator */

expand(stream)
     FILE *stream;

     {
     static char state = NOHISTORY; /* states */
     static int count; /* Count of consecutive identical characters */
     static int lastchar,nextchar;

     switch (state)
          {
          case NOHISTORY:
               /* no relevant history */
               switch (lastchar = getcnxt(stream))
                    {
                    case SYN:
                         /* must be a SYN character, check */
                         switch (getcnxt(stream))
                              {
                              case 0:
                                   /* SYN character found */
                                   return (lastchar);
                                   break;
                            default:
                                 /* bad SYN character found */
                                 printf("\n*** Bad SYN in EXPAND() ***\n");
                                 exit(0);
                                 break;
                              }
                         break;
                    default:
                         /* another character found */
                         state = SENTCHAR;
                         return (lastchar);
                         break;
                    }
               break;
          case SENTCHAR:
               /* LASTCHAR is sent, need to lookahead */
               switch (lastchar)
                    {
                    case EOF:
                         /* end of file found */
                         return (lastchar);
```

Listing Four

```
        break;
    default:
        /* any other character found */
        switch (nextchar = getcnxt(stream))
            {
            case SYN:
                /* repeating characters found */
                switch (count = getcnxt(stream))
                    {
                    case 0:
                        /* SYN character found */
                        state = NOHISTORY;
                        return (nextchar);
                        break;
                    default:
                        /* actual COUNT found */
                        state = SENDRPT;
                        count -= 2; /* adjust for 2 sent
                                       characters */
                        return (lastchar);
                        break;
                    }
                break;
            default:
                /* another character found */
                return (lastchar = nextchar);
                break;
            }
        break;
    }

            case SENDRPT:
                /* repeat sequence, send LASTCHAR COUNT times */
                if (count > 1)
                    {
                    /* keep sending LASTCHAR */
                    count--;
                    return (lastchar);
                    }
                else if (count == 1)
                    {
                    /* final LASTCHAR */
                    state = NOHISTORY;
                    return (lastchar);
                    }
                else
                    {
                    /* problem with COUNT */
                    printf("\n*** Bad COUNT in EXPAND() ***\n");
                    exit(0);
                    }
                break;
            default:
                printf("\n*** Bad STATE in EXPAND() ***\n");
                exit(0);
                break;
            }
        }

/*************************************************************************/

/* Get next character from input stream.  Must read a block of data,
   untranspose it and distribute the block a character at a time. */

getcnxt(stream)
        FILE *stream;

        {
        static char buf1[110],buf2[110],c;
        int i;
        static int done = NO,
                   bufidx = sizeof(buf1);

        if ((bufidx >= blklen) && (done == NO))
```

(Continued on next page)

```
                    /* need to read more characters into BUF1? */
{
for (i=0; i < blklen; i++)  /* read a block of encrypted text */
     {
     if ((c=getc(stream)) == EOF)  /* EOF found? */
          {
          done=YES;  /* indicate last pass of plain text */
          break;
          }
     /* fill BUF1 */
     buf1[i++]=position[c]/16;  /* char row position */
     buf1[i]=position[c]%16;  /* char column position */
     }

          if (done == NO)  /* last Block meaningless, ignore it? */
               {
               untranspose(buf1,buf2,blklen);  /* decrypt with Password */
               bufidx=0;  /* reset Block Index to start of BUF1 */
               }
          }
     if (bufidx < blklen)  /* characters still available? */
          {
          bufidx += 2;  /* point to next row/column pair */
          return (value[16*buf2[bufidx-2]+buf2[bufidx-1]]);
                                        /* return a decrypted character */
          }
     else if (done == YES)  /* no more characters available & EOF? */
          return (EOF);  /* indicate done */
     else  /* character stream problem */
          {
          printf("\n*** Synchronization error in GETCNXT() ***\n");
          exit(0);
          }
     }

/*************************************************************************/

/* Untranspose via Columns based on Password.  This routine takes a string and
   transposes the characters to row order based on a password. The password
   length is the width of the table and the password has been sorted to
   specify the column order to put back the columns before pulling out rows. */

#define MAXPWLEN 10
#define MINPWLEN  5

static int pwlen,pwcolumnorder[MAXPWLEN];

#include "PWSORT.C"

untranspose(from,to,len)
     char from[],to[];
     int len;

     {
     int i,j,k;

     i = 0;
     for (j = 0; j < pwlen; j++) /* transpose in column order */
          for (k = pwcolumnorder[j]; k < len; k += pwlen)
                                                /* transpose characters */
               to[k] = from[i++];
     }

#include "SHUFFLE.C"
```

End Listings

More dBASE Tips and Techniques

by Gene Head

I wrote a dBASE II command file called DB-SQZ.CMD that will squeeze and tokenize any command file just as DBCODE does in Ashton-Tate's RUNTIME package. My customers who already owned dBASE II could run these "scrambled" command files and not have to purchase RUNTIME. (DB-SQZ.CMD may appear in a future column or call me for a modem download.)

Unfortunately, every customer that owned dBASE II also owned an IBM PC. In their wisdom Ashton-Tate made the default command file extension different for the MSDOS/IBM version of dBASE II. CP/M command files are of the type .CMD and IBM command files are .PRG. So I have this wonderful program that protects my command files from tampering, but I can't use it unless I change every file type from .CMD to .PRG on every disk!

There should be some way to keep the filenames as they are and make dBASE II change what it looks for when it meets a DO FILENAME.

The best solution was to make dBASE II look for the file type I told it to look for; that is, have my CP/M dBASE II look for .PRG instead of .CMD command files.

This is easy enough because the file type is stored as plain ASCII text and easily patched with the POKE command. I wrote a simple command file called FLIP.IT that changed the default file type using the POKE function.

But wait. Now I want to go the other direction! Make the IBM dBASE II program look for .CMD instead of .PRG command files. Furthermore, I need each direction for several versions (2.4, 2.41, 2.41 Z80) of dBASE II. This could get confusing.

Finally, I came up with the idea of writing the program in the listing (page 72) called MAKEFLIP.XXX that will "write" a customized version of FLIP.IT for any version of dBASE II almost automatically! Now, I only need to have a copy of MAKEFLIP.XXX when I go to work in any dBASE II environment.

On any new installation *I* execute MAKEFLIP.XXX one time and it generates FLIP.IT for whatever machine it is on. Now, whenever I need to change the command file default extension I type .DO FLIP.IT.

MAKEFLIP.XXX is not only a very useful utility, but you should get some good insight into how to write code that writes code. This could be especially helpful in custom installations.

How to write dBASE code that writes dBASE code.

Before I get calls and letters telling me that my search routine is too loose and could find a bogus patch, let me say that I already know that. However, I have tried MAKEFLIP.XXX on several types of micros and most versions of dBASE II with complete success. The chances of finding an invalid patch location are too small to justify additional code. Besides, you will know right away if you found a bad patch area. *It won't work!* **DDJ**

(Listing begins on next page)

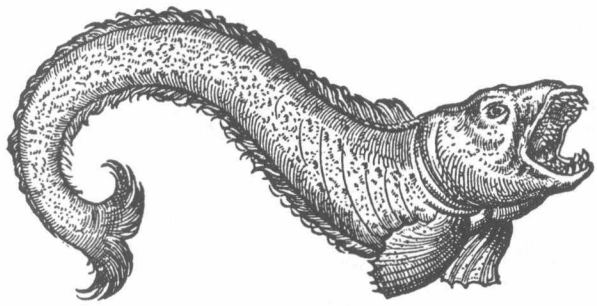
Gene Head, Head Quarters, 2860 NW Skyline, Corvallis, OR 97330.

dBASE Listing (Text begins on page 70)

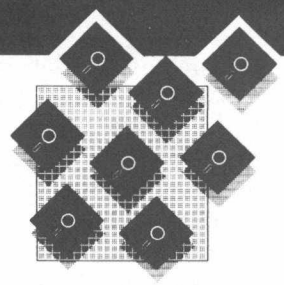
```
*
*   Another dBASE II goody from:
*
*           Head Quarters
*
*           Gene Head
*           2860 NW Skyline Drive
*           Corvallis, Oregon  97330
*           (503)  758-0279
*
*   For non-commercial use only.  If you use this utility
*   to help you develop any software that earns you money
*       (even in your own business) send me $5.00.
*   It's a fair price to pay and worth a clear conscience.
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*
*
*   Program name --> MAKEFLIP.XXX
*
*             (DO NOT change the name of this command file)
*             (Execute it as .DO MAKEFLIP.XXX or it may not work)
*
*   * * *  This program generates the command file FLIP.IT  * * *
*
*   You only need to run this program ONCE to create FLIP.IT.
*   After you create FLIP.IT put a copy on your dBASE II disk.
*   Then when you wish to change default command file extension
*   type .DO FLIP.IT
*
*   Example:
*
*   To run CP/M command files defaulted to .CMD under MS-DOS simply
*   type .DO FLIP.IT.  To change back to the .PRG default, type the
*   same thing again, .DO FLIP.IT.
*
*   Likewise, to run MS-DOS command files defaulted to .PRG under
*   CP/M, type .DO FLIP.IT.  To change back to .CMD default, again
*   type .DO FLIP.IT.
*
*
*


SET TALK OFF

*  This range works well for dBASE II versions 2.4,
*  2.41 and Z-80 2.41.  For other versions they may have to change.
STORE '12300' TO LOW
STORE '12500' TO HIGH


*  First get the range of program RAM to search

*     for the default extension

STORE 'Y' TO CHOICE
DO WHILE CHOICE = 'Y'

    STORE T TO RANGE

    ERASE

    DO WHILE RANGE
      @ 12,10 SAY 'Enter START of search area.  Suggest --> ';

      GET LOW PICTURE '99999'

      @ 14,10 SAY '  Enter END of search area.  Suggest --> ';

      GET HIGH PICTURE '99999'

      READ


      STORE VAL(LOW) TO MLOW
      STORE VAL(HIGH) TO MHIGH

        IF MLOW >= MHIGH .OR. MHIGH > 65000 .OR. MLOW < 100
           @ 20,10 SAY 'SEARCH AREA OUT OF RANGE.  TRY AGAIN . . .'
        ELSE

           STORE F TO RANGE
        ENDIF MLOW >= MHIGH .OR. MHIGH > 65000 .OR. MLOW < 100
```



```
ENDDO WHILE RANGE

ERASE

*  This is not the most accurate testing but the chances are
*     very high that it will only find what we are looking for.

@ 12,10 SAY 'SEARCHING'
STORE F TO FOUND

DO WHILE (.NOT. FOUND) .AND. MLOW < MHIGH

    IF PEEK(MLOW) <> 67 .AND. PEEK(MLOW) <> 80
     STORE MLOW+1 TO MLOW
     LOOP
    ELSE
      IF PEEK(MLOW+1) <> 77 .AND. PEEK(MLOW) <> 82
    STORE MLOW+1 TO MLOW
    LOOP
      ELSE
        IF PEEK(MLOW+2) <> 68 .AND. PEEK(MLOW) <> 71
            STORE MLOW+1 TO MLOW
            LOOP
            ELSE
            STORE T TO FOUND
            LOOP
        ENDIF PEEK(MLOW+2) <> 68 .AND. PEEK(MLOW) <> 71
      ENDIF PEEK(MLOW+1) <> 77 .AND. PEEK(MLOW) <> 82
    ENDIF PEEK(MLOW) <> 67 .AND. PEEK(MLOW) <> 80

ENDDO WHILE (.NOT. FOUND) .AND. MLOW < MHIGH

* If we found our patch area create a command file called FLIP.IT
IF FOUND
@ 12,10 SAY 'CREATING COMMAND FILE --> FLIP.IT '
STORE STR(MLOW,5) TO PATCH
SET ALTERNATE TO FLIP.IT
SET ALTERNATE ON
SET CONSOLE OFF
? [*  FLIP.IT]
? [*]
? [*  FLIP.IT WILL FLIP COMMAND FILE EXTENSION  .CMD <--> .PRG]
? [*]
? [*  FLIP.IT IS DESIGNED FOR THOSE WHO DEVELOP PROGRAMS USING CP/M]
? [*  dBASE II TO RUN UNDER MS-DOS dBASE II AND GET TIRED CHANGING]
? [*  THE COMMAND FILE EXTENSIONS FROM .CMD TO .PRG AND VICE-VERSA!]
? [*]
? [*  FLIP.IT CAUSES YOUR CP/M DBASE II TO ACCEPT .PRG AS DEFAULT]
? [*  COMMAND FILE EXTENSION.  IT CAN ALSO LET YOUR MS-DOS dBASE II]
? [*  ACCEPT .CMD AS THE DEFAULT COMMAND FILE EXTENSION.]
? [*]
? [*  IN EITHER CASE JUST TYPE .DO FLIP.IT]
? [*]
?
? [IF PEEK(]+PATCH+[) = 67]
? [ POKE ]+PATCH+[, 80, 82, 71]
? [ ? 'DEFAULT COMMAND EXTENSION --> .PRG']
? [ELSE]
? [     IF PEEK(]+PATCH+[) = 80]
? [        POKE ]+PATCH+[, 67, 77, 68]
? [        ? 'DEFAULT COMMAND EXTENSION --> .CMD']
? [     ELSE]
? [        ? 'ERROR CHANGING DEFAULT EXTENSION']
          ? [ENDIF]
          ?
          SET ALTERNATE OFF
          SET ALTERNATE TO
          SET CONSOLE ON

          STORE 'N' TO CHOICE

          ELSE

          @ 12,10 SAY 'CAN NOT LOCATE PATCH AREA WITHIN;
       THE SPECIFIED RANGE.'
          @ 14,10 SAY 'INCREASE THE SEARCH RANGE.;
        TRY AGAIN? (Y/N) --> ' GET CHOICE
          READ

          ENDIF  FOUND

          ENDDO while CHOICE = 'Y'


          ERASE

              * * * * * * * * * * * * * * * * * * * * * *
              *  end of source code for MAKEFLIP.XXX  *
              * * * * * * * * * * * * * * * * * * * * * *
```



(End Listing)



Modula-2/86, Version 1.04

Company: Logitech, 805 Veterans Blvd., Redwood City, CA 94086

Operating System: MSDOS and CP/M 86

Price: $495.00 ***(Discounts are available for educational institutions and user groups. — Ed.)***

Circle Reader Service No. 135

***Reviewed by Michael* Schmidt**

There has been much to-do about Modula-2 recently, with the theme that Modula-2 is the successor to Pascal. But how does a new Modula-2 compiler compare with an extended Pascal compiler that's been around a while? I was curious to find out.

I pitted the Logitech Modula-2/86 compiler, version 1.04 (not a public release), against the Microsoft MS-Pascal compiler, version 3.13. MS-Pascal is a highly extended version of the Pascal language, designed to facilitate systems programming. Many of the extensions of MS-Pascal correspond to standard features of Modula-2.

All tests were performed on an 8 MHz 8086 Seattle Computer Products S-100 system, running MSDOS 2.0. The system was equipped with 224K of memory, two Qume DT-8 floppy disk drives that store 1.2Mb each, and a Zenith Z19 terminal. The system was not equipped with an 8087 coprocessor.

Documentation

The Logitech manual represents what I would call a "user's guide," with certain parts of a "reference manual" in numerous appendices. With this organization, two-thirds of the documentation is in appendices! The preface refers the reader to the book *Programming In Modula-2* by N. Wirth for a definition of the Modula-2 language. However, implementation specifics and peculiarities of the SYSTEM and library modules in the Logitech implementation are not described, of course, in Wirth's book, and Logitech reduces them to the lowly status of appendices.

What do I have against appendices? The problem lies not with appendices *per se* but with the authors, who use them here as an excuse to supply substandard documentation. Gone are any requirements that material be presented in a logical order or that coherent English appear on each page or that a 70-page section show subdivisions with page numbers in the table of contents.

The largest appendix consists of the definition modules of the Logitech library, with comments. Unfortunately, these comments are not always enough! For example, the rather complex library module FileSystem seems to defy comprehension, at least mine. A similar file system appears as one of two alternate designs (one for RT-11, one for Lilith) in Wirth's book but with little supporting text—again just a definition module. It should be obvious to anyone struggling through Modula-2 definition modules that they are not self-documenting and require a little English explanation.

Part of the difficulty I had understanding these library modules result ed from undefined terminology borrowed from an alien (non-MSDOS) environment. For instance, what the heck is a "medium"? And how am I supposed to know that I should precede a filename with "DK:" to select the default drive? These peculiarities are bad enough by themselves, but they become downright unmanageable when they are not documented.

The "user's guide" portions of the Logitech documentation are well written and complete, with the following exceptions. A discussion of the runtime support (RTS) was missing alltogether. This program resides in the file M2.EXE, and handles program loading, overlays, coroutine transfers, and runtime errors. It produces a number of runtime error messages, which are not listed anywhere in the manual. Also, the section on linking overlays was not sufficiently descriptive.

The MS-Pascal documentation is more hefty. It consists of both a user's manual and a separate, much larger, language reference manual. All the information I have ever needed to know about MS-Pascal is there somewhere! The mere bulk of these manuals would tend to overwhelm the first-time user, but actually, the organization is logical, and once you have read through the manuals a few times, you can retrieve information quickly.

Neither set of manuals serves as an introduction to the language that its compiler implements. However, the MS-Pascal documentation is logically complete by itself, whereas the Logitech documentation requires *Programming In Modula-2* for a formal definition of Modula-2. The saving grace of the Logitech documentation is Logitech's technical support. Whenever I reached an impasse, a quick phone call on the Logitech 'hotline' produced my answer. With Microsoft, I am not even aware of a number I can call.

Installation

The MSDOS version of the Logitech compiler is distributed only on three 5¼-inch double-sided floppies, using 9-sector/track formatting. Logitech was not able to satisfy my request for an 8-inch distribution disk. I was able to add a minifloppy drive to my system (most MSDOS systems use the minifloppies anyhow), but for a minority of MSDOS systems this will present a problem.

The Logitech manual instructs the user to place the command

```
device = ANSI
```

in the CONFIG.SYS file. This is re-

quired only for systems with non-ANSI console devices. Because my Z19 is set to ANSI mode, no special driver is needed. Only the Logitech debugger generates ANSI escape sequences; you can operate both the compiler and linker without an ANSI console device.

The rest of the installation depends on whether you are using floppies or a Winchester. Actually, all that matters is that you have a mass-storage device that can hold about 1Mb, which is what a Modula-2 system disk requires. My 8-inch floppies were adequate. If you have a system with only 360K 5¼-inch floppies, you will need three diskettes in addition to the one holding your source code: a system disk with an editor on it, a compiler disk, and a linker/debugger disk. Good luck!

If you have adequate mass storage, you are instructed to place the following commands in your AUTOEXEC.BAT file:

```
SET M2LIB=\M2LIB\SYM
SET M2LNK=\M2LIB\LNK
SET M2REF=\M2LIB\REF
SET M2MAP=\M2LIB\MAP
```

I preceded each of the above symbols with a drive designator (M2LIB = A:\M2LIB\SYM) so that I did not have to set my default drive to the Modula system disk. The files on the distribution disks are copied to the above subdirectories or to the subdirectory \M2LOD for the executable files. Unfortunately, no corresponding symbol for this latter subdirectory exists (the command SET M2LOD=A:\M2LOD does nothing), and if the default drive does not contain the Modula system disk, the complete path name of the executable file must be given. In addition, the RTS in the file M2.EXE must be copied into a subdirectory selected by the MSDOS PATH command (in my directory structure this was A:\BIN).

Two example programs are provided on the distribution disk. When I first tried to compile them, I received an error message "cannot load. . ." during linking. What disturbed me was that this message was not in the manual. I suspected a lack of RAM and on closer examination discerned that the manual requires 192K of *program* RAM, which does not include the roughly 32K required by MSDOS 2.0. I upgraded my system to 256K and had no further troubles. To avoid such confusion, Logitech should admit to a requirement of 256K of system RAM.

The Logitech distribution is configured for an IBM PC. The source code for several modules, which might need modification for a different environment, the object files for the compiler, linker, and debugger themselves, and the assembler source code of the RTS modules are standard distribution! A rather cryptic comment in an appendix states that the user should study the source code for these and make changes as required.

Despite my different hardware configuration, I found that none of the library modules had to be recompiled. However, the process/interrupt module of the RTS proved to be incompatible with my system. It assumes a single 8259A interrupt controller, addressed at I/O ports 20 and 21H, as is the case in the IBM PC. My system has multiple cascaded 8259As and nothing at I/O ports 20 and 21H! This module plays with the interrupt controller for the TRANSFER and IOTRANSFER operations to implement an obscure feature of Modula-2 that allows numeric "priority" module parameters to turn modules into "monitors," as in Concurrent Pascal. Logitech interprets module priorities as corresponding to different interrupt masks for a single interrupt controller. Luckily, the compiler, linker, and debugger do not seem to use prioritized modules or TRANSFER statements and thus can be run without modification of M2.EXE. My system would require nontrivial modification of this assembler module to get these Modula-2 features working.

The installation of the MS-Pascal compiler is more straightforward. All that is required is to copy the three passes of the compiler, one of the library files, and the special "LARGE" version of MS-LINK to a system disk. These files can be combined on a single Pascal system disk complete with an editor—even a 360K 5¼-inch floppy. The MS-Pascal compiler (and linker!) require less than 160K total system RAM to operate.

Compiling and Linking

Definition and implementation modules are compiled separately by the Logitech compiler. The output from compiling a definition module is accessed automatically when compiling an implementation module. You can invoke all four passes of the Logitech compiler entirely from the command line with the command:

```
M2 COMP <filename>
```

The overlay mechanism of the RTS loads each pass of the compiler into memory separately. For this to work, however, the default drive when the compiler is invoked must be the drive that contains the directory \M2LOD. To get around this and to automatically invoke some of the compiler switches, I created the following command procedure (MOD.BAT) to invoke the compiler:

```
M2 A:\M2LOD\COMP
   %1/R-/S-/T-/E
```

Similarly, I created a command procedure MLNK.BAT to invoke the linker:

```
M2 A:\M2LOD\LINK %1
```

The compiler and linker use the first eight characters of a module name to search for compiled definition modules and object modules. This has the advantage of automating the search for library modules.

As is apparent from the compile and link times shown in the Figure (page 76), the Logitech compiler and linker are slow. A further annoyance stems from the Logitech compiler's organization into four passes, the first three of which can generate syntax-related error messages. Whether the compilation is aborted on an error after either of the first two passes or not can be configured by the user. This situation is made worse by the compiler's lack of certain types of error recovery. For instance, the compiler is case sensitive: all keywords are in upper case. It is very easy to forget and write "to" instead of "TO" in a FOR loop. Even though it should be an easy matter for the compiler to flag this as a warning and continue, such errors are treated as fatal.

The MS-Pascal compiler can also be invoked entirely from the command line. However, each pass of the compiler must be invoked separately; no command procedure is provided which automates this process. I have written such a command procedure (PAS.BAT) to invoke the required passes of the compiler:

```
PAS1 %1,%1,,%1;
IF ERRORLEVEL 0 DEL %1.LST
```

```
    IF ERRORLEVEL 4
        GOTO FINISHED
    PAS2
    :FINISHED
```

This command procedure would need enhancement to use PAS3, which provides a pseudo-assembly language listing of the code generated. In order to link multiple modules using MicroSoft's LINK utility, the following command format offers the greatest flexibility (the /NO parameter suppresses the automatic search for the Pascal library on the default path):

```
    LINK/NO @program
```

A "program" file must be created for each program you are developing. It would contain the following entries:

```
    program+
    module1+
    module2+
    moduleN
    program
    program
    A:\LIB\PASCAL
```

Having to maintain this additional file can sometimes be a nuisance.

Although the MS-Pascal compiler is not blindingly fast, it is certainly tolerable—over twice as fast as the Logitech compiler. The MS-Pascal compiler is organized as three passes, only the first of which generates syntax-related error messages. The error recovery of the compiler is adequate, and some common syntax errors are automatically corrected, generating only warnings. However, some simple types of errors, such as a missing END, will cascade and cause an avalanche of meaningless error messages.

The MS-Pascal compilation unit that most closely resembles the Modula-2 definition and implementation modules is the "unit" concept borrowed from UCSD Pascal. A unit also consists of a definition part and an implementation part. However, the definition portion of an MS-Pascal unit is not itself compiled; it is included as part of the source of its implementation, as well as any compilation units importing it, usually by means of the INCLUDE directive. This allows for "version" errors, resulting in changes in such definition parts.

Benchmarks

I decided to use the system clock to time my benchmarks. Although both Logitech Modula-2 and MS-Pascal provide library functions, I didn't care for the format of either. Instead I decided to write a GetTime procedure that would perform an MSDOS function call. I immediately ran into problems with MS-Pascal.

MS-Pascal provides a library function (DOSXQQ) to invoke MSDOS functions and two global variables (CRCXQQ and CRDXQQ) to pass parameters through the CX and DX registers. This function does not work properly: the two global variables are not updated from the CX and DX registers following a MSDOS call. Furthermore, even if this mechanism worked as advertised, it still would not be powerful enough to handle all MSDOS functions. The registers accessed by various MSDOS functions include the *AX, BX, CX, DX,* SI, DI, DS, and ES registers! I wrote a general purpose MSDOS interface procedure for MS-Pascal in assembly language (Listing One, page 78) which allows access to all these registers, and seems to do the job quite nicely. Logitech, on the other hand, supplies a much more versatile library function, DOSCALL, which gives full access to all MSDOS functions and worked properly on all my test cases.

I wrote a general purpose clock module in both Modula-2 and MS-Pascal (Listing Two, page 80, and Listing Three, page 82). It reflects the capabilities of MSDOS by providing a function GetTime, which returns the time through a record. I then wrote a simple stopwatch module (Listing Four, page 84, and Listing Five, page 86) which provides two procedures, Start and Stop. This module allowed me to measure the benchmarks to within a hundredth of a second.

I ran three benchmarks: the ever-popular Sieve of Eratosthenes (Listing Six, page 86 and Listing Seven, page 88), the Fibonacci number benchmark (Listing Eight, page 88, and Listing Nine, page 90), and a skeletal text filter program (Listing Ten, page 92, and Listing Eleven, page 93) that does nothing more than copy files. Both of these compilers produced very fast, highly optimized code (see the Figure). The code generator of the MS-Pascal compiler performed better on the Sieve benchmark, but not by a

benchmark & compiler	compile time (sec)	link time (sec)	code size (bytes)	linked size (bytes)	execute time (sec)
1. Sieve of Eratosthenes					
Logitech Modula-2/86	82	69	243	21474	5.25
Microsoft MS-Pascal	26	38	161	19518	4.87
2. Fibonacci Numbers					
Logitech Modula-2/86	81	68	141	21362	17.67
Microsoft MS-Pascal	25	37	107	19470	23.18
3. Text Filter					
Logitech Modula-2/86	87	69	391	21694	7.83
Microsoft MS-Pascal	27	38	353	19742	18.32

Figure

Benchmark Results.
All tests performed on 8MHz 8086, all run-time error-checking disabled.

wide margin. The Logitech compiler won the other two benchmarks by a somewhat wider margin. However, both of these languages produced bulky executable files by the time the linkers were through with them.

One thing that is not apparent from the benchmarks is that the Logitech linker produces very small executable files when none or only a few of the library modules are imported. The library module InOut (which most programs access) causes large executable files because it pulls in the FileSystem module; if no file support is needed, the Terminal module produces small executable files. This feature would be of value for writing certain types of utility programs, such as a program to continuously display the time.

Miscellaneous

Logitech supplies a *post mortem* symbolic debugger with the compiler package. This program allows the user to examine the procedure-calling sequence, last statement executed, and value of all data items following a 'core dump.' Such a core dump is performed automatically by the RTS following a runtime error or a ctrl-C typed by the user during execution. This provides significantly more diagnostic information than the simple error message produced by MS-Pascal on a runtime error. However, it is not as powerful as an interactive runtime debugger; such a product has been announced by Logitech, however.

Both Logitech and MicroSoft should be commended for the large number of useful library modules supplied with their compilers. Despite quirks in both libraries, these give the programmer a tremendous headstart in developing an application program.

Both of these compilers provide for floating-point support, either using an 8087 coprocessor or with an 8087 emulation library. I did not perform a benchmark on the emulation library, because anyone who is truly interested in fast floating point would be using an 8087, anyway.

The Logitech compiler does not support a 32-bit integer type, as does MS-Pascal. This is unfortunate because a 16-bit integer is too small for numerous applications (random number generators, for instance). This problem will undoubtedly be resolved in a future release, as recent revisions to the Modula-2 language by N. Wirth include LONGCARD and LONGINT data types.

My final complaint about the Logitech compiler, which I have already touched on, is that when a ctrl-C is typed during compilation or linking, about 15 seconds elapse and a large MEMORY.PMD file is created for the debugger in the default directory. This can be changed only by reassembling the RTS after changing one of the parameters in the source code. Of course, *post mortem* debugging of application programs requires a version the RTS with this feature left intact. I would suggest that Logitech distribute two executable versions of the RTS.

Conclusion

The Logitech Modula-2/86 package is clearly a winner. It provides a complete, relatively mature Modula-2 environment for the 8086 that is competitive with the best 8086 compilers available. In fact, a comparison of its speed on my system with published benchmarks on the Lilith shows them to be almost even in execution time. Perhaps most importantly, Logitech provides the kind of technical support necessary for a professional software development package.

On the other hand, MS-Pascal is still very much a viable systems language. It would certainly be preferable on a small system: it requires 96K less RAM and can perform adequately on a system equipped only with 5¼-inch floppies. In addition, its faster compile and link times would be appreciated on the much slower IBM PC.

I suppose the acid test is this: Which of these compilers will I be using from now on? This brings us back to the beginning of the article. Modula-2 really is the successor to Pascal, and the Logitech compiler is a thoroughly usable implementation. I have made the switch!

DDJ

(Listings begins on next page)

Listing One

```
INTERFACE; UNIT SYSTEM (
  Registers,
  DOSCALL);

TYPE

  Registers = RECORD
    ax, bx, cx, dx: WORD;
    si, di: WORD;
    ds, es: WORD;
  END;

PROCEDURE DOSCALL (VAR r: Registers);
{
    loads 8086 registers from r
    invokes MS-DOS
    loads r from 8086 registers
}

END; {SYSTEM}
        NAME    SYSTEM

        PUBLIC  DOSCALL

RGSTRS  STRUC
rax     DW      ?
rbx     DW      ?
rcx     DW      ?
rdx     DW      ?
rsi     DW      ?
rdi     DW      ?
rds     DW      ?
res     DW      ?
RGSTRS  ENDS

DOSCODE SEGMENT

        ASSUME  CS: DOSCODE

DOSCALL PROC    FAR
        PUSH    BP              ; save calling frame pointer
        MOV     BP, SP          ; get doscall frame pointer
        PUSH    DS              ; save calling data segment
        MOV     BP, 6[BP]       ; get rgstrs pointer

; load 8086 registers from structure
        MOV     AX, [BP].rax
        MOV     BX, [BP].rbx
        MOV     CX, [BP].rcx
        MOV     DX, [BP].rdx
        MOV     SI, [BP].rsi
        MOV     DI, [BP].rdi
        MOV     DS, [BP].rds
        MOV     ES, [BP].res
```

(Continued on page 80)

Listing One

```
        INT     21H             ; call MS-DOS

; load structure from 8086 registers
        MOV     [BP].rax, AX
        MOV     [BP].rbx, BX
        MOV     [BP].rcx, CX
        MOV     [BP].rdx, DX
        MOV     [BP].rsi, SI
        MOV     [BP].rdi, DI
        MOV     [BP].rds, DS
        MOV     [BP].res, ES

        POP     DS              ; restore calling data segment
        POP     BP              ; restore calling frame pointer
        RET     2               ; pop structure pointer during return

DOSCALL ENDP

DOSCODE ENDS

        END
```

End Listing One

Listing Two

```
DEFINITION MODULE Clck;

EXPORT QUALIFIED
  Time,
  Date,
  SetTime,
  SetDate,
  GetTime,
  GetDate;

TYPE
  Time = RECORD
    hour: [0..23];
    minute: [0..59];
    second: [0..59];
    hundredth: [0..99];
  END;

  Date = RECORD
    year: [1980..2099];
    month: [1..12];
    day: [1..31];
  END;

PROCEDURE SetTime (VAR t: Time);

PROCEDURE SetDate (VAR d: Date);

PROCEDURE GetTime (VAR t: Time);

PROCEDURE GetDate (VAR d: Date);

END Clck.

IMPLEMENTATION MODULE Clck;

FROM SYSTEM IMPORT
  DOSCALL;

(***********************************)

PROCEDURE SetTime (VAR t: Time);

VAR
  al: [0..255];
  cx, dx: CARDINAL;

BEGIN
  WITH t DO
    cx := 256*hour + minute;
    dx := 256*second + hundredth;
    DOSCALL (2DH, cx, dx, al);
  END;
END SetTime;

(***********************************)

PROCEDURE SetDate (VAR d: Date);

VAR
  al: [0..255];
  cx, dx: CARDINAL;
```

(Continued on page 82)

Listing Two

```
BEGIN
  WITH d DO
    cx := year;
    dx := 256*month + day;
    DOSCALL (2BH, cx, dx, al);
  END;
END SetDate;

(***********************************)

PROCEDURE GetTime (VAR t: Time);

VAR
  cx, dx: CARDINAL;

BEGIN
  WITH t DO
    DOSCALL (2CH, cx, dx);
    hour := cx DIV 256;
    minute := cx MOD 256;
    second := dx DIV 256;
    hundredth := dx MOD 256;
  END;
END GetTime;

(***********************************)

PROCEDURE GetDate (VAR d: Date);

VAR
  cx, dx: CARDINAL;

BEGIN
  WITH d DO
    DOSCALL (2AH, cx, dx);
    year := cx;
    month := dx DIV 256;
    day := dx MOD 256;
  END;
END GetDate;

END Clck.
```

End Listing Two

Listing Three

```
INTERFACE; UNIT CLOCK (
  Time,
  Date,
  SetTime,
  SetDate,
  GetTime,
  GetDate);

TYPE
  Time = RECORD
    hour: 0..23;
    minute: 0..59;
    second: 0..59;
    hundredth: 0..99;
  END;

  Date = RECORD
    year: 1980..2099;
    month: 1..12;
    day: 1..31;
  END;

PROCEDURE SetTime (VAR t: Time);

PROCEDURE SetDate (VAR d: Date);

PROCEDURE GetTime (VAR t: Time);

PROCEDURE GetDate (VAR d: Date);

END; {Clock}

IMPLEMENTATION OF CLOCK;

USES SYSTEM;

{***********************************}

PROCEDURE SetTime;

VAR
  r: Registers;

BEGIN
  WITH t DO WITH r DO BEGIN
    ax := #2D00;
    cx := BYWORD (hour, minute);
    dx := BYWORD (second, hundredth);
    DOSCALL (r);
  END;
END; {SetTime}

{***********************************}

PROCEDURE SetDate;

VAR
  r: Registers;

BEGIN
  WITH d DO WITH r DO BEGIN
    ax := #2B00;
    cx := year;
    dx := BYWORD (month, day);
    DOSCALL (r);
  END;
END; {SetDate}
```

(Continued on page 84)

Listing Three

```
{**********************************}

PROCEDURE GetTime;

VAR
  r: Registers;

BEGIN
  WITH t DO WITH r DO BEGIN
    ax := #2C00;
    DOSCALL (r);
    hour := HIBYTE (cx);
    minute := LOBYTE (cx);
    second := HIBYTE (dx);
    hundredth := LOBYTE (dx);
  END;
END; {GetTime}

{**********************************}

PROCEDURE GetDate;


 VAR
   r: Registers;

 BEGIN
   WITH d DO WITH r DO BEGIN
     ax := #2A00;
     DOSCALL (r);
     year := cx;
     month := HIBYTE (dx);
     day := LOBYTE (dx);
   END;
 END; {GetDate}

 END. {Clock}
```

End Listing Three

Listing Four

```
DEFINITION MODULE StopWatch;

EXPORT QUALIFIED
  Start,
  Stop;

PROCEDURE Start;
PROCEDURE Stop;

END StopWatch.
```

```
IMPLEMENTATION MODULE StopWatch;

FROM InOut IMPORT
  WriteString,
  WriteCard,
  WriteLn;

FROM Clck IMPORT
  Time,
  GetTime;

VAR
  t1, t2: Time;

(*****************************)

PROCEDURE Start;

BEGIN
  GetTime (t1);
END Start;

(*****************************)

PROCEDURE Stop;

(*****************************)

PROCEDURE DisplayTime (t: Time);

BEGIN
  WITH t DO
    WriteCard(hour, 8);
    WriteCard(minute, 8);
    WriteCard(second, 8);
    WriteCard(hundredth, 8);
    WriteLn;
  END;
END DisplayTime;

(*****************************)

BEGIN
  GetTime (t2);
  WriteString('start: '); DisplayTime(t1);
  WriteString('stop:  '); DisplayTime(t2);
END Stop;

END StopWatch.
```

End Listing Four

(Listing five begins on page 86)

Listing Five

```
INTERFACE; UNIT StopWatch (
  Start,
  Stop);

PROCEDURE Start;
PROCEDURE Stop;

END; {StopWatch}

IMPLEMENTATION OF StopWatch;

USES Clock;

VAR
  t1, t2: Time;

{*****************************}

PROCEDURE Start;

BEGIN
  GetTime (t1);
END; {Start}

{*****************************}

PROCEDURE Stop;

{*****************************}

PROCEDURE DisplayTime (t: time);

BEGIN
 WITH t DO BEGIN
   WriteLn(hour, minute,second,hundredth);
  END;
END; {DisplayTime}

{*****************************}

BEGIN
  GetTime (t2);
  Write('start: '); DisplayTime(t1);
  Write('stop:  '); DisplayTime(t2);
END; {Stop}

END. {StopWatch}
```

End Listing Five

Listing Six

```
MODULE Benchmark1;
(* Sieve of Eratosthenes *)

FROM StopWatch IMPORT
  Start,
  Stop;

CONST
  iterations = 10;

VAR
  i: INTEGER;

(*******************************************)

PROCEDURE Sieve;

CONST
  size = 8190;

VAR
  i, prime, k, count: INTEGER;
  flags: ARRAY[0..size] OF BOOLEAN;

BEGIN
  count := 0;
  FOR i := 0 TO size DO
    flags[i] := TRUE;
  END;
  FOR i := 0 TO size DO
    IF flags[i] THEN
      prime := i + i + 3;
      k := i + prime;
      WHILE k <= size DO
        flags[k] := FALSE;
        INC (k, prime);
      END;
      INC (count);
    END;
  END;
END Sieve;

(*******************************************)

BEGIN
  Start;
  FOR i := 1 TO iterations DO
    Sieve;
  END;
  Stop;
END Benchmark1.
```

End Listing Six

(Listing seven begins on page 88)

Listing Seven

```
PROGRAM Benchmark1 (INPUT, OUTPUT);
{ Sieve of Eratosthenes }

USES StopWatch;

CONST
  iterations = 10;
  { number of times benchmark executed }
VAR
  i: INTEGER;
{************************************}

PROCEDURE Sieve;

CONST
  size = 8190;

VAR
  i, prime, k, count: INTEGER;
  flags: ARRAY [0..size] OF BOOLEAN;

BEGIN
  count := 0;
  FOR i := 0 TO size DO BEGIN
    flags[i] := TRUE;
  END;
  FOR i := 0 TO size DO BEGIN
    IF flags[i] THEN BEGIN
      prime := i + i + 3;
      k := i + prime;
      WHILE k <= size DO BEGIN
        flags[k] := FALSE;
        k := k + prime;
      END;
      count := count + 1;
    END;
  END;
END; {Sieve}

{************************************}

BEGIN
  Start;
  FOR i := 1 TO iterations DO BEGIN
    Sieve;
  END;
  Stop;
END. {Benchmark1}
```

End Listing Seven

Listing Eight

```
MODULE Benchmark2;
(* Fibonacci series *)

FROM StopWatch IMPORT
  Start,
  Stop;

CONST
  iterations = 10;
  (* number of times benchmark executed*)
  length = 24; (* length of series *)

VAR
  i: INTEGER;
  useless: CARDINAL;

(****************************************)

PROCEDURE Fibonacci (n: CARDINAL): CARDINAL;

BEGIN
  IF n > 2 THEN
    RETURN Fibonacci(n-1) + Fibonacci(n-2);
  ELSE
    RETURN 1;
  END;
END Fibonacci;

(****************************************)

BEGIN
  Start;
  FOR i := 1 TO iterations DO
    useless := Fibonacci (length);
  END;
  Stop;
END Benchmark2.
```

End Listing Eight

Listing Nine

```
PROGRAM Benchmark2 (INPUT, OUTPUT);
{ Fibonacci Series }

USES StopWatch;

CONST
  iterations = 10;
{ number of times benchmark executed }
  length = 24; { length of series computed }

VAR
  i: INTEGER;

{****************************************}

FUNCTION Fibonacci (n: WORD): WORD;

BEGIN
  IF n > 2 THEN BEGIN
    Fibonacci := Fibonacci(n-1) +
    Fibonacci(n-2);
    END ELSE BEGIN
      Fibonacci := 1;
    END;
  END; {Fibonacci}
```

(Listing Eight begins on page 90)

Listing Nine

```
{************************************************************

BEGIN
  Start;
  FOR i := 1 TO iterations DO BEGIN
    EVAL (Fibonacci (length));
  END;
  Stop;
END. {Benchmark2}
```

End Listing Nine

Listing Ten

```
MODULE Benchmark3;
(* text filter *)

FROM StopWatch IMPORT
  Start,
  Stop;

FROM FileSystem IMPORT
  Response,
  File,
  Lookup,
  Reset,
  Close,
  ReadByte,
  WriteByte;

FROM InOut IMPORT
  WriteString,
  WriteLn;

CONST
  iterations = 10;
  FileSize = 1000;

VAR
  i: INTEGER;

(****************************************)

PROCEDURE Init;

VAR
  i: INTEGER;
  f1: File;

BEGIN
  Lookup (f1, 'DK:file1.doc', TRUE);
  FOR i := 1 TO FileSize DO
    WriteByte (f1, 'a');
  END;
  Close (f1);
END Init;
```

```
(****************************************)

PROCEDURE Filter;

VAR
  f1, f2: File;
  c: CHAR;

BEGIN
  Lookup (f1, 'DK:file1.doc', FALSE);
  IF NOT (f1.res = done) THEN
    WriteString ('file1.doc not found');
       WriteLn;
END;
Lookup (f2, 'DK:file2.doc', TRUE);
LOOP
    ReadByte (f1, c);
    IF f1.eof THEN
      EXIT;
    END;
    (* process c *)
    WriteByte (f2, c);
  END;
  Close (f1);
  Close (f2);
END Filter;

(****************************************)

BEGIN
  Init;
  Start;
  FOR i := 1 TO iterations DO
    Filter;
  END;
  Stop;
END Benchmark3.
```

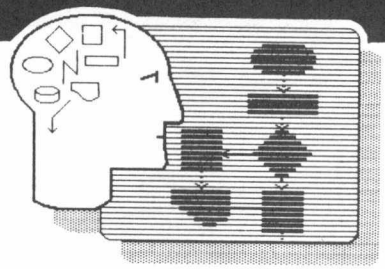
End Listing Ten

Listing Eleven

```
PROGRAM Benchmark3 (INPUT, OUTPUT);
{ text filter }

USES StopWatch;

CONST
  iterations = 10;
  FileSize = 1000;

VAR
  i: INTEGER;

{**********************************}

PROCEDURE Init;

VAR
  i: INTEGER;
  f1: file of BYTE;

BEGIN
  ASSIGN (f1, 'file1.doc');
  REWRITE (f1);
  FOR i := 1 TO FileSize DO BEGIN
    WRITE (f1, ORD('a'));
  END;
  CLOSE (f1);
END; {Init}

{**********************************}

PROCEDURE Filter;

VAR
  f1, f2: file of BYTE;
  c: BYTE;

BEGIN
  ASSIGN (f1, 'file1.doc');
  ASSIGN (f2, 'file2.doc');
  RESET (f1);
  REWRITE (f2);
  WHILE NOT EOF (f1) DO BEGIN
    READ (f1, c);
    { process c }
    WRITE (f2, c);
  END;
  CLOSE (f1);
  CLOSE (f2);
END; {Filter}

{**********************************}

BEGIN
  Init;
  Start;
  FOR i := 1 TO iterations DO BEGIN
    Filter;
  END;
  Stop;
END. {Benchmark3}
```

End Listings

THE SOFTWARE DESIGNER



by Michael Swaine

When this magazine was first published in January 1976, its title was *Dr. Dobb's Journal of Tiny BASIC Calisthenics and Orthodontia*, with the subtitle "Running Light Without Overbyte" (a phrase that one reader recently asked us to bring back). The first issue was filled with very tight code (and discussions of code) designed to fit in the limited memory of the first microcomputers. Hence "running light"; hence "without overbyte": it made sense in 1976 to squeeze every last byte out of your system. But now it's nine years and ninety-nine issues of *DDJ* later, and the memory space of the typical microcomputer has increased a thousandfold. Does it make any sense to talk about running light in 1985?

To get an answer to that question, I called Dennis Allison, the paradigmatic light runner, in Geneva.

Dennis is the "Do" of "Dobb." When and he and Bob Albrecht of *People's Computer Company* launched this magazine as a three-issue newsletter, they gave the job of naming their creation to Rick Bakalinsky, who was then *PCC*'s entire production department. Rick somehow got the idea that Dennis was "Don" and combined "Don" and "Bob" to produce "Dobb" (doubling the "b" to make it more namelike).

But Dennis is not just Do; among other things (*many* other things), he is a visiting lecturer in Geneva this month and next month will be doing something or other in Crete. Among the memorable things Dennis has done for *DDJ* over the years is a column on algorithms he wrote during the editorial tenure of Suzanne Rodriguez (whose reminiscence about Bob and Dennis appears in the "Festschrift for Dr. Dobb" elsewhere in this issue).

Dennis readily agreed that the world of microcomputer programming had changed radically since the early days.

"It's different now that memory is free," he told me. "When *DDJ* started, the Altair had 256 bytes of memory. That's a far cry from today, when for a reasonable sum of money you can get 64K of memory, and many machines have half a megabyte. That changes the whole scale of things."

Did that mean that the urge to create the great hack was no longer there? In *Hackers*, Steven Levy wrote about "the last hacker"; was he right in his assessment that commercial considerations had nudged out this motivation?

"It's still there," Dennis said. "There's a certain pride in making code as tight and as efficient as possible. I write compilers and I take pleasure in coding as efficiently as I can in writing [parts of the] compiler. But often it is done as an aesthetic exercise more than anything else. If you look at Don Knuth's code, you'll find that it's elegant and well-organized—and efficient in this same sense."

But Knuth is an academician and an artist in his work. He recently got into typefont design for purely aesthetic reasons. Is tight coding professionally important, or is it just pretty?

"It's still important to use code efficiently. The one thing that's virtually guaranteed is that you won't have enough cycles. You'll never have enough memory, because expectations [can] rise faster than costs [can] drop."

Was it, then, a question of where you invested your hacking energy?

"Yes. The rule is that 4% of the code does 50% of the work, and the place where you need to squeeze code and get the number of instructions down is in that 4%."

Tightening the critical algorithms till they squeak, rather than scrounging space for one more cell in the spreadsheet?

Dennis agreed, and then went on to talk about something near and dear to the hearts of *DDJ* readers for nine years: the virtue of publishing program listings.

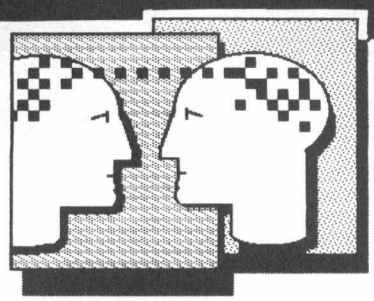
"*DDJ* is important as a journal because of the code. It was started to publish code and it has always published code. It publishes more code than any other magazine. And it's still my belief that people [programmers] learn from reading other people's code."

What they learn, of course, is how to program well: they pick up tricks, insights, algorithms, all of which are particularly well expressed when presented in the form in which they will ultimately be realized: as code. We hope to go right on publishing useful and educational code, including both programs significant in themselves and bits of code that demonstrate some exemplary algorithm or insight.

Toward the latter end, we will be reintroducing next month a column on algorithms, written by none other than Dennis Allison. The tentative title is "Allison on Algorithms." Well, it's descriptive.

With Bob Albrecht's column on "Realizable Fantasies," previewed in the "Festschrift for Dr. Dobb" elsewhere in this issue, the founders have returned to *DDJ*. **DDJ**





by Robert Blum

Buffered Disk I/O

The attempt to predict the length of a mechanical operation is easy when the system involves only a single device. The level of complication increases dramatically, however, when the system comprises two or more dependent devices. It simplifies the matter to realize that total system performance will be no greater than that of the slowest part.

A personal computer is a perfect example. The electronic portion of the computer performs its task at blinding speeds, but read a record from disk, and you quickly find the slowest part of the computer. A few techniques can minimize the mechanical delays encountered when transferring data to a disk drive. The most common examples are logical sector buffering and physical sector skewing. Both are typically a standard part of the operating system.

Further improvements are more difficult to achieve. Most require additional or improved hardware, which is impractical in many cases. This places the burden of any further improvement directly on the application program.

To increase the throughput of the CP/M, we can use any extra memory to buffer as much data as is practical. This manner of speeding things up circumvents one of CP/M's problem areas and puts all the system's resources to use—probably for the first time.

The CP/M problem has to do with the number of I/O operations necessary to write a record to disk. This was of no concern when disks were recorded in single-density format. But as improved hardware enabled higher bit rates, reliably larger sector sizes saw almost immediate use. This change allowed the total storage capacity of the media to double. There was, however, a price to be paid for this improvement.

CP/M uses a single I/O buffer to transfer data between memory and the disk drive. A 1:1 logical-to-physical

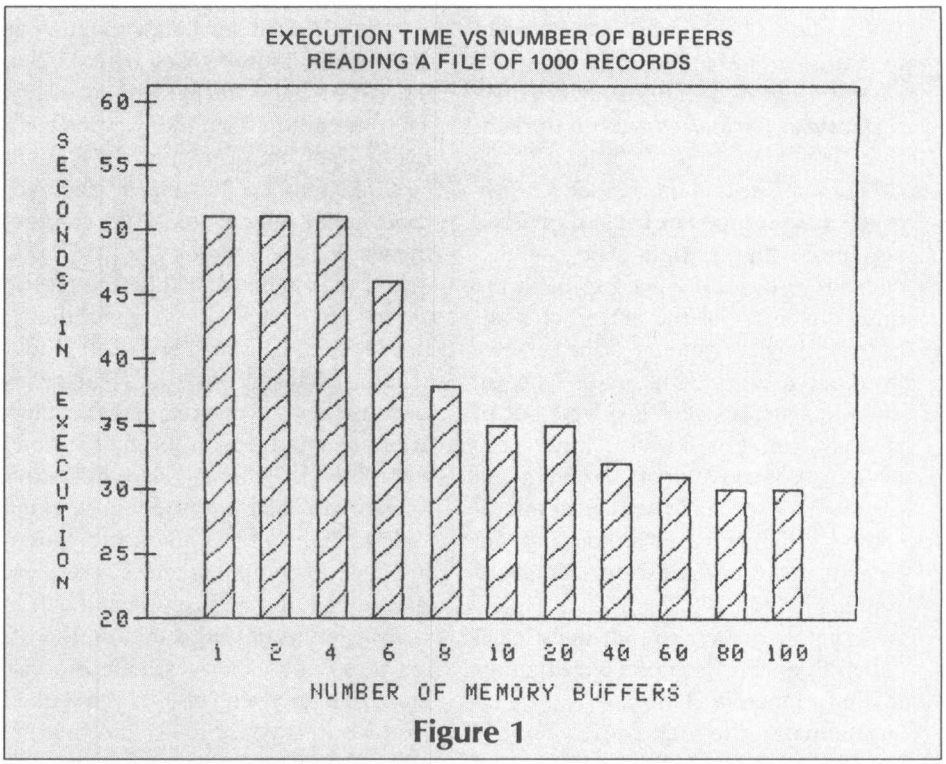


Figure 1

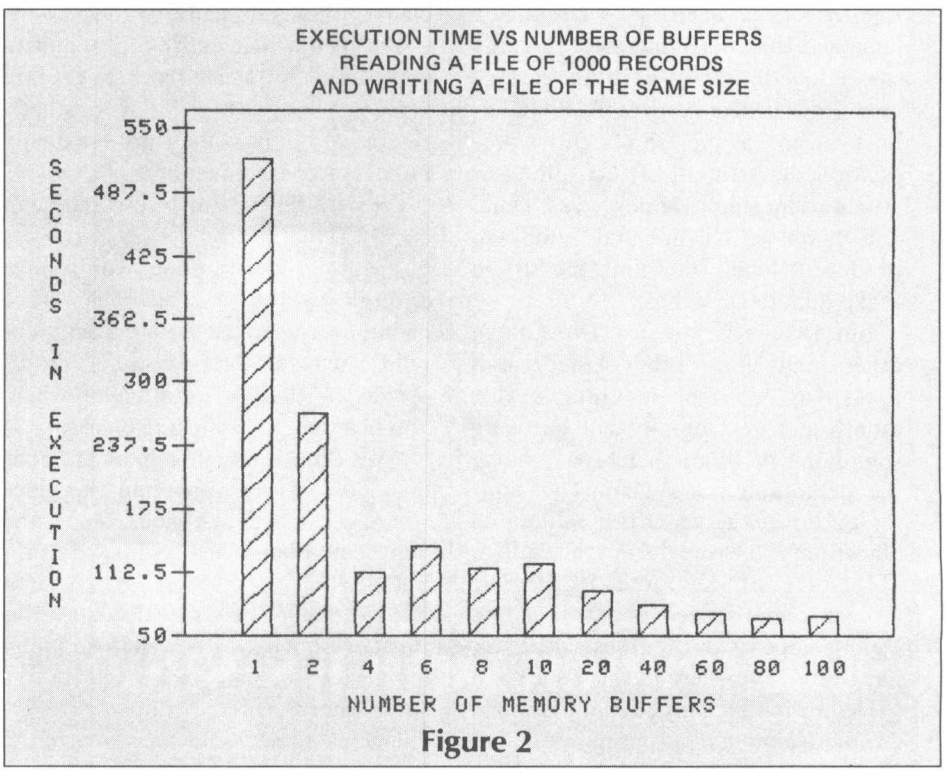


Figure 2

sector size relationship ensures transfer without buffer conflict. Changing the relationship to 4:1, however, can require up to three times as much disk I/O to transfer one logical sector of data. For a more complete explanation of buffer conflicts, refer to the December column.

Putting memory-resident data buffers to work speeding data through your machine can be as simple as specifying a parameter. When writing CBASIC as a Master's project, Gordon Eubanks made provision for the programmer to indicate how many data buffers to allocate. The last parameter of the OPEN statement (following the BUFF keyword) specifies how many record areas to set aside.

To plot the effect of using a varying number of data buffers against program execution time, I wrote two simple benchmark programs. The first (Listing One, below) requests how many data buffers to allocate, opens an input file, and executes 1000 sequential reads. Total program execution time is plotted in Figure 1 (page 98). The system I used to run the benchmark utilizes a physical buffer size of 512 bytes, which represents a logical-to-physical sector ratio of 4:1.

As you can see from Figure 1, there was no runtime improvement when from one to four buffers were allocated. When six buffers were used, however, runtime decreased markedly. An even larger decrease resulted when eight or more buffers were allocated.

The reason for the runtime improvement is that the physical sector buffer contains four logical sectors. In operation, the first program read request fills the physical sector buffer with four logical sectors. Each subsequent read request is satisfied directly from the memory buffer until it must be replenished from disk. When the data buffer exceeds the size of one physical sector, the possibility of reading two sectors in a single revolution of the disk is very good. Allocating six or more buffers reduced the program runtime by nearly 50%—obviously an improvement well worth pursuing.

Runtime improvements come even more quickly when a program progressively reads from one file and writes to another. To plot this condition, I wrote a second benchmark program (Listing 2, page 100). Its results are contained in Figure 2 (page 98). The effect of multiple data buffers on this scenario is even more dramatic.

The reason for the sudden decrease in runtime is the single I/O buffer maintained by CP/M. When only a single record buffer is chosen for the benchmark, most of the program's runtime is consumed by buffer maintenance. To understand better why this function consumes so much time, let's trace CP/M's data flow.

The first record read requires that the system perform a physical sector read from disk. The next I/O operation is a write. Because CP/M's buffer contains only data from a previous read, you can overwrite it without regard to destroying data. The next operation is another read. But before the selected physical sector can be brought in from the disk, the current CP/M buffer must be written onto disk; otherwise, the data it contains will be overwritten. The next operation is another write, but first the physical sector written onto disk during the last write operation must be read back into memory for updating.

From this study of a typical data flow, we realize that each write operation includes a pre-read of the physical sector on all but the first logical sector of each physical sector. Also included in each read is a physical sector read to refresh CP/M's I/O buffer. Data buffering of only a single record in this situation can account for an I/O reduction of at least 50%.

To conclude this discussion, I want to share an assembly language routine (Listing Three, page 102) that also performs the data buffering function. SFXIO is a part of SYSLIB3 now available in the public domain and from Echelon, Inc. One particularly interesting concept used in this routine is the transfer of information between the application program and the subroutine by use of a control block. This method is especially effective in trimming the development cycle because it clusters all the relevant control information into a fixed area of memory.

I will leave any further discussion of this subroutine to you. If you have any questions about this subroutine, please write me in care of this column. If you want to see about getting SYSLIB3 on disk, you can reach Echelon at the following address: 101 First Street, Los Altos, CA 94022 (415) 948-3820.

DDJ

CP/M Exchange

Listing One

```
REM
REM THIS PROGRAM READS A FILE OF 1000 RECORDS WITH VARYING NUMBERS
REM OF MEMORY BUFFERS FOR THE PURPOSE OF SPEED COMPARISIONS
REM

100.0
        STRING  DRIVE,\
                FILE.NAME,\
                INPUT.DATA

        INTEGER RECORD.SIZE,\
                FILE.NUMBER,\
                RECORD.COUNT,\
                LOOP.COUNTER,\
                NO.OF.BUFFERS
```

(Continued on next page)

 (Listing Continued, text begins on page 98)

Listing One

```
        DRIVE = "C:"
        FILE.NAME = "SPEED.DAT"
        RECORD.SIZE = 128
        FILE.NUMBER = 1
        RECORD.COUNT = 1000
        NO.OF.BUFFERS = 1

200.0
        INPUT "Enter number of input buffers to use: "; NO.OF.BUFFERS
        OPEN DRIVE + FILE.NAME\
                RECL RECORD.SIZE\
                AS FILE.NUMBER\
                BUFF NO.OF.BUFFERS

300.0
        FOR LOOP.COUNTER = 1 TO RECORD.COUNT
        READ #FILE.NUMBER; INPUT.DATA
        NEXT LOOP.COUNTER
        CLOSE FILE.NUMBER
        END
```

```
                              1   - 51
                              2   - 51  R S
                              4   - 51  U E
                        M B   6   - 46  N C
                        E U   8   - 38  T O
                        M F   10  - 35  I N
                        O F   20  - 35  M D
                        R E   40  - 32  E S
                        Y R   60  - 31
                          S   80  - 30
                              100 - 30
```

LISTING 1 - READ ONLY SPEED TEST

End Listing One

Listing Two

```
REM
REM THIS PROGRAM READS AND WRITES A FILE OF 1000 RECORDS WITH VARYING
REM NUMBERS OF MEMORY BUFFERS FOR THE PURPOSE OF SPEED COMPARISIONS
REM

100.0

        STRING  DRIVE,\
                INPUT.FILE.NAME,\
                OUTPUT.FILE.NAME,\
                INPUT.DATA

        INTEGER RECORD.SIZE,\
                INPUT.FILE.NUMBER,\
                OUTPUT.FILE.NUMBER,\
                RECORD.COUNT,\
                LOOP.COUNTER,\
                NO.OF.BUFFERS

        DRIVE = "C:"
        INPUT.FILE.NAME = "SPEED.DAT"
        OUTPUT.FILE.NAME = "OUTSPEED.DAT"
        RECORD.SIZE = 128
        INPUT.FILE.NUMBER = 1
        OUTPUT.FILE.NUMBER = 2
        RECORD.COUNT = 1000
        NO.OF.BUFFERS = 1

200.0
        INPUT "Enter number of input buffers to use: "; NO.OF.BUFFERS
        OPEN DRIVE + INPUT.FILE.NAME\
                RECL RECORD.SIZE\
                AS INPUT.FILE.NUMBER\
                BUFF NO.OF.BUFFERS
       CREATE DRIVE + OUTPUT.FILE.NAME\
                RECL RECORD.SIZE\
```

(Continued on page 102)

Listing Two

```
                AS OUTPUT.FILE.NUMBER\
                BUFF NO.OF.BUFFERS

300.0
        FOR LOOP.COUNTER = 1 TO RECORD.COUNT
        READ #INPUT.FILE.NUMBER; INPUT.DATA
        PRINT #OUTPUT.FILE.NUMBER; INPUT.DATA
        NEXT LOOP.COUNTER
        CLOSE INPUT.FILE.NUMBER, OUTPUT.FILE.NUMBER
        END

                                    1    520
                                    2    269
                            M B     4    113   R S
                            E U     6    138   U E
                            M F     8    117   N C
                            O F     10   123   T O
                            R E     20   94    I N
                            Y R     40   80    M D
                              S     60   73    E S
                                    80   68
                                    100  69
```

LISTING 2 - READ AND WRITE SPEED TEST

Listing Three

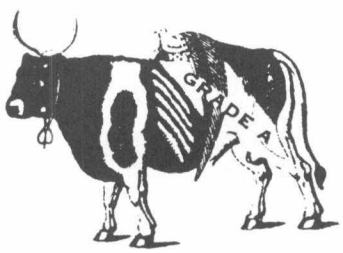
End Listing Two

```
;
; SYSLIB Module Name:  SFXIO
; Author:  Richard Conn
; SYSLIB Version Number:  3.0
; Module Version Number:  1.0
; Module Entry Points:
;       FX$GET          FX$PUT
;
;       SFXIO provides a group of routines which can perform byte-oriented
; file I/O with a user-defined buffer size.  All of these routines work with
; an I/O Control Block which is structured as follows:
;
;               Byte    Length (Bytes)  Function
;                 0             1       Number of 128-byte pages in
;                                               working buffer (set by user)
;                 1             1       End of File Flag (set and used
;                                               by SFXIO)
;                 2             2       Byte Counter (set and used by SFXIO)
;                 4             2       Next Byte Pointer (set and used by
;                                               SFXIO)
;                 6             2       Address of Working Buffer (set by user)
;                 8            36       FCB of File (FN and FT Fields set by
;                                               user, rest set by SFXIO)
;
;       The following DB/DS structure can be used in the calling program
; to implement the I/O Control Block:
;
;       IOCTL:
;               DB      8       ; Use 8 128-byte pages (1K)
;               DS      1       ; Filled by SFXIO
;               DS      2       ; Filled by SFXIO
;               DS      2       ; Filled by SFXIO
;               DW      WORKBF  ; Address of Working Buffer
;       IOCFCB:
;               DB      0       ; Current Disk (Inited by SFXIO)
;               DB      'MYFILE  '      ; File Name
;               DB      'TXT'           ; File Type
;               DS      24      ; Fill Out 36 Bytes
;       WORKBF: DS      1024    ; Working Buffer
;
;       All uses of the routines contain the address of IOCTL in DE.
; Note that if you use a buffer for input, DO NOT use it for output also!
;
```

(Continued on page 106)

Listing Three

```
;
;  EXTERNAL SYSLIB REFERENCES
;
        EXT     F$DELETE,F$MAKE,F$OPEN,F$CLOSE
        EXT     FXI$OPEN,FXO$OPEN,FXI$CLOSE,FXO$CLOSE
        EXT     F$READ,F$WRITE
        EXT     INITFCB
        EXT     HMOVB,SHFTRH,SHFTLH

;
;  CONSTANTS
;
TBUFF   EQU     80H     ; DMA BUFFER
CTRLZ   EQU     'Z'-'@' ; ^Z

;
;  MACRO ROUTINES FOR FXIO
;
PUTRG   MACRO
        PUSH    B
        PUSH    D
        PUSH    H
        ENDM

GETRG   MACRO
        POP     H
        POP     D
        POP     B
        ENDM

;
;  ENTRY POINT TO READ NEXT BUFFER FULL FROM DISK
;
FXIO0:
        LDA     EOF     ; CHECK FOR EOF
        ORA     A       ; ABORT IF ALREADY AT EOF
        RNZ             ; NZ IS ERROR CODE
        LHLD    FCB     ; PT TO FCB
        XCHG            ; ... IN DE
        LDA     BCNT    ; GET BLOCK COUNT
        MOV     B,A
        LHLD    BUFADR  ; GET ADDRESS OF BUFFER
FXIO1:
        PUSH    H       ; SAVE BUFFER PTR
        LHLD    FCB     ; PT TO FCB
        XCHG            ; ... IN DE
        CALL    F$READ  ; READ NEXT BLOCK (SECTOR)
        POP     H       ; GET PTR TO BUFFER
        JNZ     FXIO2
        LXI     D,TBUFF ; COPY INTO MEMORY BUFFER
        PUSH    B       ; SAVE COUNTER
        XCHG
        MVI     B,128
        CALL    HMOVB
        XCHG
        POP     B
        DCR     B       ; COUNT DOWN
        JNZ     FXIO1
        MVI     A,0     ; SET NO EOF
        JMP     FXIO3
FXIO2:
        MVI     A,0FFH  ; SET EOF
FXIO3:
        STA     EOF     ; SET EOF FLAG
        LHLD    BUFADR  ; PT TO FIRST BYTE AS NEXT BYTE
        SHLD    BYTENXT
        LDA     BCNT    ; SET BLOCK COUNT
        SUB     B       ; ADJUST
        MOV     H,A     ; RESULT IN HL
        MVI     L,0
```



(Continued on page 109)

Listing Three

```
        CALL    SHFTRH  ; SET COUNT
        SHLD    BYTECNT ; SET BYTE COUNT
        XRA     A       ; SET NO ERROR
        RET

;
;  ENTRY POINT TO FLUSH BUFFER TO DISK AND SET UP FOR NEXT WRITE
;
FX00Ø:
        LDA     BCNT    ; GET NUMBER OF BLOCKS
        MOV     B,A     ; ... IN B
        LHLD    BYTECNT ; NUMBER OF BYTES YET TO GO
        CALL    SHFTLH  ; SHIFT INTO H
        MOV     A,B     ; COMPUTE NUMBER TO WRITE
        SUB     H
        MOV     B,A     ; COUNT IN B
        LHLD    BUFADR  ; PT TO FIRST BYTE
FX001:
        MOV     A,B     ; CHECK FOR DONE
        ORA     A       ; Ø=DONE
        JZ      FX002
        DCR     B       ; COUNT DOWN
        PUSH    B       ; SAVE COUNT
        LXI     D,TBUFF ; PT TO WRITE BUFFER
        MVI     B,128   ; WRITE 128 BYTES
        CALL    HMOVB   ; COPY INTO BUFFER
        XCHG
        LHLD    FCB     ; PT TO FCB
        XCHG
        CALL    F$WRITE ; WRITE TO DISK NEXT BLOCK (SECTOR)
        POP     B       ; GET COUNT
        JZ      FX001
        XRA     A       ; SET ERROR CODE
        RET
;
;  ENTRY POINT TO INIT BUFFERS FOR NEXT WRITE
;
FX002:
        LHLD    BUFADR  ; SET NEXT BYTE
        SHLD    BYTENXT
        XRA     A       ; SET NO EOF
        STA     EOF
        LDA     BCNT    ; SET BLOCK COUNT
        MOV     H,A     ; ... IN HL
        MVI     L,Ø
        CALL    SHFTRH  ; SHIFT RIGHT ONE BIT
        SHLD    BYTECNT ; SET BYTE COUNT
        MVI     A,ØFFH  ; NO ERROR
        ORA     A
        RET

;
;  GET NEXT BYTE FROM BUFFER/FILE
;       ON INPUT, DE PTS TO IOCB
;       ON OUTPUT, A=CHAR AND Z FLAG SET IF PAST EOF
;
FX$GET::
        PUTRG           ; SAVE REGS
        CALL    PUTADR  ; PUT ADDRESSES
        LHLD    BYTECNT ; GET REMAINING COUNT
        MOV     A,H     ; NO MORE BYTES?
        ORA     L
        JNZ     FXGET1
        CALL    FXIOØ   ; READ NEXT BUFFER FULL
        JNZ     ERRET
FXGET1:
        LHLD    BYTENXT ; PT TO NEXT BYTE
        MOV     A,M     ; GET IT
        STA     BYTE    ; SAVE IT
        INX     H       ; PT TO NEXT
        SHLD    BYTENXT ; SET PTR
        LHLD    BYTECNT ; COUNT DOWN
        DCX     H
```

(Continued on page 112)

Listing Three

```
        SHLD    BYTECNT ; SET COUNT
OKRET1:
        CALL    GETADR  ; GET ADDRESSES
        GETRG
        MVI     A,0FFH  ; SET NZ FLAG
        ORA     A       ; NO ERROR RETURN
        LDA     BYTE    ; GET BYTE
        RET

;
;  PUT NEXT BYTE INTO BUFFER/FILE
;       ON INPUT, A=CHAR AND DE PTS TO IOCB
;       ON OUTPUT, Z FLAG SET IF WRITE ERROR
;
FX$PUT::
        PUTRG           ; SAVE REGS
        STA     BYTE    ; SAVE BYTE TO OUTPUT
        CALL    PUTADR  ; PUT ADDRESSES
        LHLD    BYTECNT ; CHECK TO SEE IF ANY ROOM LEFT
        MOV     A,H
        ORA     L
        JNZ     FXPUT1
        CALL    FX00    ; FLUSH BUFFER AND RESTART
        JZ      ERRET   ; ERROR
FXPUT1:
        LHLD    BYTENXT ; GET PTR TO NEXT BYTE
        LDA     BYTE    ; GET NEXT BYTE
        MOV     M,A     ; STORE IT
        INX     H       ; PT TO NEXT
        SHLD    BYTENXT
        LHLD    BYTECNT ; COUNT DOWN
        DCX     H
        SHLD    BYTECNT
        JMP     OKRET1  ; OK RETURN WITH BYTE AND ADDRESSES

;
;  ROUTINE TO PUT ADDRESS BUFFERS FOR LATER USE
;
PUTADR:
        XCHG            ; PT TO BUFFER WITH HL
        SHLD    BUFFER  ; PUT BUFFER ADDRESS
        MOV     A,M     ; GET BLOCK COUNT
        STA     BCNT
        INX     H
        MOV     A,M     ; GET EOF FLAG
        STA     EOF
        INX     H
        MOV     E,M     ; GET LOW COUNT
        INX     H
        MOV     D,M
        XCHG
        SHLD    BYTECNT ; PUT BYTE COUNT
        XCHG
        INX     H
        MOV     E,M     ; GET LOW COUNT
        INX     H
        MOV     D,M
        XCHG
        SHLD    BYTENXT ; PUT NEXT BYTE PTR
        XCHG
        INX     H
        MOV     E,M     ; GET LOW COUNT
        INX     H
        MOV     D,M
        XCHG
        SHLD    BUFADR  ; PUT BUFFER ADDRESS PTR
        XCHG
        INX     H
        SHLD    FCB     ; ADDRESS OF FCB
        RET

;
;  ROUTINE TO GET ADDRESS BUFFERS BACK FOR CALLING ROUTINE
;
GETADR:
```

(Continued on page 119)

```
        LHLD    BUFFER  ; PT TO BUFFER ADDRESS
        INX     H       ; SKIP BLOCK COUNT
        LDA     EOF     ; SET EOF FLAG
        MOV     M,A
        XCHG
        LHLD    BYTECNT ; SET BYTE COUNT
        XCHG
        INX     H
        MOV     M,E     ; PUT LOW COUNT
        INX     H
        MOV     M,D
        XCHG
        LHLD    BYTENXT ; SET NEXT BYTE PTR
        XCHG
        INX     H
        MOV     M,E     ; PUT LOW COUNT
        INX     H
        MOV     M,D
        RET

;
;  BUFFERS
;
BYTE:           DS      1       ; INPUT BYTE
BUFFER:         DS      2       ; STARTING ADDRESS OF I/O CONTROL BLOCK
;
;  THE FOLLOWING MIRRORS THE STRUCTURE OF THE I/O CONTROL BLOCK
;
BCNT:           DS      1       ; NUMBER OF BLOCKS
EOF:            DS      1       ; EOF FLAG (0=NOT AT EOF, 0FFH=AT EOF)
BYTECNT:        DS      2       ; NUMBER OF BYTES TO GO YET
BYTENXT:        DS      2       ; ADDRESS OF NEXT BYTE TO PUT/GET
BUFADR:         DS      2       ; ADDRESS OF WORKING BUFFER
FCB:            DS      2       ; ADDRESS OF FCB

        END
```

End Listings

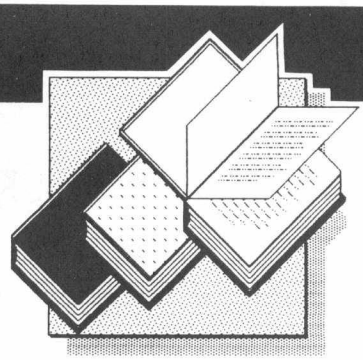
(Continued from page 121)

welded together. It can support over a hundred pounds. The 80 column width costs $19.95 and the 132 column width costs $24.95. Contact Zavie Sokoloff, 484 Lakepark Avenue, Suite 186, Oakland, CA 94610 (415) 531-0302 **Reader Service No. 133.**

BASIC

Morgan Computing Company has released version two of **Professional BASIC** and lowered the price from $345.00 to $99.00. Professional BASIC requires an IBM PC or AT. The new version includes windows into program execution and provides memory access of 640K. A $49.00 enhancement package provides 8087 and 80287 support. Contact Morgan Computing Company at 10400 N. Central Expressway, Suite 210, Dallas, TX 75231 (214) 739-5895 **Reader Service No. 117.** **DDJ**





Diana, An Intermediate Language for Ada

Edited by G. Goos, W. A. Wulf, A. Evans Jr., and K. J. Butler
Published by Springer-Verlag
$11.00, 201 pages
Reviewed by John R. Johnson

Most computer language compilers in recent years have used a concept called "common back end" in their implementation. We normally think of a compiler as a program that accepts a file of statements in some language as input and provides as output a file of machine language codes that will implement the intent of the input file on a specific computer. Many compilers were created in just this way. However, significant advantages can be gained by writing compilers for higher-level languages to provide their output in a more generic lower-level language for easy implementation on a variety of machines. The two major advantages to this approach are portability and flexibility.

Portability can be gained by using a low-level language as an intermediate or "common back end" language. My high-level compiler itself, which represents a serious programming effort, can be written in a high-level language or even in the intermediate language it produces. This simple intermediate language then can be implemented on a variety of machines to create a family of high-level compilers for many different computers. A large part of the work of high-level language implementation has to be done only once. A good example of this approach is the p-code intermediate language used in a number of implementations of the Pascal language.

Flexibility is gained by using the same intermediate language for a number of different high-level language compilers. It becomes a relatively easy task to combine modules or routines written in different source languages into one program. A good example of this is the Microsoft family of high-level language compilers: they all compile to a common relocatable assembly language, and the modules are linked with a common linker/loader to produce executable code.

To make use of this powerful technique, you need an intermediate language that is both easily implemented and capable of supporting all of the constructs of the high-level language. Because Ada contains a number of features in its specification that are straight from computer science research projects, a suitable intermediate language for Ada did not exist.

Diana, An Intermediate Language for Ada not only documents such a language, but it provides an intermediate language usable for virtually any modern high-level language such as Modula II or Pascal. The book is intended as both an introduction to Diana and a language reference manual for Diana.

This book is part of a larger series, "Lecture Notes in Computer Science," published by Springer-Verlag. The books in this series that I have seen are intended primarily for graduate students in computer science. This book is not an exception.

It is a clear and concise definition and description of a language that is a powerful tool in modern high-level compiler construction. That the book is obviously written for experienced compiler writers is not a criticism: by its nature, it would be of only limited usefulness to anyone not currently engaged in designing a high-level language compiler.

The book is well conceived and presented, and I would suggest that anyone considering writing a compiler for the Ada language buy it. It should also be in the library of any modern systems programmer. If your programming interests do not extend to compiler design, however, *Diana* would most likely fail to repay the considerable effort required to read and understand it.

Ada, An Advanced Introduction Including Reference Manual for the Ada Programming Language

by Narain Gehani
Published by Prentice-Hall, Inc.
$19.95, 291 pages
Reviewed by John R. Johnson

The Ada programming language has stirred up a great deal of interest and controversy in recent years. As a result, many books have appeared on the market to expand upon Ada. This is the best one I have seen.

The book is written for the experienced programmer, preferably someone who has had some exposure to one of the ALGOL-derived languages such as Pascal or PL/I. Gehani assumes the reader is familiar with standard programming concepts such as type definitions and control structures.

The first portion of the book is devoted to a thorough but brief description of the correct Ada syntax and form for all programming constructs not unique to Ada. Gehani covers the Ada character set, type and function declarations, and subtypes with ranges and limits. The definitions are written in the concise format normally used for language specification and documentation. Any experienced programmer should have no difficulty following the material. Gehani has illustrated each situation with several examples of correct Ada code.

He then spends the bulk of the book thoroughly discussing several features of Ada that are shared with few, if any, common languages. There is an excellent discussion of the Ada "package"

concept for information hiding and data encapsulation; this is the basic facility or modularity in Ada programs. The contents of any "package" may be altered at will as long as the imports and exports from the package remain consistent with the package specification. Ada provides for separate compilation of packages. Gehani uses many examples to illustrate the specific syntax associated with the concept.

One of Ada's unusual features is the provision for concurrency of portions of a program. The author has given us a clear and concise look at the problems of concurrency. Ada has a number of features to provide coroutines, concurrent actions, and data passing between independent concurrent tasks and procedures within a program. Once again we have numerous examples to show how to handle handshaking between asynchronous independent processes using correct Ada syntax.

Ada provides a unique and complete system for exception handling within programs. Specific exception handling routines may be added at the end of any block to process any exceptions that may arise anywhere in the block. When an exception occurs, control is transferred to the exception handler. When the exception handler is finished, the block where the exception handler resides is terminated and control passes back to the enclosing block. If the exception handler is omitted, control passes back to the exception handler in the enclosing block. This concept is presented clearly with some excellent examples.

The generic facilities of Ada are an interesting variant of the standard function library concept used in C and other separate-module compilation languages. A generic package may be "instantiated" by a declaration in a new package with appropriate parameters; the result is a unique and specific application of a generic package, such as a sort routine. This unusual approach is presented with some useful examples.

The balance of the first part of the book is devoted to a discussion of the separate compilation features and those features that may be implementation-dependent. Here, also, Gehani has used precise examples.

For all of the examples in the book, the author has used stepwise refinement to work from a general problem statement to a set of specific routines that implement the solution in Ada. It is interesting to follow the refinement process: all of the steps are included. The book would be worth buying just to trace this stepwise refinement process.

The second half of the book is the complete reference manual for the Ada programming language—reprinted from the ANSI/MIL-STD-1815 A. This is the current official definition of the Ada language. Considerably less readable than the first part of the book, it is indispensable for anyone who wishes to attempt an implementation of the Ada language.

The book includes a comprehensive bibliography for anyone interested in additional reading on the topics covered. The two halves of the book are indexed separately for reference purposes. The indexing is well done.

This book will quickly become dog-eared on the desk of any programmer working in the Ada language. When we are forced to begin using Ada in our shop, I am going to insist that every programmer be provided with his or her own copy. It is an essential tool for the new or experienced Ada programmer.

New Books

Hackers

Steven Levy
Anchor Press/Doubleday, Garden City, New York, 1984
$17.95, 458 pages
ISBN 0-385-19195-2

Steven Levy has taken on the exegesis of the Hacker Ethic. In breathless, Right Stuff prose, he follows the followers of the hacking dream from the basements of MIT through the garages of Silicon Valley right up the slopes of the Sierra, where game programmers are treated like rock stars. Steven, did you retain the film rights?

How to Copyright Software

M.J. Salone
Nolo Press, Berkeley, California, 1984
$21.95, 256 pages
ISBN 0-917316-7

It is perhaps only to be expected that a 256-page book explaining how to copyright software would cost more than twice as much as a 475-page book telling how to get it free.

The Free Software Catalog and Directory

Robert A. Froehlich
Crown Publishers, Inc., New York, New York, 1984
$9.95, 475 pages
ISBN 0-517-55448-8

Cross-referenced directories for the CPMUG and ACJ-NJ's SIG/M public domain software libraries, plus lists of bulletin boards and users' groups throughout the U.S.

Digital Deli

Steve Ditlea, Ed.
Workman Publishing Co., New York, New York, 1984
$12.95, 382 pages
ISBN 0-89480-591-6

No way to summarize this book: it contains a short piece (all the pieces in this book are short) on how to use a word processor, a quarter-page PR photo of IBM CEO John Opel, arguments for the Eniac being an Aquarius and for the Mac being short skis for the mind, cartoons, computer art, quizzes, and memoirs. The title is not original, but it fits.

The Netweaver's Sourcebook

Dean Gengle
Addison-Wesley Publishing Company, Menlo Park, California, 1984
$14.95, 326 pages
ISBN 0-201-05208-3

How refreshing to hear that "it's still perfectly socially acceptable to hate machines arbitrarily and capriciously, *especially* answering machines." This book takes a remarkably elevated perspective on networks and communications.

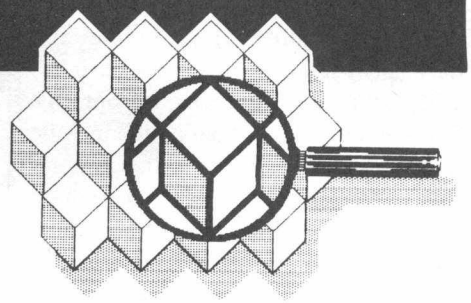
The Whole Earth Software Catalog

Stewart Brand, Ed.
Quantum Press/Doubleday, Garden City, New York, 1984
$17.50, 208 pages
ISBN 0-385-19166-9

The Brand hand is as evident here as in Whole Earth non-software catalogs, and admirers of wit, irreverence, controversy, and—um, wholearthness will be glad of that. Dr. Dobb contributed to this collection of recommended programs.

(Continued on page 119)

OF INTEREST



by R.P. Sutherland

Disk Memory

The notion of having the Library of Congress in the palm of your hand is coming closer to reality every day. A 4¾-inch compact disk ROM with **550 megabytes of storage** capacity on one side has been developed by Nippon Columbia of Kawasaki, Japan. Contact Robert Heiblim, Denon America, Inc., 27 Law Drive, Fairfield, NJ 07006 (201) 575-7810 **Reader Service No. 111.**

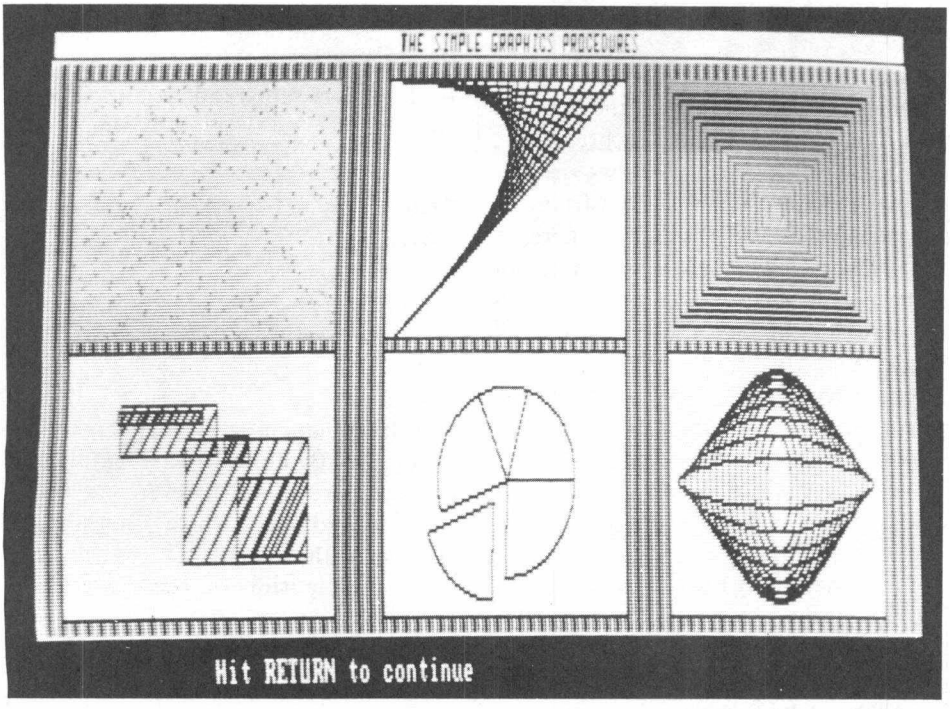
Hewlett-Packard has entered the OEM market with a 3½-inch 10 megabyte disk drive. The **HP 97501A** 3½-inch micro-Winchester is priced at $400.00 each in quantities of 10,000. For more information, write: Inquiries Manager, Hewlett-Packard Company, 1020 N.E. Circle Boulevard, Corvallis, OR 97330 **Reader Service No. 113.**

A diagnostic diskette for the IBM PC, Apple II, TRS-80, and Commodore 64 is available from Dymek Corporation. The disk is called **RID** (Recording Interchange Diagnostic), and it performs seven tests on disk drives: drive speed, radial position, hysteresis, write function, erase crosstalk, minimum signal-to-noise, and clamping. The disk is easy to use, costs $34.95, and is available from Dymek Corporation, 1851 Zanker Road, San Jose, CA 95112 (408) 947-8700 **Reader Service No. 115.**

Utilities

A high-resolution graphics system for Turbo Pascal has been released by Borland International. **Turbo Graphix Toolbox** provides procedures to create the contents of windows and to allow copying from window to window as well as scrolling of windows horizontally and vertically. There are also procedures that provide RAM storage of screen images and so permit animation in real time application, up to 500 images per second. (See photo below.) Turbo Graphix Toolbox comes with complete commented source code on disk for $49.95 from Borland International, 4113 Scotts Valley Drive, Scotts Valley, CA 95066 (408) 438-8400 **Reader Service No. 123.**

Stellation Two has disk buffering software for the IBM PC. **Invisible Optimizer** loads automatically in the unused memory space of any expansion board for the IBM PC. It works by keeping track of all disk activity and copying the most needed data into system memory. Applications then obtain the most needed data from RAM instead of from the disk drive. Invisible Optimizer is available for $69.00 from Stellation Two Inc., 26 W. Mission St., P.O. Box 2342, Santa Barbara, CA 93120 (805) 569-3132 **Reader Service No. 125.**



68000 Development

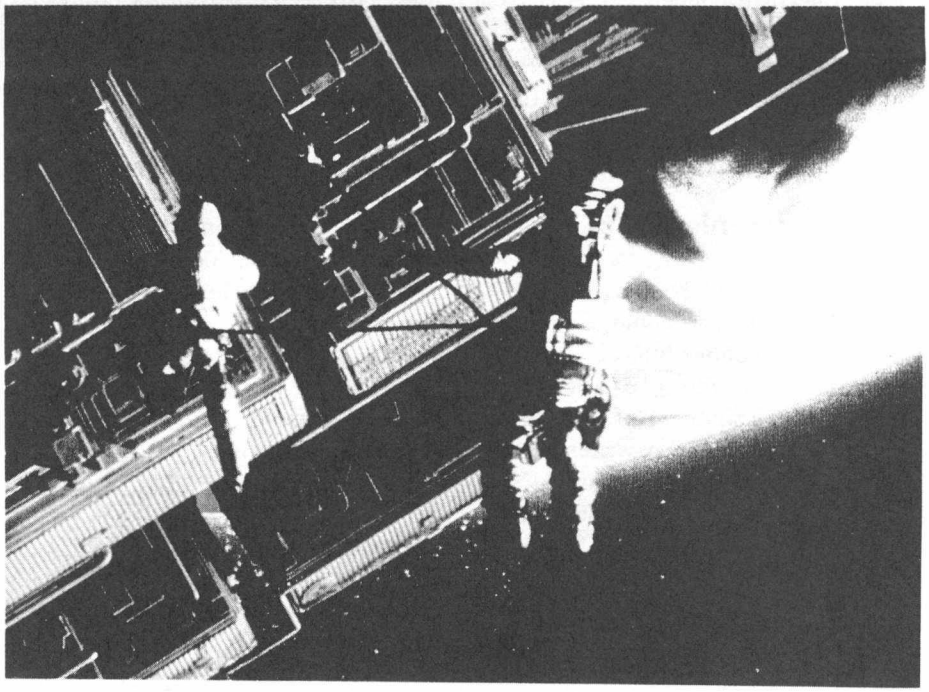
PC-68K is a new product from Language Resources that allows 68000 development on an IBM PC-XT or PC-AT. PC-68K provides a symbolic debugger, linker/locator, Motorola compatible macro assembler, and an IEEE floating point package. Pascal and C compilers are available as options. The PC-68K plug-in board has an 8 MHz 68000 CPU, a memory management subsystem, and 256K of RAM for 68000 family development, which can also be used as expansion memory by PCDOS programs. The basic PC-68K package costs $2995.00. Language Resources Inc., 4885 Riverbend Road, Boulder, CO 80301 (303) 449-8087 **Reader Service No. 119.**

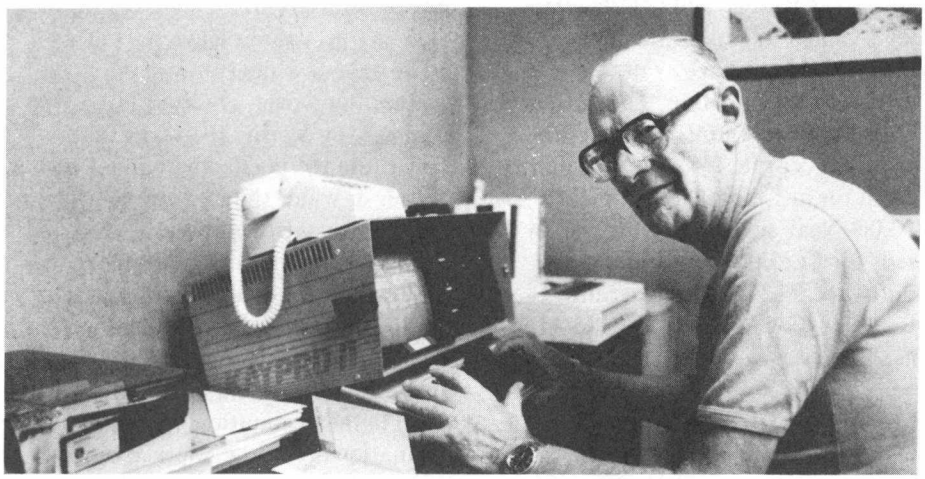
A **Modula-2** development system for the Pinnacle microcomputer has been introduced. Interesting details include a Modula-2 native code com-

piler and a 12 MHz 68000. The package is available for $3995.00 from Pinnacle Systems, 10410 Markison Road, Dallas, TX 75238 (214) 340-4941 **Reader Service No. 121.**

2010

Arthur C. Clarke's *2001: A Space Odyssey* was written, filmed, and published before 1969, when Neil Armstrong took that giant leap. In the preface to ***2010: Odyssey Two***, in reference to *2001*, Clarke mentions "uncanny instances of nature imitating art." In *2001*, the novel, the spaceship Discovery "slingshots" its way by Jupiter to reach the Saturnian moons. The Voyager space probes in 1979 used the same technique. When Stanley Kubrick set the third confrontation between Man and Monolith among the moons of Jupiter, Io, Europa, Ganymede, and Callisto were just points of light in our telescopes. The discoveries of the Voyagers have since allowed us to gaze upon the surfaces of these fantastic moons. The astronauts of Apollo 8 had already seen the film when, in 1968, they became the first men to see the other side of the moon. They were tempted to report the discovery of a large black monolith! More recently, the crew of Skylab televised their discovery that they could run around the interior in much the same way in which Frank Poole, in the film, runs around the circular track of the giant centrifuge.

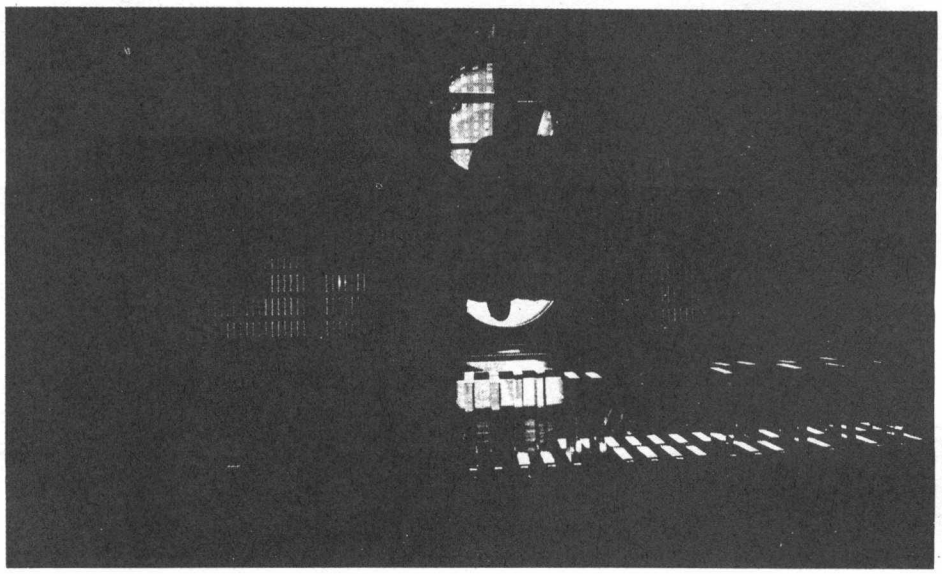




Arthur C. Clarke, one of the world's best known and most respected authors of science fiction and science fact literature, poses before one of the Kaypro II computers used to relay information between the production office and Clarke's Sri Lanka home.

Pasadena's Jet Propulsion Laboratory (JPL) claims that the vision of Jupiter in the film *2010* is better than its own. The visual effects supervisor fed the raw JPL data from the Voyager fly-by mission into a Cray. The Voyager data, processed and correlated with information about cloud vortexes, produced a detailed moving image of Jupiter.

I look forward to viewing the resurrection of HAL. (By the way, in the novel there is a character who flatly



denies the correspondence of HAL to IBM.) One Hitchcockian detail: Arthur Clarke appears as a pigeon-feeding wino on a park bench outside the White House.

Telecommunications

What do you need 2400 baud for? Well, Arthur Clarke could have halved his telephone bills across thirteen and a half time zones when he sent his film script drafts from Sri Lanka to his screenplay writer in the United States. Both Hayes and U.S. Robotics have introduced **2400 baud modems**. The Hayes costs $899.00 and the U.S. Robotics costs $895.00. Hayes is at 5923 Peachtree Industrial Blvd., Norcross, GA 30092 **Reader Service No. 101**, and U.S. Robotics is at 1123 W. Washington Blvd., Chicago, IL 60607 **Reader Service No. 103.** Where do we go from here—9600 baud?

If you can't justify $900.00 for a 2400 baud modem, consider what $200.00 will do. Business Computer Network is offering a **1200 baud modem** with SuperScout software for $200.00. Business Computer Network, Inc., is located at 1000 College View Drive, Riverton, WY 82501 (800) 446-6255 **Reader Service No. 105.** In the same spirit, Artisoft Inc. has introduced communications software for $49.95. **Envoy** is a menu-driven telecommunications program with autodial capabilities and XMODEM transfer protocols. Contact Artisoft, Inc., P.O. Box 41436, Tucson, AZ 85717 (602) 327-4305 **Reader Service No. 107.**

A newly patented device called **Shuttle Communicator** allows users to download software from an AM or FM station with a radio interfaced to the computer! Shuttle Communicator costs $70.00. (See photo below.) For additional information, contact Robert Hardwick, The Microperipheral Corporation, 2565 152nd Ave. N.E., Redmond, WA 98052 (206) 881-7544 **Reader Service No. 109.**

Miscellany

Stars and Planets on the Apple II
The Observatory is a program that turns your Apple II into an electronic celestial sphere containing stars, star clusters, galaxies, and nebulae. Halley's Comet and all of the planets are included. Users have available nine levels of magnification. The author, Gary Lassiter, claims that the program is accurate for any place on the earth for any moment of time in a 10,000 year span. The Observatory sells for $125.00 from Lightspeed Software, 2124 Kittredge, Suite 185, Berkeley, CA 94704 (415) 486-1165 **Reader Service No. 127.**

Macintosh Disk Backup
COPY II MAC makes backups of protected Macintosh software. The package includes utilities to lock/unlock, protect/unprotect, and make files visible/invisible. COPY II MAC is available for $39.95 from Central Point Software, Inc., 9700 SW Capitol Highway, Suite 100, Portland, OR 97219 (503) 244-5782 **Reader Service No. 129.**

Micromouse Contest
For those who think that they can build a small self-contained robot able to navigate quickly a complicated maze, a contest will be held by the IEEE Computer Society at the Microprocessor Forum. The Microprocessor Forum will be held March 31 to April 4, 1985, at Bally's Park Place Hotel, Atlantic City, NJ. The Micromouse Contest aims to find a robot able to negotiate a maze in the shortest period of time. The robot cannot use an energy source employing a combustion engine, cannot leave part of its body behind while running the maze, cannot jump over, climb, scratch, damage, or destroy the walls that constitute the maze, and cannot be longer or wider than 25 cm. It also cannot be controlled by radio or wire. A microcomputer must be incorporated into the design to control the sensors and drive motors, to memorize the progress of the mouse through the maze, and to calculate the shortest path to the destination. For more information about the contest, write to: Micromouse Contest, IEEE Computer Society, P.O. Box 639, Silver Spring, MD 20901.

***Micro Cornucopia* Gets Sol Libes**
This month, Sol Libes, founder of *Microsystems*, will begin a regular column in *Micro Cornucopia*. The column will cover the latest public domain software releases. *Micro Cornucopia* is the "Single Board Systems Journal" that supports the Kaypro, the Xerox 820, and the Big Boards. *Micro Cornucopia* is located at P.O. Box 223, Bend, OR 97709

Printer Pedestal
Zavie Enterprises has a desktop printer stand of a useful design. The **Printer Pedestal** comes in dark brown and is made of ¼-inch metal rods arc-

(continued on page 119)



SPECIAL PROLOG ISSUE

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

March 1985

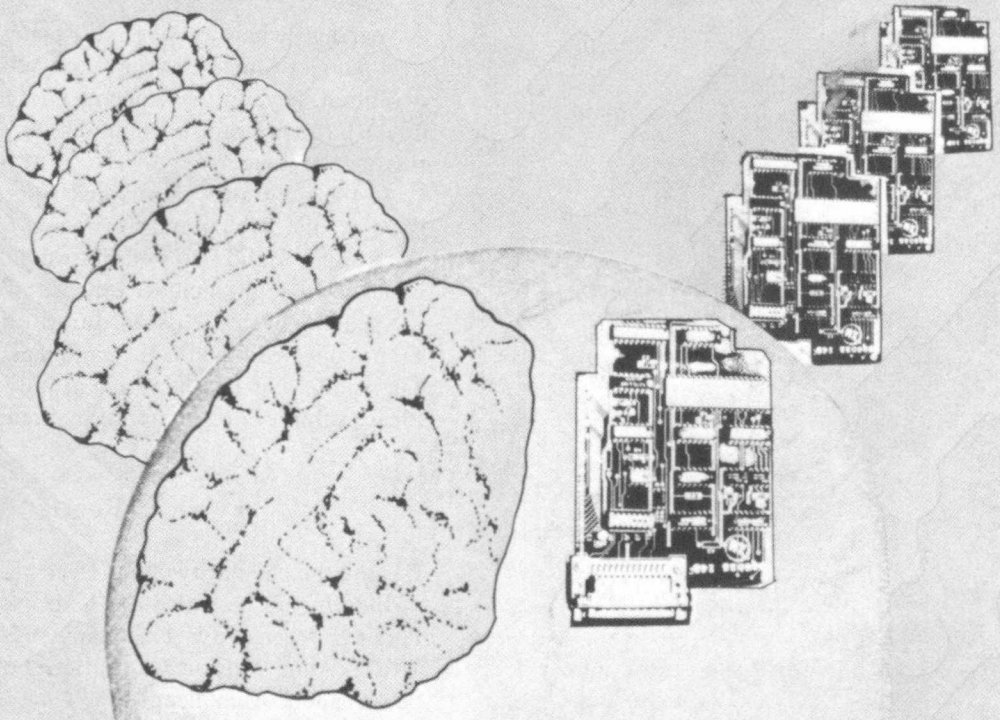
$2.95 (3.50 Cana

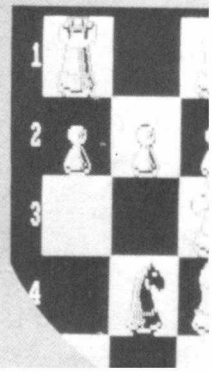
How to Program in Prolog

AI Competition Winner

New Column: The C Chest

A Proposed Public-Domain "Unix"







Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Randy Sutherland*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Polly Koch*
Typesetter *Jean Aring*

Production
Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover *Claudia Majewski*

Advertising
Advertising Director *Stephen Friedman*
Advertising Sales *Walter Andrzejewski, Shawn Horst, Beth Dudas*
DDJ Classifieds *Alex Williams*
Advertising Coordinators *Alison Milne, Lisa Boudreau*

Circulation
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Circulation Assistant *Kathleen Boyd*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounting Coordinator *Mayda Lopez-Quintana*
Accounts Payable *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

March 1985
Volume 10, Issue 3

CONTENTS

Prolog

A year ago when we were still being published by the old *People's Computer Company* in Menlo Park, I asked Mike Swaine if Artificial Intelligence was, by definition, beyond the scope of microcomputers. What resulted from that Sunday afternoon brainstorming session was the AI programming competition and this special issue.

As the programs and manuscripts rolled in we found ourselves building the issue around Prolog and we wished for a public domain tiny-Prolog to publish. We haven't found that yet, but we did get some very good Prolog pieces. Future issues of *DDJ* will include responses from Lisp people.

Speaking of future issues: June is the special telecommunications issue, as well as the copy deadline for the annual algorithms issue which will run in September. The annual Forth issue which usually runs in September will appear in October. Please submit Forth material by the Fourth of July.

Reader Ballot Results

Despite the popularity of our December 1984 focus on Unix, the article voted best in the Unix issue was not a Unix piece. Henry Seymour's "An Introduction to Parsing" received twice as many "best article" votes as the second most popular article, Alan Walworth's "Varieties of Unix." By the way, may we assume that because nobody challenged the Unix family tree on page 35, we got it right?

Mac Erratum

In the January Fatten Your Mac article, there is a discrepancy between the text on page 20 and Figure 1 on page 23. The figure correctly shows pin 8 soldered, but the text refers to pin 7 of the memory select IC. Readers who have called to point out the error inform us that the error in the text is obvious to anyone who works through and understands the article and the figures. We hope so.

This Month's Referees

Dave Cortesi, Resident Intern
Neil Jacobstein, Teknowledge
Scott Mace, Free-lance Computer Writer
Paul Reinholdtsen, Stanford University
Pekka Sinervo, Stanford Linear Accelerator

Randy Sutherland
Editor

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

Programming in Logic 36
by John Malpas
The development of logic programming from Frege to ICOT; features and applications of Prolog; implementations of Prolog. **(Reader Ballot No. 192)**

Tour of Prolog 44
by D. E. Cortesi
A description of Prolog's style from a programmer's point of view. **(Reader Ballot No. 193)**

Tax Advisor: A Prolog Program Analyzing Tax Issues 64
by Dean Schlobohm
The winner of the AI programming competition: this Prolog program assists attorneys in analyzing the tax rules in Section 318 (a) and explains its questions and conclusions. **(Reader Ballot No. 194)**

COLUMNS

C Chest 10
by Allen Holub
This new column takes over where the "C/Unix Programmer's Notebook" left off; this month, root module modifications for the Aztec CII compiler. **(Reader Ballot No. 190)**

Realizable Fantasies 30
by Richard Stallman
Richard Stallman outlines a proposal for GNU, a public domain Unix-like operating system. **(Reader Ballot No. 191)**

16-Bit Software Toolbox 96
by Ray Duncan
A Z-8001 CPU for the S-100 bus; Mac notes; a fix for the 80286 BOUNDS instruction; DOS 3.0 differences and bugs; 8086 assembly language subroutines. **(Reader Ballot No. 195)**

CP/M Exchange 108
by Bob Blum
An RSX to correct the handling of random read errors under CP/M Plus; a PIP patch for CP/M Plus. **(Reader Ballot No. 196)**

DEPARTMENTS

Editorial 6

Letters 8

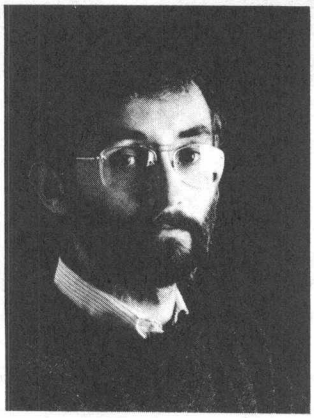
Reviews 116
Excalibur Technologies' SAVVY PC, Version 4.0.
(Reader Ballot No. 197)

Of Interest 122

DDJ Classifieds 126

Advertiser Index 128

EDITORIAL



In one of my favorite novels the very existence of a certain island seems to depend on the direction from which you approach it: sort of a human-scale quantum uncertainty effect. Early in the novel a philosopher of science and a rather unimaginative psychologist discuss the island over lunch.

"The island doesn't bother you, does it, Dr. Blank," the philosopher of science says around a bite of toast, "because you look at it one side at a time."

One side at a time. Well, yes. That's a fundamental human limitation: we can only see one side of anything at a time. It comes of having our eyes set so close together. But that limitation shouldn't keep us from paddling around the island to discover how things look from the other side.

I recently spoke before several hundred engineers at a major electronics research center. Roughly half of them had within the past year been deeply involved in a project that, to their chagrin, the company ultimately decided would not become a product. The company had probably acted in its own best economic interest in putting the kibosh on the project, although some of the engineers seemed to think the decision might have been a mistake. If the decision was not in the company's own self-interest, then of course it was a mistake. Those are the alternatives.

Well, maybe in that case. But sometimes companies—and people—deliberately act contrary to their own perceived economic self-interest. It's generally assumed that, other things being equal, some money is better than no money, and we expect others to choose some over none whenever they perceive that they have such a choice. Corporate strategies are built on such an assumption about competitors' motives. But the assumption is not always warranted in fact.

Say, for example, that a bright programmer, convinced that software should be free, sets out to develop, with a little help from his friends, a Unix-compatible operating system and subsequently distributes it free to anyone who wants it. That's not a hypothetical example; it's exactly what Richard Stallman is proposing in his "GNU Manifesto," this month's Realizable Fantasy.

Say you're one of the hordes of people who have an economic interest in the success of Unix. If Stallman's fantasy is realized, you will soon face a competitor who is giving away his product, a strategy that some might think gives him a competitive edge. What do you do?

The fantasized confrontation between Unix and GNU could occur in any part of the software market. The number of programmers who could write a word processor or spreadsheet program better than many currently on the market grows daily. Large, complex programs are not exempt from such competition: as Stallman points out, modularizing the task solves the mere size problem, and a little reverse engineering unravels much complexity.

The fact is, both commercial software (*pace* Richard Stallman) and free software—written and distributed for fun or for pride of accomplishment or for philosophical reasons—are here to stay. You can't keep people from writing programs. But free software isn't going to destroy the market for commercial software; programs are too complex to be evaluated strictly on the basis of price.

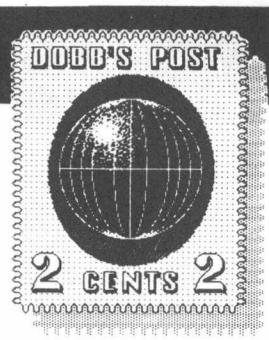
On the other hand, smart commercial software developers will keep themselves aware of what free software exists, and not borrow trouble by trying to compete with free products. If they are really smart, they will also paddle around to the other side and *imagine* what free software might be written.

What is *Dr. Dobb's* view on the issue? We're in favor of good software, free and commercial, and have been promoting both kinds since 1976. I think we see both sides.

But not at the same time.

Michael Swaine

Michael Swaine



Fantasies

Dear *DDJ*:
I read with interest your piece, "A Realizable Fantasy," in the November [1984] issue. Clearly, a product such as you describe would be potentially very useful as a productivity tool. A close approximation to such a tool may already exist!

Recently, on behalf of a client, we did a brief evaluation of a package called the Excelerator from Index Technology Corporation. The product—software + hardware—does exist and seems to function pretty much as described in the brochure. It's a very powerful utility and at least it's a start in the direction you outlined.

Not surprisingly, however, our client balked at the $9,500 price tag—nearly twice what his XT cost! I agree that in a micro environment this is an impossible price, but there is no reason to suppose that such a capability can't be built and offered at substantially less. I hope someone does it soon.

Sincerely,
R. A. Langevin
7621 Fontaine St.
Potomac, MD 20854

Dear *DDJ*:
Some humble fantasies: extra CP/M BDOS calls for full-screen I/O, multi-character key input (function keys) and a utility that translates a submit file into a .COM (or .CMD) file.

Nelson Richardson
740 West End Avenue #21
New York, NY 10025

BU Bits

Dear *DDJ*:
Ian Ashdown's BU program published in the January [1985, #99] *DDJ* is a fine contribution to the growing list of public domain programs that can perform incremental backup given correct operation of the CP/M "archive bit" (t3′). As the author of one of the best known commercial programs for performing the same function, I feel I should comment on a serious factual error in Ian's article. He states: ". . . if you use a disk utility to set t3-prime to true in the file's directory entry, you will find that the BDOS resets it to false (zero) whenever the directory entry is changed. Since this means that the file has been opened, written to, and closed (or else renamed) by the BDOS, 3t-prime is apparently an undocumented attribute bit that indicates when a file has been updated." I'm not sure where this idea got started, but for CP/M 2.2 it *just ain't so.* (Ian Ashdown is not the only notable to make this mistake: it also occurs on page 222 of David E. Cortesi's generally outstanding book *Inside CP/M.*) If you don't believe me, spend a few minutes with BU, set t3′ on a file, change the file with a standard CP/M utility, and just see if the bit is reset.

Alas, while t3′ *is* a functional archive bit under CP/M 3.0 and MPM, under CP/M 2.2 it acts just like any other attribute bit. There is, in fact, no code whatsoever in BDOS to reset this bit automatically when a file is changed, as those who have tried to implement Kelly Smith's original ARCHIVE.ASM program know full well. The BU program is almost useless under 2.2 unless BDOS is patched to install suppport for the archive bit. Our own product, Qbax, will function without patching for programs such as word processors and compilers that change files by rewriting them from scratch, but even our product requires a BDOS patch to catch random access updates in place. Such patches are extremely tricky, and while public domain software is wonderful it is also nice to have a company to rely on for support when the patch doesn't work. Some of the versions of the BDOS patch for ARCHIVE.ASM that have appeared in various places have had some serious bugs.

Ian should have gotten a tip-off from his own comments: "Speaking of file copy utilities, a few are available that, under certain circumstances, will write a file without resetting its Archive attribute. One of these, oddly enough, is DRI's own PIP.COM." Not odd, just normal for CP/M 2.2! Vanilla PIP is not a rogue elephant that somehow bypasses BDOS (like so many programs that run under that other operating system, which shall remain nameless!) In fact PIP is written in a high-level language and uses orthodox BDOS calls. PIP fails to reset the archive bit because the facility to do so automatically in BDOS *doesn't exist.* The few programs available that do reset the archive bit do it as the result of explicit code placed there by the effort of their authors.

Best,
Jim Rosenberg
President
Amanuensis, Inc.
R.D. #1, Box 236
Grindstore, PA 15442
(412) 785-2806

Acknowledgments

Dear *DDJ*:
I want to set the record straight on three items that have arisen recently concerning authorship and propriety of certain copyrighted material.

Firstly, in the January 1985 issue of *DDJ*, page 67, Walt Piotrowski incorrectly gives me credit for the original MBOOT program published in *DDJ* October 1982 page 47. As the source listing of MBOOT in that article clearly indicates, Keith Petersen, W8SDZ, is the

(Continued on page 12)

C CHEST

Pipes, Wild-Card Expansion, and Quoted Arguments

by Allen Holub

Hi. The C column is changing hands this month. So, before leaping into the thick of things, I want to talk a little about how I'd like the column to evolve. If you'd rather see this column do other things, write to me. I propose using this space for three major purposes:

(1) *Printing useful subroutines written in C.* This month's column is an example of the sort of subroutine I have in mind. In future columns, I'd like to talk about things like binary (AVL) tree management functions, operating system primitives (such as queue management routines), useful text manipulation primitives not found in the standard library (like the expression recognizer in grep discussed in *DDJ* No. 96), graphics routines, and so on. I've written compilers, operating systems, word processors, and the like, and I find myself using several pieces of these larger programs over and over again. It's those sorts of pieces that I'd like to print here. In addition to subroutines, complete small programs will appear here from time to time, such as an MSDOS version of the Unix utility "make," a simple sort utility, and the like.

Anything you're interested in is a potential topic, provided that you write and tell me what you want to know about. By the same token, if you have useful subroutines you'd like to share, please send them to me (on an IBM PC-compatible disk, if possible, or on paper, if not). While we're on the topic of participation, I've never written a bug-free program in my life. Consequently, although a program might run fine on my own system when compiled with my own compiler, I'm sure that unnoticed bugs creep in from time to time. If you find a bug, write and tell me; I'll print the correction as soon as possible. If you have a better way to do something and have a working program that does it (I'm not especially interested in theory), send it in.

(2) *Discussing technical and more theoretical issues.* Discussing some of the underlying theory of the C language and of compilers in general can help you understand a lot of the weirdness of the C language. By knowing about things such as stack frames and assembly language subroutine-calling conventions, you find that a lot of the hard-to-understand parts of the language become much clearer. This kind of knowledge also makes it easier to debug programs, so an occasional dip into theory could be useful.

(3) *Providing a general forum on C-related topics.* Want to start a public discussion on your own pet peeve? Send it in.

Adding Unix Command Line Functions

Unix provides an operating environment particularly helpful to programmers. Pipes, redirection, and wild-card expansion are all features that Unix itself provides, available to any program running under Unix regardless of the language in which that program was written. For example, when you say "grep something *.c" on the command line, Unix expands the *.c to a list of all files in the current directory that end in .c. By the time your main( ) function is executed, Unix will have filled argv with the names of these files rather than with the *.c. Similarly, when you say "foo > bar" or "foo | bar," Unix automatically takes care of sending the output of foo to the file bar (in the first case) or to the program bar (in the second).

Most operating systems aren't as nice. MSDOS does redirection, but, because it doesn't differentiate between stdout and stderr, most compilers can't let MSDOS actually do the redirection. MSDOS pipes are useable, but it doesn't do wild-card expansion. CP/M doesn't even do pipes. To make up for these shortcomings, most manufacturers of C compilers implement some or all of these Unix functions inside the C program itself (rather than in the operating system where they belong). That's one reason why a two-line program can compile into 8K of code.

Unfortunately, most compiler manufacturers don't give you full Unix compatibility. I/O redirection almost always is included, and wild-card expansion almost always is omitted. Quoted arguments (which let you get imbedded spaces into a single argv entry) usually are not implemented either. This month, we will see how to add pipes, wild-card expansion, and quoted arguments to the Aztec CII compiler. It's not hard to modify these routines to work with many other compilers. In a future column, I'll give a version that works with Lattice C.

Root Modules

Before your main( ) subroutine is activated, a certain amount of housekeeping must be done. To accomplish this, most C compilers provide a subroutine that you usually don't see. This subroutine, called a root module, first does various initializations then calls your main( ) function. The root module for CP/M compilers often is inserted into your program automatically by the linker. MSDOS compilers typically require you to provide the name of the root module as the first filename argument submitted to link.

Among other things, the root module initializes the I/O system and parses the command line into the argv array. Redirection, wild-card expansion, and so forth are also done in the root module. So, to add these functions, all you have to do is modify the root. On the down side, anything you add to the root module will increase the size of every program you compile. It's often a good idea to have several

roots around—with varying complexity and size—and to use the one most suitable to a particular application.

Different manufacturers call root modules different things (Croot, root, _main, $main, and so on). Sometimes there are two such functions. For example, the Computer Innovations compiler uses an assembly language routine $main, which calls a C routine _main, which calls main( ). Similarly, in Lattice C the assembly language routine c calls _main( ) (written in C), which in turn calls your main( ). The root module for Aztec C called croot( ) is written in C.

Fortunately, the source for the root module usually is provided with the compiler, so you can modify it as necessary. The version of croot given here differs from the distribution version as follows:

(1) Any word on the command line containing the characters * or ? is assumed to be a filename containing wild-card characters. Croot will search the current user area on the indicated drive for any files matching the indicated name. The * and ? are expanded exactly the way they are in CP/M; that is, a * will match any character repeated zero or more times, and a ? will match any single character falling in the position in a name that corresponds to the position of the question mark. No character except a period may follow a *. The expanded list of files is passed through to main( ). If you have three files on your disk with the extension .c, then the command line "program *.c" results in main( ) being passed an argc of 4; argv[1], argv[2], and argv[3] contain the names of the three files. The files are listed in no particular order.

(2) Any argument surrounded by double quotes ("arg") is treated as a single argument, even if there are embedded blanks in the string. For example, the command line:

```
program "this is one argument"
    "this is another"
```

enters main( ) with argc set to three. Argv[1] points at the string:

```
this is one argument
```

Argv[2] points at the string:

```
this is another
```

The quotes are stripped off for you. The wild-card characters * and ? are ignored if they are found in a quoted argument (i.e., they are passed through to main( ) as a * and a ?). If the quotes aren't present, croot will try to expand these characters into a filename.

(3) I/O redirection has not been modified from the Manx release version. The character > causes standard output to be "redirected" to the file whose name is immediately to the right of the character instead of to the console. The character < causes standard input to be taken from the file whose name is immediately to the left of the character instead of from the console. You have to be careful about using pipes and redirection at the same time, though (see below).

(4) The pipe character (|) causes standard output from the program to the left of the character to be used as standard input to the program to the right of the character. Two temporary files called $PIPE.IN and $PIPE.OUT are created (on the A: disk in the current user area); they're automatically deleted when they are no longer needed. The program at the end of a pipe has to support redirection, but it need not support pipes. You may use more than one pipe on a single command line. For example, the command line

```
grep -n foo | sort -d | lpr
```

executes grep first, invoking it with the arguments -n and foo. The output from grep is used as the input to the program sort, which is passed the argument -d as well. Finally, the output from sort is passed to the program lpr.

Croot does this chaining in two steps. It creates a temporary file in the first program in the chain, pretending that it saw a redirection directive (> $pipe.out) on the command line. Then when executing the function exit( ), croot loads in the next program, using execl( ), and executes it. Croot redirects the temporary file to the second program by adding a < $pipe.in to the command line sent to the second program (the name is changed before the command line is modified).

Given the above example, you could accomplish the same thing by typing:

```
grep -n foo > $pipe.out
ren $pipe.in = $pipe.out
sort -d < $pipe.in > $pipe.out
era $pipe.in
ren $pipe.in = $pipe.out
lpr < $pipe.in
era $pipe.in
```

If the temporary files can't be created, croot does not erase the $pipe.in file. However, it will attempt to abort any ongoing submit file by deleting the file $$$.sub.

Pipes may be used along with redirection. For example, the command line

```
program1 <foo | program2 > bar
```

is perfectly legal. In this case, program1 takes its input from the file foo and sends its output to program2. Program2, in turn, puts its own output into the file bar. You have to be careful, though: you can't direct output to two places at once or get input from two places at once. For example, "program1 >foo | program2" is illegal, as is "program1 | program 2 <foo."

Support Routines

Croot requires several support routines; most of these are documented in Listing One (page 14). Some of these are useful enough to merit additional explanation.

do_wild

```
int   do_wild(fname, argv, maxarg )
  char *fname, **argv ;
  int   maxarg ;
```

A general purpose directory-searching utility, do_wild( ) creates an argv-like array of pointers to strings, one string for each file in the current directory that matches the filespec "fname." It returns the number of valid elements in the argv array after the expansion; the return value can be used to update argc.

If fname contains no wild-card characters, only one name is expanded. If fname does not exist, nothing is done, and the routine returns a zero. If fname contains a * or a ?, argv will contain a pointer to a string for each possible matching name. If the string begins with x: (where x may be replaced by any valid CP/M drive id), the indicated drive is searched instead of the current drive. On entry, the parameter argv should point at the beginning of an uninitialized array of character pointers. Maxarg is the maximum permitted size of argv.

execl

```
execl( name, arg1, . . . argN,
    (char *)0 )
char *name, *arg1 . . . *argN;
```

Execl( ) chains to the program "name" by overwriting the current program and transferring control to the new program; the extension .COM is automatically appended onto "name."

Execl( ) has a variable number of arguments. Arg1 through argN are concatenated and passed through to the new program as its command line. The rightmost argument in the list must be zero. Because of the way the next program is loaded, the length of all the concatenated arg strings may be at most 79 characters. Note that execl( ) inserts a space between each string. Any characters beyond the 79 are truncated.

Execl( ) should not return. If it does return, then execl( ) can't open the specified file. Execl( ) does not print an error message in this latter case. Note that the contents of the default FCB and the default DMA buffer (at addresses 0x5c and 0x80) are undefined if execl( ) returns. Execl( ) makes no attempt to close any files, so be sure to close any opened files before the execl( ) call.

Execl( ) uses two assembly language files to load the next program in the pipe, "loader" and "loadldr"(Listing Two, page 27). Loader reads the next program into memory then transfers control. Loadldr puts the loader itself into memory.

initfcb

```
initfcb(filename, fcb)
char *filename, *fcb;
```

Initfcb( ) initializes the fcb (CP/M file control block) pointed at by "fcb" to the filename "filename." If the filename has x: as the first two characters of the string, the drive field is set to drive x as well; all other fields are filled with zeroes.

A * in the filename is expanded out to fill the rest of the field with ?s. So, the wild-card characters * and ? are treated as they are by CP/M. For example:

```
xxx.yyy     becomes   xxx     yyy
xx*.y*      becomes   xx??????y??
" " or *.*  becomes   ???????????
```

The filename argument may be terminated by either a \0 or a space.

DDJ

(Continued from page 8)

author of MBOOT and MBOOT is only an adaptation of the original MODEM-.ASM by Ward Christensen. This is an easy mistake for Walt to make and I doubt if anyone will be hearing from either Keith's or Ward's legal representatives.

Secondly, my dBASE II© command program DBSECURE.CMD is a RUNTIME © simulator, but requires the target environment to own *already* a licensed copy of dBASE II©. While DBSECURE will scramble and SCRUNCH command files like RUNTIME ©, it relies on DBASE.COM to execute the command file. This requirement is clearly indicated in the documentation.

Finally, NEWBASE.COM and NEWBASE.ASM will no longer support the SKIP-SIGNON-OPTION. The editors at *DDJ* brought it to my attention that such a feature could be used to by-pass the copyright notice of Ashton-Tate and that was not my intention.

As most authors for this magazine, I am available to try to answer questions and help solve problems with my published software. However, on behalf of all of us who put our name, address and phone number on the line, please call at reasonable hours (check time differences) and always include a self-addressed, stamped envelope when writing.

Gene Head
2860 NW Skyline Drive
Corvallis, Oregon 97330
(505)758-0279

DDJ

Listing One

```
/*                              Croot.c
 *
 *                  Wildcard expansion & pipes for Aztec CII
 *
 *       This version of Croot.c has been extensively modified by A. I. Holub.
 *       Manx Software Systems is in no way responsible for this program.
 *
 *       The portions of this program printed in boldface and are the original
 *       croot. They are Copyright (c) 1981, 1982 by Manx Software Systems and
 *       (c) 1982 by Thomas Fenwick.
 *
 *       The portions of this program not printed in boldface are
 *       Copyright (c) 1985, Allen I. Holub. These portions may be reproduced
 *       for personal, non-profit, use only. All other use is prohibited.
 *
 *       You should replace the croot in libc.lib with this module.
 */

#include "a:errno.h"
#include "a:fcntl.h"
#include "a:io.h"

#define MAXARGS         128             /* Maximum argc                     */
#define LOADERSIZE      49              /* Size of the loader in bytes      */
#define DEF_FCB         0x5c            /* CP/M Default FCB location        */
#define FCBSIZE         36              /* Size of an FCB in bytes          */

#define CBUF            ((char *)0x80)          /* CP/M command line buffer */
#define CBUFEND         (char *)(0x80+128-LOADERSIZE)

/*
 *      Various BDOS function calls:
 */

#define srchfirst(fcb)  ( bdos(  0x11, (fcb) ) & 0xff )
#define srchnext(fcb)   ( bdos(  0x12, (fcb) ) & 0xff )
#define bdosopen(fcb)   ( bdos(  0x0f, (fcb) ) & 0xff )

int badfd(), noper();

/*      Channel table: relates fd's to devices
 */

struct channel  chantab[] =
{
        { 2, 0, 1, 0, noper, 2 },
        { 0, 2, 1, 0, noper, 2 },
        { 0, 2, 1, 0, noper, 2 },
        { 0, 0, 0, 0, badfd, 0 },
        { 0, 0, 0, 0, badfd, 0 },
        { 0, 0, 0, 0, badfd, 0 },
        { 0, 0, 0, 0, badfd, 0 },
        { 0, 0, 0, 0, badfd, 0 },
        { 0, 0, 0, 0, badfd, 0 },
        { 0, 0, 0, 0, badfd, 0 },
        { 0, 0, 0, 0, badfd, 0 },
};

static char     *Argv[MAXARGS];
```

(Continued on page 16)

Listing One

```
static char     Argbuf[128];
static int      Argc;

static int      Doingpipe = 0;
static char     *Cmdtail;

/*      File names for pipe processing. The padding lets us patch the name
 *      with DDT rather than re-compiling.
 */

char    *Pipe_in  = "A:$PIPE.IN\0\0\0\0" ;
char    *Pipe_out = "A:$PIPE.OUT\0\0\0"  ;

/*------------------------------------------------------------------------*/

Croot()
{
        /*      Root module for Aztec CII. This is not the best C programming
         *      in the world. It uses gotos; what some of the code is doing
         *      is, by necessity, somewhat obscure. On the other hand it
         *      does a bunch of complex stuff relatively efficently.
         */

        register int k;
        register char *cp, *fname;

        blockmv(Argbuf, CBUF + 1, 127); /* Copy the command line somwhere  */
        Argbuf[*CBUF & 0x7f] = 0;       /*     safe and null terminate it. */

        Argv[0] = "";                   /* Initialize argv and argc        */
        cp = Argbuf;
        Argc = 1;

        while (Argc < MAXARGS)
        {
                while (*cp == ' ')              /* Skip leading blanks    */
                        ++cp;

                if (*cp == 0)                   /* At end of command line */
                        break;
                else if( *cp == '|' )           /* Process a pipe */
                {
                        Doingpipe = 1;
                        Cmdtail = ++cp;

                        /*
                         *    Pretend we saw a ">" to the Pipe_out file.
                         */

                        fname = Pipe_out ;
                        k = 1;
                        goto redir2;

                }
                else if (*cp == '>')            /* redirect output */
                {
                        k = 1;
                        goto redirect;
                }
```

(Continued on page 18)

```
                else if (*cp == '<')            /* redirect input */
                {
                        k = 0;
redirect:
                        while (*++cp == ' ')
                                ;

                        fname = cp;

                        while (*++cp)
                        {
                                if (*cp == ' ')
                                {
                                        *cp++ = 0;
                                        break;
                                }
                        }
redir2:
                        close(k);
                        k = k ? creat(fname, 0666) : open(fname, O_RDONLY) ;
                        if (k == -1)
                        {
                                perr("Can't open file for redirection <");
                                perr(fname);
                                perr(">");
                                exit(10);
                        }

                        if( Doingpipe )
                        {
                                /* The rest of the cmd tail is for the next
                                 * process so stop processing it.
                                 */

                                break;
                        }
                }

                else if (*cp == '"')
                {
                        Argv[Argc++] = ++cp;

                        while ( *cp  && (*cp != '"') )
                        {
                                cp++;
                        }
                        *cp++ = 0;
                }
                else if ( haswild(cp) )
                {
                    Argc += do_wild(cp, Argv + Argc,  MAXARGS-Argc-1);
                    goto skippast;
                }
                else
                {
                        Argv[Argc++] = cp;
skippast:
                        while (*++cp)
                        {
                                if (*cp == ' ')
```

(Continued on page 20)

Listing One

```
                                {
                                        *cp++ = 0;
                                        break;
                                }
                        }
                }
        }
        exit(main(Argc,Argv));
}

/*---------------------------------------------------------------------------*/

execl( name, args )
char    *name, *args;
{
        /*      Execl has a variable number of arguments starting
         *      with args and continuing until a 0 arg is found.
         *      It is called with:
         *
         *          execl( filename, arg1, arg2 , ... , argN, 0 );
         *
         *      All arguments must be character pointers.
         *      Execl overlays the current program with the program
         *      named "name". All the args are concatenated together
         *      with the args separated from one another by spaces.
         *      The concatenated array is used as the command line
         *      for the new process. Execv adds the .com to the end of
         *      name.
         */


        register char   *cp, *dest, **argp;

        strcpy (CBUF,  name   );
        strcat (CBUF,  ".COM" );
        initfcb(CBUF, DEF_FCB );          /* Try to open the file       */

        if( bdosopen(DEF_FCB) == 0xff )
                return(0);                /* Can't do it                */


        /*
         *      Put together a new command line from the arguments
         */


        dest = CBUF + 1;

        for(argp = &args ;  *argp && dest < CBUFEND   ; argp++ )
        {
                for(cp = *argp ;  *cp && dest < CBUFEND ;  *dest++ = *cp++ )
                        ;

                *dest++ = ' ';
        }
        *dest = '\0';

        *CBUF =  (int)dest - (int)CBUF ;

        load1dr();                        /* We won't return from load1dr */
```

```
}

/*-------------------------------------------------------------------------*/

exit(code)
{
        /*      Finish up the execution of a C program. Chain to the
         *      next process if appropriate.
         */

        register char   *tailp, *src, *dest ;

        closall_();                     /*  Close all open files          */
        unlink(Pipe_in);                /*  Get rid of the pipe temp file */

        if ( code )                     /*  Try to abort an ongoing submit */
                unlink("A:$$$.SUB");    /*      file.                      */

        if( Doingpipe )                 /*  Chain to next program in pipe  */
        {
                rename(Pipe_out, Pipe_in);

                /* Get the file name from the command tail. First strip out
                 * leading blanks, then terminate the file name with a null
                 * if necessary (the rest of the command tail will be
                 * arguments to the next file.
                 */

                while(*Cmdtail == ' ')
                        Cmdtail++;

                tailp = Cmdtail;

                while( *tailp && *tailp != ' ' )
                        tailp++;

                if( *tailp )
                        *tailp++ = '\0';

                /* Chain to the next program    */

                execl( Cmdtail, "<", Pipe_in, tailp, (char *)0 );


                /* If execl returns then we couldn't open the file */

                perr("Couldn't open file at end of pipe: ");
                perr(Cmdtail);
                perr(".COM");
                unlink("A:$$$.SUB");    /* Try to kill a submit file    */
                unlink("$$$.SUB");
        }

        boot_();
}

/*-------------------------------------------------------------------------*/

perr(str)
char    *str;
{
         /*      Print a string by brute force. We can use this
          *      whether or not stdout is open.
          */

         while(*str)
                 bdos(2, *str++);
```

```
}

/*-------------------------------------------------------------------------*/

badfd()         /* Called when a bad file descriptor is found */
{
        errno = EBADF;
        return -1;
}

/*-------------------------------------------------------------------------*/

noper()         /* Do nothing, return 0 */
{
        return 0;
}

/*-------------------------------------------------------------------------*/

initfcb(filename, fcb)
char            *filename, *fcb;
{
        /*      Initialize the fcb pointed at by "fcb" with the filename
         *      pointed to by "filename". Filename may include a drive
         *      designator (X:) or either of the wild card characters * or
         *      ?. See the text for more details.
         */

        register char   *fp;
        register int    i;

        for ( i = FCBSIZE , fp = fcb ; --i >= 0 ; *fp++ = 0 )
                ;

        if( *filename && filename[1] == ':')
        {
                /*      Fill in the drive field of the fcb
                 */

                *fcb = (char)( toupper(*filename) - 'A' + 1);
                filename += 2;
        }

        expand_name(fcb, filename);
}
/*-------------------------------------------------------------------------*/

expand_name(dest, src)
char    *src, *dest;
{
        /* Turn the filename at src into the name and type fields of an fcb
         * pointed to by dest (dest is the address of an fcb, src is an
         * ascii string).
         *
         *      "xxx.yyy"       becomes         "xxx     yyy"
         *      "xx*.x*"        becomes         "xx??????y??"
         *      "" or "*.*"     becomes         "???????????"
         *
         * The src parameter may be terminated by either a '\0' or a
         * space (' ').
```

```
         */

        register char   *dot, *end, i;

        dot = ++dest;                                   /* Skip past drive field */

        for ( i=11 ; --i>=0 ; )                         /* Dest = ????????.???   */
                *dot++ = (*src) ? ' ' : '?' ;

        dot = &dest[8];
        end = &dest[10];

        while( *src  &&  *src != ' ' )
        {
                if (*src == '*')
                {
                        if ( dest < dot )
                                while (dest < dot)
                                        *dest++ = '?';
                        else
                                while (dest <= end)
                                        *dest++ = '?';
                }
                else if (*src == '.')
                        dest = dot;
                else
                        *dest++ = *src;

                src++;
        }
}

/*-------------------------------------------------------------------------*/


#define DEF_FCB         0x5c            /* CP/M Default FCB location     */
#define srchfirst(fcb)  ( bdos(  0x11, (fcb) ) & 0xff )
#define srchnext(fcb)   ( bdos(  0x12, (fcb) ) & 0xff )


int     do_wild(fname, argv, maxarg )
char    *fname, **argv ;
int     maxarg ;
{
        register char           *dest, *src;
        register unsigned int   tmp;
        register int            i, argc = 0;
        extern   char           *alloc();

        /*      Expand fname (which may contain wild card characters)
         *      into a series of entries in argv. Return an updated argc.
         *      Maxarg is the maximum permitted value of argc.
         *      Fname may be terminated by either a '\0' or a ' '.
         */

        initfcb(fname, DEF_FCB);

        if ( (tmp = srchfirst(DEF_FCB)) != 0xff )
        {
                do
                {
                        if( (dest = fname = alloc(13)) == 0 )
                        {
```

(Continued on next page)

Listing One

```
                        perr("Not enough memory to expand wildcard");
                        exit(1);
                }

                src  = ((char *)((tmp<<5) + 0x80)) + 1;

                if ( *(fname + 1) == ':')
                {
                        *dest++ = *fname;
                        *dest++ = ':';
                }

                for( i = 1 ; i <= 11 ; i++ )
                {
                        /*      Transfer name from DMA buffer to
                         *      argument.
                         */

                        if( (*src & 0x7f) != ' ')
                                *dest++ = *src++;
                        else
                                src++;
                        if( i == 8 )
                        {
                                if ( (*src & 0x7f)  == ' ')
                                        break;
                                else
                                        *dest++ = '.' ;
                        }
                }

                *dest = '\0';
                argv[argc++] = fname;

                tmp = srchnext(DEF_FCB) ;

        }
        while( (tmp != 0xff) && (argc < maxarg) );
    }

    return( argc );
}

/*-------------------------------------------------------------------------*/
haswild(str)
char    *str;
{
        /*      Return true if str has a '*' or a '?' in it.
         */

        register int    c;

        do {
                c = *str++;
                if(c == '*'  ||  c == '?')
                        return(1);

        } while(c &&  c != ' ');

        return(0);
}
```

End Listing One

Listing Two

```
;       LOADER.ASM      Assembly language support routines for pipes
;
;          This loader is essentially that used by BDS C (written by Leor
;       Zolman). The loader is copied to just below the tpa
;       (into the default disk buffer) by loadldr. This location limits the
;       size of the command tail for the next program in the pipe to
;       128 - 49 = 79 characters (128 = buffer size, 49 = loader size).
;       We're putting the loader where we do because an 8080 doesn't have
;       a relative jump instruction. Since we are actually copying the
;       loader to the place where it will execute, any labels defined
;       at link time will be wrong (because they'll be pointing into
;       the original code, not the copy). So, we have to copy it to
;       some known place to know where to jump to when we loop. The bottom of
;       the bdos, however, is not a known place. It will vary from
;       system to system.
;
;       Before this loader is called, the calling program should put
;       the new command line into the buffer at 0x80.
;       It should also have an opened the file to be
;       loaded with the fcb for that file in the default FCB space
;       (at 0x5c).


closec  equ     16      ; CP/M function: Close file.
reads   equ     20      ; CP/M function: Read sequential
sdma    equ     26      ; CP/M function: Set DMA address

fcb     equ     5CH     ; Default FCB location
tpa     equ     100H    ; Start of transient program area
tbuff   equ     80H     ; start of command tail buffer
bdos    equ     05H     ; bdos jump vector
bdosv   equ     06H     ; pointer to bdos start address

;-----------------------------------------------------------------------
;                       LOADER                  (local)
; On entry:
;
;       D/E = the DMA address   ( 0x100 initially      )
;       B/C = points at an fcb for the opened file. This fcb is physically
;               located just beneath the loader in memory.
;       SP  = just below the bdos.
;
; On exit:
;
;       All registers are destroyed
;
loader: lxi     d,tpa   ;destination address of new program
loadl:  push    d       ;push   dma addr
        push    b       ;push   fcb pointer
        mvi     c,sdma  ;set DMA address for new sector
        call    bdos
        pop     d       ;get pointer to working fcb in DE
        push    d       ;and re-push    it
        mvi     c,reads ;read a sector
        call    bdos
        pop     b       ;restore fcb pointer into BC
        pop     d       ;and dma address into DE
        ora     a       ;end of file?
        jz      tpa-8   ;if not, get next sector (goto `load2:')
```

(Continued on next page)

Listing Two

```
        mov     d,b       ;move the fcb pointer to d/e
        mov     e,c
        mvi     c,closec ; close the file
        call    bdos

        mvi     c,sdma   ; reset DMA pointer, tbuf is at 0x80
        lxi     d,tbuff
        call    bdos
        jmp     tpa      ; and go invoke the program

load2:  lxi     h,80h    ;bump dma address
        dad d
        xchg
        jmp     tpa-46   ;and go loop (at load1)

;-------------------------------------------------------------------------
;                       LOADLDR_                    (public)
;
;       This routine loads the loader into memory, modifys the stack pointer
;       and registers as required by the loader, and passes control to the
;       loader. Since load1dr jumps directly to the loader, it will not
;       return to the calling routine.
;
; On exit;
;       B  = default FCB address   (0x5c)
;       D  = tpa start address     (0x100)
;       SP = first address of BDOS

        public  load1dr_

load1dr_:
        lxi     d,loader         ; d = src;
        lxi     h,tpa-49         ; h = dest;
        mvi     b,49             ; b = 49;       /* size of loader */

L1:     ldax    d                ; do{
        mov     m,a              ;       *d++ = *h++;
        inx     d                ;
        inx     h                ; }
        dcr     b                ; while( --b > 0 );
        jnz     L1

        lxi     b,fcb            ; b = address of fcb;
        lxi     d,tpa            ; d = dma address;

        lhld    bdosv            ; h = address of BDOS
        sphl                     ; sp = address of BDOS

        jmp     tpa-49           ; jump to the loader

        end
```

End Listings

REALIZABLE FANTASIES

The GNU Manifesto

by Richard Stallman

This month's installment of Realizable Fantasies features a guest appearance by Richard Stallman. Stallman, inventor of the original much-imitated Emacs editor and formerly of the Artificial Intelligence Lab at MIT, has worked extensively on compilers, editors, debuggers, command interpreters, the Incompatible Timesharing System (ITS) and the Lisp Machine operating system. He pioneered terminal-independent display support in ITS. Since then he has implemented one crashproof file system and two window systems for Lisp machines, and designed a third window system now being implemented; this one will be ported to many systems including GNU.—M.S.

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system that I am writing so that I can give it away free to everyone who can use it. Many other programmers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have a portable C and Pascal compiler which compiles for Vax and 68000, an Emacs-like text editor with Lisp for writing editor commands, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. When the kernel and a debugger are written, by the end of 1985 I hope, it will be possible to distribute a GNU system suitable for program development. After this we will add a text formatter, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical with Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer filenames, file version numbers, a crashproof file system, filename completion, perhaps, terminal-independent display support, and eventually, a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen.

Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at machines in the 68000/16000 class, with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on less powerful machines will be left to someone who wants to use it on them.

Why I Must Write GNU

If I like a program, I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I cannot in good conscience sign a nondisclosure or software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies. My efforts were wasted. I cannot remain in an institution where such things are done for me against my will.

So that I can continue to use computers without violating my principles I have decided to put together a body of free software sufficient to enable me to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse for preventing me from giving GNU away.

Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. Furthermore a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help. Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires that they feel like competitors with other programmers rather than like comrades. The fundamental act of friendship among programmers is the sharing of progams; marketing arrangements now in use essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather then proprietary programs, we can be hospitable to everyone *and* obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony, which is impossible if we use software that is

not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One computer manufacturer has already offered to provide a machine. We can use more. One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machine should be able to operate in a residential area, and not require sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work to GNU. For most projects, such part-time distributed work would be very hard to coordinate; the parts, written independently, would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even if Murphy creates a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part-time. The salary won't be high by programmer's standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air. This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system progamming will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one progammer or company that owns the sources and is in sole position to make changes.

Schools will be able to provide a superior educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain progams. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted. Arrangements to make people pay for using a program, including licensing of copies, always impose a tremendous cost on society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. Furthermore, only a police state can force everyone to obey. Consider the analogy of a space station where air must be manufactured at great cost: charging each breather per liter of air might be fair, but wearing the metered oxygen mask all day and all night would be intolerable even if everyone could afford to pay the bill. And the TV cameras everywhere to see if you ever took it off would be outrageous. It would be better to support the air plant with a head tax and chuck the masks. Copying all or parts of a program is as natural to a progammer as breathing, and as productive. It ought to be as free.

Some easily rebutted objections to GNU's goals

"Nobody will use it if it is free, because that means they can't rely on any support. You have to charge for the program to pay for providing the support." If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.

We must distinguish between support in the form of real programming work and mere handholding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost. If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem and you will not be at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible that there will be no available competent person, but this problem cannot be blamed on distribution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users *who know* nothing about computers need handholding, i.e., they need for *others* to do for them the things which they could easily do *themselves*, but don't know how to. Such services could be provided by companies that sell just handholding and repair service. If it is true that users would rather spend money and get a product with services, they will also be willing to buy the service, having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

"You cannot reach many people without advertising, and you must charge for the program to support that. It's no use advertising a program people can get free." There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it. On the other hand, if many people get GNU from their

friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?

"My company needs a proprietary operating system to get a competitive edge." GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefiting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems. I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.

"Don't programmers deserve a reward for their creativity?" If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only insofar as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

"Shouldn't a programmer be able to ask for a reward for his creativity?" There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customarily used in the area of software development today are based on destruction. Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount that and the ways in which the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction. The reason a good citizen does not use such destructive means to become wealthier is because, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics, or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one person to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?" I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else. But that is the wrong answer, because it accepts the questioner's implicit assumption that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing. The real reason programmers will not starve is because it will still be possible for them to get paid for programming; just not as much as now.

Restricting copying is not the only means for making a profit in software development. It is the most common means because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other methods of profit-making that are now used less often. Probably programming would not be as lucrative as it now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?" Control over the use of one's ideas really constitutes control over other people's lives; and it is usually used to make their lives more difficult. People who have carefully studied the issue of intellectual property rights (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes. For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for

whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied lengthy extracts from other authors in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented—books, which could be copied economically only on a printing press—it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: Are we really better off granting such license? What kind of act are we licensing a person to do? The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code, which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to or not.

"Won't everyone stop programming without a monetary incentive? Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way. But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is: Will anyone program with a reduced monetary incentive? My experience shows that they will. For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself. Then most of them left when offered a chance to do the same interesting work for a lot of money. What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are gone.

"We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey." You're never so desperate that you have to obey this sort of demand. Remember, millions for defense, but not a cent for tribute.

"Programmers need to make a living somehow." There are plenty of ways by which programmers can make a living without selling the right to use a program. Here are a number of examples:

- A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.
- The sale of teaching, handholding and maintenance services could also employ programmers.
- People with new ideas could distribute programs as freeware, asking for donations from satisfied users. I am told that several people are already working this way successfully.
- Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a software tax:

- Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.
- But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing—often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.
- The total tax rate could be decided by vote of the payers of the tax, weighted according to how much tax they paid in the previous year.

The consequences:

- The computer-using community supports software development.
- This community decides what level of support is needed.
- Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this in order for technical gains in productivity to translate into less work for us.

Richard Stallman, 166 Prospect Street, Cambridge, MA 02139. *Copyright © 1985 Richard Stallman. Permission is granted to make and distribute copies of this article as long as the copyright and this notice appear, and the copies are distributed at no charge.*

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 191.**

Programming in Logic

by John Malpas

Prolog's admirers claim that to use the language is to program in logic. Is the claim true, and is programming in logic something we should want to do?

Logic programming has been relatively little known in the United States until recently. A number of factors are converging, however, that augur well for an increase in its use and popularity. First of all, several interpreters offering respectable performance and support are now commercially available, and apparently other, high-performance interpreters will soon be released. Second, the lukewarm attitude of hardware manufacturers towards logic programming seems to be changing: the new Tektronix machine was released with Prolog already running on it, and Apple plans to offer a version of Prolog for the Macintosh. Third, there is an increasing demand from computer people on all levels for artificial intelligence technology, and one of the most promising areas of AI research has been the area of programming in logic.

Logic programming (see Table 1, page 37) is based on the idea that statements in first-order predicate logic, cast in Horn clause form, can be used directly as a programming language. In Horn clause form, one conclusion is followed by zero or more conditions, as follows:

```
conclusion :-
    condition1,
    condition2,
    . . . ,
    conditionN.
```

John Malpas, Pulsetrain, 747 Greenwich St., New York, NY 10014. *Copyright 1984, Pulsetrain.*

":-" is read as *if*, and the commas are read as *and*, so the whole clause can be read: *The conclusion is true if condition1 and condition2 and . . . conditionN are all true.*

Prolog is the most widespread logic programming language. A Prolog interpreter is based on a general purpose inference mechanism (typically using the resolution and unification techniques); to take advantage of this mechanism, the programmer need only declare the relationships between data items in Horn clause form. A logic program does not need overt control constructs (such as *for* or *while*) because the interpreter makes most of the control decisions.

One consequence of this approach is portability. A logic program could be made to run on all of the common versions of Prolog with relative ease. Because logic programs depend on logic, not on the implementation details of any particular interpreter, they are guaranteed to be upwardly compatible with faster interpreters or more efficient hardware. [*But see D. E. Cortesi's discussion of parallelism (page 50) for an important caveat.—Ed.*] Because the interpreter makes most of the fundamental control decisions, and because of the portability over hardware, any developments in the implementation of Prolog interpreters and in the hardware substratum are particularly interesting. There have been several such developments recently.

The first Prolog was implemented by Colmerauer in Fortran and was very slow and awkward to use (for the

development of Prolog see Table 2, below right). In 1977 Warren and Pereira wrote an interpreter/compiler in DEC-10 assembly language that showed that Prolog could be just as efficient as Lisp. A number of versions have been written since then in C for Unix machines. Clark and McCabe's original Micro-PROLOG was in assembly language on PCs, but the latest version (called Sigma-PROLOG) is written in C and runs on larger machines. There are several versions of Prolog written in Lisp for Lisp machines.

In 1983, ICOT (where the Japanese fifth generation project is being developed) announced their Personal Sequential Inference Machine, which has a processor instruction set optimized to run Prolog—a Prolog machine. When I saw this computer, it had 80MB of main memory. Warren has also followed the approach of optimizing processor microcode to run Prolog. His company (Quintus Computer) has a fast Prolog running on a VAX and a Sun workstation. Other fast Prolog interpreters are under development at UC Berkeley and Argonne National Laboratories. Prolog for machines based on parallel architecture is being developed by Shapiro in Israel, at Imperial College in London, and at ICOT.

In the area of microcomputers, there are currently three commercial versions of Prolog for the IBM PC, and Apple will announce a version of Prolog for the Macintosh this spring. There are other implementations as well, some written in Lisp and of value for learning the language. The three versions for the PC are true commercial products (for sources of Prolog see Table 3, page 38).

Features of Prolog

The fact that Prolog is an implementation of logic as a programming language makes the language distinctive in several ways.

Prolog Can Express Knowledge

Logic is a language capable of expressing situations and problems that provides a way to describe accurately decision-making behavior. Logic has precise *semantics*; that is, it is possible to understand unambiguously the meaning of a logical expression. Logical expressions can be represented by

R.A. Kowalski, *Logic for Problem Solving*, North Holland, 1979 (Much of the logic programming research done in the last five years has been inspired by this book.)

K.L. Clark and F.G. McCabe, *Micro-PROLOG: Programming in Logic*, Prentice Hall, 1984 (This is the best introduction to programming in Prolog; all examples are in Micro-PROLOG syntax.)

W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981 (This is the standard introductory text for Prolog in DEC-10 syntax.)

L. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning, Introduction and Applications*, Prentice Hall, 1984 (This contains a good introduction to symbolic logic.)

P.J. Hayes, "In Defense of Logic," *Proceedings of 5th IJCAI*, 1977

Table 1
Further Reading

1879: Gotlob Frege invents predicate calculus.

1965: J.A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *JACM* [12(1) 1965] pp. 23–41.

1968: D.W. Loveland, "Mechanical Theorem Proving by Model Elimination," *JACM* [15(2) 1968].

1971: Robert Kowalski and Donald Kuehner, "Linear Selection with Resolution Function," *Artificial Intelligence* (2 1971) pp. 227–260.

1973: Alain Colmerauer at Marseilles writes first Prolog in Fortran (see Philipe Russel, *Technical Report of the University of Marseilles*, Sept. 1975).

1974: Robert Kowalski, "Predicate Logic as a Programming Language," *Proceedings of IFIP*, 1974.

1977: David Warren and Fernando Pereira's DEC-10 Prolog interpreter/compiler, University of Edinburgh (see Warren and Pereira, "Prolog: The Language and Its Implementation Compared to Lisp," *ACM SIGPLAN Notices* [12(8) 1977]).

1980: C-Prolog for Unix systems; Micro-PROLOG for PCs from Imperial College (see K.L. Clark and F.G. McCabe, *Micro-PROLOG: Programming in Logic* Prentice Hall, 1984).

1981: Fifth generation project announced by ICOT in Japan, based on logic programming technology (see *Fifth Generation Computer Systems*, ed. T. Moto-Oka, North Holland, 1982).

1982: LM-Prolog for Lisp machines (see K.M. Kahn and M. Carlsson, "LM-Prolog User Manual, Release 1.0," *Technical Report of Uppsala University*, July 1983).

1983: E.Y. Shapiro designs Concurrent Prolog at ICOT (see Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," *Technical Report #3 ICOT*, Oct. 1983).

Table 2
Development of Logic Programming

different logic languages such as Horn clauses. An expression in one logic language is translatable into an expression in any other logic language.

Kowalski has stated that "algorithm = logic + control," which can be understood as follows: an algorithm is composed of the set of data relationships defining an application domain (that is, its logic) and the control information about how to use those data relationships. In logic programming, because the interpreter takes care of the control part of this equation, a programmer's primary responsibility is to learn the internal logic of an application domain. Writing an application program in Prolog amounts to expressing this internal logic as Horn clauses.

Prolog Describes, It Does Not Simulate

When a procedural language program is used to model a process in the world, the relationship between the program and the world is one of simulation. This means that the program is working on the same level as the process itself, and so must have large amounts of data *from* the process.

On the other hand, when a logic program is used to model a process in the world, the relationship between the program and the world being modeled is one of *description.* The program exists on a higher level than the process it describes, because it contains statements and generalizations (knowledge) *about* the process.

A program is a simulation of the world whenever the state of the computer itself (i.e., CPU registers, memory, etc.) is an integral part of the model. In the case of a descriptive program, however, the state of the computer functions only to implement the logic of the program and does not play a role in the model *per se.*

Prolog Does Not Use Destructive Assignment

Logic programming achieves its descriptive power by dispensing with the destructive assignment statement. Once a logical variable assumes a value, the programmer cannot arbitrarily decide to give it another value. In contrast, all procedural languages depend upon destructive assignment as their most essential semantic feature.

Destructive assignment is best understood as an *optimization of memory resources.* In a language based on destructive assignment, any change in a data value in the program is irrevocable; there will be no trace of previous values of the variable unless the programmer has made provisions to save them.

Prolog Statements Have Both Declarative and Procedural Interpretations

There are two ways to read a Horn clause logic program: declaratively and procedurally. Consider the example rule:

```
pays_too_much_rent (X) :-
    lives_in (X, new_york).
```

There is a procedural way to read this rule:

> To find someone who pays too much rent,
>
> find a resident of New York.

or

> To prove that John pays too much rent,
>
> verify that he lives in New York.

But it is also possible to read the rule declaratively, which enables Horn clauses to be used as a knowledge representation language:

> Someone who pays too much rent is someone who lives in New York.

or

> All people who live in New York pay too much rent.

In the declarative reading, the rule becomes a definition of a relationship

Programming Logic Systems, Inc.
31 Crescent Drive
Milford, CT 06460
- Micro-PROLOG (IBM PC, other PCs, CP/M)
- Sigma-PROLOG (written in C for Unix machines)
- APES (A Prolog Expert System Shell)

Logicware
1000 Finch Avenue West
Toronto, Ontario
Canada M3J 2V5
- M-Prolog (assembly language on IBM mainframe, cross compiled to PC, VAX, *et al.*)

Expert Systems International
1150 First Avenue
King of Prussia, PA 19406
- PROLOG-1 (assembly language on MSDOS, CP/M, VAX, Macintosh)
- ES/P Advisor (Expert System Shell)

SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
- C-Prolog (written in C for Unix machines with at least 256K addressable memory)

University of New Hampshire
- UNH Prolog (written in C for Berkeley Unix)

University of Edinburgh
- DEC-10 Prolog (assembly language)

Waterloo University, Canada
- Waterloo Prolog (assembly language on IBM mainframes)

Public Domain
- UNSW Prolog (written in C for Unix machines)

Table 3
Sources of Prolog

To give some idea of what logic programming code is like, it will be useful to examine a simple program. The simplest Horn clause is an assertion of fact, consisting of a conclusion followed by no conditions:

```
lives_in(john, new_york).
```

which can be read: *John lives in New York.* A rule is another form of Horn clause that shows the dependence of one fact on other facts:

```
pays_too_much_rent(john) :-
    lives_in(john, new_york).
```

which can be read: *If John lives in New York, he pays too much rent.* The word "john" is a constant, and these rules and facts contain explicit knowledge applicable to "john."

A logical variable in Prolog is written as a word beginning with a capital letter. The "too much rent" rule above can be generalized so that it applies to all New York residents by replacing "john" with a variable:

```
pays_too_much_rent(X) :-
    lives_in(X, new_york).
```

which may be read: *Anyone who lives in New York pays too much rent.* The variable "X" will match any constant, so this rule can be used to assert that the rent of any New York resident is excessive.

A rule utilizing variables in this way contains implicit knowledge that is applicable to all people who live in New York.

Prolog is built on the model of symbolic logic created by Frege. There are two varieties of logic to consider: propositional logic and first-order predicate logic.

Propositional logic considers only relations between (unanalyzed) propositions, and only such relations as bear on the propositions' truth or falsity. Propositions can be combined with other propositions to form new propositions via the logical connectives **and**, **or**, **not**, and **implies**. Propositional logic concerns itself with what propositions can be derived from what other propositions. For example, from proposition **P or Q**, where **P** and **Q** are propositions, we can derive **Q**. Also, from **P implies Q** and **P** we can derive **Q**.

First-order predicate logic involves more complex expressions (formulae) that contain variables and that become propositions only when values from some domain of discourse are substituted for these variables. Thus, **P(x)** becomes a proposition and can be true or false only when the variable **x** is assigned a specific value or is otherwise quantified. The ways of quantifying **x** that are particularly important for first-order predicate logic are via the universal and existential quantifiers, which turn the variable **x** into, respectively, **for every x** and **there exists an x**. First-order predicate logic is sufficiently powerful to express a good deal of mathematics and to serve as the model for a programming language. Prolog depends on the resolution technique in its implementation of first-order predicate logic.

Resolution is a decision algorithm for satisfiability in first-order predicate logic. Because the truth of a formula in first-order predicate logic depends on how values are assigned to its variables, one generally talks about the satisfiability of a formula, rather than about its truth. A formula is satisfiable if and only if it is true under some admissible assignment of values to its variables. Thus, if **P(a)** means "**a** is a prime number" and **Q(a)** means "**a** is greater than 11 and less than 17," the formula **there exists an x such that (P(x) and Q(x))** is satisfiable, namely, by the value assignment **x** = 13.

Resolution is a partial decision algorithm; it will invariably determine the unsatisfiability of an unsatisfiable formula, though it is not guaranteed to halt if the formula is, in fact, satisfiable. (This halting problem is a consequence of the fundamental unsolvability of first-order predicate logic, not a deficiency of the algorithm.) By recasting the formula, though, you can demonstrate its satisfiability (if it *is* satisfiable), throwing the halting problem into the unsatisfiability-proving half of the task.

The resolution method requires that the formula be in a particular form, variably referred to as clause form, Horn clause form, or conjunctive normal form. (These terms are not really interchangeable, but for this sketchy discussion we can regard them as so.) The basic idea is this: we eliminate all existential quantifiers by a trick that involves treating the variables they quantify as functions of a particular sort. Once the existential quantifiers are eliminated, we simply drop universal quantifiers, assuming that all remaining variables are universally quantified. And we transform the formula into a logically equivalent formula whose top-level components are all connected by **and**s, whose second-level components are all connected by **or**s, and whose third-level components are all elementary formulae or their negations. For example, one simple formula in this form is **(A or B) and (not-C or D)**.

Resolution consists of a single rule, repeatedly applied. The rule simply combines the top-level components of the formula pairwise, selecting components that have an element in common, with the proviso that in one case the element is negated. The rule throws out these conflicting elements and sticks the result together into one component. Thus, **(A or B) and (not-A or C)** resolves to **(B or C)** by throwing out the **A** and the **not-A**. If, after repeatedly banging components against each other this way, we come up empty-handed (components annihilate each other utterly), then the formula is unsatisfiable.

Unfortunately, not all the components are so simple in form; does **P(a)** cancel out **not-P(x)**? Answering that question in the particular case involves a process not unlike finding the lowest common denominator for two fractions and is accomplished by an algorithm called unification. Unification finds a substitution for **x** and **a** that is just sufficiently general to stand for both.

Resolution as originally defined is not an efficient algorithm. In 1965, J.A. Robinson defined an improved resolution algorithm that got around some of the problems of combinatorial explosion that plagued resolution, but this still left unsolved the problem of how to decide which pairs of components to resolve. Current work in AI is emphasizing heuristic strategies that take advantage of knowledge about the domain of discourse in guiding resolution.

(called "pays_too_much_rent") that holds between various data items. The translation of a rule into specific machine actions is the job of the interpreter, not of the programmer.

Prolog Can Be Used to Implement a Meta-language

Prolog is an inherently extensible language because it is possible to modify the behavior of the interpreter within the context of the language itself. Kowalski refers to this phenomenon as the meta-language/object language relationship.

A meta-language controls the interpretation of an object language. A meta-language can be implemented in the object language (Prolog itself) as a program running on top of the interpreter. The meta-language program receives queries and assertions from the user, transforms them in consultation with the knowledge base, then constructs queries in the object language and executes them.

When first confronting Prolog code, many people mistakenly assume that an end user will have to learn how to write Prolog queries in order to use a Prolog program. One way that a meta-language can be used is to give Prolog an alternative syntax; as a result, the user interface to a program is completely specifiable. Another application of a meta-language involves saving the inference path used by Prolog to answer a query, so that the user can see how an answer was generated. This ability to explain how an answer was arrived at is an essential component of expert systems. Note that this is something more than mere extensibility, which allows the programmer to add to the repertoire of the language. Here we are considering the ability of a piece of software to reason about its own reasoning processes.

Applications of Logic Programming

Prolog has been used for artificial intelligence applications as well as more conventional applications, some of which are ennumerated below.

Relational Data Bases

Because the relational data base model is a logical, well-developed formalism, it can be represented in Horn clause form in a straightforward manner. Prolog has also proved particularly useful for data base front ends. Many types of query languages have been implemented in Prolog, including QBE, SQL, relational algebra, and one similar to VisiCalc.

Software Engineering

Kowalski has shown that a program specification can be directly transformed into a logic program, provided that it is written in a logical specification language (such as DeMarco's data flow diagrams). The initial translation produces a top down design. Each piece of the design can be successively refined independently of the other pieces.

Natural Language

Colmerauer originally invented Prolog to perform natural language processing. Warren and Pereira have continued work in this field with the Definite Clause Grammar formalism (a form of top down parser for natural language). More recently, researchers at ICOT have implemented BUP (a bottom up parser for natural language).

Knowledge Representation

Knowledge representation paradigms used in artificial intelligence, such as semantic networks, frames, production rules, and objects, can all be expressed in logic and implemented in logic programming. According to Hayes, semantic networks and frames are essentially alternative syntaxes for logic.

Expert Systems

Once knowledge has been represented, an expert system can put it to use for expert consultation. Expert systems have been built with Prolog in a variety of areas, including equation solving, medicine, law, architecture, factory automation, etc.

Future applications are left as an exercise for the reader.

DDJ

A Tour of Prolog

by D. E. Cortesi

The distinction between code and data in programming is as fundamental as that between theory and observation in science—and just as false.

Prolog is a member of the general class of nonprocedural programming languages or, as I prefer to call them, the *descriptive* languages. There have been descriptive languages before—notably the simulation languages like GPSS—but most are designed to serve specific application domains, while Prolog is intended for general programming. Descriptive languages require an approach to problem solving that is radically different from the procedural approach used with an ordinary language like Pascal or BASIC.

Not many readers know Prolog or have Prolog interpreters to experiment with. When planning this issue of *DDJ*, the editors realized that it would contain several large programs written in Prolog. Because Prolog is so different, they wondered whether the readers would be able to follow the program listings.

I had been exploring Prolog in my spare time, so I volunteered to describe it from a programmer's perspective. This article is not meant to be a complete tutorial (see Clocksin and Mellish[1] for that). It is rather an orientation tour to give you enough exposure to Prolog's style so that you can make sense of the listings in the other articles.

A Caution

Be aware that the various implementors of Prolog don't agree on its syntax and writers on Prolog don't agree on the names of its parts of speech. In this overview, I've cast the examples in a presentation syntax that uses the tricks of professional typesetting for clarity. A sidebar that accompanies the article ("Prolog Syntax," page 46) explores simple mappings to the syntax schemes of common implementations. I've noted where my nomenclature varies from other writers' usage.

Prescribing and Describing

We use the familiar programming languages to *prescribe* the solutions to a set of problems. We build the prescription from two radically different materials: active code and passive data. The passage of time is an implicit part of a prescription. "To solve one of these problems," we tell the computer, "do this, then do that"; *that* can be correctly understood only in the context created by *this*.

All that changes in Prolog. In Prolog, we *describe* the solutions to a set of problems. "A solution to one of these problems," we tell the computer, "looks like this or this or this"; each *this* must be (in principle) a true description of a solution, independent of its place in the program. There is no distinction between code and data. We simply describe what is true about the universe of discourse, using identical mechanisms to describe true facts and true interpretations of the facts.

Assertions and Queries

Let's tie these abstract notions down with an informal example. The problem is one that arises in a card-playing program: defining what a card is. The honor cards are defined as follows:

David E. Cortesi, 430 Sherman Ave. #212, Palo Alto, CA 94306.

```
card(a).
card(k).
card(q).
card(j).
```

These are sentences of fact that we may *assert* (add to the program) at once. We'll deal with matters of syntax later; for the moment you should find it believable that, given those sentences, a Prolog system could conduct a dialogue along these lines:

```
card(k)?
yes.
card(w)?
not so far as I know.
```

This is a conversation with a hypothetical Prolog system. I don't know of any real systems whose dialogue is so natural nor whose negative answer is so humble. Nevertheless, we have here the essence of Prolog operation: a number of sentences are asserted (usually they are loaded from a file); the system is given a pattern to test against those sentences; and it replies "yes" or "no" depending on whether or not it can match the pattern.

It probably occurs to you that it wouldn't take a particularly clever system to hold the example dialogue. A simple comparison loop would do it. Of course, smarter dialogues are possible. For example, all systems allow the query pattern to contain variables as a way of requesting more information. (In this article, I shall indicate variables with italicized words; see the sidebar "Prolog Syntax" for the variety of ways to recognize variables in real systems.) Thus we may ask:

```
card(what )?
yes, what = a.
yes, what = k.
yes, what = q.
yes, what = j.
```

The system finds not just one but four matches to the query card(*what*)? and reports the resulting value of the variable *what* in each case.

Compound Sentences

Let's continue the example by describing the other nine playing cards. Simply to assert nine more facts, card(10) through card(2), would be in the spirit of Prolog, but Prolog would be a poor, flat notation if it didn't let us construct sentences in terms of other sentences. We can describe the remaining cards by asserting a single sentence:

```
card(x ) if
    integer(x ) and
    less(1 x ) and
    less(x 11).
```

Although awkwardly phrased, this sentence translates as the rule: "Something x is a card if it is an integer between 2 and 10 inclusive." Now we should be able to conduct the following dialogue:

```
card(6)?
yes.
```

This reveals a greater cleverness in the underlying system. To answer the question, the system had to identify 6 with the x in our latest sentence, then test the conjunction of three subordinate assertions, verifying the truth of integer(6), less(1 6), and less(6 11). Shortly, we will examine how this is done, but first let's round out the idea of descriptive programming.

Descriptive Programming

The five sentences asserted so far constitute a purely descriptive solution to the problem of defining a card. We say that they define the card *relation.* The sentences, which are independent of each other, together enumerate all possible cases of the card relation. We don't care in which order a Prolog system checks our question card(6)? It may test the question against the five sentences in any order; it will turn up the right answer regardless. This has important implications for performance (see the sidebar "Prolog in Parallel" on page 50).

Such pure, timeless description is an ideal rarely achieved in practice; working Prolog programs usually have some dependencies on their sequence (we'll see why later). Nevertheless, pure description should be the goal of a Prolog programmer, just as clean structure and avoidance of side effects are goals when using a prescriptive language.

Descriptive Arithmetic

The implementation of arithmetic in the Micro-PROLOG product[2] is an example of just how far you can go in the direction of pure description. In that system, all arithmetic is implemented through two relations built into the system. The SUM relation is defined as:

```
SUM(a b s )
```

for all sets of three numbers such that $a + b = s$. In our example, a set of sentences defined the card relation. The SUM relation is implemented so that you could almost believe it was defined by a huge data base of sentences such as:

```
SUM(0 0 0).
SUM(0 1 1).
   ...
SUM(15 9 24).
   ...
SUM(6 32761 32767).
```

and so on for all negative and fractional numbers as well. This permits you to use SUM for either addition

```
SUM(6 8 x )?
yes, x = 14.
```

or subtraction

```
SUM(x 8 14)?
yes, x = 6.
```

with equal facility.

Micro-PROLOG's built-in relation PROD is defined as follows:

```
PROD(a b c d)
```

over all numbers such that $(a * b) = (c + d)$. PROD also is implemented as if the system contained an infinite data base of sentences about numbers: PROD(16 3 47 1) and so forth. The PROD relation answers queries needing multiplication, division, or modulus:

```
PROD(16 4 prod 0)?
yes, prod = 64.
PROD(16 quotient 64 0)?
yes, quotient = 4.
PROD(16 q 65 r)?
yes, q = 4, r = 1.
```

[Extra credit puzzle: If PROD were limited to the domain of integers between 0 and 32767 inclusive—which Micro-PROLOG is not—how many sentences would its imaginary data base contain?]

Let's use SUM to define a relation we'll need later, the predecessor/successor relation:

```
succ(x y) if
   SUM(1 x y).
```

We may read this as: "The successor of a number x is y, the number 1 greater than x." The interesting thing is that, because of the descriptive nature of SUM, we can work this relation in either direction:

```
succ(7 what)?
yes, what = 8.
succ(what 14)?
yes, what = 13.
```

Therefore, the rule may also be: "The predecessor of a number y is x, the number 1 less than y."

Other Prolog systems handle arithmetic relationships much more awkwardly: they attempt to graft the arithmetic expressions of procedural languages onto Prolog syntax. In these systems, the predecessor/successor relation would be defined with syntax such as:

```
succ(x y) if
   y = x + 1.
```

Not only does this introduce the whole truckload of reserved operator symbols and operator precedence rules into Prolog—in sharp contrast to the economy of SUM and PROD—but we are not sure whether we can use a sentence of this form to query for a predecessor as well: it is unclear whether the assertion $y = x + 1$ has meaning when worked "backwards." Such a system would require an explicit predecessor rule:

```
pred(y x) if
   x = y - 1.
```

Prolog Syntax

Many Prolog interpreters have been built, most of them academic experiments, each with its own features and its own syntax as dictated by the interests and tastes of its developers and the direction of their research. One of the most influential Prolog interpreters was built for the DEC-10 at the University of Edinburgh.[3] The best Prolog tutorial text[1] is based on this "Edinburgh syntax." Because of this book, the newer products for personal computers tend to follow the Edinburgh syntax as well. However, it would be unrealistic to expect that any interpreter would follow the syntax exactly; after all, an implementor must display creativity somehow.

Another influential implementation is the Micro-PROLOG product; because it is available for both Z80-based CP/M systems and for MSDOS, Micro-PROLOG has the potential to outnumber all other Prologs put together.

There are only eight syntactic concepts in the Prolog notation: four kinds of items (numbers, variables, constants, and lists) and four higher groupings (clauses, conjunctions, facts, and rules). Even after adding whatever protocols are needed to signal the interpreter that a particular input represents a query or an assertion, you still have a very simple language. Let's see how these eight elements are formed in the two main syntactic traditions.

Although systems should differ little in the formation of literal numbers, they will differ in the magnitude and precision of the numbers supported. Some systems will support only signed integers while others allow reals.

On the other hand, systems will differ dramatically in the rules for forming variables. Edinburgh-style systems take as a variable any token that begins with an initial cap: John is a variable, but john is a constant. This allows a good range of meaningful variable-names, but it is easy to make a mistake and type an initial cap in what should be a constant, as in the assertion

```
likes(John,Mary).
```

which asserts that anyone likes everybody.

We may class the underscore as a capital letter, so that _john becomes a variable. The underscore alone is sometimes an "anonymous variable," a nameless place-holder used to name any argument that is required but not used elsewhere in the sentence.

Micro-PROLOG limits variables to the initial letters "XxYyZz" followed by zero or more digits. This restriction pinches: it makes it difficult to read sentences of more than trivial length. Contrast these versions of the QuickSort sentence:

```
qsort([A B C|Tail] Out) if
   part([A B C|Tail] Left Right) and
   qsort(Left Sleft) and
   qsort(Right Sright) and
   cat(Sleft Sright Out).

qsort([X Y Z|x] y) if
   part([X Y Z|x] z X1) and
```

```
qsort(z Y1) and
qsort(X1 Z1) and
cat(Y1 Z1 y).
```

The readability of variables is less important than it might seem at first; apparently no Prolog system retains variable-names after entering a sentence to the data base! When the sentences are listed afterward, they usually have system-generated numeric place-holders in the variable positions. Micro-PROLOG simply restricts input variables to what the system will generate anyway.

After the Prolog system has set the rules for variables and has preempted a set of special punctuation characters, whatever is left is a constant. Uncapitalized words are constants in all systems, but particular systems may use many other tokens as constants as well. Hyphenated words are usually valid constants, although some systems may take mother-in-law as an arithmetic expression. In Micro-PROLOG, certain characters are both constants and delimiters; that is, they cannot be grouped with other characters in larger tokens, but they may be used as constant tokens themselves. The square brackets fall in this category: [alone is a constant, but [a] is interpreted as three constants in a row.

Most systems allow some kind of escape so that a constant can be made from an arbitrary sequence. Edinburgh systems use the single quote ('The Tree' is a constant), while in Micro-PROLOG, the double quote ("The Tree") serves that function. This creates a problem, because in Edinburgh systems the double quote supports a glitch: it indicates a *list* composed of the quoted ASCII characters. Micro-PROLOG achieves the same result with the built-in relation STRING-OF(*c list*), meaning *list* is a list composed of the letters of constant *c*. The Micro-PROLOG relation can be worked backwards, from a list to a constant, but not so the Edinburgh syntactic glitch.

In most Prolog dialects, a list pattern is [*car* | *cdr*], but in Micro-PROLOG it's (*car*|*cdr*)—using parentheses. The Edinburgh-influenced systems use the brackets and parentheses to make a distinction:

```
cat([ ] x x ).
```

saying, in effect, that a list *as* an argument is different from a list *of* arguments. The Micro-PROLOG syntax doesn't make that distinction (for good reason, as we'll see shortly), so its equivalent formulation would be:

```
(cat ( ) X X)
```

We can convert an Edinburgh-style sentence to Micro-PROLOG form by making a series of simple transformations, one of which is to convert all square brackets to parentheses.

This brings us to the clause, the fundamental way of naming a relationship between items. In this article, I've followed the Edinburgh style, making a clause look like a procedure call in a prescriptive language. That puts the relation-name on a higher level than its arguments, so that

```
length([ ] 0).
```

With a descriptive definition of arithmetic, however, the predecessor relation is simply:

```
pred(y x ) if
   succ(x y ).
```

Unfortunately, the designers of Micro-PROLOG stopped short of implementing all the implications of descriptive arithmetic. Their interpreter insists that the SUM and PROD relations be given enough literal data so that the query has a determinate answer. This means that the question SUM(*x y* 18)? or "what numbers *x* and *y* add up to 18?" won't be answered; the answer is indeterminate because an infinity of number pairs add up to 18. I think these relations should generate random integers for the unknowns, in arbitrary sequence, until they have enough knowns to solve the query. How charming it would be to ask the system PROD(*a b p r*)? and be told "yes, $a = 2$, $b = 138$, $p = 11$, $r = 265$," or some other arbitrary but consistent answer!

Micro-PROLOG's descriptive approach to arithmetic is not important in itself; let it serve as an example of the power and economy made possible with a descriptive style of programming—and as an example of how strange and unsettling such a style initially can seem.

Getting Formal

It's time for some formal nomenclature. Writers on Prolog don't agree on the names of Prolog's parts of speech (any more than its implementors agree on a syntax), so some of these names are my own invention. The named concepts, however, are always present and easy to identify.

A Prolog program consists of a collection, usually called a *data base*, of *sentences* that define *relations*. A sentence's initial word identifies the relation of which it is a part; all the sentences that begin with that word define the relation. In a preceding example, we defined the card relation in five sentences, each beginning with the word "card."

In addition to the relations defined by the programmer, certain fundamental relations are defined by code built into the Prolog interpreter.

A *clause* is a word followed by a parenthesized list of (possibly zero) *items*: card(6), for example. We'll explore the variety of items shortly. (What I call a clause different writers have dubbed a predicate, a function, and an atom.)

Sentences have two forms. A *fact* consists of a single clause:

```
card(k).
rank(k 13).
```

The sense of a fact is that the named relation is always true of the given items. The second form of a sentence is the *rule*. It consists of a clause, an **if** symbol, and one or more clauses separated by **and** symbols:

```
succ(x y ) if
   SUM(x 1 y ).
card(x ) if
   integer(x ) and
   less(1 x ) and
```

```
less(x 11).
```

A sequence of one or more **and**-delimited clauses is called a *conjunction*. The sense of a rule is that the relation is true of these items as long as all of the clauses of the conjunction are also true.

The function of the Prolog interpreter is to answer *queries*. The form of a query is simply a conjunction, usually delimited by some kind of query symbol to trigger the interpreter into action. The interpreter answers a query by verifying each clause within it against the data base. Now, take this slowly: a clause is verified (1) when it matches a fact, or (2) when it matches the clause at the head of a rule *and the conjunction from that rule has been verified as a query*.

In other words, to answer the query card(6)? the interpreter will match the query's first and only clause to the clause at the head of the following rule:

```
card(x ) if
   integer(x ) and
   less(1 x ) and
   less(x 11).
```

As a result, it will pose the conjunction

```
integer(6) and
less(1 6) and
less(6 11)?
```

to itself as a new query.

Zowie! Recursion! This nesting of queries continues until all clauses match facts or system-defined relations or until a clause that can't be matched causes the query to fail. That's the entirety of Prolog execution: the recursive matching of clauses into the data base until all clauses have been verified or until some clause can't be matched.

In principle, there is no defined order for searching the sentences of a relation and no defined order for verifying the clauses of a conjunction. In practice, programmers rely heavily on the expectation that the system will test the sentences of a relation in the order they are asserted, not attempting the second until the first has failed. Also in practice, there is a strong assumption that the clauses of a conjunction will be verified in order from left to right.

Item Types

We've seen almost the whole of Prolog syntax: a relation that is composed of sentences, a sentence that is either a fact or a rule, and a query that is merely a conjunction to be tested one clause at a time. All that remains is to specify what sort of items we may name as the arguments of a clause. There are four: the *number*, the *constant*, the *variable*, and the *list*.

The number needs no explanation; it is simply a literal number. Some Prologs support only integer arithmetic, while others allow floating-point numbers.

The constant is a notion that may be unique to Prolog. A constant is a literal word. It doesn't stand for anything, being neither the name of something nor a character array or string. The value of a constant is simply itself, just as the value of 6 is 6, the value of "a" is "a," and the value of

seems to say "length is true of . . . ." In fact, this is a false distinction, albeit a harmless one. The interpreter's matching process does not depend on the form of the clause; it would work just as well if the relation-name were inside the parentheses:

```
(length [ ] 0).
```

Matching is just a comparison of items from left to right.

Clauses are combined to make conjunctions, sequences of clauses that are true only if all their component clauses are true. A query is a conjunction. A rule is a clause plus the conjunction that qualifies it, and a fact is a clause that needs no qualification. There is an implicit *and* between each clause of a conjunction and an implicit *if* between the head of a rule and its conjunction.

Whoever devised the Edinburgh syntax established the colon-hyphen (:-) as the if symbol, set the comma as the and symbol, and required a period at the end of an asserted sentence (but a question mark at the *start* of a query!). Thus, the Edinburgh form of length is

```
length([ ] 0).
length([Head | Tail] Len) :-
   length(Tail Tien),
   succ(Tlen Len).
```

Some systems, however, may not permit multi-line input.

The Micro-PROLOG syntax for clauses, conjunctions, and rules is rigorously logical but even less friendly:

```
((length ( ) 0))
((length (X | Y) Z)
   (length Y x)
   (succ x Z))
```

In this inner, or fundamental, syntax of Micro-PROLOG, everything is lists. A clause is a list whose first item is its naming constant. A sentence is a list of clauses. We can formalize these elements using the presentation syntax of this article:

```
clause([name | args ]) if
   constant(name ).

conjunction([ ]).
conjunction([c | etc ]) if
   clause(c ) and
   conjunction(etc ).

sentence([head | conj ]) if
   clause(head ) and
   conjunction(conj ).
```

Undoubtedly, Edinburgh-style interpreters store asserted sentences in a similar nested-list arrangement; they simply put a nice front end on it.

So does Micro-PROLOG, except that it uses a variety of front ends, each written in Prolog. Because the system allows

"card" is "card."

The function of the constant, of course, is to supply a comfortably large set of atomic symbols that need no further interpretation so we can write interesting facts in human-readable form. That different systems have different rules for forming constants is a major irritation (see the sidebar "Prolog Syntax").

The variable is a syntactic place-holder; it means "whatever matches to this position." In these hypothetical examples, I'm italicizing variables, but we obviously cannot do that with real Prologs. The syntactic space of word-like objects must be divided between constants and variables. Because different implementors draw the line in different ways, a variable in one system may be a constant in another.

List Notation

Most Prologs support the list data structure of Lisp (indeed, most Prologs are essentially special bodies on a Lisp chassis). However, a Prolog system does not construct lists; it *describes* them, and the syntax for describing a list is central to most Prolog applications.

A list is described in full by a list of items in brackets, so that [a k q [8] 10] is a list of five items (three constants, a list, and a number), while [ ] is the empty list. (Most Prologs use special brackets like these. One of Micro-PROLOG's less endearing features is that it uses ordinary parentheses.)

Often it is convenient to describe only the head of a list, referring to the remainder of the list with a variable. This is done with a *list pattern* that employs the *list constructor symbol*, usually the stile (|). Thus, the pattern [6 | *etc*] is a description of all lists having the number 6 as their first item. The pattern describes the first item exactly but says nothing about the remainder of the list except that the variable *etc* stands for it.

Now, mark two crucial rules. First, *the tail of a list is a list.* Thus, when the pattern [6 | *etc*] is matched to the list [6 5 4], *etc* stands for [5 4], a list. Should it be matched to [6 5], *etc* would stand for [5]—the list [5], not the number 5.

Second, *the empty list is the tail of all lists.* This rule supplies a tail for lists that have but one visible element, enabling the matching of list patterns to be more consistent. Thus, the pattern [6 | *etc*] describes the one-item list [6], with *etc* standing for [ ], the empty list that is the tail of all lists, even short ones.

From these rules come certain common effects. First, the empty-list pattern, [ ], matches only the empty list; no other pattern does this. It is really the only special case among lists.

Second, the pattern [*head* | *tail*] matches any nonempty list. A list pattern with a single variable preceding the list constructor matches any list that contains at least one item.

In fact, the pattern [a k *next* | *etc*] matches any list that contains at least three items, the first two being the constants a and k and the third being unspecified. The variable *etc* is identified with whatever items follow those three items (or with the empty list if there are no more).

Recursive Descriptions

How many items does a list contain? Let's describe the solutions to that problem in an example of real Prolog programming. We could enumerate all possible solutions:

the meta-variable, Prolog programs can construct and then execute other Prolog programs. In essence, Micro-PROLOG's syntax is protean: it can be whatever you wish to disguise it as. The friendliest of the distributed front ends is called SIMPLE; its form of clauses and rules is much like the presentation syntax used in this article.

Prolog in Parallel

There is an implicit *or* between the sentences that define a relation. Take the following card relation:

```
card(a).
card(k).
card(q).
card(j).
card(x ) if
    integer(x ) and
    less(1, x ) and
    less(x, 11).
```

Together these sentences say that a card is this *or* this *or* . . . this. The Prolog implementation must search through the statements to answer a question about a card, but (in principle) the order in which it tests them doesn't matter. If hardware is available to make concurrent tests of all five sentences in parallel, fine. Such parallelism is called Or-parallelism because of the implicit *or* between sentences.

Notice, too, in the last sentence, that there is (in principle) no preferred order for testing the three subordinate assertions. Proposed Prolog systems would evaluate all three assertions concurrently. This parallelism is called And-parallelism because of the *and* relation between the clauses of a conjunction. Either kind of parallelism (properly called concurrency) or both could be used to speed Prolog execution. The Japanese "Fifth Generation" project proposes to explore such highly parallel architectures for Prolog-like languages.

Notice, however, the careful use of "in principle" in the preceding paragraphs. We can express many algorithms, like the card relation and most of the other examples in this article, as sets of pure, achronic, descriptive sentences. Such descriptions executed on concurrent hardware would produce the same results as those produced under a single-thread interpreter. But it is not always easy to see the time-free, descriptive solution to a problem (especially when your mind is thoroughly steeped in the if-then patterns of procedural notations); sometimes there is no solution.

For instance, no nondeterministic description exists of a program that prompts its user and then takes input from the keyboard. There is some time t1 when no prompt is visible, some time t2 when a prompt is visible but no response has been typed, and some time t3 when the response is complete—these three times are *not* the same. Such a program *must* recognize the flow of time. Otherwise, it theoretically could read nonexistent input and prompt afterward.

As noted in the main article, similar time dependencies arise when programs assert and retract facts dynamically. To make such programs work, the available Prolog interpreters promise to evaluate the clauses of a conjunction in

```
length( [ ] 0 ).
length( [x ] 1 ).
length( [x y ] 2 ).
```

That, however, gets tedious very quickly. A general approach to problem descriptions of this sort is to enumerate only the limiting case(s) then to describe the general case *recursively*. If we apply that approach to this problem, we soon arrive at:

```
length([ ] 0).
length([h | t ] l ) if
   length(t  k ) and
   succ(k  l ).
```

In English this says: "The length of the empty list is zero, while the length of a nonempty list is one more than the length of its tail."

Please examine the two Prolog sentences that define the length relation. Do you understand them? Can you see how they are sufficient to permit the dialogue

```
length([a k q] len )?
yes, len  = 3.
```

to take place?

If they seem utterly opaque, that might be for either of two reasons. You may not be familiar with the use of recursion to solve problems. If you are baffled by the idea of describing length in terms of length, turn to the sidebar "The Recursive Method" (at right). It covers a general approach to recursion and may get you over the initial hump.

If you are comfortable with the idea of recursion but can't see how a Prolog implementation might handle the given sentences, turn to the sidebar "Executing a Query" on page 54. Watching the Prolog system work its way to the root of a query may clarify these issues.

List Operations

Let's do one more fundamental list operation, the member relation. This is the relation that is true (verifiable) if a particular item is a member of a given list. We wish to be able to ask such questions as:

```
member(j [k j 9 3 2])?
yes.
member(a [k j 9 3 2])?
not so far as I know.
```

The recursive method works well here. Because Prolog makes the head item of a list immediately visible, one limiting case is that in which the desired item is visibly the head of the list:

```
member(item [item | etc ]).
```

We may read this as: "Some item *item* is a member of any list that begins with *item*."

[This use of variables troubled me at first. It seemed too easy! I wanted to write something like

order from left to right. Programs that rely on that promise could never be executed on And-parallel hardware. In my opinion, there should be a Prolog equivalent of Ada's and-then operator to make time dependencies within a conjunction explicit.

Other cases—too elaborate to go into here—make it seem essential to test the sentences of a relation in a particular order. Present interpreters number the sentences of a relation in the sequence they are entered and promise to evaluate them in that sequence. Programmers writing code that relies on that promise are operating sometimes from necessity but more often from procedural habit. Such code will not execute reliably on Or-parallel hardware.

The Recursive Method

There are two ways to completely express a lengthy process in short terms: iteration and recursion. Iteration—the for, while, and until loops and other variations on the theme of "go back and do it again"—is a customary part of any procedural language. Thus,

```
for i := 1 to 10 do
   sum := sum + a[i];
```

is a short but complete summary of a 10-step process of summation. We may summarize the iterative design approach as: (1) decide how to solve the problem for one item and (2) decide how to repeat that solution for all the other items.

Prolog supports no iteration. Fortunately, many problems will yield to a recursive approach. We may summarize the recursive design approach this way: (1) enumerate the simple cases for which a solution is immediately known; (2) specify how to reduce a complicated case to one that is slightly simpler; (3) specify a general solution in the terms "if I knew the answer to the next-simpler case, the answer to *this* case would be . . . ."

Because step (2) supplies the means to generate the next-simpler case, and because repeated application of step (2) must eventually produce a case for which step (1) has defined an answer, we may write the general solution as "return the result of (some small function) on the result returned by applying this general solution to the next-simpler case."

Raising a number to an integral power is an example of a problem that we can handle this way. We want to describe the solutions to

```
power(x p z )
```

such that z is the result of raising x to the pth power.

Step (1): a simple case for which we know the solution immediately is

```
power(x 0 1).
```

or anything raised to the zeroeth power is 1. The complicated cases comprise all those with p greater than zero. (Actually, we know that power(x 1 x), but it turns out we don't need the knowledge.)

Step (2): to specify how to move the complicated cases one step in the direction of being a simpler case, we simply need to reduce *p* by 1, thus moving it one step toward the simple case of $p = 0$.

Step (3): applying the magic incantation "if I knew the answer to the next-simpler case, the answer to *this* case would be . . . ," we can say if we knew the answer to power(*x p-1 x-to-the-p-less-one*), we could multiply it by *x* to yield power(*x p x-to-the-p*). Or in Prolog:

```
power(x p z ) if
    sum(1 simplerp p ) and
    power(x simplerp simplerz ) and
    prod(x simplerz z 0).
```

Any recursive design should be tested for termination; an infinite regression is not a valid answer. Will the nesting ever cease? The formulation for power is unsafe in this regard. If it is queried with *p* as an integer greater than zero, it will work; each evaluation will reduce *p* by one step toward the simple case of $p = 0$, which it must ultimately reach. But *query it with a* fractional *p*, and it will skip right past $p = 0$ and continue on toward the horizon—as it will if it is queried for any *p* less than zero. To play it safe, the clauses

```
integer(p ) and
less(0, p ) and
```

should be inserted.

The speed of an iterative solution is proportional to the number of iterations. Similarly, the speed of a recursive solution is proportional to the depth of its nesting. The power algorithm just given nests *p* deep. Another recursive design for power nests to drastically fewer levels; math fans may enjoy following it:

```
power(x 0 1).
power(x 1 x ).
power(x p z ) if
    less(1 p ) and
    integer(p ) and
    prod(2 q p  r) and
    power(x q xbyq ) and
    prod(xbyq xbyq xevn 0) and
    condmul(x xevn r z ).
condmul(x xevn 0 xevn ).
condmul(x xevn 1 z ) if
    prod(x xevn z 0).
```

The main difference is that it uses a faster way to march the general case toward the simple case, dividing it by two rather than reducing it by one. This formulation also illustrates one way to handle a conditional operation in Prolog without relying on statement sequence.

Recursive solutions most often are needed to describe lists of any length. The limiting cases typically involve the empty list or the list of one item. The method of moving the general case toward the simple case is to shorten a list by dropping its head.

```
member(item [x | etc ]) if
    equal(item x ).
```

to make the comparison explicit. But that impulse is a hangover from the habits of procedural languages, where comparison is an "operation" to be "executed" upon "data." In Prolog, we don't write a *test for* equality; we write a *description of* equality. The variable-binding rules of Prolog interpreters are so implemented that the more succinct statement works. Consider matching query member(fred [mack pete])? versus the sentence member(*item* [*item* | *etc*]). The matching process begins with the binding of the constant "fred" to the variable *item*. Instantly, the remainder of the question becomes:

```
...[mack pete])?
...[fred | etc ]).
```

This soon will result in detection of a mismatch.]

We now have established the limiting case. In the general case—when the desired item isn't visible as the head item—we must hope that it appears somewhere in the tail of the list. If it does, it must be the head of some portion of the tail. So we drop the present head and try again:

```
member(item [head | rest ]) if
    member(item rest ).
```

This passes the termination test: each application of the general rule talks about a shorter list. The process sooner or later must either bring about the limiting case or else pose a query, member(*item* [ ]), that will match nothing and cause the whole query to fail.

This formulation of the membership problem will work, but it is not perfect Prolog. The two sentences aren't independent; certain queries will match both of them. I have followed up the implications of that in the sidebar "Cutting and Failing" on page 58.

Having It Both Ways

Conventional (procedural) programs are time-bound and linear, so they can flow in one direction only. Long use of procedural languages makes this seem a natural and inevitable part of programming, but it is neither. We've already noted how, when arithmetic is implemented in a descriptive way, a single relation SUM supplies both addition and subtraction. When we based a successor relation on it, the identical relation served to describe the predecessor relation as well—the predecessor being simply the successor "in reverse."

This ability to work Prolog relations backward seems strange and upsetting at first. It takes a conscious effort to recall that a reverse use is even possible and to check for it. For example, a programmer who is oriented toward procedural languages inevitably will identify the member relation with a Pascal or C function that searches for one item in a list. Such a function can be used in those languages only for that purpose, but this is not so of the member relation in Prolog. Sure, Prolog will test to see if a member exists in a list, but it is also perfectly happy reporting what members do exist in a list:

```
member(x [fred bob sam])?
yes, x = fred.
yes, x = bob.
yes, x = sam.
```

Furthermore, Prolog can be made to generate an infinite number of lists that contain a given item. The query member(*x what*)? or "What's a list containing something?" leads to an infinite recursion producing successive answers:

```
yes, what = [x | etc ].
yes, what = [head x | etc ].
yes, what = [head head x | etc ].
```

Not a useful result, but a strange and beautiful one.

Showing Off

We could use any number of enjoyable list problems for further practice: reversing them, extracting the first or last *n* items, catenating them (an especially interesting Prolog algorithm that can be worked backwards in several useful ways). But space is limited, so let's glance instead first at programs for sets and trees and finally at a sort.

A set is a list that contains no duplicates. Let's first define a relation undup(*in out*) such that *out* is the list *in* with its duplicate items removed:

```
undup( [ ] [ ] ).
```

i.e. the empty list has no duplicates;

```
undup([h | t ] [h | tnew ]) if
    remove(h t tlessh ) and
    undup(tlessh tnew ).
```

i.e. other lists are undup'd by removing all instances of the head from the tail then undup-ing the tail.

Of course, we now require a definition of the relation remove(*x in out*) such that *out* is the list *in* with all occurrences of item *x* removed:

```
remove(x [ ] [ ]).
```

i.e. removing *x* from the empty list yields the empty list;

```
remove(x [x | t ] tnew ) if
    remove(x t tnew ).
```

i.e. removing *x* from a list headed by *x* produces the tail of that list with *x*'s removed;

```
remove(x [h | t ] [h | tnew ]) if
    not-same(x h ) and
    remove(x t tnew ).
```

i.e. removing *x* from a list headed by something else (*h*) produces a list still headed by *h* but with *x*'s removed from its tail.

Naturally, it's easiest to build a set from scratch, simply banning all duplicate additions to it. Let's define the relation addset(*x in out*) such that *out* is the set *in* with item *x* added:

For example, the normal Prolog list-pattern syntax makes it easy to describe the first element of a list (*head* in the pattern [*head* | *tail*]), but there is no primitive way to describe the last item in a list. Can we describe a relation last(*list item*), meaning that *item* is the last thing in *list*?

Let's apply the recursive method. The empty list isn't a factor here because it has no last (or any other) item. The shortest list having a last element is the one-item list; we may at once assert:

```
last([only ] only ).
```

This means "the last item in the list [*only*] is the item *only* ."

A way of simplifying a list is to drop its head. If we knew the last item in the tail of a list, we would know the last item in the whole list:

```
last([x y | etc ] item ) if
    last([y | etc ] item ).
```

The solution must pass the termination test: does it advance toward the limiting case? Yes, each application of that sentence poses a new query about a list that is shorter by one item. Ultimately, it will pose a query about a *one-item list*, for which we have a fact.

A good Prolog solution should pass two more tests: independence and completeness. Are the two sentences independent; that is, do they describe distinct situations? These do, but only because the second one explicitly describes lists of two or more items. If we had written

```
last([head | tail ] item ) if
    last(tail item ).
```

then a one-item list could match to either rule. Are the sentences complete? Yes, the first covers one-item lists and the second covers all those with two or more items; thus, they cover all lists that have a last item. These two sentences constitute a good, recursive Prolog description of the last-item-of-a-list problem.

Executing a Query

A Prolog query is a conjunction or a list of clauses each of which is to be verified. Let's observe the actions of the interpreter as it executes the query

(1) length([a q j] *what*)?

against a data base of this length *relation:*

```
length([ ] 0).
length([head | tail ] len ) if
    length(tail tlen ) and
    sum(tlen 1 len ).
```

A query is verified by verifying its clauses; a clause is verified by matching it to the facts and rules of the data base.

addset(*x* [ ] [*x*]).

i.e. adding *x* to the empty set produces the set composed of *x* ;

addset(*x* [*x* | *t*] [*x* | *t*]).

i.e. adding *x* to the set already headed by *x* produces no change;

addset(*x* [*h* | *t*] [*h* | *tandx*]) **if**
not-same(*x h*) **and**
addset(*x t tandx*).

i.e. adding *x* to a set not headed by *x* is done by adding *x* to the tail of the set.

The clause not-same(*a b*), used in addset and remove, is one that the underlying system should define. It is extremely difficult to describe the not-equal relation in Prolog.

We can implement—or describe—binary trees in terms of nested lists. In fact, their Prolog description is simply a rephrasing of the theoretical definition of a binary tree. Let's say that an empty list represents the empty tree, while a three-item list [*lsub item rsub*] represents the nonempty tree. *item* is the value at the root of the tree, and *lsub* and *rsub* are the left and right subtrees. Now we can define the relation addtree(*x in out*) such that *out* is the binary tree *in* with item *x* added to it:

addtree(*x* [ ] [[ ] *x* [ ]]).

i.e. adding *x* to an empty tree produces the tree with value *x* and empty subtrees;

addtree(*x* [*l x r*] [*l x r*]).

i.e. *x* is already in the tree that has it as a value;

addtree(*x* [*l v r*] [*lnew v r*]) **if**
less(*x v*) **and**
addtree(*x l lnew*).

i.e. *x* goes into the left subtree of a tree that has a greater value;

addtree(*x* [*l v r*] [*l v rnew*]) **if**
less(*v x*) **and**
addtree(*x r rnew*).

i.e. *x* goes in the right subtree of one with a lesser value. Lovely, isn't it? No doubt you can already visualize the Prolog description of a tree search, the relation intree(*x tree*) that is true if *x* appears at some level of *tree*.

Let us return to the domain of lists and sets long enough to describe the QuickSort algorithm: the relation qsort(*in out*) such that *out* is the list *in* with its items in sorted order. First, we enumerate enough of the limiting cases to cut the recursion off at the two-item list:

qsort([ ] [ ]).

In that matching process lies much of the magic of Prolog. Many texts laboriously describe how variables are "instantiated" (I prefer to say "bound") as they are matched to constants, but the process is simply *copying with substitution*: the interpreter copies out the matched rule, substituting new values wherever a variable was matched.

The interpreter commences by comparing the first (and only) clause of the query to the first sentence in the data base:

length([a q j] *what*)
length([ ] 0)

They don't match; the mismatch comes just after the opening bracket when the constant a doesn't match the closing bracket. The interpreter tries again at the next sentence:

length([a q j] *what*)
length([*head* | *tail*] *len*) . . .

This succeeds. A new copy of the rule is made with the following substitutions:

a for *head*
[q j] for *tail*
what for *len*

The copied conjunction, with substituted values of the bound variables, reads:

(2) length([q j] $tlen_2$) **and**
sum($tlen_2$ 1 *what*)?

The subscript in $tlen_2$ indicates that the variable *tlen* is unique and local to this query.

The first clause of the new query matches the second sentence in the data base, with the following bindings:

q for *head*
[j] for *tail*
$tlen_2$ for *len*

The conjunction is copied out once more, with substitution, to produce yet another nested query:

(3) length([j] $tlen_3$) **and**
sum($tlen_3$ 1 $tlen_2$)?

Its first clause again matches the second sentence, this time binding the following:

j for *head*
[ ] for *tail*
$tlen_3$ for *len*

Its conjunction is copied out; after substitution, it yields:

(4) length([ ] $tlen_4$) **and**
sum($tlen_4$ 1 $tlen_3$)?

i.e. the empty list is sorted;

```
qsort([x ] [x ]).
```

i.e. the one-item list is sorted;

```
qsort([a z ] [a z ]) if
    leq(a z ).
qsort([a z ] [z a ]) if
    less(z a ).
```

i.e. two-item lists are sorted by exchanging their items (we assume that the leq( ) and less( ) clauses are system defined).

Now we can describe the general solution of QuickSort: the items of the input list are partitioned into two lists, which are sorted separately:

```
qsort([a b c|t ] out ) if
    part([a b c|t ] l r ) and
    qsort(l sl ) and
    qsort(r sr ) and
    catenate(sl sr out ).
```

Assuming that the catenate relation already has been described, we need only to define the partitioning scheme, part(*in l r*), such that *l* and *r* are lists containing all the items of *in* between them with every item in *l* being less than any item in *r*.

I've worked out two ways to describe the part relation. The first makes two passes over the data:

```
part(in l r ) if
    pivot(in p ) and
    partle(in p l ) and
    partge(in p r ).
```

Leaving aside the pivot relation, which describes *p* as a pivot value based on list *in*, we need definitions of partle and partge. The partle relation selects those elements of *in* that are less than or equal to *p*:

```
partle([ ] p [ ]).
partle([h|t ] p [h|z ]) if
    leq(h p ) and
    partle(t p z ).
partle([h|t ] p z ) if
    less(p h ) and
    partle(t p z ).
```

The partge relation is similar but, of course, selects only items that are greater than *p*.

It seemed somewhat inefficient to pass twice over the input list when dividing it, so I worked out an alternate scheme:

```
part(in l r ) if
    pivot(in p ) and
    partem(p in l r ).

partem(p [ ] [ ] [ ]).
```

The first clause of this conjunction matches the first sentence:

```
length([ ] tlen4) . . .
length([ ] 0).
```

As a result, the variable $tlen_4$ is bound to the literal number 0; it is effectively rewritten to 0 wherever it appears in the pending queries. Having verified the first clause of query (4), the interpreter considers its second clause, which now reads:

```
. . . sum(0 1 tlen3)
```

Assume that sum is a built-in relation; the system adds the known arguments and binds their sum to the unknown one, causing $tlen_3$ to be rewritten as 1 wherever it appears. Query (4) is now complete; the system may return to consideration of query (3).

The second clause of query (3) now reads:

```
. . . sum(1 1 tlen2)
```

After that clause has been evaluated, references to $tlen_2$ will read 2. That completes processing of query (3); the interpreter may return to query (2).

Its second clause now reads:

```
. . . sum(2 1 what )
```

Its evaluation rewrites all appearances of *what* as 3. The text of query (1) now reads:

```
length([a q j] 3).
```

Because this user-entered query has now been verified as truthful, the system can notify the user "yes" and may also display *what* = 3. That's not essential; a very simple Prolog might require the user to request a display explicitly, using a built-in relation such as:

```
length([a q j] what ) and
display(what )?
```

Had that been the case, verification of the first clause would have resulted in rewriting the query as:

```
length([a q j] 3) and
display(3)?
```

Nothing special has to be done by the system—the display argument falls out naturally from the implicit rewriting of variables as they are bound.

Thus, relatively simple matching and rewriting rules, applied recursively, produce effects all out of proportion to their size.

Cutting and Failing

Let us consider taking our member relation, defined by the following sentences:

```
partem(p [h|t] [h|l] r) if
    leq(h p) and
    partem(p t l r).
partem(p [h|t] l [h|r]) if
    less(p h) and
    partem(p t l r).
```

Finally, we may define the pivot relation in a variety of ways. The simplest is to use the first item: pivot([*h* | *t*] *h*). That causes poor performance, however, when the input is already sorted. Because, by our definition of the general case of qsort, we need to describe a pivot only for lists of three or more elements, we get a better average partition with a pivot that is the mean of the first and third items:

```
pivot([a b c|t] p) if
    SUM(a c s) and
    PROD(2 p s 0).
```

I've tested these versions of QuickSort on Micro-PROLOG, although not on lists long enough to compare their relative speeds.

Storing Data

But, I hear you asking, how can we conduct a meaningful test of a sort if the only sample data is what we can type in a query? It may be amusing to have a dialogue like

```
qsort([3 2 1] what)?
yes, what = [1 2 3].
```

but this hardly amounts to useful data processing. Where do we put stuff so we can work on it?

That appears to be a natural question, but it reveals unthinking prejudices.

Procedural languages enforce a radical distinction between code and data. Data is static; it is stored in variables; it is acted upon by code. Code is dynamic; it is expressed by statements; it takes data as input and produces data as output. These notions are so engrained in the conventional programming languages that we aren't even aware of them—until we encounter a notation like Prolog, which denies them all!

Prolog makes no difference between data and code; the distinction is simply not relevant. But, you ask, how do we store information? We do it by *asserting the truth of that information.* The asserted sentences become part of the data base. Everything is sentences; we may think of *any* sentence as either code or data. Consider that the query

```
qsort([3 2 1] what)?
yes, what = [1 2 3].
```

could be answered equally well by our algorithmic qsort definition or by a simple assertion of fact: qsort([3 2 1] [1 2 3]). In principle, there is *no difference* between the algorithmic qsort (code) and the qsort implemented by an infinite set of facts (data).

But practical problems arise from the mode of interactive use. How can we develop data during one query, yet have the same data available for another later query? Suppose, for

```
member(item [item|tail]).
member(item [head|tail]) if
    member(item tail).
```

and apply to it the query

```
member(j [j k])?
```

These sentences are not independent; the query matches both of them. It matches

```
member(item [item|tail]).
```

with *item* bound to j and *tail* bound to [k], and it also matches

```
member(item [head|tail]) . . .
```

with *item* bound to j, *head* to j, and *tail* to [k]. Which sentence would Prolog choose to use, and what would happen then?

The Prologs that run on personal computers practically guarantee that the sentences of a relation will be tested in the order they were entered to the data base. If the limiting case is entered first, as here, it will be matched first. You'll find plenty of cases where programmers rely on this implicit order of evaluation. If the interpreter is asked to find only one answer to the query, it will match to the first sentence (a fact) and, having verified the query, will reply "yes" and stop.

But in some cases, the interpreter will continue to look for any further answers to a query. In our example, it would continue on to match the query to the second sentence. That sentence is a rule, and its conjunction specifies testing the query, rewritten as follows:

```
member(j [k])?
```

This does not match the first sentence, but it does match the second one, binding *item* to j, *head* to k, and *tail* to the empty list. This results in a further nested query

```
member(j [ ])?
```

which matches neither sentence. Thus, this branch of the trail ends with failure. No additional answers have been found; the only effect has been some wasted processing time.

We could have avoided the extra work had we used the built-in "cut" relation in the first sentence. Cut has only the descriptive meaning of true, so inserting "cut( ) **and** . . ." in any conjunction has no logical effect. However, this insertion has the powerful side effect of telling the interpreter to attempt no alternative sentences for this relation! To use it in our example, we would write the first sentence as:

```
member(item [item|tail]) if
    cut( ).
```

(The clause cut( ) is written as (/) in Micro-PROLOG and !

instance, that we want to test our addset relation, retaining its output to use as input to the next test.

We start by asserting that one particular set—which we name with an unusual constant—exists and is null:

```
"The Set"( [ ] ).
```

We can query it just like any other relation:

```
"The Set"(what )?
yes, what = [ ].
```

More importantly, however, we can query it as part of a test program, test-add(*x*), that describes the effect of testing addset with item *x*:

```
test-add(x ) if
   "The Set"(in ) and
   addset(x in out ) and
   display(out ) and
   retract(["The Set" in ]) and
   assert(["The Set" out ]).
```

With this program, we begin using Prolog syntax in a linear, time-bound way. The display, retract, and assert relations all are system-defined. Each has a trivial descriptive sense and an important side effect on the Prolog environment.

Descriptively, the display(*x*) relation is simply true for any item *x*. Therefore, "display(*item*) **and**..." may be inserted in any conjunction. Its side effect is to display *x* at the console (the relation is called PP in Micro-PROLOG).

The relation retract(*list*) is true when *list* describes a sentence presently in the Prolog data base. Its side effect is to delete that sentence.

The relation assert(*list*) is true when *list* describes a valid Prolog sentence. Its side effect is to assert that new sentence, making it part of the data base. The format for describing a sentence as a list is system-dependent.

Thus, the sense of the test-add relation is: get the current value of "The Set"; add item *x* to it and display the result; replace the definition of "The Set" with the augmented set. It permits dialogues like this:

```
test-add(5)?
[5]
yes.
test-add(fred)?
[5 fred]
yes.
"The Set"(what )?
yes, what = [5 fred].
```

A test-add query is clearly time dependent. It makes no sense unless the retraction is done *after* addset and *before* assert.

This is not elegant.

This is, in fact, an indefensible kludge, so I won't try to defend it. It is how the concept of the global variable is introduced into Prolog—and how we can simulate almost any sort of data storage mechanism. For example, extending the method of test-add to make a test bed for binary tree algorithms, that we want to test our addset relation, retaining its output to use as input to the next test.

We start by asserting that one particular set—which we name with an unusual constant—exists and is null:

```
"The Set"( [ ] ).
```

We can query it just like any other relation:

```
"The Set"(what )?
yes, what = [ ].
```

More importantly, however, we can query it as part of a test program, test-add(*x*), that describes the effect of testing addset with item *x*:

```
test-add(x ) if
   "The Set"(in ) and
   addset(x in out ) and
   display(out ) and
   retract(["The Set" in ]) and
   assert(["The Set" out ]).
```

With this program, we begin using Prolog syntax in a linear, time-bound way. The display, retract, and assert relations all are system-defined. Each has a trivial descriptive sense and an important side effect on the Prolog environment.

Descriptively, the display(*x*) relation is simply true for any item *x*. Therefore, "display(*item*) **and**..." may be inserted in any conjunction. Its side effect is to display *x* at the console (the relation is called PP in Micro-PROLOG).

The relation retract(*list*) is true when *list* describes a sentence presently in the Prolog data base. Its side effect is to delete that sentence.

The relation assert(*list*) is true when *list* describes a valid Prolog sentence. Its side effect is to assert that new sentence, making it part of the data base. The format for describing a sentence as a list is system-dependent.

Thus, the sense of the test-add relation is: get the current value of "The Set"; add item *x* to it and display the result; replace the definition of "The Set" with the augmented set. It permits dialogues like this:

```
test-add(5)?
[5]
yes.
test-add(fred)?
[5 fred]
yes.
"The Set"(what )?
yes, what = [5 fred].
```

A test-add query is clearly time dependent. It makes no sense unless the retraction is done *after* addset and *before* assert.

This is not elegant.

This is, in fact, an indefensible kludge, so I won't try to defend it. It is how the concept of the global variable is introduced into Prolog—and how we can simulate almost any sort of data storage mechanism. For example, extending the method of test-add to make a test bed for binary tree algorithms in Edinburgh-style systems.) The effect, upon a successful match to this rule, would be to cut off any attempt to find an alternate sentence for member. The second sentence would be removed from consideration as soon as the first was matched. Only if the first sentence failed to match could the second be applied.

This interpretation of cut, of course, assumes that the interpreter processes sentences sequentially in the order they are asserted. If it does not, it might attempt the second sentence before the first one, and adding cut to the first accomplishes nothing.

Consider the execution of the original member relation, without cut, in a machine that has And-parallel hardware (see the sidebar "Prolog In Parallel"). The system can execute tests of both sentences of the member relation at once, and it will find that both match the query. Now what? Well, the machine might favor facts over rules, which would make it select the right path. Or it might keep an entry sequence number with each sentence and favor the one with the lower sequence number. Or it might simply follow up both paths in parallel. In the last case, the first sentence, being a fact, will terminate quickly with success, while the second will end in failure one nesting level later.

In an And-parallel machine, the effect of the cut relation is nondeterministic. One can imagine some kind of hardware feature in which execution of cut commits intersentence fratricide, the cutting sentence aborting parallel execution of any other sentence from the same relation, but would that be useful? We could not tell how far another execution unit might have gone, executing a different sentence of the relation, before the one execution unit reached the cut-clause. So programmers using cut to avoid accidental, multiple uses of a file-read or some other do-once clause—as is recommended in Prolog tutorials—should reflect that this program is not suitable for And-parallel execution.

When you are designing Prolog solutions, it is always best to describe a relation in sentences that are independent of each other in all cases, without regard to execution sequence. We could accomplish that in our example using a not-same relation:

```
member(item [item|tail ]).
member(item [head|tail ]) if
   not-same(item head ) and
   member(item tail ).
```

These sentences are complete and independent; the solution has no dependency on either order of execution or degree of parallelism.

The "fail" relation has the descriptive meaning "false—this sentence does not match." It has some uses, but it is hard to find examples that do not rely on order of execution.

One possible use of fail is to implement a not-same relation:

```
not-same(item item ) if
   cut( ) and fail( ).
not-same(x y ).
```

This is sequence dependent, as you might expect from the appearance of cut( ). If a not-same clause is being tested with arguments that are the same, the clause will match to the first sentence. The cut clause ensures that the interpreter will attempt no further sentences of not-same; the fail clause ensures that the query fails—the same items are not not-same. A query of not-same with arguments that are not the same will not match to the first sentence, so the interpreter will test it (successfully) against the second one. That sentence is a fact, so not-same is verified.

As with most definitions that use cut( ), this is utterly without meaning unless the order of evaluation of sentences is certain; thus, it is unusable in an And-parallel machine.

rithms (using a global "The Tree") is simple. More complicated schemes are also possible: we can use assert and retract to implement (*not* describe!) a sparse array. (The array is a relation with its elements stored as facts of the form array-(*sub value*), meaning the array element at position *sub* has this *value*.)

The Meta-Variable

Some Prologs permit a variable to appear in the position of the relation-name in a clause. This feature, the meta-variable, makes it possible to write Prolog algorithms *about* Prolog.

As an example of how we might use the meta-variable, consider the "over" notation of APL. In that language, a function may be applied "over" an array. For instance, in

```
+/3 1 9 6 4
```

the plus is the function being applied, and the slash means apply it "over" the following list of numbers. The effect of "over" is to insert the function symbol between every pair of numbers, so that the preceding line is equivalent to:

```
3 + 1 + 9 + 6 + 4
```

In short, "plus over" is equivalent to summation, "max over" to largest, and so on.

We can write an "over" relation in Prolog if we use the meta-variable. The relation would be over(*fun list out*) with *out* as the result of applying *fun* between every pair of items in *list*. A relation named *fun* must exist and have the form *fun* (*a b z*), meaning *a fun b* yields *z*. In addition, we need a sentence, identity(*fun id*), such that *id* is the "identity element" of the function: the value is such that *fun* (*a id a*).

Here is a collection of sample functions:

```
plus(a b z ) if
    SUM(a b z ).
identity(plus 0).
times(a b z ) if
    PROD(a b z 0).
identity(times 1).
max(a b a ) if
    leq(b a ).
max(a b b ) if
    less(a b ).
identity(max 9E99)
```

And here is the definition of "over":

```
over(fun [ ] id ) if
    identity(fun id ).

over(fun [a|t ] z ) if
    over(fun t b ) and
    fun (a b z ).
```

This is quite faithful to the APL model, even executing right-to-left through the items and returning the identity element when applied to an empty array.

Summary

Prolog is a notation that allows us to express many algorithms in neat and elegant ways and that forces others into distorted shapes (as Bob Morein puts it, if it's hard in Fortran, it's easy in Prolog—and vice versa!). Nevertheless, it is educational to rewrite familiar algorithms in a new notation.

But the greatest value of Prolog is that, in the process of understanding it, a programmer must alter and reexamine the fundamental assumptions of his or her craft. Prolog demonstrates that the distinction between code and data is artificial and unnecessary, and it exposes the many assumptions we make about the flow of time. Prolog or a notation descended from it may or may not become important in the fifth and further generations of computing; it doesn't matter. What matters is that, after learning it, you will think about programming in a whole new way.

References

1. *Programming in Prolog*, W. F. Clocksin and C. F. Mellish, Springer-Verlag, New York 1981.
2. Micro-PROLOG is a product of Logic Programming Associates, 10 Burntwood Close, SW 18 3JU, London, England.
3. Documented in *User's Guide to DECsystem-10 Prolog*, L. M. Pereira, F. Pereira, and D. Warren, and in *A Guide to Version 3 of DEC-10 Prolog*, L. Byrd, F. Pereira, and D. Warren. Both papers are published through the Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.

DDJ

Tax Advisor
A Prolog Program Analyzing Income Tax Issues

by Dean Schlobohm

The winner of our AI programming contest is a program that knows* how *it knows what it knows, and* why *it wants to know what it doesn't know.

A year ago, in March 1984, we celebrated DDJ*'s new relationship with M&T Publishing by announcing an artificial intelligence programming contest. We would award a $1000 prize, we said, to the entry that best implemented artificial intelligence techniques in a practical application on a microcomputer. We didn't say so, but we expected the "practical" criterion to direct people toward the "applied" end of AI work, and it did. Although we gave some guidelines (strive for memory efficiency, real-world usefulness, and advancing the state of the art), we left the judging criteria somewhat loose, in keeping with the nature of the definition of artificial intelligence. We didn't pretend to know exactly what we were looking for, but we figured that we'd know it when we saw it. And we did.*

It could have been a can of worms. We didn't specify processor, language, operating system, disk format. We didn't even specify that it be for a disk-based machine! For all we knew, we might get a thousand different programs for a thousand different machines.

We weren't too worried, though. The prize was not extravagant for the amount of work that was required, and we expected to receive a small number of entries. And we did. After preliminary culling, we found ourselves with only four entries. For four machines and four disk formats. From Mexico came Gerardo Cisneros' program for solving certain kinds of algebraic problems that arise in quantum mechanics. We were awed by the idea of his crunching matrices of many-electron wavefunctions on a 56 TPA CP/M system. There was Richard Grigonis's Prison Doctor, which took the overused Doctor model of Weizenbaum's Eliza program and turned it into something ingenious and new. This was only the latest in a series of entries by Richard, one of our most creative contributors and the author of last year's controversial "Sixth Generation Computers." Stephan Heumann brought us Logicon, a Commodore 64 BASIC program that simplified Boolean expressions. Stephan's program combined the formalisms of G. Spencer-Brown with some insights of his own and made clever use of color to discriminate levels of analysis. Then there was Dean Schlobohm's tax advisor program, a solid and impressive little expert system. I'd add, "just in time for doing your taxes," but I doubt that many readers of DDJ *are prepared to make use of this kind of legal tax advice.*

Judging was no trivial matter, even with so few entries. We enlisted the aid of referees and friends with the necessary machines and appropriate expertise. We ran every program past an AI expert who judged whether it embodied the techniques of AI, a programmer competent to evaluate the quality of the code, someone conversant with the subject area of the program (logic and physics, anyway; we winged it a little regarding law and forensics), a tester who could evaluate whether and how well the program actually worked, and three editors. In the end, we chose a program that demonstrated clear understanding of what applied AI techniques are and how an AI program ought to behave. It also turned out to be a good introduction to programming in Prolog, and

---

Dean A. Schlobohm, MICRO-AI, P.O. Box 91, Rheem Valley, CA 94570. *Copyright © 1984 Dean A. Schlobohm.*

this issue consequently shaped itself into a Prolog issue. One criterion that emerged in the judging process was justification: an AI program ought to be able to tell you why it did what it did. The program we chose did that, too.

We may yet print some of the runners-up. But this month we are pleased to present the winner of our AI programming contest, Dean Schlobohm's Tax Advisor. —Ed.

The prototype computer program **TA** (Tax Advisor) assists attorneys in analyzing the constructive ownership of stock rules set forth in Section 318(a).[1] Basically, Section 318(a) sets forth a number of rules under which a given person (e.g., an individual, trust, estate, partnership, or corporation) will be deemed to own stock that is actually, or constructively by the application of Section 318(a), owned by another person. For example, Section 318(a)(1) provides that an individual is considered to own the stock actually owned by his or her spouse, children, grandchildren, and parents. Similarly, Section 318(a)(2)(A) provides that stock owned by a partnership is considered to be proportionally owned by its partners.

Although the rules set forth in Section 318(a) involve little uncertainty, their application to a given factual situation is at times tedious. Furthermore, because stock that is deemed to be owned by a given person as a result of the application of Section 318(a) may, under certain conditions, be reattributed to another person, the statute provides many opportunities for the attorney to make a mistake in his or her analysis. As a result, I have developed two prototype computer programs that perform the required legal analysis.

The computer programs are written in the language Prolog. Basically, Prolog permits rules of the form:

If: **A is true; and**
B is true; and
C is true.
Then: **D is true.**

to be represented as:

```
D :-
  A,B,C.
```

One version of **TA** consists of a set of Prolog rules that describe all of the legal rules of Section 318(a). For example, assume: (1) **spouse(H,W)** means that **H** is the **spouse** of **W**; (2) **owns(I,N,C)** means that **I** actually **owns N** shares of corporation **C** stock; and (3) **num-con-own(H,N,C)** means that **H** constructively owns, as a result of Section 318(a)(1)(A), **N** shares of corporation **C** stock.
Then the rule of Section 318(a)(1)(A) that an individual is deemed to own the stock owned by his or her spouse may be expressed by the Prolog clause:

```
num-con-own(H,N,C) :-
  spouse(H,W),
  owns(W,N,C).
```

A second version of **TA** includes the facility to "explain" its reasoning. For example, the program will ask the user only for the information that it needs to reach its conclusion. If the user is unsure why the computer asks a given question, he or she can ask the computer **why** the information is needed. The computer will then set forth an explanation of its reasoning in terms of the rules that it has used up to the point the question is asked. Moreover, the user can ask the computer for **help** in response to a given question. For example, if the user enters **help** in response to the question "Does the client have any children?" the program will explain that "children" includes legally adopted children. After the program has reached its conclusion, the user can ask the computer **how** the program reached its result. This explanation will consist of the set of rules and facts that the computer used in reaching its conclusion.

Unlike other artificial intelligence programs in the legal area,[2] the major goal of **TA** has been to develop a program that could be used by a practicing attorney with little or no knowledge of computers. Thus, a substantial amount of time has gone into designing an interface so that an attorney could use **TA** without understanding the computer language in which it is written.

Both programs currently run on a Compupro 816A microcomputer under the CP/M-86[3] operating system. Because the current version of the Prolog interpreter (Prolog-86[4]) in which the programs are written can access only 128K of RAM, the explanation features have only been included in a program that contains the family attribution rules of Section 318(a)(1). The program without the explanation features, however, has correctly analyzed the examples set forth in Treasury Regulations §§1.318-2, 3, and 4.

A Brief Review of Section 318(a)

Basically, Section 318(a) sets forth rules under which certain individuals and entities are considered to own corporate stock that is, in fact, actually owned by someone else. This is called "constructive ownership of stock" or "attribution of stock ownership." Section 318(a) creates four types of relationships that may create primary attribution of ownership. These four relationships are:
(1) Attribution between members of a family;
(2) Attribution from an entity (e.g., a partnership or corporation) to person holding a beneficial interest in the entity (e.g., a partner or shareholder);
(3) Attribution from a person (e.g., a partner or shareholder) to an entity (e.g., a partnership or corporation);
(4) Attribution to owners of options to acquire stock.
Each of these will now be discussed in order, followed by a discussion of reattribution.

Attribution Between Family Members

Section 318(a)(1) provides that an individual is considered as owning stock owned, directly or indirectly, by or for:
(1) his or her spouse, other than a spouse who is legally separated from the individual under a decree of divorce or separate maintenance;
(2) his or her children, grandchildren, and parents. For purposes of Section 318(a)(1), a legally adopted child of an individual is treated as a child of such individual.

It should be noted that although a grandparent is deemed

to own stock owned by his or her grandchildren, a grandchild is not deemed to own stock owned by his or her grandparents. Furthermore, an individual is not deemed to own stock owned by his or her brothers, sisters, aunts, uncles, and other relatives.

As an example of Section 318(a)(1), assume that H, an individual, his wife, W, his son, S, and his grandson (S' son), G, each own 25 outstanding shares of Corporation X. Then H, W, and S are each considered as owning 100 shares, the 25 shares they actually own and 75 shares from the other three individuals. G, however, is considered as owning only 50 shares because the shares of his grandparents, H and W, are not attributed to him.

Attribution from an Entity to a Beneficiary of That Entity

Attribution from Partnerships and Estates

Section 318(a)(2)(A) provides that stock owned, directly or indirectly, by a partnership or an estate is considered to be owned proportionately by its partners or beneficiaries.

For example, assume that A, an individual, owns a 50% interest in a partnership. Further, assume that this partnership owns 50 out of the 100 outstanding shares of Corporation X, the remaining 50 shares being owned by A. Then A is considered to be the owner of 75 shares, the 50 shares he or she actually owns plus 50% of the shares owned by the partnership.

Attribution from Trusts

Except for "grantor" trusts and "pension" trusts, stock owned, directly or indirectly, by a trust is considered as owned by its beneficiaries in proportion to the actuarial interest of the beneficiaries in the trust.

For example, assume that a testamentary trust owns 25 of the outstanding 100 shares of stock of Corporation X. Further, assume that A, an individual, holds a vested remainder in the trust having a value, computed actuarially, equal to 4% of the value of the trust property. Then A is considered as constructively owning 1 share (4% of 25).

In the case of a "grantor" trust (e.g., a revocable trust), stock owned, directly or indirectly, by the trust will be considered as being owned by the person who is treated as the owner of the trust pursuant to Sections 671–677.[5]

Attribution from Corporations

Section 318(a)(2)(C) provides that if 50% or more in value of the stock of a corporation is owned, directly or indirectly, by a person, such person is considered as owning the stock owned, directly or indirectly, by the corporation in that proportion which the value of the stock that such person so owns bears to the value of all of the stock of the corporation.

For example, assume A and B, unrelated individuals, own 70% and 30%, respectively, in the value of the stock of Corporation M. Assume further that Corporation M owns 50 of the 100 outstanding shares of stock of Corporation O, the remaining 50 shares being owned by A. Then A is considered as owning 85 shares, the 50 shares that he or she actually owns plus 70% of the 50 shares owned by Corporation M. Because B does not own 50% or more of Corporation M, he or she is deemed to own no shares in Corporation O.

Attribution to Entities

Attribution to Partnerships and Estates

Generally, Section 318(a)(3)(A) provides that stock owned by a partner or a beneficiary of an estate is considered as owned by the partnership or estate. It should be noted that all of the shares owned by a partner or a beneficiary of an estate are attributed to the partnership or estate, not just a partner's or beneficiary's pro rata share.

For example, assume that A, an individual, has a 50% interest in a partnership. Further, assume that A owns 50 shares of stock in Corporation X. Then the partnership is considered as owning the 50 shares of Corporation X stock owned by A.

Attribution to Trusts

Except in the case of "grantor" trusts and "pension" trusts, stock that is owned, directly or indirectly, by a beneficiary of a trust is considered as owned by the trust, unless the beneficiary's interest is considered a "remote contingent interest." A "contingent interest" of a beneficiary is considered "remote" if, under the maximum exercise of discretion by the trustee in favor of the beneficiary, the value of such interest, computed actuarially, is 5% or less of the value of the trust property.

For example, assume that A, an individual, holds a vested remainder in a testamentary trust having a value, computed actuarially, of 4% of the value of the trust property. Furthermore, assume that A owns 75 shares of stock in Corporation X. Then the testamentary trust is considered to own A's 75 shares of Corporation X because A's interest is vested and not contingent.

In the case of a "grantor" trust, all stock owned by the grantor is deemed to be owned by the trust.

Attribution to Corporations

If 50% or more in value of the stock in a corporation is owned, directly or indirectly, by any person, then the corporation is deemed to own the stock owned by that person. Thus, as in the case of attribution from partners and beneficiaries to partnerships and estates, respectively, the corporation is deemed to own all of the stock owned by the shareholder, not just a proportionate amount.

For example, assume that A owns 70% of the value of Corporation M. Assume that Corporation M and A each own 50 of the 100 outstanding shares of stock of Corporation O. Then Corporation M is considered as owning 100 shares of Corporation O, the 50 shares that it actually owns and the 50 shares that A owns.

Attribution as a Result of Options

Section 318(a)(4) provides that if any person has an option to acquire stock, such stock will be considered as owned by such person. For purposes of this Section, an option to acquire such an option and each one of a series of such options is considered as an option to acquire the stock.

For example, assume that A and B, unrelated individuals, own all of the 100 outstanding shares of Corporation X, each owning 50 shares. Assume further that A has an option to acquire 25 of B's shares and has an option to acquire a further option to acquire the remaining 25 of B's shares. Then A

is considered as owning the 100 shares of stock of Corporation X.

Rules of Reattribution

Section 318(a)(5) provides certain rules for "reattribution." Except as provided below, any stock that is deemed to be owned by a person as a result of one of the four attribution rules set forth above is considered to be actually owned by such person for purposes of reapplying the above four attribution rules.

For example, if a trust owns 50% of the stock of Corporation X, stock of Corporation Y owned by Corporation X is attributed to the trust and may then be further attributed to the beneficiaries of the trust.

There are, however, two major exceptions to this rule. First, stock constructively owned by an individual as a result of the application of the family attribution rules of Section 318(a)(1) is not treated as owned by such individual for the purpose of again applying the family application rules in order to make another individual the constructive owner of such stock. For example, assume H, an individual, owns 50 shares of stock in Corporation X. Further, assume that H's son, S, and daughter, D, each owns 50 shares in Corporation X. Then S is deemed to own only 100 shares in Corporation X, the 50 shares he actually owns plus the 50 shares actually owned by his father, H. He is not deemed to own the stock owned by his sister, D.

There is a further exception to the exception for reattribution of family ownership that permits reattribution, in effect, among family members. Section 318(a)(5)(D) provides that, when an individual is deemed to own stock by either applying the family attribution rules or the option rules, then he or she is considered to own the stock by reason of the option rules. In other words, the option rules take precedence over the family attribution rules.

Finally, Section 318(a)(5)(C) provides that stock constructively owned by a partnership, estate, trust, or corporation by reason of the attribution rules set forth in Section 318(a)(3) will not be considered as owned by the entity for purposes of applying the attribution rules set forth in Section 318(a)(2) in order to make another the constructive owner of such stock. For example, if two unrelated individuals are beneficiaries of the same trust, stock owned by one of the beneficiaries that is attributed to the trust under Section 318(a)(3) will not be reattributed from the trust to the other beneficiary. It should be noted, however, that stock constructively owned by reason of Section 318(a)(2) may be reattributed under Section 318(a)(3). Thus, for example, if all of the stock of Corporations X and Y is owned by A, stock of Corporation Z held by Corporation X is attributed to Corporation Y through A.

The Section 318(a) Rules in Prolog

This part of the paper will discuss the Prolog implementation of the rules set forth in Section 318(a) and the Treasury Regulations thereunder.[6] The complete listing of the program is contained in Listing One (page 75). In Listing Two (page 79), several of the examples set forth in the Treasury Regulations under Section 318(a) are solved using **TA**.

The Interface Program—section318

The Prolog procedure **section318** *controls the general* interaction between the computer program and the user. After displaying an introductory message to the user, **TA** asks the user for the name of the client. This will be used throughout the program in communicating various information to, and asking various questions of, the user. For purposes of the remainder of this paper, the client's name will be **Client**.

TA then calls the procedure **type(Client)**. This procedure simply asks whether **Client** is an individual, partnership, estate, trust, grantor trust, or corporation. Once the user has told the computer what type of entity **Client** is, **type** then adds this information to the computer's data base. This information is later used to restrict the amount of searching that the computer must do to reach its conclusion. For example, as discussed below, if **Client** is not an individual, the program will not ask for the family members of **Client** and will automatically set the number of shares attributed to **Client** as a result of the family attribution rules set forth in Section 318(a)(1) to zero.

TA then asks the user for the name of the corporation whose stock may be attributed to **Client** under Section 318(a). For purposes of the remainder of this paper, the corporation in question will be called **Corp**.

After this is done, **TA** will attempt to find the number of shares of **Corp** stock that **Client** actually owns. The program does this by calling the procedure **num_a_own(Client, Corp, N_actual)**. This procedure will be discussed below.

After **TA** knows the number of shares of **Corp** stock that **Client** actually owns, it then calls the procedure **num_con_own(Client, Corp, N_con)**, which determines the number (**N_con**) of shares of stock of **Corp** that the **Client** constructively owns as a result of the rules set forth in Section 318(a). This procedure is the major procedure of **TA** and will be discussed in detail below.

Finally, **section318** prints out the number of shares of **Corp** stock that **Client** actually and constructively, by application of Section 318(a), owns. After this is done, **TA** asks the user if he or she wants to run the program again for a new client. If the answer is yes, the program will start over. If the user answers no, the program returns the user to the Prolog interpreter and awaits new commands.

The Prolog Procedure for Actual Ownership—num_a_own

The Prolog procedure **num_a_own** determines the number of shares of **Corp** stock that **Client** *actually* owns. The procedure first knows that a client will not actually own any stock in himself. Thus, the first clause of the procedure is simply:

```
num_a_own(Client, Client, 0) :- !.
```

If the client is not identical to the corporation, **num_a_own(Client, Corp, N_actual)** then asks the user for the number of shares of stock that **Client** actually owns in **Corp**. The program expects the user to enter an integer, and if he or she does not do so, it prints an error message and again asks the user for the number of shares of stock that **Client** actually owns.

Once **TA** knows how many shares of stock the client actually owns, it adds this fact to its data base.[7]

The Prolog Procedure for the Constructive Ownership Rule Set *Forth in* Section 318(a)—num_con_own

The procedure **num_con_own(Client, Corp, Number)** is defined as:

```
num_con_own(Client, Corp, Number) :-
   num_family(Client, Corp, N1),
   num_from(Client, Corp, N2, zzzz),
   num_to(Client, Corp, N3),
   num_options(Client, Corp, N4), !,
   Number is N1 + N2 + N3 + N4.
```

The procedure simply calls four other procedures, **num_family, num_from, num_to,** and **num_options.** As discussed below, these four procedures find the number of shares of stock that **Client** constructively owns in **Corp** by the application of the rules of Sections 318(a)(1), (2), (3), and (4), respectively.

num_con_own then simply adds the number of shares that **Client** constructively owns as a result of each of the four constructive ownership rules to find the total number of shares that **Client** constructively owns in **Corp** as a result of Section 318(a).[8]

The Rule of Section 318(a)(1)—num_family

The procedure **num_family(Client, Corp, Number, zzzz)** determines the **Number** of shares that **Client** owns in **Corp** as a result of the family attribution rules set forth in Section 318(a)(1). First, the procedure checks to determine whether **Client** is an individual. If **Client** is not an individual, the procedure quickly determines that the number of shares of stock that **Client** owns as a result of the family attribution rules is zero. In this regard, if the user has told **TA** that **Client** is a partnership, estate, trust, grantor trust, or corporation, then the computer knows that the **Client** is not an individual.

If **Client** is an individual, **num_family** then asks the user for the names of **Client's** spouse, children, grandchildren, and parents. This is done by the procedure **family_of(Client, List)** where **List** is a list of all family members of **Client**.[9]

This procedure for determining the family members of **Client** is presently very crude. In particular, a more extensive program would contain the rule that a child includes a legally adopted child of an individual. Furthermore, a more sophisticated program would contain rules for determining the various family relationships between individuals. For example, it could contain rules such that if the program knew that John is the son of Dean then the computer could automatically infer that Dean is the parent of John. These rules have not been included in the present program due to memory constraints.

Once **TA** has determined the family members of **Client**, it then calls the procedure **family_own(Client, Corp, List, M, N)**. This procedure recursively goes down the **List** of family members and determines the number of shares of stock of **Corp** that each family member actually or constructively, as a result of Section 318(a), owns. In determining the number of shares that a family member constructively owns, the program uses only the rules of Section 318(a)(2) and 318(a)(4). In other words, the computer knows that pursuant to Section 318(a)(5)(B), stock constructively owned by an individual as a result of the family attribution rules may not be considered as actually owned by the individual for purposes of again applying the family attribution rules. Furthermore, **TA** knows that the attribution rules of Section 318(a)(3) cannot apply, because those rules attribute stock ownership *to* an entity such as an estate, trust, or corporation. As a result, **family_own** simply calls **num_from** (the Section 318(a)(2) rule) and **num_options** (the Section 318(a)(4) rule).

Once **TA** determines the number of shares that each family member actually and constructively owns in **Corp**, **num_family** returns the total number of such shares.

The Rule of Section 318(a)(2)—num_from

The Prolog procedure **num_from(Client, Corp, Number, OldClient)** determines the **Number** of shares that **Client** owns of **Corp** as a result of the "from" attribution rules set forth in Section 318(a)(2). The procedure first calls **entity_of. entity_of** determines the names of all partnerships, estates, trusts, grantor trusts, and corporations in which **Client** has an interest and which may actually or constructively own stock of **Corp**. For example, assume that John is the **Client** and Corporation X is the **Corp**. Then **TA** will ask questions such as:

Is John a 50 percent or more shareholder in a corporation that owns, actually or constructively, stock in Corporation_X?

If the user answers yes (by typing **y**) to such a question, the computer will then ask the user for the name of the entity and automatically add the type of the entity to the computer's data base. For example, if the user states that Corporation Y may actually or constructively own stock in Corporation X in response to the above question, the computer will add the fact **corporation(Corporation_Y)** to its data base. Finally, **entity_of** asks the user what percentage interest **Client** has in the entity in question. This is necessary because the rules set forth in Section 318(a)(2) state that stock owned by an entity is considered to be proportionately owned by its beneficiaries.

It should be noted that **TA** currently does not automatically determine whether a client owns a 50% or more ownership interest in a given corporation. It simply asks the user whether there is such a corporation. In particular, the 50% ownership requirement must be determined by applying the Section 318(a) attribution rules. This rule presently is not in the system.

After **TA** determines the identity and percentage ownership interest of **Client** in entities that may actually or constructively own stock in **Corp**, it then calls the procedure **entity_own(Client, Corp, List, M, N, OldClient)**, which determines the number of shares that each such entity in **List** actually or constructively owns in **Corp**. Once this is done, **entity_own** determines **Client's** proportional interest in such ownership. In other words, **entity_own** first calls **nu-**

m_a_own(H, Corp, N1) to determine the number of shares, **N1**, which the entity, **H**, actually owns in **Corp**. For example, if **Client** owns a 50% interest in a partnership, **PS**, and **PS** owns 100 shares of the stock in **Corp**, **entity_own** will determine that **Client** constructively owns 50 shares of stock in **Corp** as a result of Section 318(a)(2)(A).

For each entity in which **Client** has an interest and which may actually or constructively own stock of **Corp**, **entity_own** will then use the attribution rules of Section 318(a)(2) and 318(a)(4) to determine the number of shares of stock that the entity constructively owns. These rules account for the reattribution rules set forth in Section 318(a)(5)(A). **entity_own**, however, does not include the constructive ownership rule set forth in Section 318(a)(3) because Section 318(a)(5)(C) provides that stock constructively owned by an entity by reason of the applicaton of the "to" attribution rules set forth in Section 318(a)(3) will not be considered as owned by the entity for purposes of applying the "from" attribution rules of Section 318(a)(2).

The problem of reattribution that is inherent in Section 318(a) has been easily handled in the Prolog rules because Prolog permits a recursive definition of procedures. Thus, the procedure **num_from** was able to call itself as applied to a different entity and, as a result, accounts for the reattribution rules of Section 318(a)(5)(A).

The Rule of Section 318(a)(3)—num_to

The procedure **num_to(Client, Corp, Number)** determines the **Number** of shares that **Client** constructively owns of **Corp** as a result of the "to" attribution rules set forth in Section 318(a)(3). Because these rules apply only if **Client** is not an individual, **num_to** returns zero if **Client** is an individual. Furthermore, if **Client** and **Corp** are the same, **num_to** again returns zero.

If **Client** is not an individual and is not the same as **Corp**, **num_to** then forms a list of all those beneficiaries of **Client** who may own stock, actually or constructively, in **Corp**. This procedure is similar to the procedure **entity_of** discussed above. The only difference is that if the procedure determines that the entity is a trust, it then asks the user whether the beneficiary's interest in the trust is a "remote contingent interest." If the user answers yes, the beneficiary is not included in the list of beneficiaries who may own stock in **Corp**.[10]

Once **TA** has determined the beneficiaries of **Client** who may actually, or constructively, own stock of **Corp**, **num_to** then determines the number of shares of stock that each such person actually and constructively, as a result of Sections 318(a)(1)-(4), owns of **Corp**. In other words, **num_to** calls a procedure **to_entity_own**, which then calls **num_a_own, num_family, num_from, num_to,** and **num_options**. This accounts for the reattribution rules set forth in Section 318(a)(5)(A).

The Rule of Section 318(a)(4)—num_options

Finally, the procedure **num_options(Client, Corp, Number)** handles the "options" attribution rules set forth in Section 318(a)(4). This procedure simply asks the user whether **Client** has an option, or an option to acquire an option, to acquire stock of **Corp**. If the user answers yes, **num_options** then asks the user how many shares of **Corp** stock **Client** has an option on.

Examples

Listing Two contains several computer runs for various factual situations. These examples have been taken from the examples set forth in the Treasury Regulations under Section 318(a). The user input has been underlined for clarity.

Listing Three (page 84) contains an example of a run in which the procedures **num_a_own, num_con_own, num_family, num_from, num_to,** and **num_options** have been "traced." This permits the reader to see how Prolog recursively applies the various procedures in the case of a reattribution problem.

Simple Explanations

Although the Section 318(a) program correctly determines the number of shares of stock of a given corporation that a person constructively owns as a result of the Section 318(a) attribution rules, most attorneys using the program will also want to know **how** the computer reached its conclusion. For example, if a senior partner asks an associate to determine the number of shares of stock of a given corporation that an individual owns as a result of the Section 318(a) attribution rules and the associate simply gives the number, the senior partner would undoubtedly ask the associate to prepare a memo outlining the reasoning he or she used to arrive at the number of shares. Furthermore, if the Section 318(a) program were used to prepare a Tax Court petition, the attorney would need to present a well-reasoned opinion supporting his or her position. This could be done only by explaining how the Section 318(a) attribution rules applied to the facts in question. Finally, if the attorney simply relied upon the numeric output from the program and did not review the computer's reasoning, the attorney could leave himself or herself open to a malpractice suit. Thus, any useful "knowledge-based" system in the legal field must permit the user to ask the computer to explain **how** it reached its conclusion.

The issue of how to have the computer generate useful explanations is quite complex. For example, explanations by human experts are usually tailored to their audiences. In particular, an associate would use a different (and more technical) explanation of how he or she arrived at the number of shares of stock that an individual owns as a result of the Section 318(a) attribution rules if he or she were writing a memo to a senior tax partner as opposed to explaining the problem to a client. Furthermore, in most situations, the high level domain rules that are incorporated into the computer program may not adequately contain their justifications. Although these problems present a number of interesting questions, which should be addressed by future research, they will not be considered in this paper.[11]

As will be illustrated below, most of the above complexities that arise in generating useful explanations have been avoided in **TA** because the rules set forth in Section 318(a) are justifications for themselves. In other words, the tax rules set forth in the Internal Revenue Code generally need no further justification other than the fact that Congress enacted such statutory provisions. Furthermore, the complex

problem of dealing with the interpretation of case law is avoided because there are very few cases under Section 318(a).

The remainder of this part of the paper will discuss how the Prolog program in Listing Four (page 87) implements a simple explanation facility.[12]

Answering How the Computer Reached Its Conclusion

In order to understand how **TA** explains **how** it arrives at a given conclusion, consider the following fact situation. Assume that **TA** is asked how many shares of stock of Corporation X the client, Dean, owns as a result of the family attribution rules. Furthermore, assume that Dean has only one family member, his spouse, Mary. Finally, assume that Mary actually owns 1,000 shares of Corporation X stock. Based on these facts, **TA** will explain its reasoning as follows:

START OF EXPLANATION

The number of shares of Corporation_X stock constructively owned by Dean as a result of Section 318(a)(1) is 1,000, determined as follows:
An individual is deemed to own the stock owned by his family. Section 318(a)(1).

FACT: Dean is an individual.
A spouse is a family member. Section 318(a)(1)(A)(i).
FACT: Mary is the spouse of Dean.
FACT: Mary actually owns 1,000 shares.
FACT: Dean has no more family members.

END OF EXPLANATION

As can be seen, the explanation generated by **TA** applies the rules set forth in Section 318(a)(1) to the facts in question.

In order to explain its reasoning, each rule of **TA** contains an extra argument, **H**, which contains a list of descriptions of the Prolog rules that were successful when the program finally reached its conclusion.[13] For example, when **num_a_own** determines how many shares of Corporation X stock Mary actually owns, it adds the description:

FACT: Mary actually owns 1,000 shares.

to the list of descriptions.

Furthermore, whenever a Prolog rule is successful, the rule adds a description of itself as the first element in the list for generating the explanation. For example, the predicate **spouse_own** adds the description found by **get_rule (2,Z)** to the head of the list once it has determined both that Dean has a spouse and the number of shares of stock the spouse actually owns. In particular, **get_rule(2,Z)** simply sets the first element of the list to the description:

A spouse is a family member. Section 318(a)(1)(A)(i).

Once **TA** has determined how many shares of stock Dean constructively owns as a result of the Section 318(a)(1) family attribution rules, it asks the user if he or she wants an explanation of its conclusion. If the user answers yes, **TA** calls the predicate **how**, which simply prints each element of the list **H** containing the descriptions. (See Listing Four).

Answering Why a Given Question Is Asked

In the full Section 318(a) program in Listing One, **TA** asks the user for only those facts required to reach its conclusion. To make the program more useful, the user should be able to ask the computer **why** it asks a given question. For example, when the computer asks the user:

Is Dean an individual?

the user should be able to ask why the computer needs to know this information.

In the prototype program in Listing Four, the computer will answer **why** by returning:

An individual is deemed to own the stock owned by his family. Section 318(a)(1).

In other words, **TA** is telling the user that it needs to know if Dean is an individual in order to determine whether it should apply the family attribution rules of Section 318(a)(1). If the user is still not satisfied with this explanation, the user can again type **why** in response to the question: **Is Dean an individual?** This time, the program will return:

This is what the Internal Revenue Code provides!

In other words, in the simple Section 318(a) program, the ultimate explanation of any rule is that the Internal Revenue Code contains the rule.

In order to implement the **why** feature, an additional argument, **W**, is added to each rule. This argument contains a list of all rules that have been used by the program up to the point at which the computer asks for further information. In the above example, the program has just entered the rule **num_family.**

TA first attempts to satisfy this rule by determining whether **Client** is an individual. As a result, it asks the question:

Is Dean an individual?

At this point in the program, the argument **W** contains a list with one element, the description:

An individual is deemed to own the stock owned by his family. Section 318(a)(1).

Thus, when the user types **why** in response to the question, Prolog simply returns the first element of the list, **W**, and then asks the question again. If the user again asks **why**, the program will print the next element of the list and again ask the question. When the list contains no further elements, the program will simply explain that:

This is what the Internal Revenue Code provides!

See Listing Five (page 90) for further examples of explanations.

Providing Help When the Computer Asks Questions

If the attorney using the program is not familiar with the given area of tax law, he or she may need **help** in answering some questions that the computer asks. For example, in order to keep the questions short, **TA** simply asks the question:

Does Dean have a spouse?

The user may need some assistance in determining how the word **spouse** is defined. The present version of **TA** permits the user to type **help** in response to the computer's request for information. For example, if the user types **help** in response to the question:

Does Dean have a spouse?

the computer will answer:

A spouse does not include one who is legally separated from Dean under a decree of divorce or separate maintenance. Section 318(a)(1)(A)(i).

Thus, **TA** gives the user some assistance in determining how the word **spouse** is defined for purposes of Section 318(a). The present system, however, does not further define the terms "legally separated," "decree of divorce," or "separate maintenance." For example, **TA** would not assist the user in determining whether an interlocutory decree of divorce under California law constitutes a "decree of divorce" for purposes of Section 318(a). It appears, however, that it would be possible to build further **help** facilities into the program if additional memory were available.

Finally, it should be recognized that this **help** facility may be insufficient in an actual system designed to assist attorneys because it requires the attorney to know when to ask for help. In particular, the attorney might simply assume that the word **spouse** for purposes of Section 318(a) includes any person who is married to the client even if legally separated from him or her. Thus, unless the user knew to ask for help, he or she might incorrectly answer the question.

Conclusion

This paper has discussed two Prolog programs that implement the rules contained in Section 318(a). Unlike other computer programs that have been developed in various legal fields, the major purpose of developing **TA** has been to illustrate that Prolog may be used to implement legal reasoning systems in a fairly straightforward manner. Furthermore, this paper has attempted to show that Prolog may also be used to implement simple explanation facilities for knowl-

edge-based programs.

TA achieves its degree of success, however, mostly because the tax rules of Section 318(a) are very well defined and not subject to interpretation by a large number of cases. It is presently unclear how the rules set forth in case law could be incorporated into the present framework in an area where case law is at least as important as the statutory provisions.

The prototype programs also illustrate that it is possible to design interfaces to legal reasoning programs that permit a user who is not a computer expert to use the program. In particular, more effort has gone into devising a suitable user interface than into the more complex issues of representing the rules and knowledge required to implement the Section 318(a) programs. In narrow, well-defined areas of the law, however, this method may be acceptable. If it is, programs that are useful to attorneys may be able to be developed much more quickly than currently appears possible.

Finally, it should be recognized that the explanation facilities described in this paper are extremely simple. Because the best explanation of Section 318(a) is in terms of the rules of that Section, however, the explanation facility of **TA** is probably adequate. In other situations, however, such simple explanations would clearly be inadequate.

Notes

[1] Unless otherwise stated, all Section references are to the Internal Revenue Code of 1954, as amended.

[2] See for example: L. T. McCarty, "Reflections on TAXMAN: An Experiment in Artificial Intelligence and Legal Reasoning," *Harvard Law Review* (1977) 90, 837-93; A. Hustler, Programming Law in Logic, Department of Computer Science, University of Waterloo, Research Report CS-82-13 (1982).

[3] CP/M-86 is a trademark of Digital Research, Inc.

[4] Prolog-86 is a trademark of MICRO-AI.

[5] It is unclear, however, whether the shares of stock owned by a grantor trust will also be attributed to the beneficiaries of the trust pursuant to Section 318(a)(2)(B)(i) discussed above. **TA** implicitly contains the rule that shares of stock owned by a grantor trust *are* attributed to the beneficiaries pursuant to Section 318(a)(2)(B)(i).

[6] The programs discussed in this paper are written in Prolog-86. Prolog-86 is based on the University of New South Wales Prolog, which runs under Unix (Unix is a trademark of Bell Telephone Laboratories) and is similar in many ways to the original DECSystem-10 Prolog written by L. M. Pereira, F. C. N. Pereira, and D. H. D. Warren. Prolog-86 currently runs under CP/M-86, and MSDOS and PCDOS. Prolog-86 is available through Solution Systems, 335-D Washington Street, Norwell, MA 02061. Those readers who are interested in learning more about Prolog should read the book *Programming In Prolog* by W. F. Clocksin and C. S. Mellish and published by Springer-Verlag.

[7] As a result of restrictions on numbers in Prolog-86, the implementation language of **TA**, fractional shares are not allowed. Furthermore, the number of shares must be less than approximately 30,000. These restrictions could be lifted if the Prolog interpreter permitted real numbers (i.e., fractions) and "long" integers.

[8] In Prolog, the operator **is** causes arithmetic evaluation. Thus, **N is N1 + N2 + N3 + N4** sets **N** to the sum of **N1, N2, N3,** and **N4** if **N** is previously uninstantiated.

[9] A list is an ordered sequence of elements of any length. In Prolog, a list is represented as: **[a, b, c]. a, b,** and **c** are the elements of the list. The null list is represented as **[ ]**.

[10] Section 318(a)(3)(B)(i). In this regard, **TA** does not assist the user in determining whether a beneficiary's interest is a "remote contingent interest" as opposed to a "vested" interest or a contingent interest that is not remote. In a larger system, such rules must be included because it is unlikely that a user of the system would know how Section 318(a)(3)(B)(i) defines a remote contingent interest.

[11] See for example: J. A. Goguen, J. L. Weiner, and C. Linde, "Reasoning and Natural Explanation," *Int. J. Man-Machine Studies* (1983) 19, 521–559; William R. Swartout, "XPLAIN: A System for Creating and Explaining Expert Consulting Programs," *Artificial Intelligence* (1983) 21, 285–325; and William J. Clancey, "The Epistemology of a Rule-Based Expert System—a Framework for Explanation," *Artificial Intelligence* (1983) 20, 215–251.

[12] Due to memory constraints, the Prolog program described in Listing Four only implements the family attribution rules of Section 318(a)(1). If the Prolog interpreter were able to access more than 128K of RAM, it would be fairly easy to include the simple explanation system in the full Section 318(a) program listed in Listing One.

[13] The design of the explanation facilities set forth in this paper is based on the article by K. L. Clark and F. G. McCabe with an appendix by P. Hammond, "PROLOG: A Language for Implementing Expert Systems," *Machine Intelligence* (1982) 10, 455–475.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 194.**

Listing One

```
%  THE COMPLETE SECTION 318(a) PROLOG PROGRAM
%  By Dean A. Schlobohm
%  © Copyright 1984 by Dean A. Schlobohm.  All Rights Reserved.
%    Version 1.0
%  Date:  6/11/84
%  Bugs:
%    1.  Does not include the rule that stock cannot count more
%    than once.
%    2.  When it asks for partners, etc. that own stock, you must
%    enter all partners who actually or constructively may own
%    stock.
%    3.  Does not handle "How", "Help" and "Why"
%  To Use:  type "prolog sec318.pro<RET>" then after Prolog is
%    loaded type "section318!<RET>"


% preliminary functions

append([],L,L) :- !.
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

appendall([],[]).
appendall([H],H).
appendall([H1,H2],L) :- append(H1,H2,L).
appendall([H1,H2|T],L) :-
     append(H1,H2,X),
     appendall(T,Z),!,
     append(X,Z,L).

member(X,[X|_]) :- !.
member(X,[_|Y]) :- member (X,Y).

add(X) :- X,!.
add(X) :- asserta(X),!.


%  The Interface Program.

section318 :-
     prompt(Old,' '),
     nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,
     print('     THIS PROGRAM WILL DETERMINE THE NUMBER OF SHARES
     OF'),
     print('STOCK A CLIENT ACTUALLY AND CONSTRUCTIVELY OWNS AS A
     RESULT'),
     print('OF SECTION 318(a) OF THE INTERNAL REVENUE CODE.'),
     nl,nl,
     ask('     Enter the name of the Client.  (One word)  '
     ,Client),nl,
     type(Client),
     ask('     Enter the name of the Corporation.  (One word)  '
     ,Corp),nl,
     add(corporation(Corp)),!,
     num_a_own(Client,Corp,N_actual),
     num_con_own(Client,Corp,N_con),!,nl,nl,
     print('     The number of shares of ',Corp,' stock actually
     owned by'),
     print(Client,' is ',N_actual,'.'),nl,nl,
     print('     The number of shares of ',Corp,' stock
     constructively owned'),
     print('by ',Client,' is ',N_con,'.'),nl,nl,!,
     ask('     Do you want to try another Client?  ',X),!,
     ((X = y) ; (X = 'Y')),!,nl,nl,
     prompt(_,Old),
     section318.

individual(Client) :- ind(Client).
individual(Client) :-
     (partnership(Client) ;
      estate(Client) ;
      trust(Client) ;
      grantor_trust(Client) ;
      corporation(Client)),
      !,fail.
```

(Continued on next page)

Listing One

```
individual(Client) :-
     type(Client),!,
     individual(Client).

type(Client) :-
     (ind(Client) ;
      partnership(Client);
      estate(Client) ;
      trust(Client) ;
      grantor_trust(Client) ;
      corporation(Client)),!.
type(Client) :-
     print('     Is ',Client,' an Individual, Partnership, Estate,
     Trust, '),
     prin('Grantor trust, or Corporation?  (enter i, p, e, t, g,
     or c) '),
     ratom(X),nl,
     assert_type(X,Client),!.

assert_type(i,C) :- add(ind(C)),!.
assert_type(p,C) :- add(partnership(C)),!.
assert_type(e,C) :- add(estate(C)),!.
assert_type(t,C) :- add(trust(C)),!.
assert_type(g,C) :- add(grantor_trust(C)),!.
assert_type(c,C) :- add(corporation(C)),!.

%  the rule for actual ownership

num_a_own(Client,Client,0) :- !.
num_a_own(Client,Corp,N) :-
     print('     How many shares of ',Corp,' stock does ',Client),
     prin('actually own? '),
     ratom(X),nl,
     integer(X),
     N = X,!.
num_a_own(Client,Corp,N) :-
     print('     Error -- You must enter an integer!'),nl,!,
     num_a_own(Client,Corp,N).

%  the main rule of Section 318(a)

num_con_own(Client,Corp,Number) :-
     num_family(Client,Corp,N1),       % Section 318(a)(1)
     num_from(Client,Corp,N2,zzzz),    % Section 318(a)(2)
     num_to(Client,Corp,N3),          % Section 318(a)(3)
     num_options(Client,Corp,N4),!,       % Section 318(a)(4)
     Number is N1 + N2 + N3 + N4.

%  the rule of Section 318(a)(1)

num_family(Client,Corp,Number) :-
     individual(Client),!,
     family_of(Client,List_of_family),
     family_own(Client,Corp,List_of_family,0,Number).
num_family(Client,Corp,0).

%  definition of family_of

family_of(C,L) :-
     nl,
     prin('     Does ',C,' have a spouse, child, grandchild, or
     parent? '),
     ratom(X),nl,
     ((X = y) ; (X = 'Y')),!,
     ask('     What is the name of the first family member?  '
     ,Y),nl,!,
     add(ind(Y)),
     family_of2(C,L2),!,
     append([Y],L2,L).
family_of(C,[]).

family_of2(C,L) :-
     nl,
     prin('     Does ',C,' have any more family members? '),
     ratom(X),nl,
```

```
     ((X = y) ; (X = 'Y')),!,
     ask('     What is the name of the next family member?
     ',Y),nl,!,
     add(ind(Y)),
     family_of2(C,L2),!,
     append([Y],L2,L).
family_of2(C,[]).

%  definition of family_own
%  note Section 318(a)(5)(B) is included since family_own does
%     not include num_family.

family_own(Client,Corp,[],N,N):- !.
family_own(Client,Corp,[P|T],M,N) :-
     P /= Client,!,
     num_a_own(P,Corp,N1),
     num_from(P,Corp,N2,zzzz),
     option_lieu_family(Client,Corp,P,N3), % Section 318(a)(5)(D)
     N4 is N1 + N2 - N3,
     num_options(P,Corp,N5),
     N6 is M + N4 + N5,
     !,
     family_own(Client,Corp,T,N6,N),!.
     family_own(Client,Corp,[P|T],M,N) :-
     family_own(Client,T,M,N).

option_lieu_family(Client,Corp,P,N) :-
     print('     Does ',Client,' own an option (or an option to
     acquire'),
     print('an option) to acquire the stock in ',Corp,' which
     ',P),
     prin('actually owns? '),
     ratom(X),nl,
     ((X = y) ; (X = 'Y')),!,
     print('     How many shares of ',Corp,' stock does ',Client),
     prin('have an option on? '),
     ratom(N),nl,nl,!.
```

(Continued on next page)

Listing One

```
option_lieu_family(Client,Corp,P,0) :- !.

%  the rule of section 318(a)(2)
%  entity-of returns a list ((name percent) (name2 percent2)...)

num_from(Client,Corp,0,OldClient) :-
     nl,
     print('     Do you want to consider applying the rules of'),
     prin('Section 318(a)(2) to ',Client,'? (enter y or n) '),
     ratom(X),nl,
     ((X = n) ; (X = 'N')),!.
num_from(Client,Corp,N,OldClient) :-
     ps(P),
     entity_of(Client,Corp,L1,partnership,P),
     est(E),
     entity_of(Client,Corp,L2,estate,E),
     tr(T),
     entity_of(Client,Corp,L3,trust,T),
     gt(T1),
     entity_of(Client,Corp,L4,grantor_trust,T1),
     corp(C),
     entity_of(Client,Corp,L5,corporation,C),
     appendall([L1,L2,L3,L4,L5],L),
     entity_own(Client,Corp,L,0,N,OldClient).

entity_of(Client,Corp,L,Entity,Prompt) :-
     nl,
     print('     Is ',Client,Prompt),
     prin('which owns, actually or constructively, stock in  '
     ,Corp,'? '),
     ratom(X),nl,
     ((X = y) ; (X = 'Y')),!,
     prin('     What is the name of the ',Entity,'? '),
     ratom(Y),nl,
     add(Entity(Y)),
     prin('     What percentage interest does ',Client,' own? '),
     ratom(Z),nl,
     append([Y],[Z],P1),!,
     entity_of(Client,Corp,L2,Entity,Prompt),!,
     append([P1],L2,L).
entity_of(Client,Corp,[],Entity,Prompt).

ps(' a partner in a partnership').
est(' a beneficiary of an estate').
tr(' a beneficiary of a trust (not described in Section 401(a))').
gt(' the grantor of a grantor trust').
corp(' a 50 percent or more shareholder in a corporation').

% note Section 318(a)(5)(C) is included since entity_own does
%   not include num_to

entity_own(Client,Corp,[],N,N,_) :- !.
entity_own(Client,Corp,[[H,P]|T],N,M,OldClient) :-
     H /= Client,
     H /= OldClient,
     num_a_own(H,Corp,N1),
     num_from(H,Corp,N2,Client),
     num_options(H,Corp,N3),
     N4 is (N1 + N2 + N3) * P / 100,    % this creates an error if
                                        % N1+N2+N3 > 600
     N5 is N4 + N,
     !,
     entity_own(Client,Corp,T,N5,M,OldClient).
entity_own(Client,Corp,[H|T],N,M,OldClient) :-
     entity_own(Client,Corp,T,N,M,OldClient).

%  the rule of Section 318(a)(3)
%  ben_of_entity returns a list (name name2 ...)

num_to(Client,Corp,0) :-
     not(individual(Client)),
     Client /= Corp,
     print('     Do you want to consider applying the rules of'),
     prin('Section 318(a)(3) to ',Client,'? (enter y or n) '),
```

```
     ratom(X),nl,
     ((X = n) ; (X = 'N')),!.
num_to(Client,Corp,N) :-
     not(individual(Client)),
     Client /= Corp,!,
     ben_of_entity(Client,Corp,L1,partner,partnership),
     ben_of_entity(Client,Corp,L2,beneficiary,estate),
     ben_of_entity(Client,Corp,L3,beneficiary,trust),
     ben_of_entity(Client,Corp,L4,grantor,grantor_trust),
     ben_of_entity(Client,Corp,L5,shareholder,corporation),
     appendall([L1,L2,L3,L4,L5],L),
     to_entity_own(Client,Corp,L,0,N).
num_to(Client,Corp,0).

ben_of_entity(Client,Corp,L,Prompt,Entity) :-
     Entity(Client),
     print('     Is there a ',Prompt,' of ',Client,' who owns,
     actually'),
     prin('or constructively, stock in ',Corp,'? '),
     ratom(X),nl,
     ((X = y) ; (X = 'Y')),
     ((Entity = trust,
       ask('     Is the interest a remote contingent interest?  '
     ,X2),nl,
       ((X2 /= y) , (X2 /= 'Y'))) ;
       (Entity = corporation,
       ask('   Does the shareholder own 50% or more of the stock?
    '
     ,X3),
       nl,((X3 = y) ; (X3 = 'Y'))) ;
       (Entity /= trust , Entity /= corporation)),
     prin('     What is the name of the ',Prompt,'? '),
     ratom(Y),nl,nl,
     type(Y),!,
     ben_of_entity(Client,Corp,L2,Prompt,Entity),!,
     append([Y],L2,L).
ben_of_entity(Client,Corp,[],Prompt,Entity) :- !.

to_entity_own(Client,Corp,[],N,N) :- !.
to_entity_own(Client,Corp,[H|T],N,M) :-
     H /= Client,
     num_a_own(H,Corp,N1),
     num_family(H,Corp,N2),
     num_from(H,Corp,N3,Client),
     num_to(H,Corp,N4),
     num_options(H,Corp,N5),
     N6 is N + N1 + N2 + N3 + N4 + N5,
     !,
     to_entity_own(Client,Corp,T,N6,M).
to_entity_own(Client,Corp,[H|T],N,M) :-
     !,
     to_entity_own(Client,Corp,T,N,M).

%  the rule of Section 318(a)(4)

num_options(Client,Corp,N) :-
     nl,
     print('     Does ',Client,' own an option (or an option to
     acquire'),
     prin('an option) to acquire the stock in ',Corp,'? '),
     ratom(X),nl,
     ((X = y) ; (X = 'Y')),!,
     print('     How many shares of ',Corp,' stock does ',Client),
     prin('have an option on? '),
     ratom(Y),nl,
     integer(Y),!,
     num_options(Client,Corp,M),
     N is M + Y,!.
num_options(Client,Corp,0).
```

End Listing One

Listing Two

```
: %  EXAMPLE 1.  Reg. Section 1.318-2(b).
:
: section318!

     THIS PROGRAM WILL DETERMINE THE NUMBER OF SHARES OF STOCK A
CLIENT ACTUALLY AND CONSTRUCTIVELY OWNS AS A RESULT OF SECTION
318(a) OF THE INTERNAL REVENUE CODE.
```

(Continued on next page)

Listing Two

```
     Enter the name of the Client.  (One word)  H

     Is H an Individual, Partnership, Estate, Trust, Grantor
trust, or Corporation?  (enter i, p, e, t, g, or c)  i

     Enter the name of the Corporation.  (One word)  Corporation

     How many shares of Corporation stock does H actually own?  25

     Does H have a spouse, child, grandchild, or parent?  y

     What is the name of the first family member?  W

     Does H have any more family members?  y

     What is the name of the next family member?  GS

     Does H have any more family members?  n

     How many shares of Corporation stock does W actually own?  25

     Do you want to consider applying the rules of Section
318(a)(2) to W?  (enter y or n)  n

     Does H own an option (or an option to acquire an option) to
acquire the stock in Corporation which W actually owns?  n

     Does W own an option (or an option to acquire an option) to
acquire the stock in Corporation?  n

     How many shares of Corporation stock does S actually own?  25

     Do you want to consider applying the rules of Section
318(a)(2) to S?  (enter y or n)  n

     Does H own an option (or an option to acquire an option) to
acquire the stock in Corporation which S actually owns?  n

     Does S own an option (or an option to acquire an option) to
acquire the stock in Corporation?  n

     How many shares of Corporation stock does GS actually own?
25

     Do you want to consider applying the rules of Section
318(a)(2) to GS?  (enter y or n)  n

     Does H own an option (or an option to acquire an option) to
acquire the stock in Corporation which GS actually owns?  n

     Does GS own an option (or an option to acquire an option) to
acquire the stock in Corporation?  n

     Do you want to consider applying the rules of Section
318(a)(2) to H?  (enter y or n)  n

     Does H own an option (or an option to acquire an option) to
acquire the stock in Corporation?  n

     The number of shares of Corporation stock actually owned by H
is 25.

     The number of shares of Corporation stock constructively
owned by H is 75.

     Do you want to try another Client?  n


: %  EXAMPLE 2.  Reg. Section 1.318-2(c), Example(2).
:
     THIS PROGRAM WILL DETERMINE THE NUMBER OF SHARES OF STOCK A
CLIENT ACTUALLY AND CONSTRUCTIVELY OWNS AS A RESULT OF SECTION
318(a) OF THE INTERNAL REVENUE CODE.

     Enter the name of the Client.  (One word)  A
```

(Continued on page 82)

Listing Two

```
     Enter the name of the Corporation.  (One word)  Corporation

     How many shares of Corporation stock does A actually own?  75

     Does A have a spouse, child, grandchild, or parent.  n

     Do you want to consider applying the rules of Section
318(a)(2) to A?  (enter y or n)  y

     Is A a partner in a partnership which owns, actually or
constructively, stock in Corporation?  n

     Is A a beneficiary of an estate which owns, actually or
constructively, stock in Corporation?  n

     Is A a beneficiary of a trust (not described in Section
401(a)) which owns, actually or constructively, stock in
Corporation?  y

     What is the name of the trust?  Trust

     What percentage interest does A own?  4

     Is A a beneficiary of a trust (not described in Section
401(a)) which owns, actually or constructively, stock in
Corporation?  n

     Is A the grantor of a grantor trust which owns, actually or
constructively, stock in Corporation?  n

     Is A a 50 percent or more shareholder in a corporation which
owns, actually or constructively, stock in Corporation?  n

     How many share of Corporation stock does Trust actually own?
25

     Do you want to consider applying the rules of Section
318(a)(2) to Trust?  (enter y or n)  n

     Does Trust own an option (or an option to acquire an option)
to acquire the stock in Corporation?  n

     Does A own an option (or an option to acquire an option) to
acquire the stock in Corporation?  n

     The number of shares of Corporation stock actually owned by A
is 75.

     The number of shares of Corporation stock constructively
owned by A is 1.

     Do you want to try another Client?  n


: %  EXAMPLE 3.  Reg. Section 1.318-2(c), Example(4).
:
     THIS PROGRAM WILL DETERMINE THE NUMBER OF SHARES OF STOCK A
CLIENT ACTUALLY AND CONSTRUCTIVELY OWNS AS A RESULT OF SECTION
318(a) OF THE INTERNAL REVENUE CODE.

     Enter the name of the Client.  (One word)  A

     Enter the name of the Corporation (One word)  Corporation_O

     How many shares of Corporation_O stock does A actually own?
50

     Does A have a spouse, child, grandchild, or parent?  n

     Do you want to consider applying the rules of Section
318(a)(2) to A?  (enter y or n)  y

     Is A a partner in a partnership which owns, actually or
constructively, stock in Corporation_O?  n
```

```
     Is A a beneficiary of an estate which owns, actually or
constructively, stock in Corporation_O?  n

     Is A a beneficiary of a trust (not described in Section
401(a)) which owns, actually or constructively, stock in
Corporation_O?  n

     Is A the grantor of a grantor trust which owns, actually or
constructively, stock in Corporation_O?  n

     Is A a 50 percent or more shareholder in a corporation which
owns, actually or constructively, stock in Corporation_O?  y

     What is the name of the corporation?  Corporation M

     What percentage interest does A own?  70

     Is A a 50 percent or more shareholder in a corporation which
owns, actually or constructively, stock in Corporation_O?  n

     How many shares of Corporation_O stock does Corporation_M
actually own?  50

     Do you want to consider applying the rules of Section
318(a)(2) to Corporation_M?  (enter y or n)  n

     Does Corporation_M own an option (or an option to acquire an
option) to acquire the stock in Corporation_O?  n

     Does A own an option (or an option to acquire an option) to
acquire the stock in Corporation_O?  n

     The number of shares of Corporation_O stock actually owned by
A is 50.

     The number of shares of Corporation_O stock constructively
owned by A is 35.

     Do you want to try another Client?  y


     THIS PROGRAM WILL DETERMINE THE NUMBER OF SHARES OF STOCK A
CLIENT ACTUALLY AND CONSTRUCTIVELY OWNS AS A RESULT OF SECTION
318(a) OF THE INTERNAL REVENUE CODE.


     Enter the name of the Client.  (One word)  Corporation M

     Enter the name of the Corporation.  (One word)  Corporation O

     How many shares of Corporation_O stock does Corporation_M
actually own?  50

     Do you want to consider applying the rules of Section
318(a)(2) to Corporation_M?  (enter y or n)  n

     Do you want to consider applying the rules of Section
318(a)(3) to Corporation_M?  (enter y or n)  y

     Is there a shareholder of Corporation_M who owns, actually or
constructively, stock in Corporation_O?  y

     Does the shareholder own 50% or more of the stock?  y

     What is the name of the shareholder?  A

     Is there a shareholder of Corporation_M who owns, actually or
constructively, stock in Corporation_O?  n

     How many shares of Corporation_O stock does A actually own?
50

     Does A have a spouse, child, grandchild, or parent?  n

     Do you want to consider applying the rules of Section
318(a)(2) to A?  (enter y or n)  n

     Does A own an option (or an option to acquire an option) to
acquire the stock in Corporation_O?  n
```

(Continued on next page)

Listing Two

```
     Does Corporation_M own an option (or an option to acquire an
option) to acquire the stock in Corporation_O?  n

     The number of shares of Corporation_O stock actually owned by
Corporation_M is 50.

     The number of shares of Corporation_O stock constructively
owned by Corporation_M is 50.

     Do you want to try another Client?  n
```

End Listing Two

Listing Three

```
: %  An Example From Reg. Section 1.318-4(c)(1)
:
: trace[num a own,num con own,num family,num from]!
: trace[num to,num options]!
:
: section318!

     THIS PROGRAM WILL DETERMINE THE NUMBER OF SHARES OF STOCK A
CLIENT ACTUALLY AND CONSTRUCTIVELY OWNS AS A RESULT OF SECTION
318(a) OF THE INTERNAL REVENUE CODE.

     Enter the name of the Client.  (One word)  Corporation Y

     Is Corporation_Y an Individual, Partnership, Estate, Trust,
Grantor trust, or Corporation?  (enter i, p, e, t, g, or c)  c

     Enter the name of the Corporation.  (One word)  Corporation Z

C|>num_a_own('Corporation_Y', 'Corporation_Z', _6)
     How many shares of Corporation_Z stock does Corporation_Y
actually own?  0

E|<num_a_own('Corporation_Y', 'Corporation_Z', 0)
C|>num_con_own('Corporation_Y', 'Corporation_Z', _4)
C||>num_family('Corporation_Y', 'Corporation_Z', _26)
R||>num_family('Corporation_Y', 'Corporation_Z', 0)
E||<num_family('Corporation_Y', 'Corporation_Z', 0)
C||>num_from('Corporation_Y', 'Corporation_Z', 0, zzzz)

     Do you want to consider applying the rules of Section
318(a)(2) to Corporation_Y?  (enter y or n)  n

E||<num_from('Corporation_Y', 'Corporation_Z', 0, zzzz)
C||>num_to('Corporation_Y', 'Corporation_Z', 0)
     Do you want to consider applying the rules of Section
318(a)(3) to Corporation_Y?  (enter y or n)  y

R||>num_to('Corporation_Y', 'Corporation_Z', _28)
     Is there a shareholder of Corporation_Y who owns, actually or
constructively, stock in Corporation_Z?  y

     Does the shareholder own 50% or more of the stock?  y

     What is the name of the shareholder?  A

     Is A an Individual, Partnership, Estate, Trust, Grantor
trust, or corporation?  (enter i, p, e, t, g, or c)  i

     Is there a shareholder of Corporation_Y who owns, actually or
constructively, stock in Corporation_Z?  n

C|||>num_a_own('A', 'Corporation_Z', _136)
     How many share of Corporation_Z stock does A actually own?  0

E|||<num_a_own('A', 'Corporation_Z', 0)
C|||>num_family('A', 'Corporation_Z', _137)
```

Listing Three

```
     Does A have a spouse, child, grandchild, or parent?  n

E|||<num_family('A', 'Corporation_Z', 0)
C|||>num_from('A', 'Corporation_Z', 0, 'Corporation_Y')

     Do you want to consider applying the rules of Section
318(a)(2) to A?  (enter y or n)  y

R|||>num_from('A', 'Corporation_Z', _138, 'Corporation_Y')

     Is A a partner in a partnership which owns, actually or
constructively, stock in Corporation_Z?  n

     Is A a beneficiary of an estate which owns, actually or
constructively, stock in Corporation_Z?  n

     Is A the grantor of a grantor trust which owns, actually or
constructively, stock in Corporation_Z?  n

     Is A a 50 percent or more shareholder in a corporation which
owns, actually or constructively, stock in Corporation_Z?  y

     What is the name of the corporation?  Corporation X

     What percentage interest does A own?  100

     Is A a 50 percent or more shareholder in a corporation which
owns, actually or constructively, stock in Corporation_Z?  n

C||||>num_a_own('Corporation_X', 'Corporation_Z', _242)
     How many shares of Corporation_Z stock does Corporation_X
actually own?  50

E||||<num_a_own('Corporation_X', 'Corporation_Z', 50)
C||||>num_from('Corporation_X', 'Corporation_Z', 0, 'A')

     Do you want to consider applying the rules of Section
318(a)(2) to Corporation_X?  (enter y or n)  n

E||||<num_from('Corporation_X', 'Corporation_Z', 0, 'A')
C||||>num_options('Corporation_X', 'Corporation_Z', _244)

     Does Corporation_X own an option (or an option to acquire an
option) to acquire the stock in Corporation_Z?  n

R||||>num_options('Corporation_X', 'Corporation_Z', 0)
E|||<num_options('Corporation_X', 'Corporation_Z', 0)
E||<num_from('A', 'Corporation_Z', 50, 'Corporation_Y')
C||>num_to('A', 'Corporation_Z', 0)
R||>num_to('A', 'Corporation_Z', _139)
R||>num_to('A', 'Corporation_Z', 0)
E||<num_to('A', 'Corporation_Z', 0)
C|||>num_option('A', 'Corporation_Z', _140)

     Does A own an option (or an option to acquire an option) to
acquire the stock in Corporation_Z?  n

R|||>num_options('A', 'Corporation_Z', 0)
E||<num_options('A', 'Corporation_Z', 0)
E|<num_to('Corporation_Y', 'Corporation_Z', 50)
C||>num_options('Corporation_X', 'Corporation_Z', _29)

     Does Corporation_Y own an option (or an option to acquire an
option) to acquire the stock in Corporation_Z?  n

R||>num_options('Corporation_Y', 'Corporation_Z', 0)
E||<num_options('Corporation_Y', 'Corporation_Z', 0)
E|<num_con_own('Corporation_Y', 'Corporation_Z', 50)

     The number of shares of Corporation_Z stock actually owned by
Corporation_Y is 0.

     The number of shares of Corporation_Z stock constructively
owned by Corporation_Y is 50.

     Do you want to try another Client?  n
```

End Listing Three

Listing Four

```
%  THE SECTION 318(a) PROLOG PROGRAM WHICH ANSWERS HOW, HELP AND
%  WHY
%  By Dean A. Schlobohm
%  (c) Copyright 1984 by Dean A. Schlobohm.  All Rights Reserved.
%  Version 1.0
%  Date:  6/11/84
%  Bugs:
%    1.  Only includes the Section 318(a)(1) family attribution
%    rules.
%    2.  Very crude English interface.
%  To Use:  type "prolog sec318-1.pro<RET>" then after Prolog is
%  loaded type "section318!<RET>"


% preliminary functions

append([],L,L) :- !.
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

appendall([],[]).
appendall([H],H).
appendall([H1,H2],L) :- append(H1,H2,L).
appendall([H1,H2|T],L) :-
     append(H1,H2,X),
     appendall(T,Z),!,
     append(X,Z,L).

member(X,[X|_]) :- !.
member(X,[_|Y]) :- member (X,Y).

add(X) :- X,!.
add(X) :- asserta(X),!.


%  The Interface Program.

section318 :-
     prompt(Old,' '),
     nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,
     print('     THIS PROGRAM WILL DETERMINE THE NUMBER OF SHARES
     OF'),
     print('STOCK A CLIENT CONSTRUCTIVELY OWNS AS A RESULT OF
     SECTION '),
     print('318(a)(1) OF THE INTERNAL REVENUE CODE.'),
     nl,nl,
     ask('     Enter the name of the Client.  (One word)  '
     ,Client),nl,
     ask('     Enter the name of the Corporation.  (One word)  '
     ,Corp),nl,
     add(corporation(Corp)),!,
num_family(Client,Corp,N_con,[],H)
     ,!,nl,nl,
     print('     The number of shares of ',Corp,' stock
     constructively'),
     print('owned by ',Client,' as a result of Section 318(a)(1)
     is ' ,N_con,'.'),
     nl,nl,!,
     ask('    Do you want to know how I reached this conclusion? '
     ,Z),!,
     ((Z = y) ; (Z = 'Y')),nl,
     how(num_family(Client,Corp,N_con,W,H)),nl,nl,
     prompt(_,Old).

individual(Client,W,H) :-
     prin('     Is ', Client, ' an individual? '),
     ratom(X),nl,!,
     respond1(X,individual(Client,W,H),W).

respond1(y,individual(Client,_,[H]),_) :-
     concat(['    FACT:  ',Client,' is an individual.'],H),
     asserta(ind(Client,_,[H])),!.
respond1(why,individual(Client,W,H),[]) :-
     print('This is what the Internal Revenue Code provides!'),!,
     nl,
```

(Continued on next page)

Listing Four

```
     individual(Client,[],H),!.
respond1(why,individual(Client,W,H),[P|T]) :-
     print(P),nl,!,
     individual(Client,T,H).
respond1(help,Y,Z) :-
     print('Section 318(a)(1) only applies to individuals.'),!,nl,
     Y.
respond1(n,individual(Client,_,[H]),_) :-
     concat(['    FACT:  ',Client,' is not an individual.'],H),
     asserta(non_ind(Client,_,[H])),!,fail.
respond1(X,Y,Z) :-
     print('I do not understand!  Enter y, n, help, or
     why.'),!,nl,
     Y.

%  the rule for actual ownership
num_a_own(C,C,0,W,[H]) :- !.
num_a_own(Client,Corp,N,W,[H]) :-
     print('     How many shares of ',Corp,' stock does ',Client),
     prin('actually own? '),
     ratom(X),nl,
     integer(X),
     concat(['    FACT:  ',Client,' actually owns ',X,'
     shares.'],H),
     N = X,!.
 num_a_own(Client,Corp,N,W,[H]) :-
      print('     Error -- You must enter an integer!'),nl,!,
      num_a_own(Client,Corp,N,W,[H]).

 %  the rule of Secton 318(a)(1)

 num_family(Client,Corp,N,W,H) :-
      get_rule(1,Z),
      append(W,[Z],X),
      individual(Client,X,Y),!,
      spouse_own(Client,Corp,N1,X,Q),
      family_own(Client,Corp,N2,X,R),
      N is N1 + N2,
      appendall([[Z],Y,Q,R],H),!.
 num_family(Client,_,0,_,H) :-
      get_rule(1,Z),
      non_ind(Client,_,Y),!,
      append([Z],Y,H),!.

 %  definition of spouse_own

 spouse_own(Client,Corp,N,W,H) :-
      get_rule(2,Z),
      append(W,[Z],A),
      spouse(Client,L1,A,H1),
      append([Z],H1,H2),
      ind_own(Client,Corp,L1,N,H2,H3),
      appendall([[Z],H1,H3],H),!.
 spouse_own(Client,_,0,_,[H]) :-    concat(['    FACT:  ',Client,'
      has no spouse.'],H),!.

 %  definition of family_own

 family_own(Client,Corp,N,W,H) :-
      get_rule(3,Z),
      append(W,[Z],A),
      family(Client,L1,A,H1),
      append([Z],H1,H2),
      ind_own(Client,Corp,L1,N1,H2,H3),!,
      family_own(Client,Corp,N2,W,H4),!,
      N is N1 + N2,
      appendall([[Z],H1,H3,H4],H),!.
 family_own(Client,_,0,_,[H]) :-
      concat(['    FACT:  ',Client,' has no more family
      members.'],H),!.

 %  definition of ind_own
 %  note Section 318(a)(5)(B) is included since ind_own does
 %    not include num_family.
 %  does NOT include the other attribution rules!
```

```
ind_own(Client,Corp,[],0,_,[]):- !.
ind_own(Client,Corp,[P],N,W,H) :-
     P /= Client,!,
     num_a_own(P,Corp,N,W,H1),
     !,
     H = H1,!.
ind_own(Client,Corp,_,0,W,W) :- !.

%  the definition of a family for purposes of Section 31

spouse(C,[L],W,[H]) :-
     prin('     Does ',C,' have a spouse? '),
     ratom(X),nl,
     respond2(X,spouse(C,[L],W,[H]),W),!.

spouse1(C,[L],_,[H]) :-
     prin('     What is the name of the spouse? '),
     ratom(L),nl,
     concat(['    FACT:  ',L,' is the spouse of ',C,'.'],H),!.

family(C,[L],W,[H]) :-
     prin('     Does ',C,' have a child, grandchild or parent? '),
     ratom(X),nl,
     respond2(X,family(C,[L],W,[H]),W),!.

family1(C,[L],_,[H]) :-
     prin('     What is the name of the child, grandchild or
     parent? '),
     ratom(L),nl,
     concat(['    FACT:  ',L,' is a family member of
     ',C,'.'],H),!.

respond2(y,Relation(Client,Person,_,H),_) :-
     concat([Relation,'1'],Z),
     Z(Client,Person,_,H),!.
respond2(why,Relation(Client,Person,_,H),[]) :-
     print('This is what the Internal Revenue Code provides!'),!,
     nl,
     Relation(Client,Person,[],H),!.
respond2(why,Relation(Client,Person,_,H),[P|T]) :-
     print(P),nl,!,
     Relation(Client,Person,T,H).
respond2(help,spouse(Client,Person,W,H),_) :-
     print('A spouse does not include one who is legally separated
     from'),
     print(Client,' under a decree of divorce or separate
     maintenance.'),
     print('Section 318(a)(1)(A)(i).'),
     !,nl,
     spouse(Client,Person,W,H).
respond2(help,family(Client,Person,W,H),_) :-

     print('A child includes a legally adopted child.  Section
     318(a)(1)(B).'),
     !,nl,
     family(Client,Person,W,H).
respond2(n,Relation(Client,_,_,[H]),_) :- !,fail.
respond2(X,Y,Z) :-
     print('I do not understand!  Enter y, n, help, or
why.'),!,nl,
     Y.

%  definition of rules to use in how and why

get_rule(1,'An individual is deemed to own the stock owned by
     his family.  Section 318(a)(1).').
get_rule(2,'A spouse is a family member.  Section
     318(a)(1)(A)(i).').
get_rule(3,'A child, grandchild or parent
     is a family member.  Section 318(a)(1)(A)(ii).').

% definition of "how"

how(num_family(Client,Corp,N,_,[H|T])) :-
     print('                START OF EXPLAINATION'),
     nl,
     print('     The number of shares of ',Corp,' stock
     constructively owned'),
     print('by ',Client,' as a result of Section 318(a)(1) is '
     ,N,','),
     print('determined as follows:'),nl,
     print(H),!,
```

(Continued on next page)

Listing Four

```
     howl(T),!.
how(_) :-
     print('     I can not give you an explaination!'),!.

howl([]) :-
     nl,
     print('                    END OF EXPLAINATION'),
     !.
howl([H|T]) :-
     nl,
     print(H),
     howl(T),
     !.
```

End Listing Four

Listing Five

```
     THIS PROGRAM WILL DETERMINE THE NUMBER OF SHARES OF STOCK A
CLIENT CONSTRUCTIVELY OWNS AS A RESULT OF SECTION 318(a)(1) OF THE
INTERNAL REVENUE CODE.

     Enter the name of the Client.  (One word)  Dean_Schlobohm

     Enter the name of the Corporation.  (One word)  Corporation

     Is Dean_Schlobohm an individual?  why

An individual is deemed to own the stock owned by his family.
Section 318(a)(1).

     Is Dean_Schlobohm an individual?  why

This is what the Internal Revenue Code provides!

     Is Dean_Schlobohm an individual?  y

     Does Dean_Schlobohm have a spouse?  why

An individual is deemed to own the stock owned by his family.
Section 318(a)91).

     Does Dean_Schlobohm have a spouse?  why

A spouse is a family member.  Section 318(a)(1)(A)(i).

     Does Dean_Schlobohm have a spouse?  why

This is what the Internal Revenue Code provides!

     Does Dean-Schlobohm have a spouse?  help

A spouse does not include one who is legally separated from
Dean_Schlobohm under a decree of divorce or separate maintenance.
Section 318(a)(1)(A)(i).

     Does Dean_Schlobohm have a spouse?  y

     What is the name of the spouse?  Mary_Schlobohm

     How many shares of Corporation stock does Mary_Schlobohm
actually own?  300

     Does Dean_Schlobohm have a child, grandchild or parent?  why

An individual is deemed to own the stock owned by his family.
Section 318(a)(1).

     Does Dean_Schlobohm have a child, grandchild or parent?  why

A child, grandchild or parent is a family member.  Section
318(a)(1)(A)(ii).
```

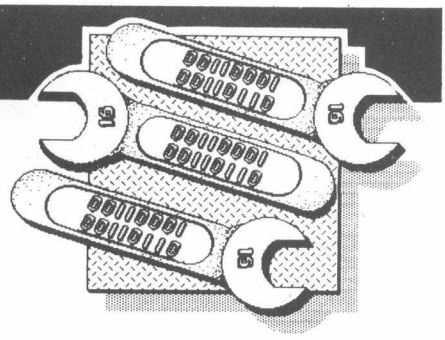
(Continued on page 92)

Listing Five

```
     Does Dean_Schlobohm have a child, grandchild or parent?  why

This is what the Internal Revenue Code provides!

     Does Dean_Schlobohm have a child, grandchild or parent?  help

A child includes a legally adopted child.  Section 318(a)(1)(B).

     Does Dean_Schlobohm have a child, grandchild or parent?  y

     What is the name of the child, grandchild or parent?
Dean_F_Schlobohm

     How many share of Corporation stock does Dean_F_Schlobohm
actually own?  100

     Does Dean_Schlobohm have a child, grandchild or parent?  n

     The number of shares of Corporation stock constructively
owned by Dean_Schlobohm as a result of Section 318(a)(1) is 400.

     Do you want to know how I reached this conclusion?  y

                    START OF EXPLANATION

     The number of shares of Corporation stock constructively
owned by Dean_Schlobohm as a result of Section 318(a)(1) is 400,
determined as follows:

An individual is deemed to own the stock owned by his family.
Section 318(a)(1).

     FACT:  Dean_Schlobohm is an individual.

A spouse is a family member.  Section 318(a)(1)(A)(i).

     FACT:  Mary_Schlobohm is the spouse of Dean_Schlobohm.

     FACT:  Mary_Schlobohm actually owns 300 shares.

A child, grandchild or parent is a family member.  Section
318(a)(1)(A)(ii).


FACT:  Dean_F_Schlobohm is a family member of Dean_Schlobohm.

FACT:  Dean_F_Schlobohm actually owns 100 shares.

FACT:  Dean_Schlobohm has no more family members.

                    END OF EXPLANATION
```

End Listings

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

Z-8001 CPU for S-100 Bus

Yes, gentle reader, we occasionally *do* discuss some other kind of microprocessor besides the Intel family. This month, we'd like to announce the availability of an S-100 bus Z-8001 CPU card called the Super Z-10 from Way Engineering. The CPU runs at 10 MHz, is fully compatible with the IEEE 696 standard, and includes two on-board RS-232 ports, 32K of phantom EPROM, and a complete Forth monitor in EPROM. You can use the Z-8001 CPU card with Compupro high-speed static RAM boards and with most popular S-100 hard disk controllers. The introductory cost of the Super Z-10 CPU is $1000.00.

Digital Research CP/M-8000 is available in a customized version for the Super Z-10 and includes the usual CP/M utilities (PIP, ED, DDT, and so on) as well as a C compiler, assembler, and linker. The BIOS is written in C and assembly language, and the source code is supplied. A disk-based Forth development system that runs under CP/M-8000 will also be available in the near future. CP/M-8000, which requires a hard disk and 256K of RAM, costs $350.00. For more information, contact Way Engineering at 2011 Tulip Tree Lane, La Canada, CA 91011 (213) 245-1480. I am taking delivery of one of these systems within the next few days, so you should see a detailed report in a later column.

MacFeedback

Mike Cohen of the Bronx writes: "In the December 1984 issue, you incorrectly state that the Macintosh 68000 assembler requires two machines. It simply isn't true, as I've used that assembler along with Consulair's excellent Mac C compiler on a single Macintosh with no problems. True, Apple provides a debugger, which does require two machines, but the Macsbug debugger (which is also provided on the same disk) requires only one machine. It does have some problems in 128K, since it can't remain resident while the assembler or editor are being used. There are no such problems on a 512K Macintosh, since I keep the full screen debugger resident while editing, assembling, and compiling programs.

"Also, the statement that most high-level languages are 'incredibly weak, bug-ridden, and slow' is untrue of everything other than MS-Basic. MacPascal is one of the best versions of Pascal I've seen on any micro and reasonably fast—it can calculate 1500 Factorial (using extended precision math) in an amazing 10 seconds. Consulair Corp.'s Mac C compiler is fast, powerful, and bug-free. It's reasonably Unix-compatible, although using the Mac Toolbox library rather than STDLIB results in much smaller applications (a text editor can be easily done in less than 4K).

"To get things started with 68000 software, here are two functions I wrote for use with the Mac C compiler to provide the standard file 'Open' and 'Save as' dialogs. Most of the code was output directly from Mac C (particularly the in-line trap instructions and parameter saving).

"The function OldFileName(Count, typeList,Reply) will display the open dialog, which lists all files of the specified type. NewFileName(DefaultName,Reply) will display the save dialog, with the default filename displayed for editing. Both functions return a zero if the request was cancelled, otherwise returning nonzero, and Reply (pointer to struct SFReply, as defined in Inside Macintosh and packages.h supplied with Mac C) will contain the filename as volume reference number of the file selected." See Listing One (page 102) for Mr. Cohen's contribution.

Well, having tried or observed friends being tried by MacPascal, MacBasic, and MacForth, I'll stand by my earlier statements about those languages at least. They don't meet my criteria for "insanely great." On the Apple assembler, I was in error. I based my previous column on Apple's own product announcement, which clearly stated "two machines required," and had not received the actual software when I submitted the column to *DDJ* for publication. The Macsbug debugger, by the way, is insanely clever. When loaded, it simply requests some space on the heap and then hides itself there like any other data object, grabbing back control when an exception is detected.

More Mac News

As far as program development tools go for the Mac, there is hope for the rest of us after all. Mainstay of Agoura Hills, CA, has announced MacASM, a macroassembler for the Macintosh with an integral full-screen editor and resource compiler that costs only $125.00. I already have received several favorable letters from *DDJ* readers about this product. You can contact Mainstay at (818) 991-6540.

But the neatest news of all is the impending release of Neon from Kriya Systems, which we recently saw demonstrated at the Las Vegas Comdex Apple booth. This is a high-level, object-oriented language, a la Smalltalk, that will sell for only $150.00 and allows full access to the Macintosh's Toolbox and graphics capabilities. Best of all, the package includes extensive source code, a 600-page manual, and a toll-free license to distribute Neon embedded in application programs. Based on a 32-bit implementation of Forth, Neon makes available to

the programmer the full spectrum of Forth commands and extensibility. The scheduled release date of Neon is February 15, so if all goes well, it should be available by the time you read this. You can reach Kriya at 505 N. Lake Shore Drive, Suite 5510, Chicago, IL 60611.

At the other end of the spectrum, we have the advertisements heralding "CP/M for the Macintosh" from IQ Software in Fort Worth, TX. I'll bet this advertisement made the Apple software gurus gnash their teeth and pull their hair out! I can't imagine anything more antithetical to the whole MacMilieu than DIR, PIP, ED, and STAT.

80286 Opcode of the Month

One of the opcodes added in the 80286 was the instruction called BOUNDS, which checks whether a number falls within limits and generates a hardware interrupt 5 if it does not. This nifty opcode was added for the sake of compiler writers who wanted an efficient way to check the validity of an array index at runtime. Don't rush to use this instruction in your assembly language programming experiments on the IBM PC/AT, however, or you may get an unpleasant surprise.

It seems that way back when the PC/AT was just a gleam in the eye of the president of the Entry Systems Division (in fact, no such division even existed then), somebody at either IBM or Microsoft assigned interrupt 5 to the Print Screen ROM BIOS routine. So, when the 80286 executes a BOUNDS instruction, detects a value outside of the specified range, and executes interrupt 5 . . . you guessed it! To add insult to injury, the 80286 assumes that the interrupt 5 handler will fix up the offending value, so it executes the BOUNDS instruction again. Of course, the ROM BIOS program to print the screen carefully saves and restores all the registers, including the one containing the nasty value that caused the interrupt, because it doesn't know anything about BOUNDS exceptions or fix ups or anything else for that matter. The result: one screen dump after another until you give up and restart the computer.

IBM has distributed a fix for this problem in its Software Support Center news bulletin. The fix involves installation of a new interrupt 5 handler that does the following when it receives control:

(1) It uses the address pushed on the return stack to peek into the program that was executing to determine whether a BOUNDS opcode caused the exception.

(2) If the cause was, in fact, a BOUNDS opcode, then it performs both the appropriate fix up and an interrupt return.

(3) Otherwise, it transfers control to the Print Screen ROM BIOS routine.

Not very aesthetically pleasing, but this fix seems to work.

DOS 3.0 Differences

In my last column, I discussed some of the documented differences between PCDOS 2.0 and DOS 3.0. Here's news about one I missed. The unexplained failure of a once perfectly happy program prompted me to investigate the behavior of the Find First and Find Next functions (04EH and 04FH, respectively). Much to my surprise, I found that the definition of the Find Next function has been changed in DOS 3.0 for no obvious reason. Furthermore, the change is not documented on DOS Technical Manual page vi, which is supposed to warn the programmer about new or modified DOS function calls.

Under DOS 2.0, function 4FH did not take any register arguments but assumed that the information from the previous call to Find First or Find Next was in the current DTA. Under DOS 3.0, the manual says that function 4FH requires DS:DX to "contain the information from a previous 'Find First' call." Because there's no conceivable reason why function 4FH would want the original filename string address, I interpret this rather vague wording to mean that DS:DX should point to the same buffer that you set the DTA to before calling Find First. In the case of my program at least, the problem turned out to be not function 4FH at all but the fact that function 4EH does not return register AX = 0 for a successful call as it did under DOS 2.0. Apparently, under DOS 3.0, you absolutely *must* check the carry bit first; if it isn't set, don't assume that register AX will contain anything in particular.

Naturally, a day or two after I'd wasted the afternoon tracking down the above, I got a letter from Dan Daetwyler serving up the same information in his usual lucid way. He went on to write: "The DOS 3.0 writers finally saw fit to give us an interface to the resident PRINT function. It's described as interrupt 2FH in the manual. I had a critical need for this function, and it was one of the reasons I quickly hopped on the V3.0 bandwagon. So I read the manual and then tried to implement the function. The manual states that the AL register (and for some calls, some other registers) must be set up before the interrupt is issued. A set of values for AL are given, with specific details of the return values, other registers, etc. Not bad!

"What's the problem? Simple. Nowhere is the AH register mentioned. But it turns out that unless AH contains a binary one at interrupt time, you will have no chance at all of getting the thing to work! I found it out by the cut and try method, and lucked out. Since then I've disassembled the PRINT module and confirmed my 'guess.' Not so incidentally, this interrupt is apparently going to be used for

```
Declarations: globals
     char c0;int x0;
     char c,c1,ca[100];int x,x1,xa[100];

Declarations: locals
char c1,c11,ca1[100];int x1,x11,xa1[100];
```

Test #	Z80A 4 MHz BDS C v1.43 msec	286 6 MHz C86 v1.33 msec	Ratio	statement
1	5	1	5.00	c='a';
2	6	1	6.00	x=1;
3	6	1	6.00	x=1234;
4	15	4	3.75	if(c)x=1;
5	15	4	3.75	if(x)x=1;
6	13	5	2.60	c <<= 1;
7	21	5	4.20	c <<= 4;
8	11	4	2.75	x <<= 1;
9	19	5	3.80	x <<= 4;
10	33	5	6.60	c >>= 1;
11	69	5	13.80	c >>= 4;
12	31	4	7.75	x >>= 1;
13	67	5	13.40	x >>= 4;
14	8	2	4.00	c=ca[0];
15	17	10	1.70	c=ca[c0];
16	8	2	4.00	x=xa[0];
17	22	6	3.67	x=xa[x0];
18	8	2	4.00	c=c0;
19	8	1	8.00	x=x0;
20	11	4	2.75	c=c+1;
21	5	3	1.67	c++;
22	9	2	4.50	x=x+1;
23	9	2	4.50	x++;
24	4	1	4.00	c1='a';
25	11	1	11.00	x1=1;
26	11	1	11.00	x1=1234;
27	20	4	5.00	if(c1)x1=1;
28	27	4	6.75	if(x1)x1=1;
29	16	5	3.20	c1 <<= 1;
30	24	5	4.80	c1 <<= 4;
31	25	4	6.25	x1 <<= 1;
32	33	5	6.60	x1 <<= 4;
33	37	5	7.40	c1 >>= 1;
34	72	5	14.40	c1 >>= 4;
35	46	4	11.50	x1 >>= 1;
36	81	5	16.20	x1 >>= 4;
37	12	2	6.00	c1=ca1[0];

Test #	Z80A 4 MHz BDS C v1.43 msec	286 6 MHz C86 v1.33 msec	Ratio	statement
38	21	10	2.10	c1=ca1[c0];
39	22	3	7.33	x1=xa1[0];
40	33	6	5.50	x1=xa1[x0];
41	8	2	4.00	c1=c0;
42	15	2	7.50	x1=x0;
43	11	4	2.75	c1=c1+1;
44	4	3	1.33	c1++;
45	24	2	12.00	x1=x1+1;
46	17	2	8.50	x1++;
47	253	18	14.06	movmem(&ca[0],&ca[50],1);
48	274	21	13.05	movmem(&ca[0],&ca[50],5);
49	218	47	4.64	strcpy(&ca[50],&ca[0]); /*1 char*/
50	764	192	3.98	strcpy(&ca[50],&ca[2]); /*11 chars*/
51	14	14	1.00	dummy(); /*just returns*/

Notes:
1. All times accurate to 1 microsecond.
2. Z80A tests were run at 4 MHz in an IMS International 8000 CPM80 system. 80286 tests were run at 6 MHz (no wait states) on a Lomas Data Products LDP2 MSDOS 2.1 system.
3. For BDS-C both -o and -e optimizations were used.
4. Test program generation and result reporting done with T/Maker.

Figure 1

Thomas Morgan's Benchmarks

some of the TopView or file-sharing functions, and if you issue the interrupt with the AH register containing large values (0BBH range), you end up going to a totally different function. Undocumented, of course!

"I'm getting a bit tired of getting manuals filled with good advice (it really is) on programming techniques and then finding out that the operating system violates these 'good practices' right and left. The PRINT module, for example, not only includes a significant array of self-modifying code (ugh), but it even 'relocates' (within its own area . . . why?) its interrupt service routines then does adjustment of the addresses to allow for the self-imposed relocation! (Double ugh.) I'm sure I'll eventually find out that there is a rationale for this, but I strongly suspect it will be a 'cop-out'. I've never yet found a reason for flagrant violation of good programming practices! This case seems to be caused by a desire to save a couple of hundred bytes of memory. Well, everyone knows that DOS is growing fast enough, without wasted memory, but it also appears that the memory could have been saved by simple reorganization of the source module. If I ever get the courage to complete the disassembly, I'll find out!"

Microsoft Assembler Bug of the Month

The program in Listing Two (page 102) will assemble without any errors; however, when you try to link it, you get the error message:

```
Invalid Object Module.
  Input file: TEST.OBJ(A)
  pos: 00044 Record Type: A0
```

In search of further enlightenment, we looked this message up in the DOS 3.0 manual (page A-58) and found the advice:

> Explanation: LINK. Object module(s) was incorrectly formed or unobserved errors occurred during compilation. The disk may be bad.
> Action: Recompile the module to either the same or a different disk.

There's no explanation at all of what the various record types for the Linker consist of or what "pos: 00044" is supposed to mean (if you knew the latter, you could probably figure out the offending instruction regardless of the inscrutable error message).

Repeated attempts at assembly and linking, on different disks, using different versions of DOS and/or the Linker, produced the same or other confusing error messages. To make a long story short, either the Assembler or the Linker can't cope with a DD data declaration statement inside the invocation of a structure macro. Our thanks to the boys at Microsoft for yet another interesting puzzle; we trashed an entire day figuring out that this was the only problem in a 5000-line graphics driver.

Some 80286 Timings

Thomas Moran writes: "Here are some comparisons of individual C statement times using BDS C on a Z80 and C86 on an 80286. Most of the C programs I run seem to be around three times as fast on the 286 as the Z80. (They run slightly faster on the Z80 than on the IBM PC.)

"Perhaps the most interesting results are (1) the C86 code for a subroutine call takes as long to execute on the 80286 as the BDS C code on the Z80, and (2) strcpy is a horror in both cases. One often codes *p++ . . . for scanning and copying since C has no language constructs to handle strings. Perhaps a smart compiler could realize that a subroutine could be replaced by a few machine-repeated move or scan instructions." Thomas's timings are presented in the figure on page 100.

16-Bit Software Toolbox Installment

For my own code contribution this month, I'm providing two 8086 assembly language subroutines of general interest (Listing Three, page 104). The first, BIN_TO_ASC, converts a signed 32-bit integer to an ASCII string in the desired radix (the radix must be in the range 2 – 36). The upper part of the 32-bit value is passed in DX and the lower part in AX; the radix is passed in CX; and SI contains the last byte address of the area to store the converted ASCII string. It is the responsibility of the user to make sure that the destination buffer is large enough for any number that might be printed in the desired radix.

DIVIDE is a general purpose 32-bit by 16-bit unsigned divide subroutine. The 32-bit dividend is passed with the upper part in DX and the lower part in AX; the divisor is passed in CX. The 32-bit quotient is returned in DX:AX, and the remainder in BX. Contrast this with the 8086's built-in unsigned DIV instruction, which also divides DX:AX by a 16-bit value but can return only a 16-bit quotient; this is inadequate for an output conversion routine that must extract digits from a 32-bit argument. The code for DIVIDE may look a little obscure at first, but it will become clearer if you simply consider each of the registers DX, AX, and CX as a single digit. Then you'll find that it works exactly like normal long division done by hand.

DDJ

(Listings begin on next page)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**

Listing One

```
Mike Cohen's standard file "Open" and "Save" dialogs for the Macintosh

;Consulair Mac C Compiler 1.0
     STRING_FORMAT 0
     INCLUDE M68KLIB.D
OldFileName:
     CLR.L     -(SP)
     LINK      A6,#-20
     MOVEM.L   D0/D1/D2,-16(A6)
     PEA       -20(A6)
     DC.W      $A874      ;GetPort
     MOVE.W    #80,-(SP)
     MOVE.W    #80,-(SP) ;location
     CLR.L     -(SP)      ;Prompt
     CLR.L     -(SP)      ;FileFilter
     MOVE.W    D0,-(SP)   ;numTypes
     MOVE.L    D1,-(SP)   ;TypeList
     CLR.L     -(SP)      ;DlgHook
     MOVE.L    D2,-(SP)   ;Reply
     MOVE.W    #2,-(SP)   ;routine selector for GetFile
     DC.W      $A9EA      ;Pack3
     MOVE.L    -20(A6),-(SP)
     DC.W      $A873      ;SetPort
     MOVE.L    -8(A6),A0
     MOVE.B    (A0),D0
     EXT       D0
     EXT.L     D0
     RETX
     XREF      OldFileName
;

NewFileName:
     CLR.L     -(SP)
     LINK      A6,#-16
     MOVEM.L   D0/D1,-12(A6)
     PEA       -16(A6)
     DC.W      $A874      ;GetPort
     MOVE.W    #80,-(SP)
     MOVE.W    #80,-(SP) ;location
     PEA       @1         ;Prompt
     MOVE.L    D0,-(SP)   ;OrigName
     CLR.L     -(SP)      ;DigHook
     MOVE.L    D1,-(SP)   ;Reply
     MOVE.W    #1,-(SP)   ;routine selector for PutFile
     DC.W      $A9EA      ;Pack3
     MOVE.L    -16(A6),-(SP)
     DC.W      $A873      ;SetPort
     MOVE.L    -8(A6),A0
     MOVE.B    (A0),D0
     EXT       D0
     EXT.L     D0
     RETX
@1:  DCB       8
     DC.B      'Save As:'
     XREF      NewFileName
```

End Listing One

Listing Two

```
Demonstrates obscure Microsoft assembler/linker bug.

        title   'Show Structure Bug'
        page    60,132

dseg    segment para public 'DATA'
```

(Continued on page 104)

 (Listing Continued, text begins on page 96)

Listing Two

```
fred    struc                   ;declare a structure macro
        dd      1 dup (?)       ;consisting of a double-word
fred    ends                    ;data declaration

harry   fred    <>              ;invoke the structure macro

dseg    ends

cseg    segment para public 'CODE'

pgm     proc    far
        ret                     ;this is a Zen program...
pgm     endp

cseg    ends

        end     pgm
```

End Listing Two

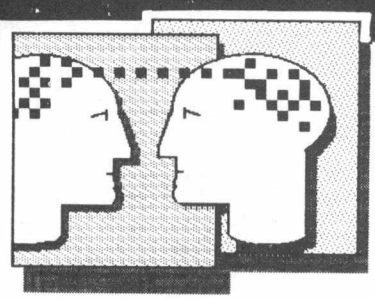
Listing Three

```
General purpose Binary-to-ASCII conversion and unsigned divide subroutines.

;
; Convert 32 bit binary value to ASCII string.
; Copyright (C) 1984 Ray Duncan
;
; Call with  DX:AX = signed 32 bit value
;            CX    = radix
;            SI    = last byte of area to store resulting string
;                    (make sure enough room is available to store
;                     the string in the radix you have selected.)
;
; Destroys AX, BX, CX, DX, and SI.
;
bin_to_asc proc near            ;convert DX:AX to ASCII.
                                ;force storage of at least 1 digit.
        mov     byte ptr [si],'0'
        or      dx,dx           ;test sign of 32 bit value,
        pushf                   ;and save sign on stack.
        jns     bin1            ;jump if it was positive.
        not     dx              ;it was negative, take 2's complement
        not     ax              ;of the value.
        add     ax,1
        adc     dx,0
bin1:                           ;divide the 32 bit value by the radix
                                ;to extract the next digit for the
                                ;forming string.
        mov     bx,ax           ;is the value zero yet?
        or      bx,dx
        jz      bin3            ;yes, we are done converting.
        call    divide          ;no, divide by radix.
        add     bl,'0'          ;convert the remainder to an ASCII digit.
        cmp     bl,'9'          ;we might be converting to hex ASCII,
        jle     bin2            ;jump if in range 0-9,
        add     bl,'A'-'9'-1    ;correct it if in range A-F.
bin2:   mov     [si],bl         ;store this character into string.
        dec     si              ;back up through string,
        jmp     bin1            ;and do it again.
bin3:                           ;restore sign flag,
        popf                    ;was original value negative?
        jns     bin4            ;no, jump
                                ;yes,store sign into output string.
        mov     byte ptr [si],'-'
```

```
bin4:   ret                     ;back to caller.
bin_to_asc endp
;
;
;
; General purpose 32 bit by 16 bit unsigned divide.
; Copyright (C) 1984 Ray Duncan
;
; This must be used instead of the plain machine unsigned divide
; for cases where the quotient may overflow 16 bits (for example,
; dividing 100,000 by 2).  If called with a zero divisor, this
; routine returns the dividend unchanged and gives no warning.
;
; Call with DX:AX = 32 bit dividend
;           CX    = divisor
;
; Returns   DX:AX = quotient
;           BX    = remainder
;           CX    = divisor (unchanged)
;
divide  proc    near            ; Divide DX:AX by CX
        jcxz    divl            ; exit if divide by zero
        push    ax              ; 0:dividend_upper/divisor
        mov     ax,dx
        xor     dx,dx
        div     cx
        mov     bx,ax           ; BX = quotient1
        pop     ax              ; remainder1:dividend_lower/divisor
        div     cx
        xchg    bx,dx           ; DX:AX = quotient1:quotient2
divl:   ret                     ; BX = remainder2
divide  endp
;
;
```

End Listings

CP/M EXCHANGE



by Robert A. Blum

Steve Russell of SLR Systems, the author of Z80ASM, recently sent along a disk containing a problem report that he had sent to DRI. The disk contained a program to demonstrate the problem he had found with CP/M Plus and the RSX that he had written to correct the problem.

While installing for a construction company an accounting package that was written in a high-level language, Steve encountered inconsistent results when he attempted to determine the current disk file sizes. Although the software publisher proved to be of little help, Steve did learn that this strange occurrence seemed also to appear under MP/M but never with CP/M 2.2—and that the disk files in question were being accessed with random I/O BDOS calls.

After a great deal of probing and prodding into how CP/M Plus was responding to the application program's system calls, Steve found a problem with the way the CP/M Plus file system was treating random read errors. This situation was compounded further by the fact that errors sometimes were not reported at all and that incorrect data was being placed into the FCB.

The disk I received contained a sample data file, 257 sectors (records) in length, and a program FIND that would produce the problem. When run, the FIND program showed that the random record logic of CP/M Plus became confused on the second consecutive random read to an unallocated sector of an existing extent other than extent 0. However, if an interceding valid read is done or a read to an unallocated record of an unwritten extent is done, the problem does not occur.

The FIND program (Listing One, page 112) tries to read several sectors from the file XXX. Each read attempt is documented by displaying the record number in hex and the return error code in A. The FCB contents then are dumped to find out what is happening.

Follow along in Figure 1 (page 109) as each read is documented. The first read is to record zero, which is an allocated record of extent zero; it correctly completes with a zero return code and valid FCB contents. The second read is to record number 257, which is one record beyond the last record written to the file. The return code for this read is reported correctly as one, indicating that the last read operation was on an unallocated record of a written extent. The third read is to an allocated record of extent zero, providing once again a valid error return code of zero and correct FCB contents.

The error occurs on the second of the next two reads. The fourth read is to record number 500, an unallocated record of a written extent. The read is flagged correctly as an error, and the FCB contents are also correct. Now, the error occurs. The fifth read is to record number 260, which again is to an unallocated record of a written extent. This time, the error return code is reported incorrectly as zero, or no error, and the record count field of the FCB incorrectly contains 80h. In reality, both the return code and the record count field should have been one.

So, the problem seems to happen only when two consecutive random reads to an unallocated record of a written extent are attempted. To correct the problem, Steve wrote an RSX (Listing Two, page 114) to monitor the BDOS function calls made by the application program. When a random read is found, the return error code is examined, and when a nonzero error return code is found, a dummy read to record zero of the same file is done. This interceding read keeps the incorrect error return code and FCB contents from resulting, should the same type of read occur a second consecutive time. You might want to examine Figure 2 (page 110), which displays the results of the FIND program after the installation of the RSX.

CP/M Plus: PIP Patch

Gary Silvey of Digital Research recently sent along a patch for the CP/M Plus version of PIP to set the verify option permanently on, as I described for CP/M 2.2 in the November issue of this column. As before, DRI does not recognize this patch, even though it comes from a DRI employee. Also, the patch is known to work only with the specific version of the PIP program mentioned here. Before setting out to install this patch, make a backup copy of your disk in case it becomes necessary to return to your original version of PIP.

Figure 3 (page 112) lists the procedure that I followed to install the verify patch on my system. The first step is to verify that your PIP program is version 3.0; this is the only version of PIP with which this patch is known to work successfully. To verify your version number, load PIP.COM into memory with either SID or DDT and display memory between 210h and 21fh; the version number, which should be 3.0, should begin at 21Ch. The next step is to change the single byte at 1001h from 1Fh to 37h. The final step is to save the modified program back onto disk under a new name ready for testing.

To ensure that I had entered my patches correctly, I timed how long it took to copy the same file with an unaltered version of PIP using the verify option and again with the newly altered version of PIP. The two times were practically identical, which was verification enough for me.

Gary also sent along the timings from his testing of the changes; these once again point out how much quicker

disk I/O is under CP/M Plus.

For your information, I copied HELP.HLP to a file called H to check out the PIP version 3.0 patch. Under my nonbanked version of CP/M Plus, it took 13:42 sec for the copy to complete with the patched PIP and 21:71 sec with unpatched PIP. On my banked system, copying the same file with the patched and unpatched PIP yielded times of 8:00 and 18:24 sec, respectively. Using the unpatched version of PIP on the banked system with the V option yielded 18:25 sec. **DDJ**

(Listing begins on page 112)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 196.**

```
READING RECORD 0000, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
00 00 00 00

READING RECORD 0101, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 02 02 80 01
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 01 01 00

READING RECORD 0064, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
64 64 00 00

READING RECORD 01F4, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 03 02 80 00
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
74 F4 01 00

READING RECORD 0104, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 02 02 80 80
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04 04 01 00

READING RECORD 0000, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
00 00 00 00

READING RECORD 0100, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 02 02 80 01
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 01 00

READING RECORD 0104, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 02 02 80 01
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04 04 01 00


READING RECORD 0202, ERROR CODE RETURNED IN A IS 04
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 02 02 80 01
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02 02 02 00

READING RECORD 0104, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 02 02 80 01
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04 04 01 00
```

Figure 1 Results of Random Read Before RSX Installed.

```
READING RECORD 0000, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
00 00 00 00

READING RECORD 0101, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
00 01 01 00

READING RECORD 0064, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
64 64 00 00

READING RECORD 01F4, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
00 F4 01 00

READING RECORD 0104, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
00 04 01 00

READING RECORD 0000, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
00 00 00 00

READING RECORD 0100, ERROR CODE RETURNED IN A IS 00
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 02 02 80 01
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 01 00

READING RECORD 0104, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
00 04 01 00


READING RECORD 0202, ERROR CODE RETURNED IN A IS 04
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
02 02 02 00

READING RECORD 0104, ERROR CODE RETURNED IN A IS 01
36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ
00 58 58 58 20 20 20 20 20 20 20 20 00 02 80 80
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13_14 15
00 04 01 00
```

Figure 2

Results of Random Read After RSX Installed.

```
B>SID PIP.COM
CP/M 3 SID - VERSION 3.0
NEXT MSZE PC   END
2300 2300 0100 C8FF
#D210,21F
0210  20 33 20 50 49 50 20 56 45 52 53 20 33 2E 30 20  3 PIP VERS 3.0
#S1001
1001 1F 37
1002 D2 .
#WPIPNEW.COM,100,2300
0044h record(s) written.
#GO
```

Figure 3

Pip verify patch

CP/M Exchange (Text begins on page 108)

Listing One

```
        TITLE   DEMONSTRATE FAULTY RANDOM RECORD ERROR
DFCB    EQU     5CH

;
; THE SAMPLE FILE NAME IS RATHER GENERIC, 'XXX'
; TO RUN THIS PROGRAM, JUST TYPE A>FIND XXX
;

START
;
;FIRST OPEN THE FILE
;

        LD      DE,DFCB
        LD      C,0FH
        CALL    5
;
;DID IT OPEN ?
;
        INC     A
        RET     Z       ;NOPE, RETURN
;
;OK, MAKE SURE HIGH BYTE OF RANDOM RECORD IS 0
;
        XOR     A
        LD      (DFCB+35),A

        LD      SP,MYSTACK
;
;FIRST, LETS SET UP A STRING OF RECORD NUMBERS TO READ
;WITH A RANDOM READ TO SEE IF THEY REACT DIFFERENTLY
;
        LD      HL,RECORD_TABLE         ;TABLE OF RECORD NUMBERS
        LD      A,TABLE_LEN             ;LENGTH OF TABLE
LOOP1
        PUSH    AF              ;SAVE COUNT
        LD      E,(HL)          ;LOAD RECORD # DESIRED
        INC     HL
        LD      D,(HL)
        INC     HL
        PUSH    HL              ;SAVE TABLE POINTER
        EX      DE,HL
        LD      DE,RECNUM       ;ENCODE RECORD NUMBER TO HEX-ASCII
        CALL    HEXWOUT
        LD      (DFCB+33),HL    ;SET RECORD NUMBER
        LD      DE,DFCB
        LD      C,21H           ;RANDOM READ
        CALL    5
        LD      DE,RESULT       ;ENCODE VALUE RETURNED IN A
        CALL    HEXOUT
                CALL    PRINT_LINE      ;PRINT LINE ON THE CONSOLE
                POP     HL              ;RESTORE POINTER AND COUNTER
                POP     AF
```

```
                DEC     A
                JR      NZ,LOOP1        ;IF MORE RECORDS, JUMP
        ;
        ;FINISHED.......
        ;
                JP      0

        RECORD_TABLE
                DW      0               ;FIRST READ RECORD ZERO
    DW      257             ;READ RECORD 257 (ONLY 0-256 EXIST...ERROR REPORTED
    DW      100             ;READ RECORD 100 (MAIN PURPOSE, BACK TO EXT 0)
    DW      500             ;READ RECORD 500 (ONLY 0-256 EXIST...ERROR REPORTED
    DW      260             ;READ RECORD 260 (ONLY 0-256 EXIST...OOPS! NO
R!)
        DW      0               ;BACK TO VALID EXT INFO
        DW      256             ;LAST RECORD OF FILE, FOUND
        DW      260             ;READ 260, ERROR RETURNED
        DW      514             ;READ RECORD 514 ("  "  "  ,NOTE, UNWRITTEN EXT
        DW      260             ;READ RECORD 260 ("  "  "  ,THIS TIME ERROR
REPORTED)
TABLE_LEN       EQU     ($-RECORD_TABLE)/2

HEXWOUT
        LD      A,H
        CALL    HEXOUT
        LD      A,L
HEXOUT
        PUSH    AF
        RRA
        RRA
        RRA
        RRA
        CALL    HEX2
        POP     AF
HEX2
        AND     0FH
        OR      30H
        CP      3AH
        JR      C,HEX3
        ADD     7
HEX3
        LD      (DE),A
        INC     DE
        RET

PRINT_LINE
        LD      DE,RECLINE
        LD      C,9
        CALL    5
;
;NOW PRINT OUT THE FCB
;
        LD      HL,DFCB
        LD      DE,FCBASCII
        LD      B,36
LOOP5
        LD      A,(HL)
        INC     HL
        CALL    HEXOUT
        INC     DE
        DJNZ    LOOP5
        LD      DE,FCBASC
        LD      C,9
        JP      5

FCBASC  DB      0DH,0AH
        DB      '36 HEX BYTES OF FCB CONTENTS AFTER ATTEMPTED READ',0DH,0AH
FCBASCII
        DS      36*3,20H
        DB      0DH,0AH,'$'

RECLINE DB      0DH,0AH,'READING RECORD '
RECNUM  DB      '    , ERROR CODE RETURNED IN A IS '
RESULT  DB      '  $'

        DS      64
MYSTACK

        END
```

End Listing One

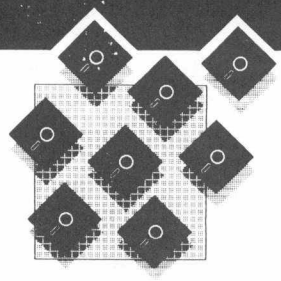
(Listing two begins on next page)

 (Listing Continued, text begins on page 108)

Listing Two

```
            TITLE  RANDOM READ RSX SOURCE LISTING
        ;START OF PREFIX
        DB      0,0,0,0,0,0     ;SERIAL NUMBER
        JP      START
NEXT    JP      $-$
PREV    DW      0
REMOVE  DB      0
NONBNK  DB      0
RSXNAME DB      'TRACE EM'
        DB      0,0,0
;
;START OF RSX CODE
;
START
        LD      A,C             ;PUT BDOS FUNCTION CODE IN A
        CP      21H             ;IS IT A RANDOM READ
        JR      NZ,NEXT         ;SKIP IF NOT RANDOM READ
START2
;
;       DO READ, BUT IF IT RETURNS AN ERROR CODE 1
;       SEEK SECTOR 0 BEFORE GOING ON TO CORRECT POSSIBLE
;       INACCURACY OF SUBSEQUENT RANDOM READ
;
        LD      (USERSAVE),SP   ;SAVE USER STACK POINTER
        LD      SP,MYSTACK      ;LOAD A LOCAL STACK POINTER
        PUSH    DE              ;SAVE FCB ADDRESS
        CALL    NEXT            ;ALLOW REQUESTED OPERATION TO COMPLETE
;
;       THE USER REQUESTED OPERATION IS NOW COMPLETE
;
        CP      1               ;WAS THE RETURN CODE DATA UNAVAILABLE
        JR      NZ,START8       ;NO - BYPASS OUR FIX
;
;       SAVE ALL THE USER INFORMATION AND PREPARE FOR SEEK TO SECTOR ZERO
;
        EX      (SP),HL         ;SAVE HL AND LOAD FCB ADDRESS
        LD      DE,33           ;GET INDEX TO R0 POSITION OF FCB
        EX      DE,HL
        ADD     HL,DE           ;ADD R0 POSITION INDEX TO FCB ADDRESS
        PUSH    HL              ;SAVE ON STACK
        XOR     A               ;PUT ZERO INTO A
        LD      C,(HL)          ;PUT FCB R0 BYTE INTO C
        LD      (HL),A          ;ZERO FCB R0 BYTE
        INC     HL              ;POINT AT R1 BYTE
        LD      B,(HL)          ;PUT FCB R1 BYTE INTO B
        PUSH    BC              ;SAVE BC ON STACK
        LD      (HL),A          ;ZERO FCB R1 BYTE
        INC     HL              ;POINT AT R2 BYTE
        LD      B,(HL)          ;PUT FCB R2 BYTE INTO B
        PUSH    BC              ;SAVE BC ON STACK
        LD      (HL),A          ;ZERO FCB R2 BYTE
        LD      C,21H           ;GET RANDOM READ FUNCTION CODE
        CALL    NEXT            ;EXECUTE IT
;
;       NOW RESTORE ALL THE ORIGINAL USER INFORMATION
;
        POP     AF              ;PULL R0, R1, AND R2 INFO FROM THE STACK
        POP     BC              ;*
        POP     HL              ;PULL THE FCB ADDRESS FROM THE STACK
        LD      (HL),C          ;PUT THE ORIGINAL RANDOM READ INFO BACK
        INC     HL              ;INTO THE FCB
        LD      (HL),B          ;*
        INC     HL              ;*
        LD      (HL),A          ;*
        LD      A,1             ;RESTORE THE ERROR CODE
        POP     HL              ;CORRECT STACK
START8
        LD      SP,(USERSAVE)   ;RESTORE USER STACK POINTER
        RET                     ;AND BACK WE GO
USERSAVE DW     0
        DS      64
MYSTACK
        END
```

End Listings



SAVVY PC, Version 4.0

Company: Excalibur Technologies Corporation, 800 Rio Grande Blvd. NW, Albuquerque, NM 87104, (505) 242-3333
Computer: IBM PC and most compatibles
Price: $395
Circle Reader Service No. 131
Reviewed by Steve King

SAVVY PC is misrepresented—Excalibur Technologies advertises its product as "the artificial intelligence data base," but the data base manager constitutes only a small part of the total contents of the SAVVY PC package. SAVVY is actually a complete programming language unto itself. Excalibur Technologies seems to have included the SAVVY PC data base manager more as an example of the SAVVY programming language's capabilities. Don't get me wrong; the SAVVY PC data base manager is as powerful as any I've tried, including KnowledgeMan. But the SAVVY programming language is the heart of the whole SAVVY PC package.

Fourth and Fifth Generation Languages

The mainframe world is undergoing a computer language revolution that partially parallels a similar change in the mini- and microcomputer worlds. Fourth generation languages are replacing the mainframe world's liturgical high-level programming language, COBOL.

From a microcomputer world view, it appears anachronistic that a basically unstructured language, such as COBOL, continues to enjoy extensive use for applications programming. (Of course, the microcomputer world also continues to suffer with Microsoft BASIC in its many flavors.) But the existence of so much mainframe COBOL code makes it economically unfeasible to discard the language in favor of such structured languages as Pascal or C, which have become so popular for mini- and microcomputers. However, fourth generation languages (FGLs) are now helping to solve this mainframe problem: they allow programmers to develop application programs much more rapidly than high-level languages, and moreover, many mainframe FGLs can access COBOL data files.

What is an FGL? It's the procedural and query language supplied with a data base management system (DBMS); examples are dBASE II or KnowledgeMan. FGLs probably began as macros to expedite data base management procedures, but many of them have evolved into fairly complete programming languages. Just as the dBASE II language was the first to gain wide use and popularity in the microcomputer world, the SQL (pronounced "sequel") language, developed by IBM as part of the SQL/DS DBMS, was the first to gain wide use and popularity in the mainframe world. The KnowledgeMan version of SQL is modeled after the IBM mainframe version.

When I first received SAVVY PC, I expected to find another DBMS with its accompanying FGL. Instead, I found a highly structured and modular language that uses spoken English terminology. Excalibur Technologies calls SAVVY "a natural language programming environment and set of tools for solving data processing problems [by] using natural language programming." The Excalibur folks wrote the SAVVY PC data base manager in the SAVVY programming language (from here on referred to as SAVVY). SAVVY is not quite a true "fifth generation" language, which would allow a spoken English interface with a computer ("Hey, computer, baby, look in the name and address data base and get me the phone number for Dr. DooBy's ComPuter JunkYard and HamBurger EmPorium . . ."), but it's as close to that goal as any programming language I've seen yet.

Pattern Recognition

Pattern recognition capabilities, one of the foundation structures of the artificial intelligence world, allow Excalibur Technologies to tout SAVVY PC as "the artificial intelligence data base." The folks at Excalibur structured SAVVY around their Adaptive Pattern Recognition Processor (APRP). This 4K program core recognizes the commands you give SAVVY PC. If you've made a typing error or used improper syntax with a command, the APRP will either correct your error or suggest changes to the command by providing a menu of possible correct choices.

The SAVVY APRP is fast. Excalibur claims that it does not look up each word in the SAVVY dictionary (more about that later) as you key entries but uses an algorithmic trick to provide immediate response to entry errors. In fact, development of the APRP preceded the SAVVY language and SAVVY PC DBMS.

You may turn off the APRP and switch SAVVY from the free mode to the literal mode at any time. When you write a program with SAVVY, the manual recommends that you work in the literal mode so the APRP won't search for words it doesn't know. Until I got used to SAVVY, I found myself frustrated several times when I forgot to turn off the APRP. SAVVY often rewrote what I thought was a new word because the program considered my word a typing error.

The second contribution that the artificial intelligence world has made to the application software world is the natural language interface. The SAV-

VY RETRIEVER reportedly allows you to query a SAVVY PC data base using plain old English. I had hoped to describe how RETRIEVER works and to evaluate it for this review, but Excalibur didn't release it in time for the Dr.'s publication deadline.

The Data Base Manager

The SAVVY PC package comes with a thick, IBM-sized, detailed reference manual/slide case for SAVVY and a booklet for the SAVVY SAMPLER to introduce you to the DBMS. You also receive a master program disk and the sampler disk. Both disks contain the files SAVVY.COM, SAVVY.OVR, and DATABASE.SVY.

Excalibur wrote SAVVY PC in MMSForth (this despite John Dvorak's statement in his *InfoWorld* column, "Inside Track," that no decent program was ever written in Forth). SAVVY PC shows its Forth heritage by storing data base procedures, report formats, and the data files in DATABASE.SVY. Although the program does not use the DOS environment, DATABASE.SVY resides on a standard DOS format disk, and SAVVY PC can import and export standard DOS files. When you write a SAVVY program that doesn't use the data base manager, SAVVY PC creates a separate storage file that you name (FILENAME.SVY).

After SAVVY boots and presents its sign-on messages, it displays the prompt: "What would you like me to do now?" You may either enter the menu system by typing MAIN MENU or enter direct commands to SAVVY. The data base manager booklet and sampler disk take you through the basics of using the DBMS and generating reports. Because the DATABASE.SVY file on the sampler disk contains almost 213K, you barely have enough disk space to finish the examples before running out of storage room. SAVVY gives you adequate warning when you are about to run out of disk space. It also provides the JANITOR command to perform general disk housekeeping chores and to compact your data. However, you can avoid the disk space problem by copying DATABASE.SVY to drive B: and modifying the DEMO.BAT batch file to reflect this change.

When you tell SAVVY PC to generate a new data base, the program provides a blank form on the screen and allows you to structure the layout of your data entry screen by "painting" the screen. SAVVY PC also guesses at your data types (i.e., ANSWER, COMPUTED, DATE, GENERAL, INTEGER, MONEY, NOTE, NUMBER, PHONE, SSN, STATE, or ZIP) when you create the data entry screen. Later, the program allows you to clean up things by detailing data types, field length, and limits for data entry. You may also build relationships with other data files and specify whether data entered into your new data base will update related data bases. After you complete your design work, SAVVY PC takes from two to six minutes to generate the data base. All SAVVY's activities scroll past you as it works. Although not documented in the SAVVY manual, if you press <Ctrl><Num Lock>, you can pause SAVVY's activities to determine exactly what the program is doing to create your data base—a good tool to assist in learning the SAVVY language.

[A warning about another undocumented feature: <Ctrl><Break> exits SAVVY in a hurry. Once, while reviewing a program I was working on, I accidentally pressed <Ctrl><Break> instead of <Ctrl><Num Lock>. After some disk activity, SAVVY dumped me back to DOS. SAVVY had closed the open files, so no harm was done. However, I experimented further and found that if I hit <Ctrl><Break> while SAVVY was building a new data base, all my work would be lost.]

As time has permitted over the last six months, I have been developing a ranch management package for my wife, who is a horse trainer. I had been using KnowledgeMan (KMan), which allows much quicker and easier program development than a high-level language does, but hadn't found the time to design simplified data entry forms. KMan's traditional data base data entry format reinforced my wife's dislike of computers. She also had trouble phrasing data base queries with KMan's SQL language. After her first few frustrating attempts with the KMan ranch management package, she refused to use the program.

When I received SAVVY PC, I decided to give the ranch management project one more try—but with SAVVY PC instead of KMan. SAVVY PC simplifies the data base design process (you construct the entry screens as you structure the data base), which also speeds up the whole programming effort. I completed the SAVVY PC version of the ranch management data base in less time than it took to finish the KMan version only partially. After a great deal of coaxing, my wife tried the new program. She was pleasantly surprised. Although she doesn't dislike computers any less, at least now she uses the data base I designed for her.

SAVVY Programming

SAVVY comes with a dictionary of more than 220 words. The dictionary does not contain just the usual data base terms; it also holds all the available SAVVY commands, items, tasks, and functions. When you write a SAVVY program, you use existing SAVVY words to create new words for the dictionary; i.e., you may extend SAVVY as you would extend C or Forth. Because SAVVY words may contain up to 80 characters, you can be fairly descriptive when creating new terms.

Commands simply represent direct orders to SAVVY. However, the APRP affects the way you interact with SAVVY. For example, after you give the command COPY and press <Enter>, SAVVY responds with "COPY the _ to _." The cursor is positioned at the first underline (underlines are displayed as reverse video blocks on your CRT). After you type the name of the item you wish copied and press <Enter>, SAVVY moves the cursor to the second underline. Type the second item and <Enter>. Items may be either constants or variables; SAVVY doesn't differentiate between the two. The final result could indicate: COPY the FIRST CONSTANT to LAST VARIABLE.

Tasks and functions contain groups of commands that perform specific operations (they may also contain other tasks and functions); these groups provide SAVVY with its modular programming approach. Tasks and functions are identical except that a task stands on its own and a function requires operands or arguments to perform its chores.

For example, when SAVVY created the ranch management data base pro-

gram for my wife, I had specified that the horse's AGE field be calculated. My wife wants the data base to calculate the horse's age automatically after she enters the animal's date of birth. (By convention, all horses become one year older at the first of each year, rather than on their actual birth dates; this custom greatly simplifies the calculation.)

While watching the screen as SAVVY generated the ranch management data base, I saw that the program created a task called RANCH MANAGEMENT, COMPUTE THE AGE. After SAVVY finished creating the data base, I told it to detail the RANCH MANAGEMENT , COMPUTE THE AGE. Because SAVVY showed that the task contained only the line "COPY the '' to AGE," I used the SAVVY Task Editor to create the actual age calculation task as follows:

```
Task RANCH MANAGEMENT ,
   COMPUTE THE AGE
   1 COPY the DATE OF BIRTH to
     BIRTHDATE
   2 TAKE the first 6 characters from
     BIRTHDATE
   3 COPY the BACK to BIRTH YEAR
   4 GET DATE and time into TO-
     DAY'S DATE
   5 TAKE the first 15 characters from
     TODAY'S DATE
   6 COPY the BACK to THIS YEAR
   7 SUBTRACT the BIRTH YEAR
     from THIS YEAR
   8 COPY the DIFFERENCE to AGE
End.
```

Each time you enter a word that is not in the SAVVY dictionary (such as BIRTHDATE or BIRTH YEAR), the program informs you that it does not know the word and asks if you wish to define it. Answer "yes" and continue to enter your program. SAVVY automatically numbers the lines. Because no task or function may be more than 100 lines long, you are forced to write programs in modular blocks and to structure them in a Forth-like top down fashion.

In the above example, line 1 copies the animal's birthdate to a new item. Because you enter BIRTHDATE in the form of DD/MM/YY, line 2 can split it into the FRONT portion, DD/MM/, and the BACK portion, YY, to obtain the BIRTH YEAR. Lines 4 and 5 obtain the TODAY'S DATE from the operating system; the GET DATE command also retrieves the system time. Because this command uses a HH:MM:SS DD/MM/YY format, you must cut the TODAY'S DATE variable after the fifteenth character to obtain THIS YEAR. Line 7 performs the calculation, and line 8 copies it to the AGE.

After you finish writing a task or function, press <Alt><Q> and SAVVY will compile it. The program informs you of the nature and location of any syntax errors that the task or function may contain.

Programming in SAVVY is similar to programming with Forth. Basically, you extend SAVVY and create new tasks and functions with the existing SAVVY words, then SAVVY compiles the new tasks and functions. When your program runs, SAVVY interprets the compiled words one by one. But unlike Forth, SAVVY's terminology is easy to remember and understand because it's plain old English. You also don't have to worry about manipulating Forth's stack or reverse Polish notation.

On the Negative Side

The only mathematical functions that SAVVY contains are add, subtract, multiply, and divide. If your programs require higher math functions, you will have to write them yourself.

SAVVY contains no way of adding comments or remarks to programs. Because of SAVVY's natural language terminology, I'm sure Excalibur Technologies didn't consider remarks necessary. But remarks often make it easier to understand long tasks or functions. However, SAVVY lends itself well to pseudocoding, which greatly simplifies program design.

Like its parent Forth, SAVVY makes no provisions for arrays and, as stated earlier, doesn't use a stack as a partial substitute for arrays. However, the program contains such extensive string-handling functions for character and bit manipulation, you probably won't need arrays. Unlike BASIC's RIGHT$, MID$, LEFT$, and so on, SAVVY's string-handling commands are easy for even novice programmers to understand and use.

As massive and as well written as SAVVY's documentation is, there's just not enough of it. You must watch SAVVY build data bases to learn the details of writing programs with the language. The data base manager booklet contains only 4½ pages about extending a SAVVY-generated data base. Also, the manual contains only the particulars of programming with SAVVY and supplies no information about integrating programs into SAVVY PC-created data base programs.

A member of Excalibur Technologies informed me that a 600+ page tutorial for an earlier version of SAVVY exists. The company is updating the tutorial to meet the requirements of the present version of SAVVY and will release the book as soon as possible.

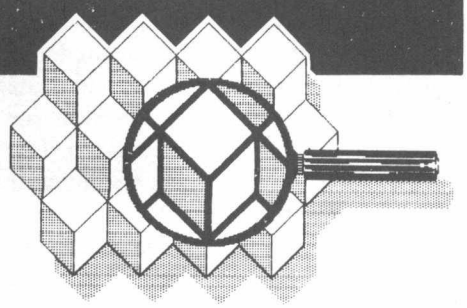
Where to Next?

Because SAVVY also contains a fairly complete set of communications commands, a friend and I plan to write a remote bulletin board system (RBBS) with SAVVY. Most IBM PC RBBSs use variations of the Capitol Computer Club (CPC) RBBS, which was written in BASICA. I have seen several variations (A – D) of the CPC RBBS, version 12.3, on different systems and have also seen announcements that version 13 is imminent. An RBBS SYSOP (SYstem OPerator) recently confided in me that although CPC RBBS has been evolving for over two years, it still contains bugs. My friend and I believe that we can develop a SAVVY RBBS in just two or three months; almost any FGL, and particularly SAVVY, reduces programming time extensively.

I have pretty much stopped programming with BASIC, Turbo Pascal, and KnowledgeMan and switched to SAVVY. When the tutorial becomes available, I believe SAVVY will become one of the easiest ways to teach novice computer users about the black art of computer programming. When they reviewed SAVVY PC for the massive data base comparison series that appeared in *PC* Magazine, John F. and Barbara E. McMullen stated that programming in SAVVY PC is fun—and they were right.

DDJ

OF INTEREST



by R. P. Sutherland

Turing's Man

A reading of J. David Bolter's book ***Turing's Man: Western Culture in the Computer Age*** has put our issue on artificial intelligence in a larger context for me. *Turing's Man* is a study of the impact that electronic logic machines are having upon our culture.

Bolter traces the history of what he calls "defining technologies" from ancient Greece until today. He does this by showing the correspondence between technologies and philosophical/theological metaphors. He points out, for example, that the Greeks, who were craftsmen, had gods who were potters and spinners. The poet who described the Fates as spinning the threads of human lives was influenced by the fact that in ancient Greece spinning was a defining technology. Plato compared the physical universe to a spindle. Likewise, during the Middle Ages the weight-driven clock provided a new metaphor for the motions of the planets. Descartes' mechanistic descriptions of the world were compelling to his contemporaries because they were familiar with clocks and gears. With the rise of the steam engine in the nineteenth century, the earth was compared to a large heat engine.

In the twentieth century's headlong rush to make a machine think like a human, we have defined the human in terms of a machine. Arguing for a synthesis of man and computer, Bolter proposes to replace the term "artificial intelligence" with "synthetic intelligence." Bolter concludes that Turing's man is the most complete synthesis of the human and the technological in Western history, and he predicts that the computer as a defining technology will give way to a technology that we cannot yet imagine. We may not realize Turing's 1950 prediction by the end of this century, but the fact that we are thinking along these lines implies that the computer as a metaphor for the way we understand ourselves is firmly established.

Bolter has graduate degrees in the classics as well as in computer science. *Turing's Man* (ISBN 0-8078-1564-0) is published by The University of North Carolina Press, Post Office Box 2288, Chapel Hill, NC 27514. **Reader Service No. 101.**

AI

Solution Systems, the publisher of Prolog-86, is sponsoring a **contest for programmers** interested in artificial intelligence. The purpose of the contest, "Artificial Intelligence Concepts," is to generate material to help Prolog programmers and to promote Prolog. Programmers who wish to enter should develop expert systems and/or natural language processing systems designed to illustrate methods and concepts of Prolog programming. Applications with a strong commercial potential are discouraged. The application judged to be the most significant will receive a cash reward of $1000.00. All participants will receive diskettes containing all contributions made by other contestants. Submit applications on 5¼-inch MSDOS diskettes to Solution Systems, 335 Washington Street, Norwell, MA 02061, accompanied by a one-page description. All submissions must be in by April 30, 1985. The winners will be announced in June 1985. For details call Solution Systems at (617) 659-1571. **Reader Service No. 103.**

Sun Microsystems has expanded the use of its workstations by adding AI packages to its Third Party Referral Program. Quintus Computer Systems, Inc., Palo Alto, CA, has released Quintus Prolog, which includes an interface for calling C programs. Lucid, Inc., Palo Alto, CA, is offering Lucid Common Lisp for developing portable knowledge-based applications on Sun Workstations. The Knowledge Engineering System (KES) from Software Architecture and Engineering, Inc., Arlington, VA, is an integrated set of utilities designed to allow newcomers to the field of AI to design expert systems. Smart System Technology, McLean, VA, offers DUCK, a high-level Lisp-based applications development environment for designing systems based on logic reasoning. JoAnn Kahn is the manager of Sun's Third Party Programs and can be reached at Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043. **Reader Service No. 105.**

A Canadian company has developed a micro-version of **MPROLOG**, which, it says, opens the door to low-cost AI applications. MPROLOG (Modular Programming in Logic) allows a programmer to create a major program for a mainframe computer on a microcomputer by working with modules. So, by working on an IBM PC, one can develop an AI application one module at a time and execute the full application on a VAX or IBM mainframe or a 68000-based machine. MPROLOG for microcomputers is $950.00 for a paid-up license or $260.00 a month to lease. For further information: Logicware Inc., 1000 Finch Avenue West, Suite 600, Toronto, Ontario M3J 2V5 (416) 665-0022. **Reader Service No. 107.**

Software

"The arcade game is all but dead," according to Synapse Software President Jon Loveless who has just released two **electronic novels**: Essex and Mindwheel. I haven't "read" one yet, but they are supposed to put the user in the center of the action so that the user's

decisions determine the outcome of the plot. The novels are to be marketed through bookstores as well as software stores for less than $50.00 each and are available for IBM PC, Atari, Apple IIe, Apple IIc, Macintosh, and Commodore. Synapse Software Corporation is located at 5221 Central Avenue, Richmond, CA 94804 (415) 527-7751. **Reader Service No. 109.**

Speaking of novels, **RACTER** is a software program that has authored a published book: ***The Policeman's Beard is Half Constructed***. The book features stories, essays, poems, and interviews created by *RACTER*. A sample poem: "More than iron, more than lead, more than gold I need electricity./I need it more than I need lamb or pork or lettuce or cucumber./I need it for my dreams." The creators, Tom Etter and Bill Chamberlain, claim that it produces original conversation that is also grammatical, without the use of stock phrases. *RACTER* is available for MSDOS or CP/M for $69.95 from John D. Owens Associates, Inc., 12 Schubert Street, Staten Island, NY 10305 (212) 448-6298. **Reader Service No. 111.** *The Policeman's Beard is Half Constructed* (ISBN 0-446-38051-2) is published by Warner Software/Warner Books, 666 Fifth Avenue, New York, NY 10103. **Reader Service No. 113.**

A computerized thesaurus called **Synonym Finder** is available for MSDOS running WordStar or Multimate and CP/M 80 running WordStar. Synonym Finder sells for $124.95 from Writing Consultants at 11 Creek Bend Drive, Fairport, NY 14450 (716) 377-0130. **Reader Service No. 115.**

Word Publishing has announced the release of **NARNIA**, a video game based on the *Chronicles of Narnia* by C. S. Lewis. NARNIA is available for Apple II and Commodore 64 computers. Availability: Waldenbooks and Computerland stores, or send $39.95 to Word Inc., Waco, TX 76703. **Reader Service No. 117.**

Thoroughbred Handicapper II is a computerized system that analyzes thoroughbred horse races and helps to predict the winners. The handicapping system is based on a statistical analysis of horse races. Thoroughbred Handicapper II is available for the IBM PC family for $49.95. Contact The Software Bottling Company of New York, 29-14 23rd Avenue, Long Island City, NY 11105 (718) 728-2200. **Reader Service No. 119.**

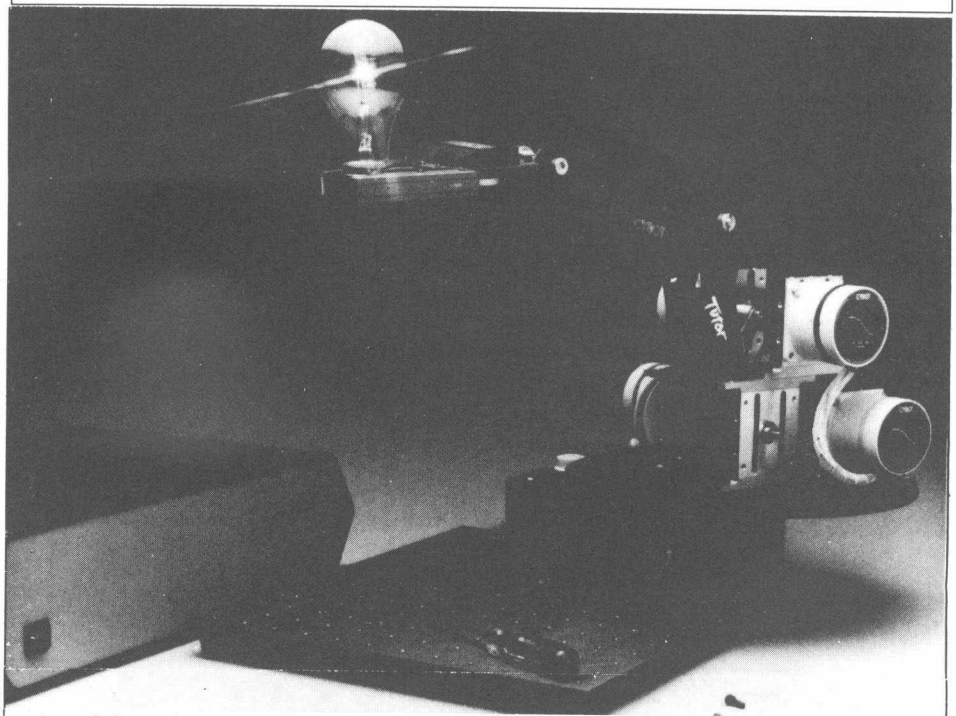
A brain wave to electronic speech system for the ZX-81/TS-1000 and Apple computers permits brain wave control over robotic devices. We cannot verify this claim, but for more information write to Rosetronix, P.O. Box 7464, Chula Vista, CA 92012.

Hardware

The **Integral Personal Computer** from Hewlett-Packard is a 68000 Unix-based machine. This portable computer has a built-in ThinkJet printer, a 3½-inch (710K) disc drive, a 9-inch electroluminescent display, and a full-size keyboard. Price: under $5,000.00. Hewlett-Packard, P.O. Box 10301, Palo Alto, CA 94303.



The Spectravideo ***Bondwell 12*** *computer runs CP/M 2.2 on a Z80a CPU. The price of $995.00 includes a voice synthesizer, two floppy disk drives, and bundled software. Christopher Chang, President, Spectravideo, 3300 Seldon Court #10, Fremont, CA 94539 (415) 490-4300.* **Reader Service No. 125.**



A modular robot called ***Tutor*** *is available as a kit from Cybot, Inc. for $3,395.00. The three circuit boards used to control the Tutor robot are designed to the S-100 standard. Cybot. Inc.. 733 7th Avenue. Kirkland, WA 98033.* **Reader Service No. 129.** **DDJ**

HUMAN INTERFACE DESIGN

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

#102 April 1985

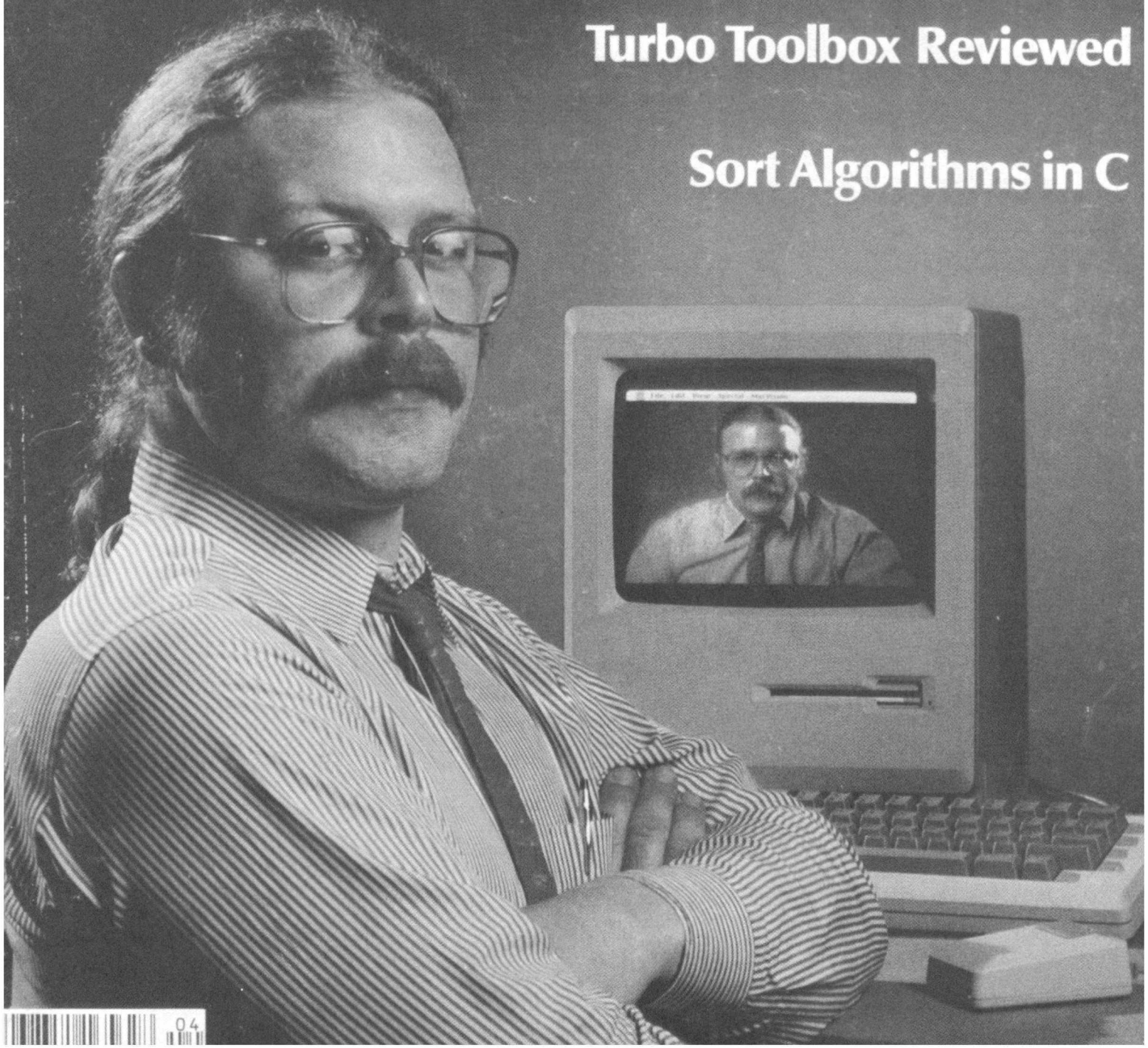
$2.95 (3.50 Canada)

80286 Xenix

Three Printer-taming Programs

Turbo Toolbox Reviewed

Sort Algorithms in C



Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Randy Sutherland*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Polly Koch*
Typesetter *Jean Aring*

Production
Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover *Tom Upton*

Advertising
Advertising Director *Stephen Friedman*
Advertising Sales *Walter Andrzejewski, Shawn Horst, Beth Dudas, Michele Beaty*
DDJ Classifieds *Alice Abrams*
Advertising Coordinators *Lisa Boudreau, Jay Horvath*

Circulation
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Circulation Assistant *Kathleen Boyd*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounting Coordinator *Mayda Lopez-Quintana*
Accounts Payable *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank. For subscription problems, call: outside of CA 1-800-321-3333; within CA 1-619-485-6535.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

April 1985
Volume 10, Issue 4

CONTENTS

Dr. Dobb's Board of Referees

Dennis R. Allison, Stanford University
Ian Ashdown, Professional Engineer
Joe A. Barnhart, Hewlett-Packard
S. M. Bellovin, Bell Laboratories
Robert Blum, *DDJ* Contributing Editor
Patrick Burnstad, Lands End Computers, IEEE, SVCS
Wayne Chin, Hewlett-Packard
David D. Clark, Pennsylvania State University
David E. Cortesi, *DDJ* Contributing Editor
Keith Coye, PicoNet
Mel Cruts, Trilogy
Bob Desinger, Hewlett-Packard
Ray Duncan, Laboratory Microsystems, Inc.
Michael T. Enright, Cal Omega, Inc.
Z. Z. Fisher
Jim Fleming, UNIR
Angel Gomez, Tele Comp, Inc.
Kim Harris, Forthright Enterprises
James E. Hendrix, University of Mississippi
William P. Hogan
Allen Holub, *DDJ* Contributing Editor
David W. Hopper, Senes Consultants Ltd.
John R. Johnson, Professional Microware, Inc.
James Christopher Johnston, NASA Lewis Research Center
Michael P. Kelly, Design Software
John P. Keys, MicroSoft
David Kirkland, Baker and Botts
Richard G. Larson, University of Illinois at Chicago
Ben Laws, North Texas State University
Mohamed el Lozy, Harvard University
Patrick Lynch, Tandem Computers, Inc.
Leonard K. Nicholson, ADM,IEEE
Donald L. Rastede,
Richard Relph, ELXSI
John Rogers, Fortune Systems
Doug Rosenberg, Westminster Software
Prof. David Ross, University of Iowa
Darryl E. Rubin, Rolm Corporation
Joseph Sharp, Micro Science Associates
Henri Socha, SochaLogical Research
Charles T. Springer, Information Appliance, Inc.
John Taber, IBM
Scott D. Thomas, Informatics General Corp.
Allen Tigert, Kontron Medical Electronics
Matthew H. Trask, Phoenix Computer Products Corp.
Bob Wheeler, Custom Hardware
Charles Wilde, Aton International, Inc.

Cover Credit

Tom Upton created our cover this month with the help of MacVision, a graphic digitizer from Koala Technologies Corporation.

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

MONTY Plays Scrabble: What We Can Learn about Human Factors Engineering from a Game-Playing Computer *by Terry A. Ward* — **18** Designers of software can make it easier for the end user to interact with their programs. Some principles of human factors engineering can be learned from a small computer that plays Scrabble **(Reader Ballot No.192)**.

Shortcut to SCISTAR: For Prowriter Users *by Amer Nelson* — **24** How to print special characters with an Osborne 1/ Prowriter system **(Reader Ballot No. 193)**.

fx80char: A Character Editor for the Epson FX-80 *by David D. Clark* — **28** A screen oriented editor for developing alternate character sets; commands for downloading the new characters to the printer **(Reader Ballot No. 194)**.

A Magic Mushroom for the Epson Alice *by Gary Palmer* — **58** An anthropologist prints vertically extended phonetic symbols for transcriptions of the stories of the Coeur d'Alene Indians **(Reader Ballot No. 195)**.

Let's Mouse Around *by Vincent and Ronald G. Parsons* — **74** The Microsoft Mouse with Turbo Pascal **(Reader Ballot No. 196)**.

COLUMNS

Realizable Fantasies *by Michael Swaine and Bob Albrecht* — **12** Liberating the essence of the Mac from Apple's closed architecture **(Reader Ballot No. 190)**.

Dr. Dobb's Clinic *by D. E. Cortesi* — **14** DRI phases out unconfigured versions of CP/M; critique of Modula-2; shadowy printing; well- and ill-behaved functions **Reader Ballot No. 191)**.

Software Designer *by Mike Swaine* — **86** Dave Winer of Living Videotext and Thom Hogan of Business Software discuss human factors engineering and software design **(Reader Ballot No. 197)**.

C Chest *by Allen Holub* — **90** Reasons for not using the Shell sort in K&R; qsort.c, a general purpose Quicksort routine **(Reader Ballot No. 198)**.

16-Bit Software Toolbox *by Ray Duncan* — **102** ED/ASM-86; a TEE filter for MSDOS 2.x; MacAsm; 80286 Xenix 3.0 **(Reader Ballot No. 199)**.

CP/M Exchange *by Bob Blum* — **112** A patch to enable PIP v. 1.5 to accept null command lines from a submit file; a stand-alone $$$.SUB file generator improves batch procedures **(Reader Ballot No. 200)**.

DEPARTMENTS

Editorial **6**

Letters **8**

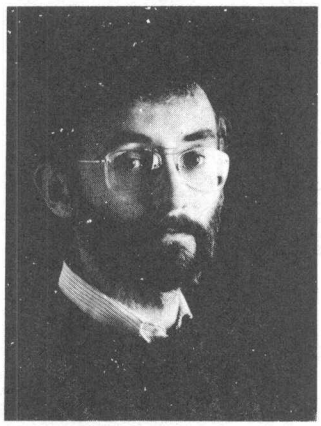
Reviews **80** SPSS/PC; Borland's Turbo Toolbox

DDJ Classifieds **107**

Of Interest **122** **(Reader Ballot No. 201)**

Advertiser Index **128**

EDITORIAL



Changes come and trip all your traps. You subscribe to *Micro* and you get *Dr. Dobb's Journal*. You just get used to thinking of yourself as a "serious computerist" and you find yourself a subscriber to a magazine "for advanced programmers."

With this issue, we welcome several thousand readers of the sadly-defunct *Micro* magazine to *Dr. Dobb's*. *Micro*, like *Microsystems* and *Microcomputing*, died last fall in a bad season for many technical computer magazines.

I'm sorry about *Micro*, but I'm happy to see this influx of readers serious about the 6502 and 68000 microprocessors. We've been publishing more and more 68000 material recently, but aside from an occasional Commodore 64 piece, we've neglected the 6502 programmer. The *Micro* readers present *Dr. Dobb's* with an opportunity to address an audience it's lost touch with in recent years. (For an indication of the kind of 6502 material we have published in the past, see the note at the end of the "Realizable Fantasies" column elsewhere in this issue.)

Welcome, *Micro* readers.

Changes have come to *Dr. Dobb's* as well in the past year. This is the twelfth issue of the magazine published under the *M&T* banner (the hundred-and-second in all). I became editor-in-chief then (and thanks, by the way, to all those I failed to thank in my February editorial), and I instituted some new features in the magazine, including our first operating-systems issue (on *Unix*) *and our* unconventional coverage of the Mac (see our January how-to on home-fattening your Mac or this issue's Realizable Fantasy on liberating the Mac).

While making changes, I've tried not to lose track of what *Dr. Dobb's* was founded to do: to provide software tools for advanced programmers. In 1976, those programmers were hackers and in 1985 they tend to be professional software designers, but after attending last fall's Hackers' Conference I've decided that the distinction is moot.

Changes come thick and fast in the microcomputer industry, but in the thick of it all, some things must hold fast. Without *Micro*, without *Microsystems*, without *Microcomputing*, where does the programmer interested in 8-bit systems go? We promised, in our May and October issues, not to abandon the 8-bit computer, and we haven't. We promised not to forsake public-domain and low-cost software, and we haven't. (We've also pointed out that purveyors of public-domain and commercial software must coexist.)

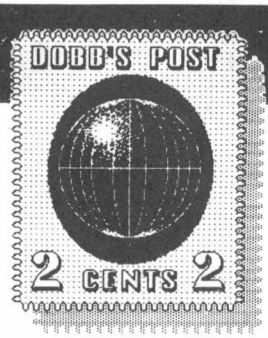
But there are changes yet to come. While we will not abandon our current readers, we'll be exploring new territory in 1985, ranging beyond the fields we know. The only way we can do this is to publish more material that is of broader applicability—not by lowering the technical level, but by giving more emphasis to algorithms, to techniques, to portable software, and less to machine-specific code. I claim that the overall effect will be to raise, not lower the technical level of the magazine without making it harder to understand. We'll see if I'm right.

We'll also give more attention to the craft of software design in 1985, and to the aesthetics of design. If writing utilities produces utilitarian code, then I think that programmers should be writing efficiencies, elegances, beauties in addition to utilities, because we need more efficient, more elegant, more beautiful software. We'll encourage that.

And yes, I deliberately worked into this editorial the themes of my other eleven editorials. I'm sorry; I learned recursion at an early age and was warped for life.

Michael Swaine

Michael Swaine



Chubby Mac

Dear *DDJ*,
I would like to congratulate you for the excellent January [1985, #99] issue of *Dr. Dobb's* featuring "Fatten Your Mac" by Tom Lafleur and Susan Raab. Thanks to that article, I was able to upgrade my Mac for much less than Apple's list price: I paid $200 total or around $11 per memory chip.

Having never seen a hardware technical manual for the Mac that's equivalent to "Inside Macintosh" for software, I found the diagnostics part of the article very useful. While doing the upgrade, I encountered socket connection problems that were solved by following the continuity test procedure mentioned.

Do you know what dealers or customers do with the 128K boards, for those who upgrade their boards the official Apple way? I was thinking that if I decide to pursue this and upgrade Macs for friends and for money, I will probably need some extra 128K boards as a safety net. If you know people who have a 128K board that they are going to throw away, let me know.

I am so happy to have a 512K Mac now that I decided to enter a 1-year subscription to *Dr. Dobb's*. Once again, congratulations to *Dr. Dobb's* and the authors.

Sincerely,
Tho AuTruong
908-B Menlo Avenue
Menlo Park, CA 94025

PUBlic Service

Dear *DDJ*,
A minor correction is needed to our article "CP/M 2.2 Goes Public (November, 1984, #97)". In Listing 2, page 70, the PUBLIC.ASM utility to set or reset the PUBlic file attribute bit contains a bug, discovered by Greg Platt, that prevents it from making very large files PUBlic.

Here's the fix; Update the version equate to read:

```
vers    equ     1$1
```

and insert the following code in the SAMEXT routine preceeding its "ret" instruction:

```
    rnz         ; v 1.1
    inx   h     ;extent is 0,
                  check overflow (s2)
                  ext.
    inx   h
    mov   a,m
    ani   7fh
;
    ret         ;end of SAMEXT
                  routine
```

PUBPATCH—the BDOS patch that implements PUBlic file processing—is unaffected by the bug.

Sincerely,
Bridger Mitchell
Derek McKay
Plu*Perfect Systems
Box 1494
Idyllwild, CA 92349

Dear *DDJ*,
The article "CP/M 2.2 Goes PUBlic" by Mitchell and McKay, in your November [1984, #97] issue was most informative and useful. However, the code text published with it had a few problems. Some specifics from Listing One: the location definitions CKFILPOS and SETSTAT were multiply defined (to be sure, the definitions were the same, but still rather irregular); 'nxbyt:' was referenced both as itself and 'nxtbyte', typo or program error I can't tell, since only the first six characters are frequently used. More serious was the use of a font in which both lower case 'L' and the number one used the same representation. This is common on typewriters, but rare on computer printers. Line 253 is definitely ambiguous as printed, although it can be deciphered in context.

At any rate, I was able to enter the program text, make the necessary corrections, translate it to true Z80 code, and assemble it with the Z80MR assembler from MicroCornucopia. Installation under ZCPR3 required a little research to find the correct addresses, but they were there. Thus I no longer need to have long text files in the same user area as my PW or WS editor; this is a real convenience in the rather crowded directories on my Kaypro10. My sincere thanks to the authors!

Gratefully
Bonnell Frost
5559 Colt Drive
Longmont, CO 80501

Grep Again

Dear *DDJ*,
A belated thank you for grep.c [*DDJ* #96, October 1984], which I finally got around to typing in and compiling on my Osborne 1 with C/80.

C/80 Version 3.0 from The Software Toolworks will not compile grep.c as listed because it has no:

a) linker
b) typedef statement
c) FILE type
d) #defined NULL
e) stdin
f) stdout
g) stderr
h) fgets
i) stdio.h
j) capability to call a function with a different number of

(Continued on page 17)

REALIZABLE FANTASIES

by Mike Swaine and
Bob Albrecht

Liberating the Mac

The Apple Macintosh embodies a frustration and an irony: an irony for anyone who knows the history of the company and observes that it is IBM rather than Apple that is selling an open-architecture machine; a frustration for anyone who knows the sense of power and satisfaction that comes from popping the cover off an Apple II to plug in a card and has come up against the closed design of the Macintosh.

After setting an example for open design with the Apple II, Apple has created in the Mac an appliance computer "for the rest of us," but some of the brightest people feel particularly excluded from that "us." That's unfortunate, since the Mac implements a brilliant user interface using a powerful and well-designed microprocessor. But for experienced programmers, that interface can get in the way, and the bulk of the bandwidth of the microprocessor is swallowed up by the graphics. What Apple has done in producing the Mac is remarkable, but one of the things it has done is to make evident all the possible things it didn't do. Some programmers seem to feel that the Mac has too much potential to be left to Apple, and have put forth various schemes for liberating the essence of the machine for their own version of "us." This month's Realizable Fantasies are all schemes for the liberation of the Mac.

The Hacker's Mac

Lee Felsenstein has been talking for the past six months about something called "the hacker's Mac." The hacker's Mac is not in essence a commercial product or even a proposed commercial product, but a project. Lee's intent appears to be to engage the imaginations of engineers and programmers who may grudgingly admit that they like the Mac, but who know in their hearts that they would have done it better.

The hacker's Mac is a consensus design project with the objective of developing a computer that does all that the Macintosh does and more. Anyone who cares to contribute ideas to the design, may. Clearly that's not a way to develop a commercial product; for one thing, nobody could hope to compete commercially head-on with Apple without Apple's resources and Apple's Macfactory. Well, sure. The endeavor may be, though, a way to learn the design of the Mac so intimately that you can find ways to extend, improve upon the technology. All the Mac experts a year from now will not be Apple employees and alumni. Did Burrell Smith and Andy Hertzfeld and Bill Atkinson get something more than money out of their work on the Mac? Sure, and Lee seems to think that there's no reason that others can't get an education from creating a Maclike computer on their own—or with some help from friends. So he's kicked off the hacker's Mac project.

As Lee has pointed out to me, a monthly magazine is an inappropriate medium for issuing progress reports on an ongoing project, particularly one that could move along as quickly as this one could. I shall report, then, only that Lee introduced the hacker's Mac project at the Hacker's Conference in the Marin headlands last fall and led the *Homebrew Computer Club* in sketching out the technical desiderata of the machine in January. It is February as I write this; suffice it to add that you can follow the progress of the hacker's Mac project on line by calling Gordon French's BBS (the French Connection) at 408-736-6181. By press time, this board should have hacker's Mac info; you can also learn more about the project by sending a self-addressed stamped envelope to Lee at Golemics, 2600 10th, Berkeley CA 94710.

Free the ROM 64

Steve Jasik, a veteran of Control Data and Sorcim, has written a program he calls MacNosy, "the disassembler for the rest of us."

MacNosy (Nosy for short) is an interactive disassembler for 1M Lisa systems and Fat Macs. According to Steve, it lets programmers recover the source code, or a reasonable facsimile thereof, for any Macintosh application program, ROM or "code" type resources on the "System" file. It splits programs into procedures and referenced data blocks, and maintains journal files of its activity for later playback or trading with your friends. Steve brought a copy around to *DDJ* and demonstrated its capabilities, and, whatever Nosy's strengths and weaknesses, it was clear that Steve, like the hacker's Mac consensus designers, is working to liberate the Mac.

Here's how Steve views his project: "Nosy is more of a decompiler than a disassembler. Its purpose is the recovery of source code in a human-readable format that reflects the intent of the code's author. To achieve this goal, it treats the program it is disassembling as a tree of procedures. It does a tree walk to locate the regions that are code; the remaining regions are marked as data. At present it does not gather enough information to determine which data areas are really code (like procedures passed as parameters), and human help is then needed to give this information to it. One then repeats the tree search and continues, until the

program is in its original format, or some reasonable representation."

Steve sees the disassembler as appropriate for anyone who wants to understand 68000 and Macintosh programming techniques by studying others' code and for developers who want to "find out what the ROM routines are doing to or for them." He also sees it as a tool for people who want to remove copy protection code from programs they want to run from a hard disk, and for programmers to find out how easy it would be for someone to remove copy protection code from their programs.

You can get the current version of Nosy for $70 or maybe less from Free the ROM 64 in Menlo Park, California.

Blue Light Special

Then there's the machine everybody's calling the Jackintosh: Jack Tramiel's Atari's ST's version of who the rest of us are. The ST, running Digital Research's Maclike GEM user interface, will be sold in K-Marts at a price that will take desktop icons out of Yuppiedom for the first time. Ron Jeffries (of *The Jeffries Report*) is impressed with GEM and with the ST, and seems to think that the machine will do well. Will it? Will icons play in Hamtramck? Only The Marketplace knows for sure. (And is it true that DRI is convinced that MSDOS machines represent the real market for GEM? Sigh.)

"Liberation" is a slippery word; it has taken us from considerations of open *vs.* closed architectures to K-Mart Blue Light Specials in the Computer Department and Jack "business is war" Tramiel leading a war of interface liberation. Anyway, although Atari's marketing strategies are a different kind of liberation of the Macintosh from reverse-engineering the machine or disassembling the ROMs, the ST is evidence that the user interface that Apple liberated from the workstations of Xerox PARC isn't waiting for the Mac to carry it to the heartland.

Virtual Slots

But is all this Apple-tweaking really necessary? As Tiny BASIC author Tom Pittman said in these pages two months ago, the Mac presents us with a new environment for hacking; if a few years ago you could accept the "closed architecture" of integrated circuits as opposed to discrete components, why not accept today the closed case of the Mac and concentrate on exploring its rich software environment? You can't plug boards into the Mac, but you can add desk accessories. And external enhancements.

And therein (in externals) lies the basis for Apple management's recent claim that Apple has opened the Mac. Apple's new LAN, goes the argument, gives the Mac "virtual slots."

The network is certainly open. Apple has taken the lid off AppleTalk for developers, and many have taken advantage of that fact. 3Com, for example, has developed an AppleTalk-to-Ethernet link. The protocols for AppleTalk are all public. Apple is making the multilevel network open at all levels, in the hope to "spur developers to provide innovative products for AppleTalk." It seems to be happening.

Do virtual slots offset what some see as the Mac's deficiencies as a developer's dream machine? Is Apple the true liberator of the Macintosh? Or is that just Apple's fantasy?

Free-floating Fantasy

And now for something completely different. Everybody wants to define or redefine "hacking." Steven Levy devoted a book to getting a handle on the term; the subsequent Hacker's Conference wrestled with the definition; old-time programmers wonder whether the term has been irrenewably tarred by the brush of illegal-entry pranks and mischief. Perhaps we need to liberate the word from definition. Undefine it. Perhaps this liberating will happen as a matter of course, whatever we do. Perhaps liberating the word "hacking" is some part of what "hacking" (or "to hack") means.

Consider the denumerable set of all possible meanings of the term "to hack." Call the cardinality of this set *n*. Proposition: *n* is the cardinality of the natural numbers, i.e., infinity (in the aleph-one sense). Defend the proposition by enumeration.

Well, let's see. (1) to hack is to undefine "to hack." (2) to hack is to do more with less. (3) um, reader involvement here is welcome.

"Hackers don't grow older; they just get lazy or preoccupied or enmeshed in the bottom line. What was done once can be done again." — *Laran Stardrake.*

Micro Subscribers Please Note:

MacNosy is not the first disassembler for an Apple machine we've published. In 1976 we printed a description and a listing of a 6502 disassembler, both written by Steve Wozniak and Allen Baum. People to whom good ideas come easily tend to be free with their ideas, and Woz, who recently left Apple over philosophical differences, has always been a believer in sharing knowledge. Over the years he has given away several pieces of 6502 code through the pages of *Dr. Dobb's*, including the disassembler, floating-point arithmetic and conversion routines, a (very simple) number-guessing game, and BASIC routines for renumbering and appending.

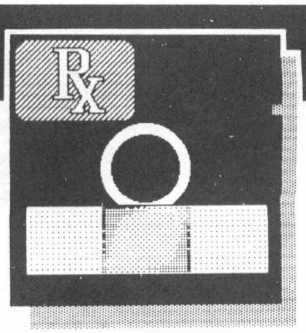
It's unlikely that we'd reprint the articles in *Dr. Dobb's*. We want to run *current* 6502 material, and these pieces may be most appealing for their historical interest (although it's instructive to read Woz's code). But we could put together a reprint package if there's enough interest. We'd have to charge enough to cover the postage and printing and handling—probably three dollars. But we could throw in Andy Hertzfeld's *Lazarus*, a program for resurrecting BASIC programs on the Apple II.

If you'd like to have such a reprint, write to The Woz Papers, c/o this magazine, and let us know. If interest is great enough, we'll do it.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

DR. DOBB'S CLINIC



by D. E. Cortesi

DRI Drops Homebrew Systems

An item in a recent *BUSS* gave us a mild panic. It seemed to say that Digital Research had withdrawn CP/M Plus from sale. We called some people we know in Pacific Grove and found that the news wasn't that bad. CP/M Plus is alive; it's homebrewing that's been given up for dead.

DRI used to sell its various DOSses in generic form: the base code, unconfigured to any hardware, needing a complete BIOS to run. Lots of hobbyists bought CP/M 2.2 that way; we and a few others bought CP/M Plus in the same form.

No more. Today, most of DRI's retail sales are of preconfigured systems (for the IBM PC, mostly) and of languages such as DR LOGO, which need no configuration. The generic packages aren't moving, our source said; they burden the inventory and bring in no revenue. So, DRI is phasing out retail sale of all generic systems. An added-value vendor who wants to package and resell a DOS in quantity may license it, of course.

But sales of unconfigured systems to individuals are ending. Wait, we protested, there are still a few of us who want to roll our own. What about us? Well, DRI doesn't mind you getting a copy of a generic system, but they're determined to shed the inventory burden. They are "talking to" third parties who may want to take over the sale of generic systems as a sideline, but nothing has been settled.

If you think you might ever want to buy a single copy of CP/M Plus or CP/M 68K or Concurrent CP/M in generic form, write to the marketing folks at Digital Research (P. O. Box DRI, Monterey, CA 93942) and tell them. They think all the homebrewers have bought PCs and retired.

Criticism of Modula-2

One of the oldest and most vigorous SIGs sponsored by the ACM is SIGPLAN, the Special Interest Group on Programming Languages. Its monthly newsletter, *SIGPLAN Notices*, has carried some of the most influential language papers ever published; yet because it isn't a formal journal, it also carries some of the funniest and some of the most controversial papers as well.

One item in the December 1984 issue falls into the latter category. "Some Concerns About Modula-2" by David V. Moffat of North Carolina State University appears to be intended as an antidote to some of the gushier praise lavished on Modula-2 of late. Some of you may not have access to *SIGPLAN Notices*. Because we wouldn't want you to miss out on a solid linguistic controversy, we thought we'd summarize Moffat's points here.

His first point is that, as a result of its relegating all I/O to procedures written in the language itself, Modula-2's I/O syntax is very low level compared to other languages. It "forces the programmer to use a different procedure for each type, for each number of parameters, and for each default," Moffat writes, whereas languages that contain built-in I/O statements, however inconsistent (think of Pascal's pseudo-procedure WRITE), are a lot easier to use.

He says that the standard I/O modules are, in his opinion, "a hodgepodge of oversights, inconsistencies, duplicate names, and hasty patches." And he gives examples: you cannot write real numbers to the terminal while accessing a file with the standard module InOut; the I/O modules export many similar names, "so the simple Pascal . . . statement 'write(s)' becomes 'Terminal.WriteString(s).' "

Moffat's third point is an involved analysis of the portability of modules as object code. Because of the linker's version checking, and because any implementor will have recompiled the standard I/O modules one or more times, he concludes that the modules will not port, which forestalls substantial traffic in precompiled modules. But he doubts that distribution of modules in source form will work either, partly because developers are reluctant to release source but also "because the 'standard' modules are not really standard. There is no guarantee, and indeed no real expectation, that a procedure in one version of a module will have the same syntax or semantics—or even exist—in another version of that 'same' module."

Moffat notes the absence of built-in provisions for varying strings, for sets large enough to cope with the alphabet, and for high-precision numeric types. And although he admits that "all of these types can be simulated in modules, an activity that is too often beside the point of the program to be written, although a hacker's delight," their absence as efficient built-in types and the presence of "lots of features for systems programming" make it "plain that Modula is a systems programming language, not a general purpose language."

Because the "effective size of the language is very great" (Moffat notes 40 reserved words, 26 standard identifiers, 130 identifiers in the standard modules, and "some implementations provide yet 200 more!"), he deems Modula-2 confusing and ineffective as a teaching language. Because it's neither general purpose nor good for teaching, he concludes that "Modula-2 is not the successor to Pascal

or a substitute for Pascal," as many have called it. He points to UCSD Pascal as "a genuine extension and improvement of Pascal."

We'd enjoy seeing your informed comments, pro or con, on Modula-2 and will print any that seem enlightening. However, if you feel moved to debate with Moffat specifically, you should read the original paper and direct your comments first to *SIGPLAN Notices*; we're just some interested bystanders.

Headless No More

A few readers wrote to commiserate with us regarding the decapitation of our Shugart 860, dramatically described in the December 1984 issue. To our surprise, they didn't think much of the Tandon half-height either. Rex Backus of St. Helena, CA, made a typical remark: "I really wonder if you were just lucky to get a working 848-2 with only one try or if Tandon is doing a better job with those drives than they did two years ago. . . . I sent two drives back for replacement before I got a pair to work." The answer is probably yes, Tandon has improved; their latest drives, we hear, are quite different from their first efforts.

However, these comments made us wonder. How many other people are suffering from such hardware problems? More to the point, why are they keeping it a secret? That's what this column is for; we *love* to collect anecdotal reports of the reliability (or lack thereof) of hardware and software. How are the Qume and Mitsubishi drives?

One reader offered us more than sympathy. Chuck Schuetz called to say that when he'd worked for Shugart he'd had a hand in the design of the 860; that he knew exactly what we were talking about; and that he'd like to fix our busted drive. He took the headless drive for a weekend and returned it working perfectly.

Chuck told us a lot about the 860. He maintains that it ended up as a really excellent drive in its final versions but allows that it took a number of MLCs (machine-level changes) to reach that state. For example, the earlier versions included a small boss to force an intentional bow in the diskette jacket; this boss had to be ground flat in later versions. In fact, most of the drive's problems related to the positioning of the diskette jacket in the frame. The drag and varying speed that were our original problems most likely were due to a maladjustment that let the eject slider bear against the corner of the jacket when the drive was closed.

Schuetz is willing to advise other people with distressed 860 drives. He can get spare parts at reasonable cost and can handle a limited number of repairs as his spare time permits. He'd prefer your initial contact with him to be by mail; write him at 3675 Desoto Avenue, Santa Clara, CA 95051.

Shadowy Printing

OK, the Intern needs more help. Our Diablo 1650, sturdy companion through zillions of words, is getting

flaky. The figure below shows the problem. It shows lots of shadow-printed Os. As you'll see if the reproduction is good, the amount of shadow varies from letter to letter. Some overstrikes are offset a little, some a lot.

The problem seems to be a slight random variation in the carrier's horizontal position. It shows the worst in shadow printing, but it can also be seen as a slightly "dirty" look in proportionally spaced text. The problem is not in radial control of the typewheel, because it can occur when the typewheel isn't turning (as in the figure), and it isn't a typewheel problem, because it affects all typewheels including new ones. It's a problem of rapid, incremental, horizontal movement; it probably has an inertial component, since running the printer at half speed (300 baud not 1200) improves the output.

We took it to a local Xerox America service franchise. They replaced the carriage and the horizontal drive motor. They claim they swapped out all the boards. At our urging, they checked the power supply under load. They kept the printer two weeks. They presented a large bill and said, "There's nothing wrong with it now."

Baloney. Shadow printing still looks awful. Are there any devilishly good printer techs out there who could tell us why?

Well-Behaved Comments

Back in October we outlined a method by which a C compiler could (1) detect which functions absolutely couldn't be recursive and (2) allocate their local variables in common pools of static storage. Several readers were moved to comment on the scheme.

David S. Tilton, whose letter last summer started the whole train of thought, found a hole in our scheme. Although we set out to identify all the well-behaved functions (WBFs), we ended up identifying only one of them. Tilton says, "Consider three functions which each make one call. A calls B, B calls C, and C calls B. Clearly B and C are part of a recursive loop and are therefore not WBFs. It is also clear that B and C constitute a closed set; the call from A to B cannot be part of a recursive loop, and A is definitely not recursive." But our method would not classify it as a WBF.

In part, that's because we merged two distinct kinds of ill-behavior. We viewed as "potentially recursive" functions that called themselves or each other (functions B and C in Tilton's example) and also functions that called external or pointer-based names. A compiler in principle could recognize the limited scope of the former (although neither we nor Tilton can think of a simple algorithm to do so), but it's a peculiarity of C that the latter behavior is intractable. If A calls B, B calls C, and C is declared external, then we must assume that the external function C *might* call either A or B or both—we just can't tell.

James Jones of Norman, OK, pointed out that the method might be simpler if we reversed it, searching for the ill-behaved functions (IBFs) instead of the well-behaved ones. A direct IBF (DIBF) would be a function that was itself objectionable—declared external, called itself, or called a pointer. Then a general IBF would be any function that called a DIBF at any level. In pseudo Prolog:

```
IBF(f) if member(f,DIBF).
IBF(f) if CALLS*(f,g) and
   IBF(g).
```

Here CALLS* signifies the transitive closure over the CALLS(a,b) relation. We can find the transitive closure in order n-squared time, Jones reminds us, using Warshall's algorithm. (The final phase of the method we described bore a strong resemblance to Warshall's algorithm.) Jones cites David Gries' *Compiler Construction for Digital Computers* (Wiley, 1972, page 39) as a source for Warshall's algorithm, but heck, it's easy to describe.

Suppose i and j are both integers between one and some maximum n, and suppose that you've defined a relationship R(i,j) that is true for some pairs (i,j) and not others. In our case, the relationship is CALLS(i,j), and it's true when function i calls function j. We can express the complete relationship in the form of a matrix of bits, B, such that B[i,j] is zero where R(i,j) is false and one where it's true.

We really want not CALLS(i,j) but CALLS*(i,j), the transitive closure of CALLS: the relation that is true if function i calls function j directly *or* through any number of intermediate functions. Warshall's algorithm to convert matrix B into matrix B* (using pseudo-C notation) is:

```
for(i = 1; i < = n; i + +)
   for(j = 1; j < = n; j + +)
      if B[j,i] then
         B[j,*] | = B[i,*];
```

The innermost line signifies the bitwise ORing of row i into row j; it could be spelt out as:

```
for(k = 1; k < =n; k + +)
   B[j,k] | = B[i,k];
```

However, the algorithm is n-squared only when the rows of B can be ORed as single objects, preferably with a fast assembly subroutine.

Jones and Jim Howell of San Jose,

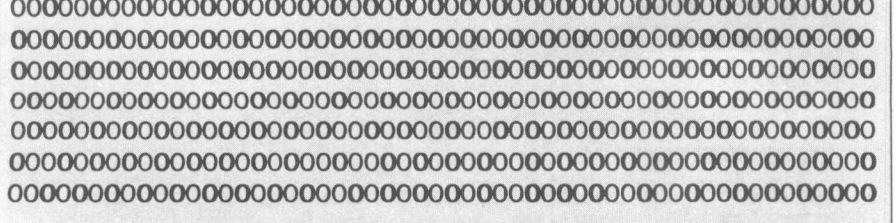


Figure
The irregular shadowing produced by the Intern's printer.

CA, both commented that the Motorola 6809 CPU shouldn't be tarred with the same brush we used for the 8080; the 6809 handles stack-allocated variables very well and wouldn't benefit much from their being static. Howell notes that, lacking a smart compiler, C programmers can create their own data type

```
#define LOCAL static
```

and use it in the functions that they know to be nonrecursive:

```
fun(arg)
{
   LOCAL int i,j;
```

Then, Howell says, "when porting to a machine that can efficiently access variables on the stack, just change one line to

```
#define LOCAL auto
```

or even to

```
#define LOCAL /* nothing */
```

All the local variables are now automatic."

Both Howell and Brian McKeon of Saugus, MA, noted that our method doesn't take account of functions that could be called as the result of interrupts. Paul Canniff of King of Prussia, PA, commented that stack-allocated locals, especially arrays, conserve storage, and a compiler that makes locals static may produce programs that are too big. Clearly, any compiler that used our method would have to provide a switch to turn it off. Alternatively, the compiler could take an explicit declaration of "auto" as barring static allocation.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 191**.

(Continued from page 8)

arguments than defined.

To compile grep.c with C/80:

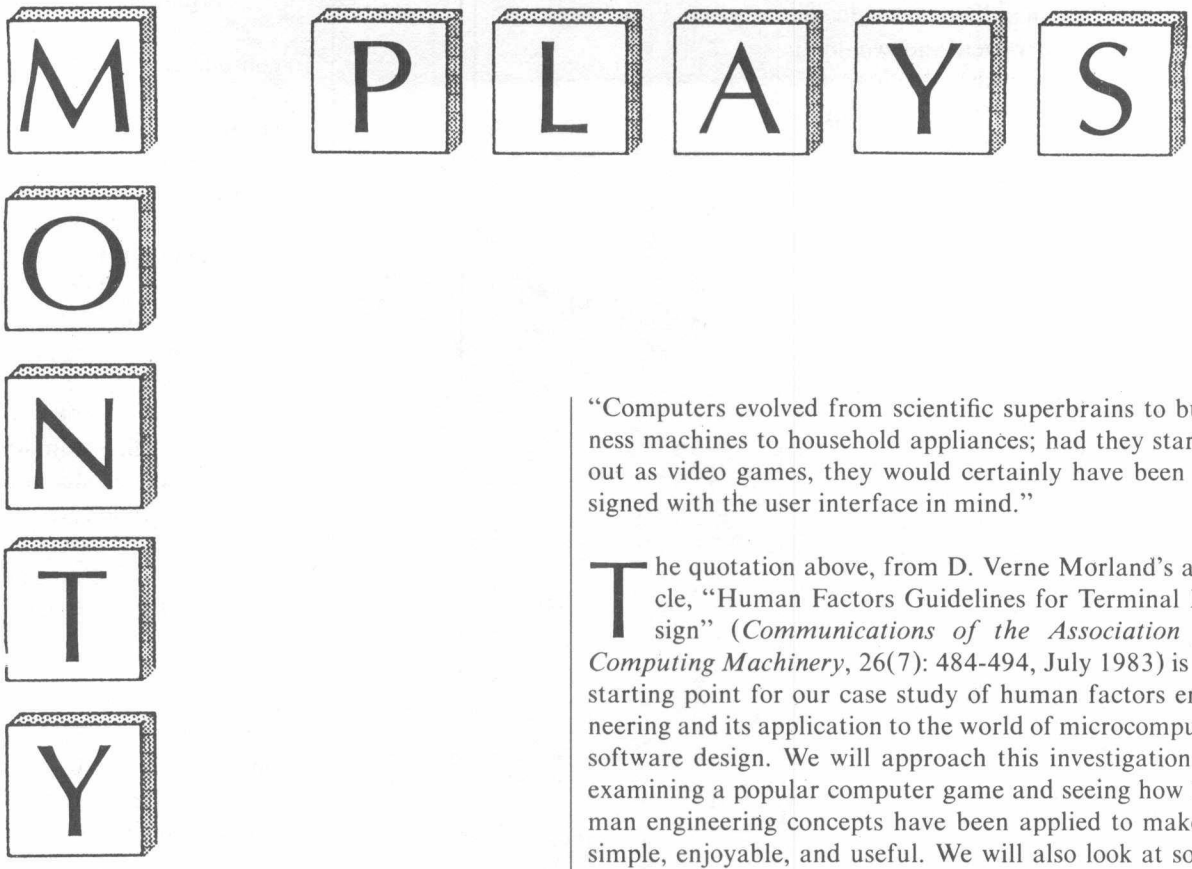
In TOOLS.H:
1. After #define NUL 0,
   add #define NULL NUL
2. Add #define EOF –1
3. Add #define TOKEN struct
   token
4. As explained in Section 8 of the C/80 Version 3.0 documentation add the following:
   ```
   #define FILE int
   #define stdin fin
   #define stdout fout
   #define stderr 0
   extern int fin, fout;
   ```
5. In the typedef statement on page 58 delete the words "typedef" and "TOKEN".

In TOOLS.C:
1. Delete the two #include directives
2. On page 61 add the argument "boln" to the omatch( ) call. The modified line should read:
   ```
   while(*lin && omatch(&lin,
      pat, boln))
   ```
3. On page 64 change the expression
   ```
   (dstart – dest < maxccl)
   ```
   to
   ```
   (dest – dstart < maxccl)
   ```

In GREP.C:
1. Delete the line #include "a:stdio.h"
2. Change #include "b:tools.h" to #include "tools.h"
3. Add #include "printf.c"
4. Add #include "tools.c"
5. On page 77 change
   ```
   if(matchs(line, exprv[j]))
   ```
   to
   ```
   if(matchs(line, exprv[j],0))
   ```
6. On page 83 change
   ```
   bdos(11)
   ```
   to
   ```
   bdos(11,0)
   ```
7. At the end of grep.c add #include "STDLIB.C"

Thank you,
Robert C. Briggs
1337 Rossway Court
Los Altos, CA 94022

DDJ



"Computers evolved from scientific superbrains to business machines to household appliances; had they started out as video games, they would certainly have been designed with the user interface in mind."

The quotation above, from D. Verne Morland's article, "Human Factors Guidelines for Terminal Design" (*Communications of the Association for Computing Machinery*, 26(7): 484-494, July 1983) is the starting point for our case study of human factors engineering and its application to the world of microcomputer software design. We will approach this investigation by examining a popular computer game and seeing how human engineering concepts have been applied to make it simple, enjoyable, and useful. We will also look at some microcomputer software products and consider how and where we might apply these same concepts to them.

Human factors engineering is that branch of systems analysis and design that deals with the interface between the computer user and the software/hardware combination. On the hardware side, human factors engineering helps designers determine where they should place special function keys on the keyboard, what color monitor is least fatiguing, and so on. On the software side, it helps programmers to design program menus, or choices to accomplish certain tasks. Finally, human factors engineering addresses the user—problems of operator error (and how to handle these errors) and whether a computer or a program is sufficiently user friendly.

Talk of "computer literacy" can sometimes be a mask for a botched and inhuman design.

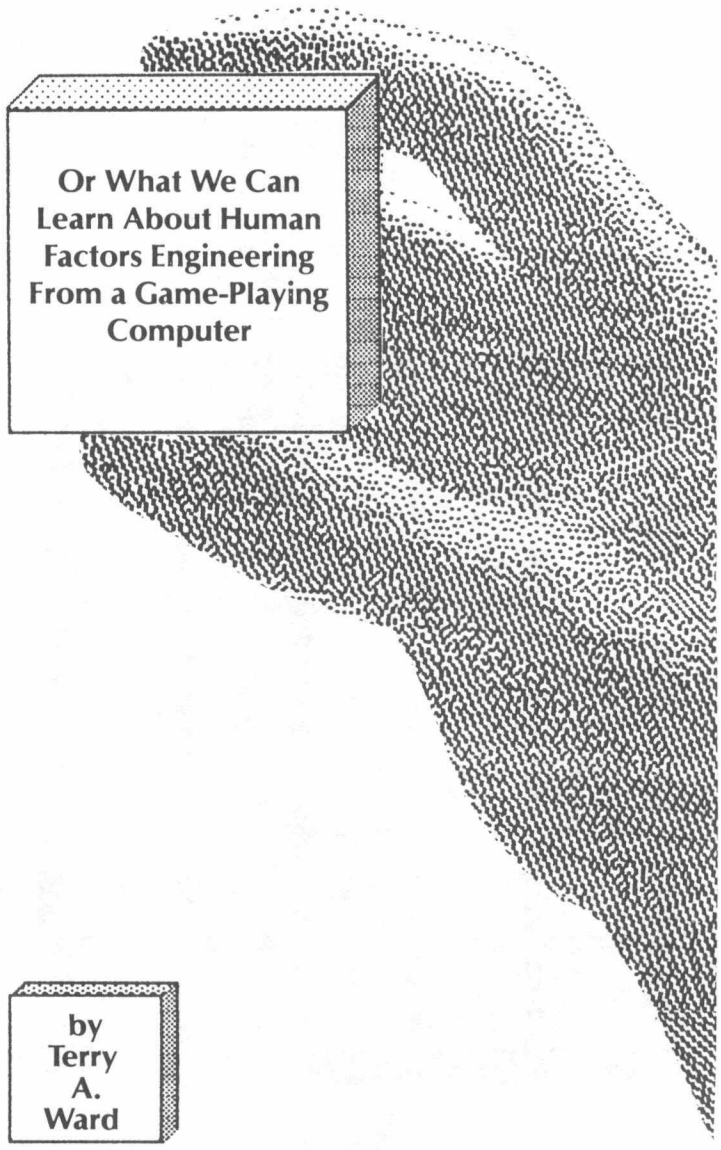
Although human factors engineering probably should be an integral part of any computer hardware or software design, the numerous variations of keyboard layout and the contorted sequences of keystrokes required to operate some programs would indicate that human engineering has *not* been applied uniformly. For every powerful, accessible, easy to use system available, many others perform as if no one ever tried using them before making them available for sale.

Scrabble is a trademark of the Selchow & Righter Company

Terry A. Ward, Academic Computing Services, University of Northern Iowa, Cedar Falls, IA 50614.

SCRABBLE:™

Or What We Can Learn About Human Factors Engineering From a Game-Playing Computer



by Terry A. Ward

The Product

Let's look at a popular microcomputer-based game system for playing Scrabble. For those unfamiliar with its rules, Scrabble is a crossword-puzzle game that can be played by two to four people. The players form interlocking words on the Scrabble playing board using letter tiles with various point values. The more common letters such as I or E are very low scoring (1 point each), while letters such as Q are worth more (in this case, 10 points). Players compete by using their letters and the various bonus squares on the board to maximize their score while minimizing their opponents' opportunities.

The computer variation of Scrabble involves MONTY, a self-contained, game-playing computer manufactured by the Ritam Corporation. Microcomputer versions of the game are available on diskette for both the Apple and the Radio Shack computer systems and have a similar operation. Further information on the game appears at the end of this article.

Basically, the self-contained MONTY (as we'll refer to the game from now on) looks like an oversized calculator with a membrane keyboard, which has some specialized characters, and a small 4 × 8 LCD display window. Figure 1 (page 20) shows a stylized view of the MONTY keyboard and display area. MONTY measures 6 × 10 inches and weighs about two pounds. Within this small frame is packed a formidable Scrabble challenger—and an excellent example of human engineering.

The Human Elements

Human factors engineering has certain principles or axioms that are applicable to any computer application, whether it be an extensive payroll program or a game-playing computer. The first rule or axiom is to identify your system users. This may seem obvious when stated so baldly, but many software designers neglect to do this. Consider any number of word processing or text editing software systems. The hardware includes a keyboard and a video display screen, and a great many software products for text editing refer to "screens" of information. However, screens are a convenience only to the hardware or to the programmer. Any user of word processing does not deal with screens. Writers write *pages* of books, secretaries write *pages* of letters, and even programmers write

pages of documentation. The system is oriented toward programmer or hardware conventions that have no meaning to the word processing audience.

Designers also fail to take into account the variety of people likely to use a computer system. A program written for other programmers (such as Unix or CP/M) can be terse, concise, and not particularly user friendly. But the average computer user who desires simply word processing capabilities or spreadsheet functions should not have to deal with obscure names for common functions. Programs for these users must provide easier access. CP/M shells (such as Unica or MicroShell) and menu-driven systems (such as the Morrow or Osborne systems) are attempts to match established programs with a new user audience. If the program is intended for everyone from programmers to the company vice president, it must provide some means of accommodating a wide spectrum of computer experience and computer literacy.

Furthermore, software should make complex tasks manageable. Obviously, a trivial program may need little, if any, human engineering. Operating real programs, however, is often sufficiently complex to require some assessment of the human element. In general, any application program can be subdivided into manageable tasks for the human operator. For example, WordStar, an intrinsically complex program, has a mind-boggling variety of commands, but remains a top-selling word processing program. This is because WordStar makes information available in the form of help screens to anyone who fails to remember the arbitrary keystrokes required for operation; sophisticated users may omit the help screens. This on-line help alleviates the problem of program complexity. As we will see in our case study of MONTY, this game uses the keyboard itself as a reminder for its functions.

Finally, programs should weigh the consequences of operator error. If the consequences are truly catastrophic (such as file deletion or program abort), the program should not take such action lightly. Release 3.0 of CP/M (also known as CP/M Plus) has instituted automatic relogging of diskettes to compensate for the common user mistake of switching disks without doing a control-C. When you do that, the dreaded "BDOS ERROR" appears and the system must be restarted or rebooted. In general, programs should try to prevent accidental disasters due to operator error. MONTY does this without appreciably slowing down the game-playing process.

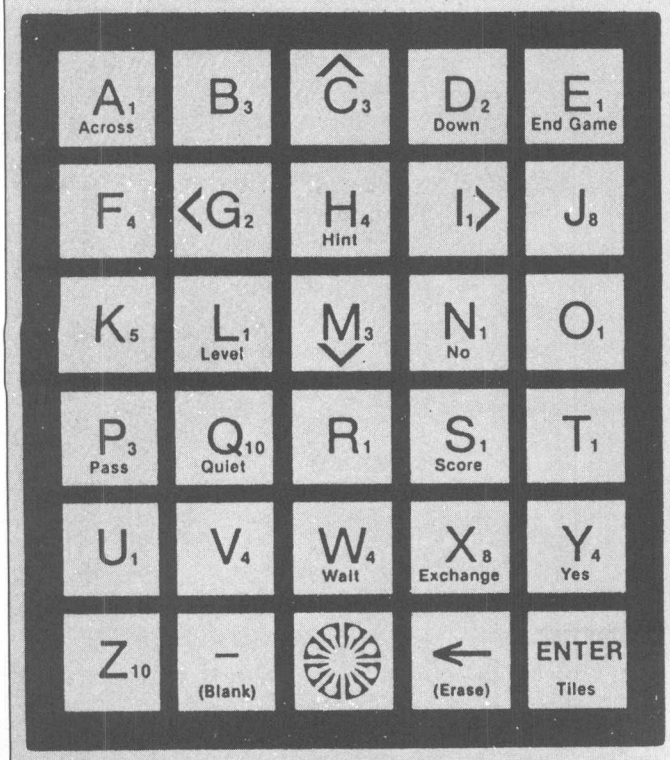


Figure 1
MONTY keyboard layout: A 4 x 8 character LCD screen provides a window to the entire Scrabble board. The commands used in playing the game are printed in contrasting colors to make them more apparent. The point value of the letters is also reproduced on the keyboard.

Software Design Principles

We have noted that our first general axiom is to consider the human elements in the human-machine interface. The second principle of human engineering is to design the product as much as possible to meet human needs—for feedback, consistency, memory aids, simplicity, and so on. We will now consider specific rules to follow when designing software.

The first rule is to provide some means of feedback as to the program's current status. Whenever the computer is doing something that may take a few moments, the user should be aware that the machine is still working. For example, a word-processing program will often appear totally "dead" when it is doing a global search-and-replace or advancing to a distant portion of the text. This can be both annoying and disconcerting; novice users may wonder as time passes if something is wrong. A simple display of a flashing asterisk could reassure them that all is well.

The second rule is to be consistent in terminology and program design. If we require the user to enter information using the Return key, we should refer to it in our instructions as the Return key—not the Carriage Return or the Enter key. While you and I may know that carriage return, enter, and return are all synonymous, individuals new to computing will hesitate before assuming that the three names refer to the same key, as they rightly should.

A third rule is to minimize the memory demands placed on human operators. We should not expect people to use our programs if they require more memorization than they are worth. The spreadsheet SuperCalc makes help available throughout the program for users who forget the exact requirements to specify a format, for example. This frees them to handle the task of problem definition.

The fourth software design rule is to keep the program simple. Programmers are notorious for preferring the la

est and most complex versions with "all the bells and whistles," but users simply want to get the job done. The program Bank Street Writer, while not in the same league as WordStar, nonetheless provides usable word processing capabilities to many people. I have even seen a review of Bank Street Writer written by a six-year-old child—an obvious endorsement of keeping it simple!

A fifth rule is to match the program to the operator's skill level. Many software programs assume that all users are equal: everyone must venture through countless help screens or menus to accomplish even the simplest task. For example, the SCSS conversational statistics package is very user friendly, but you must answer an abundance of questions to obtain even the simplest results. To plot two variables requires close to two dozen instructions, while a similar task in the statistical package MINITAB requires but three commands. Providing help screens on request only will accommodate both novice and advanced statistical users.

Finally, our last rule for software design might be paraphrased as: "The user is always right." Programs should filter and correct user mistakes. Often user error simply indicates poor program design. For example, our word processor requires that users remember that the Home button *really* means Down Arrow. Anyone who uses the key incorrectly is obviously guilty of user error, but the program actually invites just such an error. Always maintain a user or operator orientation. We are writing programs for people to use, not purely academic exercises.

MONTY—A Case Study

As we noted above, MONTY is a hand-held computing device that can play Scrabble with up to three other opponents. Let's look at the design of this computer game with an eye toward the first axiom of human factors engineering: identify your audience.

Obviously, MONTY is designed for people who wish to play Scrabble. Beyond that, the audience may range from computer analysts to ten-year-old children to adults who are totally unfamiliar with computers. I know individuals in each of those categories who have successfully played MONTY with only minimal instruction. How is MONTY designed to appeal to this wide audience?

To begin, the keyboard contains virtually all the commands needed to play the game. MONTY commands are superimposed on many of the alphabetic keys, printed in a contrasting color to distinguish them from the alpha characters. These commands are fully spelled out: one may "exchange" tiles or place them "across" the board. This is in contrast to many microcomputer programs that would abbreviate these commands as "xchng" or "a." This user-friendly keyboard allows virtually anyone to begin playing with little if any training. MONTY has taken into account the breadth of possible users and the diversity of their backgrounds without confusing one or patronizing the other.

In MONTY, the major catastrophic errors are ending the game prematurely, exchanging tiles (and thus losing a turn), and impossible plays. When a player proposes to end the game or exchange tiles, MONTY asks the user again if he or she is sure of this request. Only upon being told "yes" does the program implement this action. When players attempt an impossible play (such as placing tiles where they cannot legally go), MONTY returns them to the beginning of their turn. If you attempt to use a letter that you do not have, MONTY informs you of this and tells you which letter is missing. In all cases, the user is placed back into a logical position from which to try again. This is in contrast to many computer programs that will abort and place you in an unknown environment (such as CP/M) upon operator error.

Obviously MONTY has followed the first axiom of human factors engineering. We will now investigate how well our game-playing computer has utilized the six rules encompassed by the second axiom: design the product to meet human needs.

Provide Feedback

MONTY can take up to three minutes to find a word and make its play. A counter that is constantly displayed on the screen provides feedback while the program is calculating. When the counter reaches zero, MONTY plays its word. This feedback assures the opponent that all is proceeding normally. Likewise, requests for information are always accompanied by a listing of expected responses (such as yes/no or across/down). This feedback ensures that the user is aware of what is going on with the program.

Be Consistent

MONTY always requests information in a similar format or style when prompting for response. An inappropriate response is always greeted with a beep, and the request is issued again. It also uses the natural terminology of its users rather than the contrived terminology of computerdom.

Minimize Memory Demands

MONTY excels in this particular design goal. Embossing the keyboard with the commands reduces human memory demands to the bare minimum. By replacing human memory requirements with good software design, the programmer frees the user to concentrate on the task at hand—beating a machine at Scrabble.

Keep It Simple

MONTY is totally self-contained. It requires that the player know only a few commands in addition to the rules of Scrabble. This is in contrast to much of the currently available microcomputer software. Programmers have a tendency to fill any available storage space with functions, relevant or not. The range of options is simply too great, and most people use only a limited subset of a machine's capabilities. Limiting the options to those most commonly used will encourage people to use the software rather than intimidate them with an instructional manual the size of a telephone directory.

Match the User's Skill Level

Here is an area where much of the available microcomput-

Level A:	frog	
	suet	
Level B:	azo	
	eta	
Level C:	viol	
	averters	
Level D:	qintar	(unit of currency)
	hegira	(a journey)
	wadi	(a normally dry stream bed)

Figure 2
MONTY offers four levels of play. Levels may be increased or decreased without restarting the game or program. Listed are typical words encountered at each level of play.

er and mainframe software succeeds. MONTY presents all players with the same information and keyboard. It then accommodates itself to player proficiency by providing four levels of play (see Figure 2, above) and adjusting the rewards accordingly. MONTY rewards particularly good plays with a brief musical selection (such as the "1812 Overture" or "Hail to the Chief"); at level D, 50 points per play are required for "Hail to the Chief," while at lower levels of play it takes a less impressive score. This feature relates to the sixth and final design principle.

Sustain Operator Orientation

This last principle is by no means the least. In fact, it may be the most important of the half dozen presented. Programs are designed for people, not other programmers. Unix, for example, is an incredibly powerful software operating system. It is, however, generally quite unsuited for the non-programmer. CP/M is likewise an excellent operating system, but with its PIPs and CON:s and LST:s it is designed for the already computer literate. This is perhaps one of the great appeals of BASIC as a computer language or environment. Everything that needs to be done can be done in the one environment of the BASIC interpreter. You write your programs, you test them, you edit them and you even save and retrieve them from one, consistent environment. This is in contrast to the typical mainframe situation where you must always know exactly where you are. Editing commands are not recognized by the job control scanner and BASIC commands are likely to result in bizarre error messages if typed in at the wrong moment. In my experience as a user consultant, this is perhaps the major complaint of most people using computers. The machine forgets that it exists for the sake of human operators and not the other way around.

Conclusions

This venture into human factors engineering via a game-playing computer should give some insight into the reasons people complain about computers and their associated software. We are quite often to blame: many programs are *not* easy to use. Computer illiteracy often reflects programmer insensitivity to people's needs and desires. Although microcomputer sales are still quite healthy, I wonder whether we will see increasing disenchantment with software that doesn't work. MONTY and other well-designed programs can provide a starting point to enter the world of software engineering for human beings.

"MONTY Plays Scrabble"
Manufacturer: Ritam Corporation
P. O. Box 921
Fairfield, IA 52556
(515) 472-8262
Format: Apple II, II Plus, or IIe; Radio Shack TRS-80 Model 1, 3, or 4; self-contained version
Price: Apple, $39.95; Radio Shack, $34.95; self-contained, $129.95
Description: MONTY has a basic understanding of Scrabble strategy and, combined with a vocabulary of up to 44,000 words, can provide a formidable opponent to word game enthusiasts.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 192**.

Shortcut to SCISTAR: For Prowriter Users

by Amer W. Nelson

Thomas and Fletcher's excellent article, "SCISTAR: Greek and Math Symbols with WordStar" (*DDJ*, August 1984), taught me what I needed to know to incorporate those symbols, as well as super/subscripts and underlining, into my writing. The real advantage of Thomas and Fletcher's method is their use of a universal command set common to all printers. The drawbacks, as I see them, are that MailMerge must be used and that a "hunt and peck" typist like me requires a lot of time to produce files like SCI.CMD and SCI.DAT.

Further information on how to incorporate Greek and math symbols into WordStar files.

These drawbacks caused me to search for a shortcut to SCISTAR that my wife and I could use on our personal Osborne1/Prowriter system. I am a security analyst with a small investment company, and my wife is an EEG technologist. Because neither of us produces lengthy scientific papers, our use of Greek characters, math symbols, and super/subscripts is limited. (I use an occasional β to denote the relative volatility of a stock, an occasional M_1 in reference to the money supply, and a few superscripts for footnotes; my wife makes use of the Greek characters α, β, δ, κ, μ, and θ to denote brain wave patterns and Ω, the symbol for ohms of resistance.)

Amer W. Nelson, 12030 36th Ave., NE, Seattle, WA 98125.

The shortcut I will describe has the disadvantage of being tailored strictly to the Prowriter. Nonetheless, I believe that experienced users who understand Thomas and Fletcher can make those changes necessary to incorporate the shortcut in their own computer/printer configuration.

As Table 1 (page 25) shows, I have retained or only slightly modified many of Thomas and Fletcher's WordStar patches for the Prowriter. I elected to use WordStar's own boldfacing (hence, no patch for that feature), and I chose pica (10 CPI) to be my standard print with compressed 17 CPI pitch as an alternate. I used the command ^PQ—related to USR1:—to enter the Prowriter's Incremental Print Mode before typing special characters and the command ^PR—related to USR4:—to return to the Prowriter's Logic Seek Print Mode (see the model 8510A user's manual, page 46).

It's particularly important to imbed your ^PQs and ^PRs between paragraphs. When the Incremental Print Mode is on, printing is unidirectional, but when operating in the Logic Seek Print Mode, printing is bidirectional. If you try to bounce back and forth between them in the middle of a paragraph, you may find one line printing on top of another. To avoid such a problem and to conserve paper, I make it a general rule to enter the Incremental Print Mode before typing any paragraph where I intend to use special characters (be

they Greek, math, or super/subscripts) and to return to the Logic Seek Print Mode at the end of that paragraph. Because I use WordStar's boldfacing and underscoring feature—the latter through RIBBON: and RIBOFF:—I need not change print modes just to print in boldface or to underline.

Table 2 (page 26) contains specific commands that produce special characters: Greek, math, super/subscripts, and a few others. Remember, however, once you are through using one of the special characters and wish to return to English characters, you must type a ^PE. Thus, to print something like πr^2, you would type ^PW/^PEr^PWQ^PE. Similarly, to print a chemical representation of molecular composition like H_2SO_4, you would type H^PV^PWQ^PE^PVSO^PV^PWS^PE^PV.

Also remember that you cannot mix boldface and underlining with special characters on the same line. As Thomas and Fletcher point out, the Prowriter will print an alpha when it receives a 20H.

One advantage of my shortcut is that you can print the Greek gamma without jumping through hoops. Without using MailMerge, you can send a " (22H) as a variable. Moreover, if you're trying to type eight or nine pages with only a few special characters imbedded in the text, you can print out your file much more quickly in the bidirectional mode, using the unidirectional (Incremental Print) mode only as needed.

Table 3 (page 27) contains some additional information, unrelated directly to this article, that may nevertheless be of interest to Osborne1 users. If you're interested in learning C language programming, it would be nice to know how to type (and print) a { or a }.

This works too! (Refer to page 29 of the August 1984 *Dr. Dobb's Journal.*) Typing x ^PT ^PW (space) ^PE ^PT (space) ^PWF ^PE (space) ^PW!Q ^PE is an inequality. produces the following print:

$x^{\alpha} \geq \beta^2$ is an inequality.

Nice, huh?

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 193.**

Incremental Print	USR1:	ESC[	02H,1BH,5BH	^PQ
Logic Seek Print	USR4:*	ESC]	02H,1BH,5DH	^PR
Special Characters	USR2:	ESC&	02H,1BH,26H	^PW
ASCII (English)	USR3:	ESC$	02H,1BH,24H	^PE
Underscore On (toggle)	RIBBON:	ESCX	02H,1BH,58H	^PY
Underscore Off (toggle)	RIBOFF:	ESCY	02H,1BH,59H	^PY
Superscripting &	ROLUP:	ESCr(LF)	03H,1BH,72H,0AH	Use ^PVS or ^PTS
Subscripting	ROLDOW:	ESCf(LF)	03H,1BH,66H,0AH	(toggles)
Normal <CRLF>	PSCRLF:	ESCf(CR)(LF)(LF)	05H,1BH,66H,0DH,0AH,0AH	
Initialization String	PSINIT:	(CR)ESCT12	05H,0DH,1BH,54H,31H,32H	
Alternate Print	PALT:	ESCQ	02H,1BH,51H	17 CPI
Standard Print	PSTD:	ESCN	02H,1BH,4EH	10 CPI

*You may find my order strange, but I associated ^PR with *Return* to Logic Seek and ^PE with *English.*
(I did not use PSFINI:. I just made sure that I was in Logic Seek Mode at the end of printing.)

Table 1.
Patches installed on WordStar disc for use with the Prowriter

Type:	To get:	
^PW<SPACE>	α	(alpha)
^PW!	β	(beta)
^PW"	γ	(gamma)
^PW#	δ	(delta)
^PW$	ϵ	(epsilon)
^PW%	ζ	(zeta)
^PW&	η	(eta)
^PW'	θ	(theta)
^PW(	ϑ	(iota)
^PW)	κ	(kappa)
^PW*	λ	(lambda)
^PW+	μ	(mu)
^PW,	ν	(nu)
^PW-	ξ	(xi)
^PW.	o	(omicron)
^PW/	π	(pi)
^PW0	ρ	(rho)
^PW1	s	(sigma)
^PW2	τ	(tau)
^PW3	υ	(upsilon)
^PW4	ϕ	(phi)
^PW5	χ	(chi)
^PW6	ψ	(psi)
^PW7	ω	(omega)
^PW8	Δ	(upper-case delta)
^PW9	Γ	(upper-case gamma)
^PW:	Σ	(upper-case sigma)
^PW;	Λ	(upper-case lambda)
^PW<	Ω	(upper-case omega)

Most upper-case Greek characters can be printed by typing normal upper-case English characters: upper-case alpha is A, upper-case beta is B, and so on. For those upper-case Greek characters that the Prowriter can't handle, like pi and phi, just leave two spaces and do them by hand.

When you are through with Greek characters or any other special characters, type a ^PE. For instance, to print $\phi\upsilon\kappa\ \alpha\ \delta\upsilon\kappa$, type ^PW43)^PE <space>^PW(space)^PE<space>^PW#3)^^PE.

Other available symbols (characters) include the following:

Type:	To get:	
^PW>	$\surd$	(square root sign)
^PW?	$\square$	(upper square)
^PW@	$\uparrow$	(arrow up)
^PWA	$\downarrow$	(arrow down)
^PWB	$\leftarrow$	(arrow left)
^PWC	$\rightarrow$	(arrow right)
^PWD	$\pm$	(plus or minus)
^PWE	$\neq$	(not equal)
^PWF	$\geq$	(greater than or equal)
^PWG	$\leq$	(less than or equal)
^PWH	$\approx$	(approximates)

Table 2.
Special Character Commands

(Table 2 continued)

Type:	To get:	
^PWI	·	(dot for multiplication sign)
^PWJ	⊕	(earth symbol)
^PWK	∞	(infinity)
^PWL	∴	(therefore)
^PWM	½	(one half)
^PWN	¼	(one quarter)
^PWO	°	(degree or super zero)
^PV^PWO	$_0$	(sub zero)
^PWP	^{1}	(super 1)
^PV^PWP	$_1$	(sub 1)
^PWQ	^{2}	(super 2)
^PV^PWQ	$_2$	(sub 2)
^PWR	^{3}	(super 3)
^PV^PWR	$_3$	(sub 3)
^PWS	^{4}	(super 4)
^PV^PWS	$_4$	(sub 4)
^PWT	^{5}	(super 5)
^PV^PWT	$_5$	(sub 5)
^PWU	^{6}	(super 6)
^PV^PWU	$_6$	(sub 6)
^PWV	^{7}	(super 7)
^PV^PWV	$_7$	(sub 7)
^PWW	^{8}	(super 8)
^PV^PWW	$_8$	(sub 8)
^PWX	^{9}	(super 9)
^PV^PWX	$_9$	(sub 9)
^PWY	$^($	(super open parenthesis)
^PV^PWY	$_($	(sub open parenthesis)
^PWZ	$^)$	(super close parenthesis)
^PV^PWZ	$_)$	(sub close parenthesis)
^PW[	$^+$	(super plus)
^PV^PW[	$_+$	(sub plus)
^PW\	$^-$	(super minus)
^PV^PW\	$_-$	(sub minus)
^PW]	$^.$	(super period)
^PV^PW]	$_.$	(sub period)
^PW^*	*	(super asterisk)
^PV^PW^*	$_*$	(sub asterisk)
^PW_	$^/$	(super slash)
^PV^PW_	$_/$	(sub slash)

*The trailing ^ is *not* a "control." It is a caret, the upper-case 6.

Type:		To get:	
<CONTROL> equal sign	^=	`	(a back accent or grave)
<CONTROL> comma	^,	{	(a left-hand brace)
<CONTROL> period	^.	}	(a right-hand brace)
<CONTROL> slash	^/	~	(a tilde as used in Spanish)

The left- and right-hand braces are particularly useful to those interested in learning C programming.

Table 3.
Some Characters not on the Osborne1 Keyboard

fx80char: A Character Editor for Epson FX-80 Printers

by David D. Clark

When Epson announced the FX-80/100 series of dot matrix printers, microcomputer users were attracted immediately because of the variety of desirable features offered at a low price. One of the things that attracted me was the ability to redefine the standard character font. Such a capability is useful, if only because technical papers almost always seem to require characters that your printer normally can't print.

The FX-80 also makes a good draft printer. If you prepare a document for printing with a letter quality printer such as a Diablo or NEC, printing early drafts can be tedious because the printers are so slow; the normal character set of a dot matrix printer, however, may not correspond to the characters available on the print element used with the letter quality printer. With the Epson, you can redefine the dot matrix characters to correspond exactly to those on the print element of the slow printer. In my case, I have access to a letter quality printer at work and use the FX-80 at home. By redefining the standard characters, I can get a reasonable facsimile of how the document will look when printed at my office while still working at home.

The most underused printer feature may be the redefinable character set. This tool makes the feature more usable.

The program described in this article, fx80char, is designed to aid in the development of alternate character sets for use on the Epson FX-80/100 series of printers. Written in the Eco-C version of C for CP/M, the program provides a screen-oriented character editor, the ability to read and write disk files containing character set definitions, and commands to download new character sets to the printer.

David D. Clark, 126 Birchview Dr., Piscataway, NJ, 08854.

Commands

After fx80char is started, it displays a menu of commands. To select a particular command, you just type the highlighted character. It is not necessary to hit a <Ret> after typing the character. This section discusses each of the commands in alphabetical order.

The Activate command is used to toggle the use of the printer's ROM or RAM character set; it is assumed that the ROM set is active when the program starts. The first use of the Activate command causes the printer to use the character set stored in RAM for subsequent printing. Successive uses of this command will toggle between the two character sets. The program always displays the active character set after executing this command.

The Edit command is used to create or alter a particular character. When you enter the character editing routine, you will be asked for the number of the character to edit. This number is simply the decimal number associated with the ASCII character.

A number is requested because sometimes you can't enter the desired character from the keyboard.

After you enter the number of the character, the character itself, as currently defined, is displayed on an enlarged "easel" for editing. You can move the cursor around within the easel, as well as "flip" the state of the individual elements. Elements displayed in reverse video represent the pins of the print head that will be fired and where ink will appear when the character is printed.

There are two equivalent groups of movement commands. The first is similar to that used by WordStar: Ctrl-E moves up, Ctrl-D moves right, Ctrl-X moves down, and Ctrl-S moves left. You can use Ctrl-F to flip the state of the element upon which the cursor rests. The second allows you to use a numeric keypad for editing: 8 moves the cursor up; 6 moves it right; 2 moves it down; 4 moves it left; and you can toggle the current element by tapping 5. The two groups of keys may be mixed at any time.

When alterations to a character are complete, typing a Q or Ctrl-Q updates the character definition and exits the editing function. During the update process, the character is converted from the form used for convenient display to the actual form used when sending definitions to the printer. The definition is checked for two types of errors: characters nine elements high and characters with consecutive printing of horizontally adjacent dots. Because the FX-80/100 can print only characters that are eight or fewer cells in height, definitions with nine cells are not allowed. Similarly, although eleven horizontal positions are available, the FX-80/100 can print no two consecutively. If an inadmissable definition is detected during the conversion process, a message to that effect is displayed. No further action, however, is taken: the character will be converted but the definition will not print correctly. Attempting to send such a definition to the printer will produce indeterminate results.

The Initialize command is used to clear the buffer that holds the character definitions in the program's memory. This command has absolutely no effect on the printer or on the contents of its memory; it simply inserts housekeeping information into the buffer and sets all definitions to blanks. This command gives you a "clean slate" to work with. It is a good idea to execute this command before building a new character definition file from subsets of previously defined files on disk (see the Read command below).

The Load command loads the *printer* RAM with the contents of the *printer* ROM. It does not transfer data between the computer and the printer. It is useful primarily to check the proper functioning of the printer's RAM. You also can use it in conjunction with the Overlay command to replace parts of the standard character font.

The Overlay command is used to redefine all or part of the printer's character set using definitions from the program's character buffer. This command displays a menu with options to redefine all characters, a range of characters, or a single character.

If you request that a range of characters or a single character be redefined, you must select the limits of the range or the single character as decimal numbers, just as in the Edit command. If the selected characters are not within the range of legal characters (0–225), or if the low end of a range is greater than the upper end, an error message is displayed and no characters are changed.

The Print command is used to print out the entire character set. After you have Overlayed the printer's RAM character set with newly defined characters, use this command to see what they look like. Remember, to be printed, redefined characters must be Overlayed in the printer's RAM, and the RAM character set must have been Activated.

This command exposes one of the program's foibles. Character 127, the alternate zero, is not printed: in its place is a blank. I couldn't figure out how to print the alternate zero under program control. You must set a switch inside the printer itself (see your printer's user manual).

The Read command is used to retrieve previously defined fonts from files on the disk. You first are asked for the name of the file. After you enter the name, a new menu is displayed giving you the option of reading all of the characters, a range of characters, or a single character from the file.

If you select to read a range of characters or only a single character, you again are asked to supply the limits of the range or the number of the character. If any of the numbers are outside the legal range, or if the lower limit exceeds the upper limit, an error message is displayed and no action takes place. If the program cannot open the file or read its contents, an error message to that effect is displayed. Check the spelling of the filename.

As mentioned above, if you intend to create a font by building it up through successive Read commands on a group of character definition files, invoke the Initialize command before you start. If not, the newly built definition may redefine character 0 to a blank or possibly crash the program, system, and printer. In addition, you may find it impossible to Read the uninitialized font definition after it has been stored on disk.

The Write command is used to store the contents of the character definition buffer into a file on the disk. You will be asked to enter a name for the file. If the file already exists, its contents are lost when the new file overwrites it. If the program cannot create the file or detects some I/O error, an error message is displayed. It is a good idea to make sure you have enough space on the disk for all the files you intend to create *before* you start the program. If the program crashes because of insufficient disk space, all of your work since the last Write command will be lost.

The Write command saves the entire character definition buffer. It cannot operate with only a range of characters or a single character.

The Quit command is used to exit the program. Remember to save any newly created or altered definitions with the Write command before Quitting or you will lose all your work.

Using the Program

Before you start the program, be sure that the switch in the printer that determines how the 2K buffer will be used is set so you can store your character definitions. To start the program, type fx80char followed by a carriage return. You can also enter a filename following the program name; fx80char will assume that the file is a character definition file and attempt to read it into the program's character buffer.

The usual sequence of actions during a session with fx80char involves Activating the printer's RAM character set, Reading a character definition, Editing some of the characters, Overlaying the new characters in the printer, and Printing the new character set. You usually repeat the Edit-Overlay-Print cycle several times during a session as you make changes to the character definitions. When you are finished, you probably will want to Write your revised character definitions to a file for permanent storage.

When you want to use a different character set from another program (for example, to print a draft copy of a document), run fx80char, Read the new character set from a file, then Overlay the new character set. After you exit from fx80char, the altered character set will be used for printing until you turn off the power or run fx80char again and replace the set.

Some of my friends and I have been using this program for some time. We have yet to find any errors. The most common mistake we make seems to be Quitting the program before Writing revised definitions to a file.

The Program

The source text for the program is divided among five files written in the Eco-C version of C for CP/M. These consist of the header file FX80.H (Listing One) and the files FX80CHAR.C, ACTIONS.C, EDCHAR.C, and PRNOUT.C (Listings Two through Five, respectively).

FX80.H (Listing One, page 34) contains the global #define, data structure, and variable declarations. This file should be included in all C source modules making up the program. In addition, *one and only one* of the C source modules must define the macro MAIN before inclusion of FX80.H. This defines the global character buffer pointer, struct chrary *chrs, in that file. All other files (those without the macro MAIN) will compile only a declaration of *chrs as an external variable.

struct chrary defines the structure of the elements that will make up the program's global character buffer. It is fairly straightforward. The chrnum member contains the number of the character, which corresponds to the numbers requested in the Edit, Overlay, and Read commands. The attrib element contains the attribute byte for the character as currently defined: the program uses only two values for this byte to distinguish between printing with the upper or lower eight pins of the print head. The program generates no spacing data for proportional characters, and ary simply contains the sequence of bytes needed to define the character to the printer.

The variable chrs is a pointer to a variable of type struct chrary. When the program starts, an array large enough to hold NUMCHARS elements of type struct chrary is allocated; chrs points to the first element. The variable then is used globally throughout the program.

FX80CHAR.C (Listing Two, page 36) contains the main program function and miscellaneous utility routines. The function main( ) attempts to allocate space for the global character buffer and to assign the variable *chrs to point to it. If the allocation fails, the program will abort. If the allocation succeeds, the program will zero (not initialize) the character buffer. The character buffer is allocated dynamically to decrease the size of the code file (and to speed up program assembly and linkage).

The program then checks to see if a command line argument has been specified. If so, the program interprets the argument as the name of a character definition file and attempts to read it into the global character buffer. When all initialization is complete, the function mainmenu( ) is invoked to handle the command loop.

ACTIONS.C (Listing Three, page 42) contains most of the functions that actually implement the commands available from the main menu level. These are all pretty obvious, many simply sending sequences of commands to the printer.

EDCHAR.C (Listing Four, page 50) contains the character editor function. After the user enters the number of the character to be edited, the element is "unpacked" into the static external variable cary. This is a two-dimensional array with one element of type unsigned, corresponding roughly to each bit in a variable of type struct chrary. Unpacking the bits into this variable makes the editing process much simpler and faster because individual bits of the character definition need not be accessed. The character display and cursor movement are fairly obvious. At the end of the edit, the character is checked for errors in definition and repacked.

PRNOUT.C (Listing Five, page 56) contains the function to print the character set. printall( ) loops through all 256 characters, testing whether the character falls into one of five classes. If it is a normal printable character or its italic counterpart, the character simply is sent to the printer. If it is a high-order control character or something other than a special low-order control character, the appropriate series of commands is sent to the printer. If the loop counter corresponds to character 127, a space is skipped. If it is a special character, as determined by the function isspecial( ), the pspecial( ) function is called. (Special characters correspond to things like the escape and carriage return.) This function basically does a brute force translation from the character number to the commands necessary to print a graphic version of the character.

External Functions

fx80char uses several external routines that are not members of the standard library. With a few exceptions, these are used to control the video properties of the terminal upon which the program is running.

The video control functions are from my C function library. Most programmers have a similar library containing routines to access functions available on their particular terminals. The functions revon( ) and revoff( ) are used to turn on and off the reverse video mode of the terminal. The gotoxy(column, line) function is used to position the cursor at an arbitrary location on the screen. clrline(line) clears the entire line that is specified as the function's argument. The function clreos(column, line) clears the screen, from the requested coordinates to the end of the screen, and leaves the cursor at the specified coordinates. Similarly, clrscrn( ) clears the *entire screen* and leaves the cursor at the home position.

My video routines assume that the home position has coordinates of (0, 0). Column numbers increase to the right and line numbers increase as the cursor moves down the screen. The program assumes it is running on a terminal with 24 lines of 80 col-

umns each.

The program also uses some functions from the Eco-C standard library that are not available in the Unix C library. The _bdos( ) function, which provides direct access to the CP/M BDOS, is used for raw input from the keyboard and output to the printer. Similar functions are available in most CP/M-based versions of C. The setmem( ) function quickly initializes a block of memory to a specified value.

One additional item is worth noting here. The version of fopen( ) used in the listings does not use standard parameters. Unix C terminates lines with a single newline character, but CP/M uses the carriage return-line feed (CR/LF) pair of characters. Most CP/M-based implementations of C provide an automatic translation of CR/LF to LF when reading text from a disk. Conversely, single LF characters are translated to the CR/LF sequence when writing a text file. In Eco-C, the mode argument is declared to be "binary" by appending the character "b" to the standard mode (see functions readfl( ) and writefile( ) in Listing Three). This convention is used when no special handling of CR/LF sequences is desired.

Conclusions

If you are looking for things to improve, a couple of places in the program are ripe for diddling. First, the definition of struct chrary is rather wasteful. Declaring the attrib and ary fields as char could save about half the space now required. Another area that could be improved is the character-packing function, packit( ), in the file EDCHAR.C (Listing Four). As currently implemented, the program will not create characters that can be proportionally printed. This is due to the way packit( ) assigns the attribute byte.

It would also be simple to make the program a little more forgiving of user errors. You could have the program prompt users if they try to Quit before Writing their changes to disk. Another good idea might be to automatically Initialize the character buffer when it is allocated.

The program as it appears in the listings does not use any printf( )s. This made the program about 1.5K smaller at the expense of a couple of clumsy statement sequences. If you make changes that include calls to printf( ), you also might want to clean up some of those statements.

This program has made creating and printing alternate characters for the FX-80 much easier. You can change characters and immediately see the results, which is much faster and less frustrating than writing a program to send a fixed series of commands to the printer.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 194.**

fx80char (Text begins on page 28)

Listing One

```
/*
**      fx80.h -- global header file for fx80 character font editor
*/

#ifndef VOID
#define VOID            int
#endif

#define REVISION        "7 April 1984"
#define TITLE           "Character Font Editor -- FX-80 Vers. 1.0"

#define STATUSLINE      0               /* status line line number */
#define MENULINE        4               /* line at which menus start */
#define PROMPTLINE      20              /*,line for prompts */
#define NAMELENGTH      16              /* max length of file name */

#define WIDTH           11              /* width of character cell */
#define HEIGHT          9               /* height of character cell */
#define NUMCHARS        256             /* number of possible characters */

#define BOXX            (unsigned) 30   /* left edge of character box */
#define BOXY            (unsigned) 5    /* top of character box */
#define YOFF            (BOXY + HEIGHT + 1)

#define BELL            7
#define ESC             27
#define CTRLE           5
#define CTRLX           24
#define CTRLS           19
#define CTRLD           4
#define CTRLF           6
#define CTRLQ           17
```

(Continued on page 36)

Listing One

```
struct  chrary
        {
                unsigned        int     chrnum;
                unsigned        int     attrib;
                unsigned        int     ary[WIDTH];
        };

#define ELSIZE          (sizeof(struct chrary))
#define ARRAYSIZE       (NUMCHARS*ELSIZE)


#ifdef  MAIN            /* being included in main program file */

struct  chrary  *chrs;  /* pointer to global character array */

#else                   /* being included in subsidiary function file */

extern  struct  chrary  *chrs;

#endif
```

End Listing One

Listing Two

```
/*
**      fx80char.c -- character font editor for Epson FX80 printers.
**
**      This program can be used to create and alter character fonts
**  for downloading to Epson FX80 and FX100 printers.
**
**      David D. Clark
**      7 April 1984
*/

#include        <stdio.h>

#define         MAIN            /* notify the fx80.h header that this is
                                   the main program file */
#include        <fx80.h>


main(argc, argv)
int     argc;
char    *argv[];
{

        /* allocate character buffer area */
        if ((chrs = (struct chrary *) calloc(NUMCHARS, ELSIZE)) == NULL)
        {
                fputs("No room to allocate edit buffer.\n", stderr);
                exit(1);
        }

        /* try to read in an initial font file if requested */
        if (argc > 1)
        {
                readfl(argv[1], 0, NUMCHARS);
        }

        mainmenu();
        clrscrn();
        exit(0);
}
```

(Continued on page 38)

Listing Two

```
/*
**      mainmenu -- outer most command level.  This is the actual

**  command p ocessing loop.
*/

char    *mnul[] =       /* menu for outer-most command level */
        {
          " -- Activate/deactivate RAM character set.\n",
          " -- Edit the currently selected character.\n",
          " -- Initialize the edit character array.\n",
          " -- Load printer RAM with ROM character set.\n",
          " -- Overlay new characters in printer.\n",
          " -- Print the character set.\n",
          " -- Read character file.\n",
         " -- Write character file.\n",
          " -- Quit.\n"
        };

char    *cmdl =         /* acceptable user responses to this level */
        {

          "AEILOPRWQ"
         };

 int    ncmdl = 9;      /* number of acceptable commands */

 VOID   mainmenu()
 {
 int    cmd;            /* command character */

        while (TRUE)    /* loop forever */
        {
                clrscrn();
                shortstat();
                cmd = domenu(mnul, cmdl, ncmdl);
                switch (cmd)
                {
                        case 'A':
                                activate();
                                break;

                        case 'E':
                                edchar();
                                break;

                        case 'I':
                                inichars();
                                break;

                        case 'L':
                                loadram();
                                break;

                        case 'O':
                                overlay();
                                break;

                        case 'P':
                                printall();
                                break;

                        case 'R':
                                readfile();
                                break;

                        case 'W':
                                writefile();
                                break;
```

```
                        case 'Q':
                                return;

                }
        }
}


/*
**      The following functions are utilities used throughout the program.
*/

/*
**      domenu -- print out specified menu and get an acceptable response.
*/

int     domenu(mnu, cmds, ncmds)
char    *mnu[],
        *cmds;
int     ncmds;
{
int     c;
unsigned        i;

                fputs("\n\n\n", stdout);
                for (i = 0; i < ncmds; i++)
                {
                        fputs("\t\t", stdout);
                        revon();
                        putc(cmds[i], stdout);
                        revoff();
                        fputs(mnu[i], stdout);
                }

                do
                {
                        c = (islower(c = mygetc()) ? toupper(c) : c);
                        if (index(cmds, c) == NULL)
                        {
                                c = NULL;
                                putc(BELL, stdout);
                        }
                } while (c == NULL);

                return (c);
}


/*
**      selchar -- select a character by number.
*/

unsigned        selchar()
{
char    charstr[6];

        gotoxy(0, PROMPTLINE);
        fputs("character number?  ", stdout);
        if (fgets(charstr, 5, stdin) == NULL)
                return (32);    /* have to return something */
        return ((unsigned) atoi(charstr));
}


/*
**      pputc -- put a character to the printer.
*/

VOID    pputc(c)
int     c;
{
        _bdos(5, c);            /* send char directly to printer */
}
```

(Continued on next page)

Listing Two

```
/*
**      mygetc -- get raw keyboard input.
*/

int     mygetc()
{
int     c;

        while (!(c = _bdos(6, 0x00ff)))
                ;                               /* wait for a character */
        return (c);
}


/*
**      pause -- wait for the user to type a character.
*/

VOID    pause()
{
        fputs("Type any character to continue.  ", stdout);
        mygetc();
        putc('\n', stdout);
}


/*
**      cantopen -- print error message when unable to open "file".
*/

VOID    cantopen(file)
char    *file;
{
                fputs("Can't open ", stderr);
                fputs(file, stderr);
               putc('\n', stderr);
                pause();
}


/*
**      filerr -- print error message after file i/o error.
*/

VOID    filerr()
{
        fputs("\nFile i/o error\n", stderr);
        pause();
}


/*
**      shortstat -- print short status line.
*/

VOID    shortstat()
{
        clrline(STATUSLINE);
        fputs("\t\t", stdout);
        revon();
        fputs(TITLE, stdout);
        revoff();
}
```

(Continued on page 42)

Listing Two

```
/*
**      longstat -- print a long status line.
*/

VOID    longstat(charnum)
int     charnum;
{
char    curchar[6];             /* ASCII representation of charnum */

        clrline(STATUSLINE);
        revon();
        fputs(TITLE, stdout);
        gotoxy(42, STATUSLINE);
        fputs("Char. No. ", stdout);
        itoa(curchar, charnum);
        fputs(curchar, stdout);

        gotoxy(58, STATUSLINE);
        fputs("Char = ", stdout);
        revoff();

        if (isprint(charnum & 0x7f))
                if (charnum > 127)
                {
                        revon();
                        putc(charnum, stdout);
                        revoff();
                }
                else
                        putc(charnum, stdout);
        else
        {
                revon();
                fputs("Can't show", stdout);
                revoff();
        }
}
```

End Listing Two

Listing Three

```
/*
**      actions.c -- action routines for character editor
*/

#include        <stdio.h>
#include        <fx80.h>


/*
**      activate -- activate the RAM character set.
*/

static  ramact = FALSE;         /* assume ROM set active at start of program */

VOID    activate()
{
        gotoxy(0, PROMPTLINE);
        if (!ramact)
        {
```

```
                pputc(ESC);
                pputc('%');
                pputc(1);
                pputc(0);
                ramact = TRUE;
                fputs("RAM character set active.\n", stdout);
        }
        else
        {
                pputc(ESC);
                pputc('%');
                pputc(0);
                pputc(0);
                ramact = FALSE;
                fputs("ROM character set active.\n", stdout);
        }
        pause();
}


/*
**      inichars -- initialize the global character array.
*/

VOID    inichars()
{
unsigned        i;
struct  chrary  *ptr;

        gotoxy(0, PROMPTLINE);
        fputs("Initializing character array...\n", stdout);

        setmem(chrs, ARRAYSIZE, 0);
        for (i = 0, ptr = chrs; i < NUMCHARS; i++, ptr++)
                ptr->chrnum = i;

        pause();
}


/*

**      loadram -- load ROM character set into RAM.
*/

VOID    loadram()
{
        gotoxy(0, PROMPTLINE);
        pputc(ESC);
        pputc(':');
        pputc(0);
        pputc(0);
        pputc(0);
        fputs("Printer RAM loaded from ROM.\n", stdout);
        pause()
}


/* menu and responses for the overlay and read commands */

char    *mnu2[] =
        {
          " -- All characters.\n",
          " -- Range of characters.\n",
          " -- Single character.\n"
        };

char    *cmd2 =
        {
          "ARS"
        };
```

(Continued on next page)

fx80char (Listing Continued, text begins on page 28)

Listing Three

```
int     ncmd2 = 3;


/*
**      overlay -- overlay the characters in the printer RAM with
**  the characters in chrs.
*/

VOID    overlay()
{
int     cmd;
unsigned        i, rlo, rhi;
struct  chrary  *myptr;

        clrscrn();
        shortstat();
        cmd = domenu(mnu2, cmd2, ncmd2);
        gotoxy(0, PROMPTLINE);
        switch (cmd)
        {
                case 'A':
                        fputs("Overlaying entire character set...\n", stdout);
                        for (i = 0, myptr = chrs; i < NUMCHARS; i++, myptr++)
                                defchar(myptr);
                        break;

                case 'R':
                        clreos(0, (PROMPTLINE - 1));
                        fputs("For the low end of the range,\n", stdout);
                        rlo = selchar();
                        clreos(0, (PROMPTLINE - 1));
                        fputs("For the high end of the range,\n", stdout);
                        rhi = selchar();
define:                 if (rlo >= 0 && rhi < NUMCHARS && rlo <= rhi)
                        {
                                myptr = chrs;
                                /* skip unwanted characters */
                                while (myptr->chrnum != rlo)
                                        myptr++;
                                for (i = rlo; i <= rhi; i++, myptr++)
                                        defchar(myptr);
                        }
                        else
                                fputs("Illegal character range.\n", stderr);
                        break;

                case 'S':
                        rlo = rhi = selchar();
                        goto define;
        }

        pause();
}


/*
**      defchar -- define a character to the printer.
*/

VOID    defchar(p)
struct  chrary  *p;
{
unsigned        j;

        pputc(ESC);
```

(Continued on page 48)

```
        pputc('&');
        pputc(0);
        pputc(p->chrnum);
        pputc(p->chrnum);

        pputc(p->attrib);
        for (j = 0; j < WIDTH; j++)
                pputc(p->ary[j]);
}


/*
**      readfile -- read a file of character definitions.
*/

VOID    readfile()
{
char            name[NAMELENGTH];
int             cmd;
unsigned        rlo, rhi;

        gotoxy(0, PROMPTLINE);
        fputs("Read what character file?  ", stdout);
        fgets(name, NAMELENGTH, stdin);
        clrscrn();
        shortstat();
        cmd = domenu(mnu2, cmd2, ncmd2);
        gotoxy(0, PROMPTLINE);
        switch (cmd)
        {
                case 'A':
                        readfl(name, 0, NUMCHARS);
                        break;

                case 'R':
                        clreos(0, (PROMPTLINE - 1));
                        fputs("For the low end of the range,\n", stdout);
                        rlo = selchar();
                        clreos(0, (PROMPTLINE - 1));
                        fputs("For the high end of the range,\n", stdout);
                        rhi = selchar();
doread:                 if (rlo >= 0 && rhi < NUMCHARS && rlo <= rhi)
                                readfl(name, rlo, rhi);
                        else
                                fputs("Illegal character range.\n", stderr);
                        break;

                case 'S':
                        rlo = rhi = selchar();
                        goto doread;
        }
}


/*
**      readfl -- do the actual read of a character definition file.
**  Only read characters from "lo" to "hi" into the global character
**  buffer.
*/

VOID    readfl(name, lo hi)
char    *name;
unsigned        lo, hi;
{
char    c, *ptr;
unsigned        i;
FILE    *fd;

        if ((fd = fopen(name, "rb")) == NULL)
```

```
        {
                cantopen(name);
                return;
        }

        fputs("Reading...", stdout);

        /* skip past unwanted characters */
        for (i = 0, ptr = (char *) chrs;
             (i < (lo*ELSIZE)) && ( c = getc(fd)) != EOF);
             i++, ptr++
            )
                ; /* do nothing */

        if (lo > 0 && i < (lo*ELSIZE - 1))
                filerr();

        /* now read the characters into the array */
        for (
             i = (lo*ELSIZE); /* ptr is already where it should be */
             (i < (hi*ELSIZE + ELSIZE)) && ((c = getc(fd)) != EOF);
             i++, ptr++
            )
                *ptr = c;

        if (hi > 0 && i < (hi*ELSIZE + ELSIZE - 1))
                filerr();

        fclose(fd);
}


/*
**      writefile -- write a file of character definitions.
*/

VOID    writefile()
{
char    c, *ptr;
char    name[NAMELENGTH];
unsigned        i;
struct  chrary  *myptr;
FILE    *fd;


        gotoxy(0, PROMPTLINE);
        fputs("Write characters to what file?  ", stdout);
        fgets(name, NAMELENGTH, stdin);
        putc('\n', stdout);
        if ((fd = fopen(name, "wb")) == NULL)
        {
                cantopen(name);
                return;
        }

        fputs("Writing...", stdout);
        for (i = 0, ptr = (char *) chrs; i < ARRAYSIZE; i++, ptr++)
        {
                if (putc(*ptr, fd) == EOF)
                {
                        filerr();
                        fclose(fd);
                        return;
                }
        }

        fclose(fd);
}
```

End Listing Three

(Listing four begins on next page)

Listing Four

```
/*
**      edchar.c -- character editing functions for the fx80
**  character editor program.
*/

#include        <stdio.h>
#include        <fx80.h>


/* unpacked version of the character being edited */
static  unsigned        cary[WIDTH][HEIGHT];


/*
**      edchar -- the main character editing function.
*/

VOID    edchar()
{
int     c;
unsigned        myx, myy, charnum;

        gotoxy(0, PROMPTLINE - 1);
        fputs("Edit", stdout);
        charnum = selchar();
        clrscrn();
        drwbox();
        longstat(charnum);
        unpack(charnum);
        flashit(charnum);
        myx = 1;
        myy = 1;
        boxgotoxy(myx, myy);

        do
        {
                switch (c = (islower(c = mygetc()) ? toupper(c) : c))
                {

                        case '8':               /* up */
                        case CTRLE:
                                if (myy < HEIGHT)
                                        boxgotoxy(myx, ++myy);
                                break;

                        case '2':               /* down */
                        case CTRLX:
                                if (myy > 1)
                                        boxgotoxy(myx, --myy);
                                break;

                        case '4':               /* left */
                        case CTRLS:
                                if (myx > 1)
                                        boxgotoxy(--myx, myy);
                                break;

                        case '6':               /* right */
                        case CTRLD:
                                if (myx < WIDTH)
                                        boxgotoxy(++myx, myy);
                                break;

                        case '5':               /* toggle pixel */
                        case CTRLF:
                                if (cary[myx - 1][myy - 1])
                                {
                                        if ((myx - 1)%2)
```

(Continued on page 52)

```
                                        putc('|', stdout);
                                else
                                        putc(' ', stdout);
                                putc('\b', stdout);
                                cary[myx - 1][myy - 1] = 0;
                        }
                        else
                        {
                                revon();
                                putc(' ', stdout);
                                revoff();
                                putc('\b', stdout);
                                cary[myx - 1][myy - 1] = 1;
                        }
                        break;

                case 'Q':               /* quit */
                case CTRLQ:
                        c = 'Q';
                        break;

                default:
                        putc(BELL, stdout);
                }
        } while (c != 'Q');

        packit(charnum);
        clrscrn();
}


/*
**      drwbox -- draw the character easel.
*/

VOID    drwbox()
{
unsigned        i, j;

        gotoxy(BOXX, BOXY);
        putc('+', stdout);
        for (i = 0; i < WIDTH; i++)
                putc('-', stdout);
        putc('+', stdout);

        for (i = 1; i <= HEIGHT; i++)
        {
                gotoxy(BOXX, BOXY + i);
                putc('|', stdout);
                for (j = 0; j < WIDTH; j++)
                        putc(' ', stdout);
                putc('|', stdout);

         }

         gotoxy(BOXX, YOFF);
         putc('+', stdout);
         for (i = 0; i < WIDTH; i++)
                 putc('-', stdout);
         putc('+', stdout);
 }


 /*
 **      unpack -- unpack the selected character.
 */

 VOID    unpack(cnum)
 unsigned        cnum;
 {
```

```
unsigned        i, j, mypack, loj, hij;
struct  chrary  *myptr;

        for (i = 0; i < WIDTH; i++)
                for (j = 0; j < HEIGHT; j++)
                        cary[i][j] = 0;

        myptr = chrs;
        for (i = 0; i < cnum; i++)
                myptr++;

        if (myptr->attrib & 128)
        {
                loj = 1;
                hij = HEIGHT;
        }
        else
        {
                loj = 0;
                hij = (HEIGHT - 1);
        }

        for (i = 0; i < WIDTH; i++)
        {
                mypack = myptr->ary[i];
                for (j = loj; j < hij; j++)
                {
                        cary[i][j] = mypack & 1;
                        mypack >>= 1;
                }
        }
}


/*
**      packit -- pack the selected character.
*/

VOID    packit(cnum)
unsigned        cnum;
{
unsigned        i, j, mypack, loj, hij, wantlo, wanthi, adjacent;
struct  chrary  *myptr;

/* move pointer to appropriate character */
myptr = chrs;
for (i = 0; i < cnum; i++)
        myptr++;

/* determine if upper or lower 8 pins requested */
wantlo = 0;
for (i = 0; i < WIDTH; i++)
        wantlo |= cary[i][0];

if (wantlo)
{
        loj = 0;
        hij = HEIGHT - 1;
        myptr->attrib = 11;
}
else
{
        loj = 1;
        hij = HEIGHT;
        myptr->attrib = 139;
}

/* pack the character */
for (i = 0; i < WIDTH; i++)
{
        mypack = 0;
        for (j = loj; j < hij; j++)
        {
                mypack >>= 1;
                if (cary[i][j])
                        mypack |= 128;
        }
```

(Continued on next page)

```
                myptr->ary[i] = mypack;
        }

        /* determine if upper 8 pins requested */
        wanthi = 0;
        for (i = 0; i < WIDTH; i++)
                wanthi |= cary[i][(HEIGHT - 1)];

        /* determine if adjacent pins requested */
        adjacent = 0;
        for (i = 0; i < (WIDTH - 1); i++)
        {
                if (adjacent = (myptr->ary[i] & myptr->ary[i + 1]))
                        break;
        }

        /* check for definition violation */
        if (adjacent || (wanthi && wantlow))
        {
                gotoxy(0, PROMPTLINE);
                fputs("This definition is illegal.\n", stderr);
                pause();
        }
}

/*
**      flashit -- draw selected character.
*/

VOID    flashit(cnum)
unsigned        cnum;
{
unsigned        i, j;

        for (i = 0; i < WIDTH; i++)
                for (j = 0; j < HEIGHT; j++)
                {
                        boxgotoxy(i + 1, j + 1);
                        if (cary[i][j])
                        {
                                revon();
                                putc(' ', stdout);
                                revoff();
                        }
                        else if (i%2)
                                putc('|', stdout);
                        else
                                putc(' ', stdout);
                }
}


/*
**      boxgotoxy -- goto x,y position in character editing box.
*/

VOID    boxgotoxy(x, y)
unsigned        x, y;
{
        gotoxy(x + BOXX, YOFF - y);
}
```

End Listing Four

(Listing five begins on page 56)

```
/*
**      prnout.c -- the function to print out the entire character
**  set of the FX80 printer.  The function printall() will print the
**  characters taking into account special handling required to
**  print characters stored in the locations of control codes.  The
**  only character not printed is that stored in 127, the alternate
**  zero.  I can't figure out how to do it.
*/


#include        <stdio.h>
#include        <fx80.h>


/*
**      printall -- print the entire character set.
*/
VOID    printall()
{
unsigned        i;

        for (i = 0; i < NUMCHARS; i++)
        {
                if (!(i%64))            /* line feed every 64 characters */
                        pputc('\n');

                if (isprint(i & 0x7f))  /* normal Roman or Italic */
                        pputc(i);
                else if (i >= 128)      /* high order control code */
                {
                        pputc(ESC);
                        pputc('6');
                        pputc(i);
                        pputc(ESC);
                        pputc('7');
                }
                else if (isspecial(i))  /* special low order control code */
                        pspecial(i);
                else                    /* low order control code */
                {
                        if (i == 127)
                                continue;       /* swallow it */

                        pputc(ESC);
                        pputc('I');
                        pputc('1');
                        pputc(i);
                        pputc(ESC);
                        pputc('I');
                        pputc('0');
                }
        }
        pputc('\n');
        pputc('\n');
}


/*
**      isspecial -- return TRUE if argument is low order control
**  code requiring special handling to print.

*/

int     isspecial(i)
unsigned        i;
{
        return ((i >= 7 && i <= 15) || (i >= 17 && i <= 20) ||
                i == 24 || i == 27);
}
```

```
/*
**      pspecial -- print control codes that require special handling.
*/

VOID    pspecial(i)
unsigned        i;
{
        pputc(ESC);     /* all select an international character set */
        pputc('R');

        switch (i)
        {
                case 7:
                        pputc(7);
                        pputc(91);
                        break;
                case 8:
                        pputc(7);
                        pputc(93);
                        break;
                case 9:
                        pputc(7);
                        pputc(92);
                        break;
                case 10:
                        pputc(7);
                        pputc(124);
                        break;
                case 11:
                        pputc(5);
                        pputc(36);
                        break;
                case 12:
                        pputc(7);
                        pputc(35);
                        break;
                case 13:
                        pputc(4);
                        pputc(93);
                        break;
                case 14:
                        pputc(4);
                        pputc(125);
                        break;
                case 15:
                        pputc(1);
                        pputc(92);
                        break;
                case 17:
                        pputc(2);
                        pputc(126);
                        break;
                case 18:
                        pputc(4);
                        pputc(91);
                        break;
                case 19:
                        pputc(4);
                        pputc(123);
                        break;
                case 20:
                        pputc(4);
                        pputc(92);
                        break;
                case 24:
                        pputc(2);
                        pputc(92);
                        break;
                case 27:
                        pputc(5);
                        pputc(124);
        }

        pputc(ESC);             /* Return to USA font */
        pputc('R');
        pputc(0);
}
```

End Listings

A Magic Mushroom for the Epson Alice

A Practical Program for Printing Phonetic Characters

by Gary B. Palmer

I am using an Apple II Plus microcomputer and an Epson MX-100 printer with Graftrax Plus to print stories of the Coeur d'Alene Indians in their native language; this requires phonetic characters. Still spoken fluently by a handful of elderly Indians on the reservation in northern Idaho, Coeur d'Alene belongs to a family of languages known for their difficult phonetic systems. Linguists who record Indian languages of the American Northwest have designed a character set that is visually elegant but extremely difficult to reproduce on a typewriter.

Table 1 (page 59) shows the Coeur d'Alene characters. Entering a single phonetic character with a customized linguistic element may require the typist to use as many as six keystrokes, often involving keys located in awkward positions. With phonetically complex languages, the tedium and frustration of keyboard entry send the probability of errors soaring.

An exploration of phonetic printing and rubberized fonts for the Epson MX-100.

To those of us working with Indian languages, microcomputers promise convenient transcription, rapid processing, and accurate printing.

I begin on the reservation by recording the stories on audio cassette tape. After transcribing a text, using phonetic symbols, I work with the Coeur d'Alenes to translate the story into English. I use the word processor Applewriter II to enter both the Indian text and the translation into the computer, but I do not use the computer to translate the text.

Gary B. Palmer, Center for Computer Applications in the Humanities, University of Nevada, Las Vegas, NV 89154.

What a Linguist Needs

Automatic printing of bilingual texts on the Epson MX series of printers requires graphics software. Such software is readily available for the Apple Macintosh, which prints excellent characters on the ImageWriter dot matrix printer. Other late model dot matrix printers, such as the Epson FX series that replaced the MX, permit loading of user-defined fonts. This is a great convenience for linguists, but if your available printer belongs to the MX series, that is small comfort. Although the MX-100 does not permit loading of fonts, you can still print user-defined fonts by sending graphics commands to the printer.

Linguists have a great interest in displaying alternate character sets on the CRT screen. On the Apple II, this is easily accomplished with the appropriate 80 column card or with a patch to the word processor's character set if the characters are created by the software. But when you wish to produce a manuscript, the creation of alternate character sets on the printer becomes an issue. It is important to realize that the printer and the screen are independent, a fact that is not at all transparent to newcomers.

Linguists who wish to switch from one language to another at will, mix-

ing codes in lines of text, will discover that the limitations of the standard keyboard require them to select a set of characters to represent the phonetic symbols for text entry: each key must correspond to an ASCII code *number*—the only kind of signal acceptable to the printer.

Only 52 keys are available for alphabetical use on the Apple keyboard (ASCII numbers 65 to 90 and 97 to 122). Moreover, the Apple II Plus will send only the first 128, not enough to support an additional concurrent set even if the new set shares many symbols with the standard set. Because the control codes below ASCII 32 are needed to drive the printer, and because standard punctuation marks, numerals, and underlining are commonly needed in the text, the set of characters that can be sacrificed to provide alternative characters is quite small: #, $, %, &, *, +, /, <, =, >, @, [,], ^, {, ¦, }, ~. Of these, several (@, [,], ^, {, ¦, }, ~,) require awkward combinations of shift and control characters that are difficult to remember and to enter, at least with Applewriter II Plus. This inconvenience defeats our original goal of facilitating manuscript preparation.

How To Get It

The solution lies not on the keyboard but in the software. In the text file, the standard character set can represent the phonetic set; a printer driver written in APPLESOFT BASIC defines the phonetic characters. Code inserted in the text signals character set changes. A signal to change sets can be entered as one or more nonalphanumeric symbols, and each unique signal makes available at least 52 new shapes. Because the printer can create a practically unlimited number of shapes, it is possible to access many alternate character sets from a single text.

A special problem arises if a character exceeds the standard 7/72-inch height. Because the MX series lacks reverse line feeds and expanded character matrices, the vertical extension must be printed before the remainder of the character. This means the upper story of *every* extended character on a line must be printed before advancing the line feed.

The program described in this article creates vertically extended fonts and generates one alternate character set that can be mixed at will with English. The program does word wrap, avoids breaking words at the end of lines, numbers pages, and permits variable line spacing. Definitions are entered as data lines (twelve data points/character).

To produce a phonetic symbol, it gets one character at a time from the text file, replacing the ASCII number with a string of graphics commands. Each string of code, representing a phonetic symbol, is placed in one element of an array of 60, the length of a line. Decoding and printing the graphics set is slower than printing English but still very practical compared to performing the same task with a modified typewriter.

Convenient Text Entry

Entry codes must be easy to remember and to type. I use a *system that* assigns a single key to frequently occurring phonetic features, such as stressing and rounding. This avoids the necessity of remembering a separate key for each stressed vowel or rounded consonant.

The system requires no unusual substitutions (for one who understands Coeur d'Alene phonetics); it does not resort to otherwise little-used capital Zs , Qs, numbers, or special keys. These can all be kept for their normal uses or for character set change flags. Table 2 (below) lists some representative characters from

These are the Coeur d'Alene vowels: a ɛ ə i ɪ o ɔ u.
These are the Coeur d'Alene consonants: b c ć c̆ c̆́ d gʷ h j
kʷ k̓ʷ l ĺ l̆ m m̓ n n̓ p p̓ q q̓ qʷ q̓ʷ r r̓ R R̓ Rʷ R̓ʷ s s̆ t t̓ w w̓
xʷ x̱ x̱ʷ y y̓ ʔ. These are English vowels often found in
Coeur d'Alene texts: a ɛ ə i ɪ ɔ o u ʌ. These are English
consonants often found in Coeur d'Alene texts: ð θ ŋ. Any
vowel can be printed with stress, as in the following: á é
ə́ í ɪ́ ó ɔ́ ú ʌ́.

Table 1
Coeur d'Alene Phonetics

CONSONANTS			VOWELS		
Entry	WPL	Phonetic	Entry	WPL	Phonetic
c	c	c	e	e	ɛ
c'	c'	c̆	e!	e!	ɛ́
C	C	c̆	E	E	ə
C'	T	c̆́	i!	i!	í
gw	gW	gʷ	I	I	ɪ
'	H	ʔ	o	o	ɔ
j	j	j	O	O	o
kw	kW	kʷ	u	u	u
l'	l'	ĺ			
L	L	l̆			
q'	q'	q̓			
qw	qW	qʷ			
q'w	q'W	q̓ʷ			
R	R	R			
R'	R'	R̓			
Rw	RW	Rʷ			
R'w	R'W	R̓ʷ			
s	s	s			
S	S	s̆			
w	w	w			
xw	xW	xʷ			
X	X	x̱			
Xw	XW	x̱ʷ			

Table 2
Phonetic Characters for the Coeur d'Alene Language, in Entry Code, after Processing by A WPL Filter Program, and in Final Form.

the set of Coeur d'Alene phonetics.

All entries bear some resemblance to the actual phonetic symbols that they represent. All are either lower-case letters, capitals, or simple double keystrokes called "digraphs." Speakers of Coeur d'Alene pronounce several phonemes with a catch in the throat, so I appended the apostrophe symbol to those phonemes for all such glottal sounds. Likewise, several phonemes are produced with a rounding of the lips, so I appended the lower-case w for all such labial sounds. Because the symbols are easy to remember and the same keyboard pattern serves for both languages, text entry is simple.

Any good word processor permits similar entry, but Applewriter II has another useful feature: it comes with its own programming language, called Word Processing Language (WPL), which is designed for string manipulation. When preparing long text files for the printer driver, I use a WPL filter to convert some digraphs into single characters. It also prefixes lines of English and Coeur d'Alene with change codes. Table 2 shows the two transformations required to produce phonetic text from my entry code. Although not strictly necessary, the WPL filter does simplify text entry.

Graphics Fonts on the MX-100

A few hours of experimentation will disclose the difficulty of printing text files with Epson graphics characters. Taking pity on me, a colleague called my attention to the Gutenberg software, which he thought might work. Having just stretched the department budget to purchase a computer and printer, however, we could not go back again for an unproven software item costing over $300. For those with a CP/M card, the Fancy Font program would serve the purpose, but the cost made this option unavailable to us. Not wishing to risk buying software of uncertain utility, I opted for writing a printer driver: a program to act as a filter to replace ASCII characters with graphics.

Creating special fonts requires some basic arithmetic. On the MX-100, English consonants normally occupy up to 7/72 inch above the baseline and 2/72 inch below (see Figure 1, below). The remaining 3/72 inch in each normal line feed of 12/72 inch provides the space between lines. All vowels extend only 5/72 inch above the baseline except i, which extends 7/72 inch. Each character is printed within the space of five columns of dots. A trailing sixth column provides spacing between letters.

Standard fonts are mapped in the printer, and when the printer slot is addressed (normally with PR#1), a single ASCII code number prints a character. For example, the command PRINT CHR$(65) prints A. The Apple printer and newer Epson printers such as the FX-80 can load user-defined fonts and then treat them in much the same way as standard character sets. With the MX-100, however, alternate fonts require graphics code from the computer each time a character is printed.

On the MX-100, each column in a

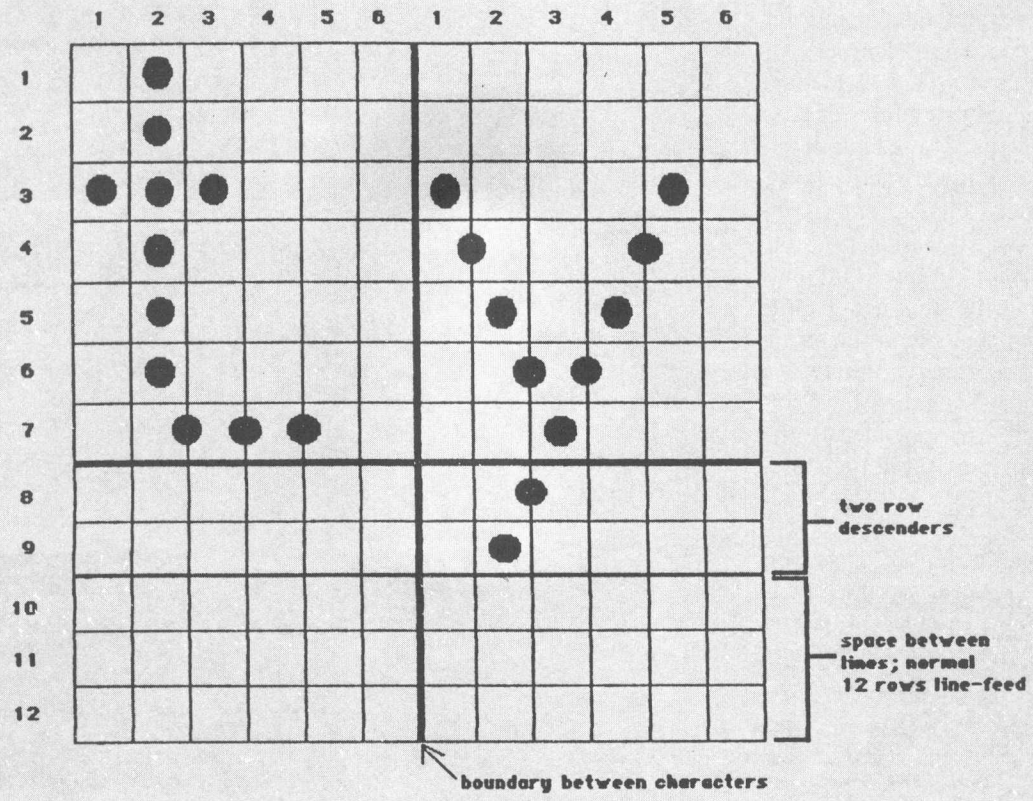


Figure 1
Standard Lower-Case t and y on the Epson MX-100

character's six-column matrix is sent with a code number. Code specifying the shape of these columns must be prefixed by a graphics command. A high-density mode also must be called to generate well-defined characters; because this mode moves the printing head only half a column at a time, the graphics command creates twelve-column characters. Take as an example the command: PRINT CHR$(27) + CHR$(76) + CHR$(12) + CHR$(0). CHR$(27) says to treat the following characters as commands rather than ASCII symbols. CHR$(76) calls the high-density graphics mode. CHR$(12) sets aside twelve columns of dots. CHR$(0) signifies that less than 256 columns follow.

The high-density mode produces definition and density equal to that of standard fonts printed in emphasized mode. In the printer driver (see Listing One, page 66), the twelve data points necessary to define the overlapping columns of a phonetic symbol are provided in DATA lines 1050 through 1170. Each data point represents a column seven dots high. Figure 2 (page 63) illustrates data points for the graphics character "schwa." Although the Epson MX-100 manual describes graphics procedures, it does not mention the use of the high-density mode for designing attractive fonts.

The complete character is constructed by concatenation. As the READ G statement in line 1000 finds each data point, the statement G$(M) + CHR$(G) appends it to the string of data that represents the twelve printed columns. Then the twelve columns in G$(M) are appended to the graphics command GC$ in the statement G$(M) = GC$ + G$(M). Finally, the command PRINT G$(M) will print the phonetic symbol. However, this command is never given explicitly, because the value of G$(M) is first transferred to yet another array element that represents the top or bottom sector of a character.

Although standard characters can extend as high as 7 rows above the baseline, a graphics column can be only 6/72 of an inch high. Likewise, the graphics column extends only 1/72 of an inch below the baseline, compared to the 2/72 drop of the standard j, g, or y. This is because the Apple II Plus can address only seven bits; it prints only six dots above and one dot below. For many purposes, seven rows will suffice, but if one needs taller fonts, there is a solution.

Vertical Extension of Fonts

Some applications, such as Chinese characters, are too complex for a seven-by-six dot matrix. But for Epson Alices, the printer driver offers a magic mushroom: smaller line feeds can make fonts grow infinitely tall. Two normal lines spliced together make one tall line, and ten lines spliced together make a very tall line. Split characters are aligned horizontally by printing lines in multiple arrays of corresponding elements and are joined vertically by controlling line feeds.

In the Coeur d'Alene set, diacriticals signify phonetic features; the most frequent is the stress mark on vowels, represented in entry code with !. If the MX-100 supported reverse line feeds, we could backspace with PRINT CHR$(8);, send a one-half reverse line feed, print the character, and send a half line feed for-

ward. Because reverse line feeds are unavailable, the stress mark must be printed first, followed by a partial line feed, and then the vowel.

Similarly, each character with a vertical extension has a top sector and a bottom sector. If the stress mark on top is T$(1), the vowel on the bottom is B$(1), and PLF$ represents a partial line feed command, then PRINT T$(1):PRINT PLF$:PRINT B$(1) will print the full phonetic symbol.

I designed the diacriticals so that a partial line feed of 4/72 inch adjusts them to vowels of standard height (see Figure 3, page 63). The command is TF$ = CHR$(27) + CHR$(65) + CHR$(4) (see line 1240 in the listing). Following the escape character, CHR$(65) resets the line spacing and CHR$(4) sets the spacing at four rows of dots.

To avoid a cascade effect, the program prints an entire line of diacriticals first, followed by a partial line feed, and then a line of base characters. The diacriticals must be stored in the array T$(X), where X is the character counter. Baseline characters are stored in the array B$(X). If a phonetic symbol lacks a diacritical, a blank must be printed in the top sector to maintain proper alignment between the top and bottom of subsequent complex symbols; this is accomplished by initializing the whole T$(X) array with blanks before getting a line.

The code for a stress mark, !, is entered into the text file after its vowel. We then send the corresponding graphics mark to its proper destination over the vowel by placing it in T$(X – 1) rather than T$(X). X is decremented by one so the next character will be assigned to T$(X) rather than T$(X + 1) (see line 200 in the listing). The same procedure is used for the graphics apostrophe in line 250.

When vertically extended characters are represented by a single character in the text file, both the top and bottom sectors of the character can be assigned directly to corresponding array elements. For example, T represents the standard c with a diacritical cap that looks like a v with a nested apostrophe. The graphics cap is assigned to the T$(X) array in line 270, while the standard Epson c goes into B$(X) (see Figure 4, page 63). A column can be raised one row by doubling its number. To raise an entire character one row, double the data point values for each of the twelve columns.

The example in Figure 4 illustrates vertical extension by setting a graphics character on top of a standard font. It would work just as well in reverse, if that produced the desired shape, or you could use graphics in both the top and the bottom sectors, as I have done with the character "eth" (see Figure 5, page 63). Overstriking, superscripting, and subscripting provide additional possibilities accessible to either the top or bottom sectors in this flexible system; however, I have only been able to achieve overstriking on standard characters.

The Program

Because characters consume more memory than numerals, text processing runs more slowly than calculations. You can accelerate a program by placing frequently used subroutines near the beginning and by storing string constants in variables; initialization routines are placed near the end. Thus, program execution first GOSUBs to line 850, where it sets the margins, length of lines, and flags, then it dimensions the top and bottom sector arrays, T$(X) and B$(X).

Beginning with line 980, two nested FOR/NEXT loops define the phonetic set. The inside or N loop beginning on line 990 reads one data line to build a graphics string by concatenation; each data point represents one high-density column. At line 1030, the column descriptor string is prefixed with the graphics command. The outside or M loop beginning with line 980 cycles the process through all the data lines.

Commonly used sequences of printer commands for emphasis, superscripting, and partial and normal line feeds are assigned to string variables from line 1190. This segment also defines the two overstrike characters ɬ and x.

From line 1320, a DOS subroutine catalogs the text diskette, prompts for the name of the text file, verifies the name, opens the file, and opens the printer port, PR#1. Notice in lines 1500 and 1510 that the command to the printer precedes the PRINT D$;"READ";T$ command to the disk drive. Why this particular order is necessary remains one of those mysteries of the silicon cosmos, but the program will not work properly if the statements are reversed.

After the catalog has been listed and the file name entered, the program prompts for the desired line spacing with a default value of 0 for single spacing. The IF/THEN statement at line 1500 rejects spacings less than zero or greater than nine.

Before getting the first character from the text file, the program performs one more task: a subroutine beginning on line 650 assigns blanks to all elements of the T$(X) and B$(X) arrays. The blanks in the top sector remain, unless replaced by diacriticals or partial fonts, to guarantee proper horizontal alignment of diacriticals to character base sectors.

After initialization, execution drops to line 100 to get the first character (A$) from the text file and to increment the character counter X. Because both arrays have been initialized with blanks, a blank in A$ does not require an assignment; execution is routed to line 90, which checks for line length. If A$ is not a blank, lines 120 and 130 check whether A$ is a flag for English (ENG$ = "^") or for Coeur d'Alene (CDA$ = "+"). If A$ is a flag, the result is stored in FLAG$ and execution returns to line 100 to get another character.

Because flags are not printed, the counter is decremented by one. The flag variable in line 150 directs subsequent traffic. When FLAG$ is English, A$ is assigned directly to an element of the B$(X) array at line 150; T$(X) remains a blank, as initialized. When the flag variable switches to Coeur d'Alene, execution branches to line 180 where a graphics or backspace-and-overstrike sequence corresponding to A$ is assigned to B$(X).

All paths to the printing subroutine pass through carriage-return and end-of-line checks. If A$ is a carriage

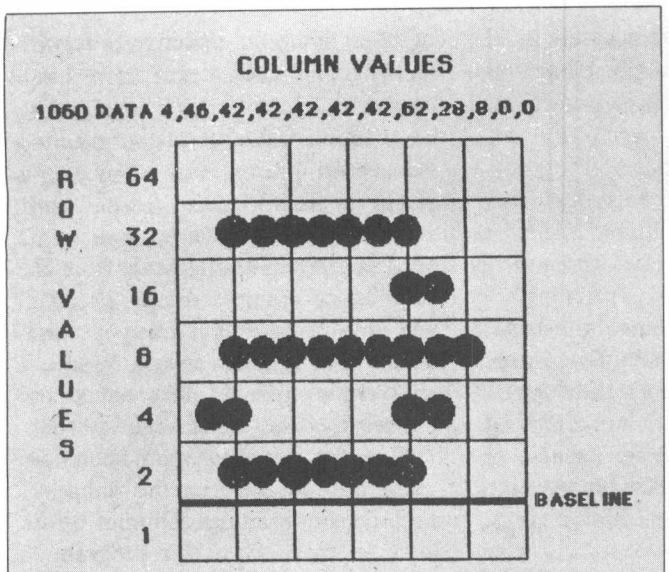


Figure 2
Data Points for the Character "Schwa" (Compare column values with Listing One, line 1060.)

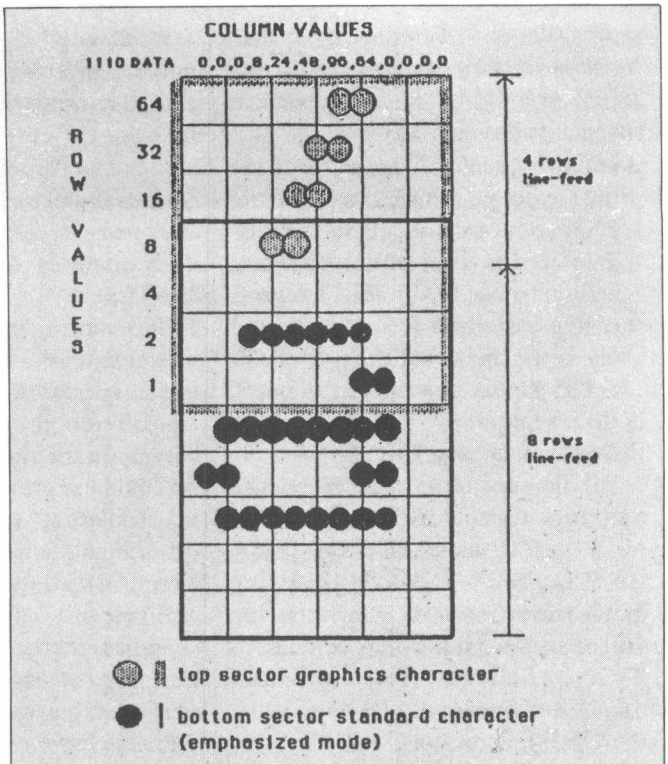


Figure 3
Alignment of Graphics Stress Character by Means of Partial Line Feed (Compare Listing One, lines 200 and 1110.)

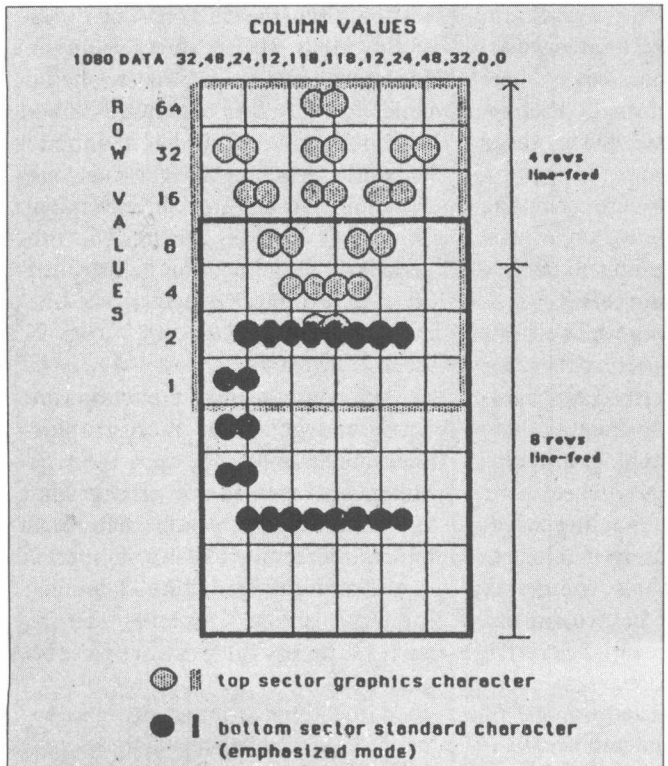


Figure 4
Partial Overlay of Graphics Diacritical on Standard Lower-Case c to Print Composite Character (Compare Listing One, lines 270 and 1080.)

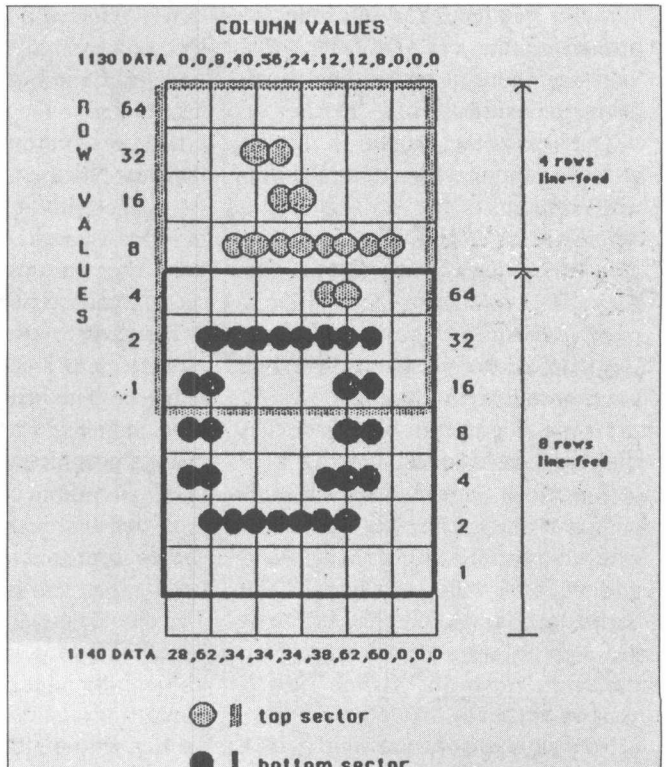


Figure 5
Vertically Extended Graphics Character "Eth" (Compare Listing One, lines 320, 1130, 1140.)

return, the end-of-line variable EOL is set as the current value of X. Because no word segment remains at the end of the line, the variable SEG is set to 0. Execution branches to the printing routine at line 410.

If a line contains more than 60 characters, the check at line 90 causes a branch to line 690; there a reverse checking procedure looks for the last blank in the line to set the end-of-line variable. It also determines the length of the trailing word segment to be carried over to the next line.

When execution passes to the printing subroutine at line 410, PRINT SPC(LM); prints the left margin. The FOR/NEXT loop beginning in program line 450 prints the top sector diacriticals contained in T$(X). Line 460 prints the partial *line feed of four* rows and increments the line feed counter, LF. To complete a line, it remains only to print the baseline characters. PRINT BF$; resets the line spacing to eight rows. For unknown reasons, the eight-row line feed cannot be printed with a single print command as is done for the four-row line feed. The left margin is printed again, and the FOR/NEXT loop beginning at program line 500 prints the baseline.

The assignments made in the loop at line 610 transfer the line-final word segment to the beginning of the following line of text. The loop beginning with program line 660 reinitializes the remainder of the two line arrays. Subsequent lines set X to the length of the segment and reinitialize the counters FIRST and SEG.

Now the program cycles back to the line feed check in line 80. More arithmetic is needed. The length of a page is 66 lines; after the last line, we print two blank lines, a line number, and six more lines for a total of nine with a bottom margin of six. The figure nine represents the bottom margin in the program. At line 780, the assignment FILL = 66 − BM − LINES says to subtract nine and the line count from 66 lines. After the command PRINT NLF$ returns the line feed status to normal, the fill lines and bottom margin are printed. Line 800 centers the page number, which is printed in line 810. The line feed counter is reinitialized to the value of the top margin (four), and the page counter PG is incremented by one.

Summary

The program in the listing prints vertically extended phonetic characters on the Epson MX-100 with Graftrax Plus by storing graphics commands in two arrays for each line. Flags inserted in the text file route ASCII codes to English or phonetic program segments. In the phonetic branch, the ASCII codes are replaced by graphics, overstrike, or superscript code sequences. A phonetic symbol may combine these modes with standard characters.

Formatting features include margins, page numbering, word wrap, and variable line spacing. English and phonetic fonts mix freely within lines of text. While perhaps not as flexible as graphics font programs that print one row at a time, a program such as this, which overlays whole lines, should print more rapidly—an important consideration when long texts are involved. If you use the same character set repeatedly, the once-only effort of programming data lines is minor compared to the advantages of using a familiar word processor for text entry and formatting.

Although these graphics routines might be unnecessary with new printers that support downloading of fonts, the system of storing partial characters in arrays and splicing lines by means of partial overlays is adaptable even to these printers. Because the program stores a phonetic character as a string of commands, it is also adaptable to daisy wheel printers, especially those that support reverse line feeds. A Coeur d'Alene orthography similar to the one created on the Epson could be printed on a daisy wheel printer with only three physically modified characters. The program would require only one line array, rather than the two arrays required to print vertically extended symbols on the MX-100.

Although many Chinese or Kanji characters would fit within the vertical format of two arrays, large and complex fonts of Chinese, Japanese, or other symbol systems may require splicing of multiple arrays. You can create wider characters by using more than twelve columns in the graphics command prefix. For example, a Kanji character could probably fit within a matrix of 16 dot rows by 32 high-density columns. Aside from the problem of graphics design, practical use of fonts with hundreds or thousands of characters would require a different system of text entry and character-to-keyboard assignment.

Once you understand the binary method of specifying the shape of high-density graphics columns in data lines, adaptation of this program to print other two-sectored fonts is relatively simple. Owners of MX series Epson printers with an interest in special character sets may find it essential. Modification of escape sequences will make it available to other printers. If you have a secret urge to print Sanskrit, runic, or Elf script, the tools are now available. Please send me listings of your data statements.

Postscript

Listing Two (page 70) contains some of the more commonly used special characters of the International Phonetic Alphabet taken from Robert Albright's article "The International Phonetic Alphabet" (*International Journal of American Linguistics,* vol. 24, no. 1, January 1958). The program and data lines can be substituted as needed into the sections of Listing One titled "*Assign Fonts To Array Elements*" and "*Graphics Fonts.*" The choice of an appropriate entry character for each graphics character will depend upon the organization of the user's orthography. Few applications would require all the characters. Avoid the numbers 7, 8, and 9 in graphics data statements for the Epson MX printers; they result in beeps and unpredictable graphics.

DDJ

(Listing begins on page 66)

Magic Mushroom for the Alice (Text begins on page 58)

Listing One

```
 10  ONERR GOTO 1630
 20  TEXT : HOME
 30  REM ------------------------------------------------------------
 40  GOSUB 850: GOSUB 650: REM INITIALIZE
 50  REM ------------------------------------------------------------
 60  REM              GET CHARACTER AND SET LANGUAGE FLAGS
 70  REM ------------------------------------------------------------
 80  IF LINES >  = 66 - BM THEN  GOSUB 760: REM END-OF-PAGE
 90  IF X = LL THEN  GOSUB 690: GOSUB 410: GOTO 80: REM EOL,PRINT,PG
100  GET A$: X = X + 1
110  IF A$ = " " THEN 90
120  IF A$ = ENG$ THEN FLAG$ = ENG$: X = X - 1: GOTO 100
130  IF A$ = CDA$ THEN FLAG$ = CDA$: X = X - 1: GOTO 100
140  IF A$ = CR$ THEN EOL = X:SEG = 0: GOSUB 410: GOTO 80: REM PRINT
150  IF FLAG$ = ENG$ THEN B$(X) = A$: GOTO 90
160  GOSUB 180: GOTO 90
170  REM ------------------------------------------------------------
180  REM                ASSIGN FONTS TO ARRAY ELEMENTS
190  REM ------------------------------------------------------------
200  IF A$ = "!" THEN :T$(X - 1) = G$(7):X = X - 1: RETURN
210  IF A$ = "E" THEN B$(X) = G$(2): RETURN
220  IF A$ =  CHR$ (101) THEN B$(X) = G$(13): RETURN
230  IF A$ = "W" THEN B$(X) = SS$: RETURN
240  IF A$ = "H" THEN B$(X) = G$(1): RETURN
250  IF A$ = "'" THEN T$(X - 1) = G$(6):X = X - 1: RETURN
260  IF A$ = "C" THEN B$(X) =  CHR$ (99):T$(X) = G$(5): RETURN
270  IF A$ = "T" THEN B$(X) =  CHR$ (99):T$(X) = G$(4): RETURN
280  IF A$ = "L" THEN B$(X) = LT$: RETURN
290  IF A$ = "S" THEN B$(X) =  CHR$ (115):T$(X) = G$(5): RETURN
300  IF A$ = "X" THEN B$(X) = BX$: RETURN
310  IF A$ = "I" THEN B$(X) = G$(3): RETURN
320  IF A$ = "D" THEN T$(X) = G$(9):B$(X) = G$(10): RETURN
330  IF A$ = "G" THEN B$(X) = "O" +  CHR$ (8) + "-": RETURN
340  IF A$ =  CHR$ (106) THEN B$(X) = A$:T$(X) = G$(5): RETURN
350  IF A$ = "N" THEN B$(X) = G$(8): RETURN
360  IF A$ =  CHR$ (111) THEN B$(X) = G$(11): RETURN
370  IF A$ = "O" THEN B$(X) =  CHR$ (111): RETURN
380  IF A$ = "U" THEN B$(X) = G$(12): RETURN
390  B$(X) = A$: RETURN
400  REM ------------------------------------------------------------
410  REM                              PRINT
420  REM ------------------------------------------------------------
430  REM ** PRINT TOP SECTOR
440  PRINT  SPC( LM);: REM LEFT MARGIN
450  FOR PR = 1 TO EOL - 1: PRINT T$(PR);: NEXT
460  PRINT TF$: REM 4 ROWS LINE-FEED
470  PRINT BF$;: REM 8 ROWS LINE-FEED
480  REM ** PRINT BOTTOM SECTOR
490  PRINT  SPC( LM);: REM LEFT MARGIN
500  FOR PR = 1 TO EOL - 1: PRINT B$(PR);: NEXT
510  PRINT :LINES = LINES + 1
520  REM LINE-SPACING
530  IF S = 0 THEN  GOTO 590
540  PRINT NLF$;
550  FOR L = 1 TO S
560  PRINT :LINES = LINES + 1
570  IF LINES >  = 66 - BM THEN  GOSUB 760: GOTO 590: REM PAGINATION
580  NEXT
590  REM ** CARRY OVER WORD SEGMENT
600  IF SEG = 0 THEN 650
610  FOR FIRST = 1 TO SEG
620      T$(FIRST) = T$(EOL + FIRST)
630      B$(FIRST) = B$(EOL + FIRST)
640  NEXT FIRST
650  REM ** INIT ARRAYS
660  FOR X = FIRST TO LL:T$(X) = " ":B$(X) = " ": NEXT
670  X = SEG:FIRST = 1:SEG = 0: RETURN
680  REM ------------------------------------------------------------
690  REM                          FIND END-OF-LINE
700  REM ------------------------------------------------------------
710  FOR EOL = LL TO 1 STEP  - 1
720      IF B$(EOL) = " " THEN SEG = LL - EOL: RETURN
730  NEXT EOL
```

(Continued on page 68)

Magic Mushroom for the Alice (Listing Continued, text begins on page 58)

Listing One

```
740  EOL = LL + 1: RETURN
750  REM ------------------------------------------------------------
760  REM                          PAGINATION
770  REM ------------------------------------------------------------
780  FILL = 66 - BM - LINES : PRINT NLF$;
790  IF FILL > 0 THEN  FOR L = 1 TO FILL: PRINT : NEXT
800  PRINT : PRINT : CENTER = LM + LL / 2 - 3
810  PRINT  SPC( CENTER);"- ";PG;" -": FOR L = 1 TO 6: PRINT : NEXT
820  FOR L = 1 TO TM: PRINT : NEXT
830  LINES = TM:PG = PG + 1: RETURN
840  REM ------------------------------------------------------------
850  REM          INITIALIZE VARIABLES AND DIMENSION ARRAYS
860  REM ------------------------------------------------------------
870  VTAB 10: HTAB 12: PRINT "INITIALIZING"
880  TM = 4:BM = 9:LINES = TM:PG = 1: REM TOP AND BOTTOM MARG, PAGE
890  LM = 10:RM = 70: LL = RM - LM: REM MARGINS, LENGTH OF LINE
900  CDA$ = "+": ENG$ = "^": REM COEUR D'ALENE AND ENGLISH FLAGS
910  FLAG$ = "+": REM INIT TO COEUR D'ALENE
920  DIM T$(LL): DIM B$(LL): REM TOP- AND BOTTOM-OF-LINE SECTORS
930  REM ------------------------------------------------------------
940  REM                        GRAPHICS FONTS
950  REM ------------------------------------------------------------
960  DIM G$(13): REM FONTS
970  GC$ =  CHR$ (27) +  CHR$ (76) +  CHR$ (12) +  CHR$ (0): REM
          GRAPHICS COMMAND
980  FOR M = 1 TO 13: REM READ FONTS
990      FOR N = 1 TO 12: REM READ COLUMNS
1000         READ G
1010         G$(M) = G$(M) +  CHR$ (G)
1020     NEXT N
1030     G$(M) = GC$ + G$(M)
1040 NEXT M
1050 DATA 32,96,78,78,72,72,120,48,0,0,0,0: REM G$(1) GLOTTAL STOP
1060 DATA 4,46,42,42,42,42,42,62,28,8,0,0: REM G$(2) SCHWA
1070 DATA 0,0,0,0,32,62,62,2,2,0,0,0: REM G$(3) IOTA
1080 DATA 32,48,24,12,118,118,12,24,48,32,0,0: REM G$(4) EJEC PALATAL
1090 DATA 32,48,24,12,6,6,12,24,48,32,0,0: REM G$(5) PALATAL
1100 DATA 0,0,0,0,80,80,96,96,0,0,0,0: REM G$(6) EJECTIVE
1110 DATA 0,0,0,8,24,48,96,64,0,0,0,0: REM G$(7) STRESS
1120 DATA 62,62,32,32,33,33,33,33,63,30,0,0: REM G$(8) NASAL VELAR
1130 DATA 0,0,8,40,56,24,12,12,8,8,0,0: REM G$(9) ETH TOP SECTOR
1140 DATA 28,62,34,34,34,34,38,62,60,0,0,0: REM G$(10) ETH BOT SECT
1150 DATA 16,48,34,34,34,34,34,54,62,28,0,0: REM G$(11) OPEN "O"
1160 DATA 14,14,18,18,32,32,18,18,14,14,0,0: REM G$(12) LAX MD-CNT VL
1170 DATA 20,62,42,42,42,42,42,42,0,0,0,0: REM G$(13) LOW-MID "E"
1180 REM ------------------------------------------------------------
1190 REM                        PRINTER COMMANDS
1200 REM ------------------------------------------------------------
1210 CR$  =  CHR$ (13)
1220 EM$  =  CHR$ (27) +  CHR$ (69): REM  EMPHASIS
1230 OFF$ =  CHR$ (27) +  CHR$ (70): REM TURN OFF SUP/SUBSCRIPT
1240 TF$  =  CHR$ (27) +  CHR$ (65) +  CHR$ (4): REM 4 DOTS LINE-FEED
1250 BF$  =  CHR$ (27) +  CHR$ (65) +  CHR$ (8): REM 8 DOTS LINE-FEED
1260 NLF$ =  CHR$ (27) +  CHR$ (65) +  CHR$ (12): REM NORMAL LN-FEED
1270 RS$  =  CHR$ (27) +  CHR$ (64): REM RESET
1280 SS$  =  OFF$ +  CHR$ (27) + "S" +  CHR$ (0) +  CHR$ (119) +
          CHR$ (27) + CHR$ (72) + EM$: REM SUPERSCRIPT COMMAND
1290 BX$  =  CHR$ (27) + "-" +  CHR$ (1) +  CHR$ (120) + CHR$ (27) +
          "-" +  CHR$ (0): REM BARRED "X"
1300 LT$  =  CHR$ (108) +  CHR$ (8) +  CHR$ (126): REM "L" BKSP TILDA
1310 REM ------------------------------------------------------------
1320 REM                             DOS
1330 REM ------------------------------------------------------------
1340 D$ =  CHR$ (13) +  CHR$ (4)
1350 PRINT D$;"MONI"
1360 VTAB 10: HTAB 10: PRINT "TURN ON PRINTER"
1370 VTAB 11: HTAB 10: PRINT "INSERT TEXT DISKETTE"
1380 HTAB 10: PRINT "PRESS ANY KEY TO CONTINUE";
1390 GET Q$: HOME
1400 PRINT D$;"CATALOG"
1410 INVERSE : PRINT "RIGHT DISKETTE? Y OR N> ";
1420 GET Q$: IF Q$ = "Y" THEN  PRINT : GOTO 1460
1430 NORMAL : HOME : VTAB 12: HTAB 10
```

(Continued on page 70)

Magic Mushroom for the Alice

Listing One

(Listing Continued, text begins on page 58)

```
1440  PRINT "INSERT CORRECT DISKETTE"
1450  GOTO 1380
1460  INPUT "TEXT FILE NAME> ";T$: NORMAL
1470  GOSUB 1550: REM GET LINE-SPACING
1480  PRINT D$;"VERIFY ";T$
1490  PRINT D$;"OPEN ";T$
1500  PRINT D$;"PR#1";
1510  PRINT D$;"READ ";T$
1520  PRINT EM$;
1530  FOR L = 1 TO TM - 2: PRINT : NEXT : RETURN
1540  REM ------------------------------------------------------------
1550  REM                          LINE-SPACING
1560  REM ------------------------------------------------------------
1570  HOME : VTAB 8: HTAB 10: PRINT "DEFAULT SPACING> 0. RETURN"
1580  PRINT : HTAB 18: PRINT "SPACING> ";: INPUT S$
1590  IF S$ = "" THEN S = 0: RETURN
1600  IF  ASC (S$) < 48 OR  ASC (S$) > 57 THEN 1550
1610  S =  VAL (S$): RETURN
1620  REM ------------------------------------------------------------
1630  REM                          TERMINATION
1640  REM ------------------------------------------------------------
1650  IF  PEEK (222) = 5 THEN EOL = X + 1: GOSUB 410: GOSUB 760:
      PRINT RS$: PRINT D$;"PR#0": PRINT "END OF FILE": END
1660  IF  PEEK (222) = 6 THEN : HOME : VTAB 8: HTAB 12: INVERSE :
      PRINT "FILE NOT FOUND": FOR T = 1 TO 2000: NEXT : NORMAL :
      GOSUB 1380: GOSUB 650: GOTO 60
1670  PRINT RS$: PRINT D$;"PR#0"
1680  PRINT "DOS ERROR STATUS: "; PEEK (222): END
1690  REM ------------------------------------------------------------
1700  REM                    PHONETIC COEUR D'ALENE
1710  REM             COPYWRITE © ALL RIGHTS RESERVED
1720  REM
1730  REM            GARY B. PALMER, ASSOCIATE PROFESSOR
1740  REM
1750  REM          DEPT OF ANTHROPOLOGY AND ETHNIC STUDIES
1760  REM         UNIVERSITY OF NEVADA, LAS VEGAS, NV 89154
1770  REM
1780  REM                       DECEMBER 31, 1983
1790  REM ------------------------------------------------------------
1800  REM  PRINTS ENGLISH AND PHONETIC COEUR D'ALENE.
1810  REM  MIXES CHARACTER SETS IN SAME LINE.
1820  REM  TEXT CODE CHARS: "+" = CDA, "^" = ENG.
1830  REM  CREATES VERTICALLY EXTENDED FONTS BY SPLICING TWO ARRAYS.
1840  REM  DOES WORD-WRAP, PAGINATION, AND VARIABLE SPACING.
1850  REM  ALL LOWER CASE CHARS EXCEPT "J" AND "O" PASSED TO PRINTER.
1860  REM  UPPER CASE A,B,F,J,K,M,P,Q,V,Y,Z
1870  REM  SPECIAL CHARACTERS C,D,E,G,H,I,L,N,O,S,T,U,W,X,!,'
1880  REM  REQUIRES APPLE ][ PLUS AND EPSON MX-100.
1890  REM  MARGINS: 4 TOP, 6 BOTTOM, 10 LEFT, 70 RIGHT
```

End Listing One

Listing Two

```
t       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 32,32,127,127,32,32,0,0,0,0,0,0: REM G$(M)
        DATA 0,0,12,14,2,2,2,0,0,0,0,0: REM G$(M+1)

d       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 0,2,2,2,2,2,2,126,126,0,0,0: REM G$(M)
        DATA 28,30,2,2,2,2,2,30,31,1,1,0: REM G$(M+1)

j       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 0,2,2,62,62,0,0,0,0,0,0,0: REM G$(M)
        DATA 28,28,4,28,24,0,0,0,0,0,0,0: REM G$(M+1)
```

(Continued on page 72)

Magic Mushroom for the Alice

(Listing Continued, text begins on page 58)

Listing Two

```
?       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 32,96,78,78,72,72,120,48,0,0,0,0: REM G$(M)
        (See also, Listing 1, lines 240, 1050)

m       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 28,28,32,32,28,28,33,33,31,30,0,0: REM G$(M)

n       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 62,62,32,32,32,32,62,31,1,1,3,0: REM G$(M)

p       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 1,1,63,62,32,32,32,32,62,30,0,0: REM G$(M)

q       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 62,62,32,32,33,33,33,33,63,30,0,0: REM G$(M)
        (See also Listing 1, lines 350, 1120)

r       IF A$ = "any char" THEN B$(X) = CHR$(108) + CHR$(8)
        + CHR$(126)
        (See also, Listing 1, lines 280, 1300)

h       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 30,30,2,2,2,2,2,0,0,0,0,0: REM G$(M)
        DATA 30,30,0,0,0,0,25,25,15,6,0.0: REM G$(M+1)

l       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 64,64,127,127,0,0,0,0,0,0,0,0: REM G$(M)
        DATA 0,0,30,30,2,2,0,0,0,0,0,0: REM G$(M+1)

k       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 2,6,12,24,56,108,70,66,0,0,0,0: REM G$(M)

f       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 2,2,30,62,34,34,50,16,0,0,0,0: REM G$(M)

[       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 63,63,16,48,32,32,32,32,32,0,0,0: REM G$(M)
        DATA 14,15,1,1,3,3,0,0,0,0,0,0: REM G$(M+1)

Φ       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 6,6,40,40,62,62,40,40,6,6,0,0: REM G$(M)
        DATA 16,16,10,10,30,30,10,10,16,16,0,0: REM G$(M+1)

β       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 30,30,16,16,16,18,30,12,0,0,0,0: REM G$(M)
        DATA 31,31,5,4,4,4,28,24,0,0,0,0: REM G$(M+1)

Θ       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 31,63,34,34,34,34,34,34,63,31,0,0: REM G$(M)
        DATA 12,14,2,2,2,2,2,2,14,12,0,0: REM G$(M+1)

∂       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 0,0,8,40,56,24,12,12,8,8,0,0: REM G$(M)
        DATA 28,62,34,34,34,34,38,62,60,0,0,0: REM G$(M+1)

┘       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 2,2,2,2,2,6,4,62,62,0,0,0: REM G$(M)

℘       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 12,12,10,10,10,10,12,4,0,0,0,0: REM G$(M)
        DATA 62,63,41,41,41,40,56,16,0,0,0,0: REM G$(M+1)

z       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 68,76,92,116,102,71,1,1,1,0,0,0: REM G$(M)

∫       IF A$ = "any char" THEN T$(X) = G$(M): B(X) =
```

```
        G$(M+1): RETURN
        DATA 0,0,0,6,14,24,16,16,0,0,0,0: REM G$(M)
        DATA 1,1,3,30,28,0,0,0,0,0,0,0: REM G$(M+1)

ɜ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 108,108,66,66,82,114,108,76,0,0,0,0: REM G$(M)

ɾ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 1,29,63,34,34,34,34,34,34,0,0,0: REM G$(M)

ɹ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 1,1,33,35,38,46,58,50,34,2,0,0: REM G$(M)

ç       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 56,125,69,69,71,71,68,68,0,0,0,0: REM G$(M)

ʎ       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 32,48,25,15,6,6,15,25,48,32,0,0: REM G$(M)
        DATA 0,0,12,14,2,2,28,12,0,0,0,0: REM G$(M+1)

ɥ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 32,32,126,126,40,40,40,40,14,14,0,0: REM G$(M)

ç       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 48,120,72,72,78,78,96,32,0,0,0,0: REM G$(M)

ʁ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 62,126,72,72,72,104,46,14,0,0,0,0: REM G$(M)

ʮ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 112,112,16,16,16,16,127,127,1,1,0,0: REM G$(M)

ʊ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 56,60,6,2,34,38,60,56,0,0,0,0: REM G$(M)

ʁ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 126,126,18,18,18,18,18,126,108,0,0,0:
        REM G$(M)

i       IF A$ = "any char" THEN T$(X) = G$(M): B$(X) =
        G$(M+1): RETURN
        DATA 0,0,24,24,0,0,0,0,0,0,0,0: REM G$(M)
        DATA 16,18,62,62,18,16,0,0,0,0,0,0: REM G$(M+1)
```

All the remaining vowels are produced entirely within the bottom line, as in the barred-u character which follows:

```
ʉ       IF A$ = "any char" THEN B$(X) = G$(M): RETURN
        DATA 16,16,60,62,18,18,18,62,60,16,16,0: REM G$(M)

ɑ       DATA 28,29,35,38,46,58,50,98,92,28,0,0

ɒ       DATA 28,28,34,34,62,62,42,42,46,44,0,0

ɔ       DATA 16,48,34,34,34,34,34,54,62,28,0,0

ɒ       DATA 62,62,20,54,34,34,34,34,62,28,0,0

ø       DATA 20,62,42,42,42,42,42,42,0,0,0,0

æ       DATA 36,46,42,42,42,62,62,42,42,46,44,0

ə       DATA 4,46,42,42,42,42,42,62,28,8,0,0

ɐ       DATA 32,60,30,42,42,42,42,42,58,18,0,0

ɯ       DATA 60,60,2,2,60,60,2,2,60,62,2,0

ɤ       DATA 44,62,18,18,62,44,0,0,0,0,0,0

ɑ       DATA 14,14,18,18,32,32,18,18,14,14,0,0

ɑ       DATA 28,62,34,34,34,34,54,20,62,62,0,0
```

End Listings

Let's Mouse Around

A Turbo Pascal/Microsoft Mouse Sketching Program for IBM PCs

by Vincent and
Ronald G. Parsons

A mouse makes drawing with a computer easy. Here's a program that makes using a mouse easy.

The program in the listing (page 75) is fun to use and illustrates techniques for using the Microsoft Mouse with Turbo Pascal. The program requires the IBM Color/Graphics adapter, a Microsoft Mouse, and Turbo Pascal.

By moving the mouse around, anyone can be an artist. When the program starts, an X-shaped cursor appears in the middle of a blue screen. A line is drawn on the screen as the mouse is moved with either mouse button depressed.

Below the drawing area is a set of options for changing colors, erasing portions of the screen, clearing the screen, and quitting the program. If the mouse cursor is placed between the [ ] after an option, the option is executed. Choosing "Back" changes the background color, "Palette" selects another palette of colors for drawing, and "Color" changes the color of the lines to be drawn. The "Eraser" option changes the cursor to a square and erases the portion of the drawing area in the square when a button is down. The "Clear" option clears the drawing area and "Quit" ends the drawing session.

Some of the more interesting techniques for using the Microsoft Mouse with Turbo Pascal are discussed below.

The Microsoft Mouse function 9 (Set Graphics Cursor Block) allows the program to redefine the mouse cursor shape and the position of the "hot spot" of the cursor. The hot spot is the position relative to the upper left corner of the 16 × 16 pixel cursor block where the position of the cursor is defined. The call to this function and function 12 (Set User-Defined Subroutine Input Mask) requires slightly different parameters than calls to the other mouse functions. The fourth parameter of functions 9 and 12 is an offset of an array and subroutine, respectively. The segment address of this array or subroutine must be passed to the mouse interrupt routine in register ES. We elected to make the segment address a global variable to keep the procedure parameters for the rest of the functions unchanged.

If you want to define a new cursor shape, a screen and cursor mask must be defined and placed in two contiguous arrays in storage and a pointer to these masks passed to function 9. Several different cursor shapes are defined in the Microsoft Mouse manual. The examples are given for BASIC. However, the storage order of arrays is different in Turbo Pascal than it is in BASIC. Thus, the two-dimensional array defined in the listing has the parameters in reverse order from the equivalent BASIC arrays in the mouse document. The horizontal and vertical hot spot locations are passed in m3 and m4, respectively. The hot spot can be within the range of −16 to 16 from the upper left corner of the cursor block.

When you use medium resolution

Vincent and Ronald G. Parsons, 9001 Laurel Grove Dr., Austin, TX 78758.

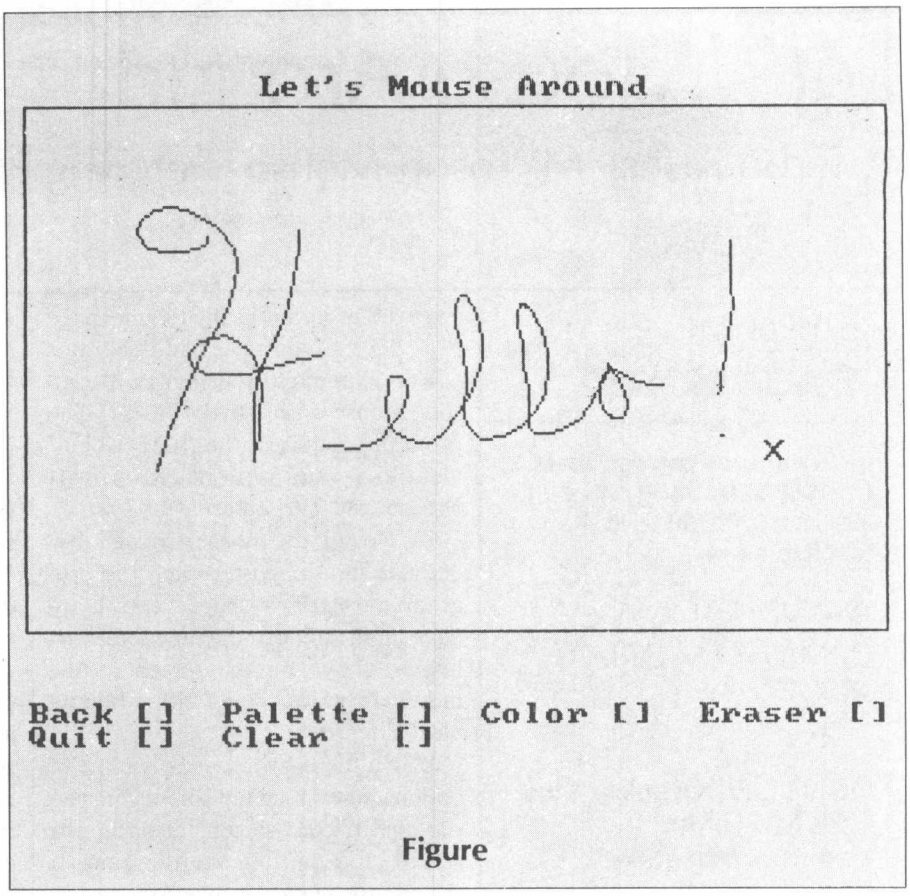
graphics mode with Turbo Pascal and the mouse, the x coordinate of the mouse must be divided by two before the Turbo Pascal Plot or Draw procedures can use it. Thus, the x coordinate is calculated as

```
x := m3 div 2;
```

The cursor must be hidden before you modify any portion of the screen containing the cursor. Mouse function 2 (Hide Cursor) performs this duty. If this is not done, the portion of the modified screen behind the cursor block will disappear when the cursor is moved. Functions 1 and 2 must be used in pairs because these functions increment and decrement a cursor flag and the cursor is displayed only when the cursor flag is zero. The initial value of the flag is –1.

Turbo Pascal permits fast development of programs, and the Microsoft Mouse makes the programs easier to use. DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 196**.



Figure

Let's Mouse Around Listing (Listing Continued, text begins on page 74)

```
{ Mouse_sketch routine for Turbo Pascal and Microsoft Mouse }
{ Uses redefined cursor and Medium resolution graphics in color }
{  by Ronald G. Parsons and Vincent L. B. Parsons }

Program Mouse_sketch;

type regpack = record
                 ax,bx,cx,dx,bp,si,di,ds,es,flags : integer;
               end;
     cursordef = array[0..1,0..15] of integer;

var  oldx,oldy,x,y : integer;
     i,esreg,m1,m2,m3,m4 : integer;
     cursor1,cursor2 : cursordef;
     background,palettenum,color : integer;
     exit,eraser : Boolean;

procedure mouse (var m1,m2,m3,m4:integer); {Microsoft Mouse interrupt routine}
var regset : regpack;
begin
  with regset do
  begin
    ax := m1;       {Set registers for mouse interrupt}
    bx := m2;
    cx := m3;
    dx := m4;
    if ((m1=9) or (m1=12)) then es:=esreg; {Set ES for functions 9 and 12}
  end;
  intr(51,regset); {Perform mouse interrupt 51}
  with regset do
  begin
    m1 := ax;       {Set parameters on return}
    m2 := bx;
    m3 := cx;
    m4 := dx;
```

(Continued on next page)

Let's Mouse Around Listing

(Listing Continued, text begins on page 74)

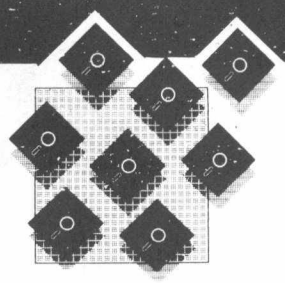
```
  end;
end;

procedure initialize;
begin

  cursor1[0, 0]:=$07e0;  {Screen mask} {Diagonal Cross}
  cursor1[0, 1]:=$0180;
  cursor1[0, 2]:=$0000;
  cursor1[0, 3]:=$c003;
  cursor1[0, 4]:=$f00f;
  cursor1[0, 5]:=$c003;
  cursor1[0, 6]:=$0000;
  cursor1[0, 7]:=$0180;
  cursor1[0, 8]:=$07e0;
  cursor1[0, 9]:=$ffff;
  cursor1[0,10]:=$ffff;
  cursor1[0,11]:=$ffff;
  cursor1[0,12]:=$ffff;
  cursor1[0,13]:=$ffff;
  cursor1[0,14]:=$ffff;
  cursor1[0,15]:=$ffff;

  cursor1[1, 0]:=$0000;  {Cursor mask}
  cursor1[1, 1]:=$700e;
  cursor1[1, 2]:=$1c38;
  cursor1[1, 3]:=$0660;
  cursor1[1, 4]:=$03c0;
  cursor1[1, 5]:=$0660;
  cursor1[1, 6]:=$1c38;
  cursor1[1, 7]:=$700e;
  cursor1[1, 8]:=$0000;
  cursor1[1, 9]:=$0000;
  cursor1[1,10]:=$0000;
  cursor1[1,11]:=$0000;
  cursor1[1,12]:=$0000;
  cursor1[1,13]:=$0000;
  cursor1[1,14]:=$0000;
  cursor1[1,15]:=$0000;


  cursor2[0, 0]:=$0000;  {Screen mask} {Eraser}
  cursor2[0, 1]:=$0000;
  cursor2[0, 2]:=$0000;
  cursor2[0, 3]:=$1ff8;
  cursor2[0, 4]:=$1ff8;
  cursor2[0, 5]:=$1ff8;
  cursor2[0, 6]:=$1ff8;
  cursor2[0, 7]:=$0000;
  cursor2[0, 8]:=$0000;
  cursor2[0, 9]:=$0000;
  cursor2[0,10]:=$ffff;
  cursor2[0,11]:=$ffff;
  cursor2[0,12]:=$ffff;
  cursor2[0,13]:=$ffff;
  cursor2[0,14]:=$ffff;
  cursor2[0,15]:=$ffff;

  cursor2[1, 0]:=$0000;  {Cursor mask}
  cursor2[1, 1]:=$7ffe;
  cursor2[1, 2]:=$4002;
  cursor2[1, 3]:=$4002;
  cursor2[1, 4]:=$4002;
  cursor2[1, 5]:=$4002;
  cursor2[1, 6]:=$4002;
  cursor2[1, 7]:=$4002;
  cursor2[1, 8]:=$7ffe;
  cursor2[1, 9]:=$0000;
  cursor2[1,10]:=$0000;
  cursor2[1,11]:=$0000;
  cursor2[1,12]:=$0000;
  cursor2[1,13]:=$0000;
```

(Continued on page 79)

```
  cursor2[1,14]:=$0000;
  cursor2[1,15]:=$0000;

  background:=Blue;
  palettenum:=2;
  color:=3;
  GraphColorMode;
  GraphBackground(background); {Medium resolution graphics - Blue background}
  Palette(palettenum);
  eraser:=false;
  exit:=false;
  m1:=0;
  mouse(m1,m2,m3,m4);        {Reset mouse}
  m1:=1;
  mouse(m1,m2,m3,m4);        {Show cursor}
  m1:=9;
  m2:=7;                     {Horizontal hot spot}
  m3:=4;                     {Vertical hot spot}
  m4:=ofs(cursor1[0,0]);
  esreg:=seg(cursor1[0,0]); {Set pointer to cursor definition}
  mouse(m1,m2,m3,m4);        {Set graphics cursor}
  oldx:=160;                 {Center of screen}
  oldy:=100;

  gotoxy(1,1);
  write('          Let''s Mouse Around   ');
  gotoxy(1,23);
  write('Back []  Palette []  Color []  Eraser []');
  gotoxy(1,24);
  write('Quit []  Clear   []');
  draw(0,10,319,10,3);
  draw(319,10,319,169,3);
  draw(319,169,0,169,3);
  draw(0,169,0,10,3);
  GraphWindow(1,0,317,168); {Limit drawing window}
end; {initialize}

procedure changeback;
begin
  background:=background+1;
  if background>15 then background:=0;
  GraphBackground(background);
  delay(300);
end;

procedure changepalette;
begin
  palettenum:=palettenum+1;
  if palettenum>3 then palettenum:=0;
  palette(palettenum);
  delay(300);
end;

procedure changecolor;
begin
  color:=color+1;
  if color>3 then color:=0;
  delay(300);
end;

procedure makeeraser;
begin
  if eraser then
  begin
    m1:=9;
    m2:=7;                     {Horizontal hot spot}
    m3:=4;                     {Vertical hot spot}
    m4:=ofs(cursor1[0,0]);
    esreg:=seg(cursor1[0,0]); {Set pointer to cursor definition}
    mouse(m1,m2,m3,m4);        {Set graphics cursor}
  end
  else
  begin
```

```
    m1:=9;
    m2:=1;                    {Horizontal hot spot}
    m3:=1;                    {Vertical hot spot}
    m4:=ofs(cursor2[0,0]);
    esreg:=seg(cursor2[0,0]); {Set pointer to cursor definition}
    mouse(m1,m2,m3,m4);       {Set graphics cursor}
  end;
  eraser:=not eraser;
  delay(300);
end;

procedure checkoption;
begin
  if m2 <> 0 then
  begin
    if ((m4 > 175) and (m4 < 183)) then
    begin
      if ((m3 >  81) and (m3 < 103)) then changeback;
      if ((m3 > 273) and (m3 < 295)) then changepalette;
      if ((m3 > 433) and (m3 < 455)) then changecolor;
      if ((m3 > 609) and (m3 < 631)) then makeeraser;
    end;
    if ((m4 > 183) and (m4 < 191)) then
    begin
      if ((m3 >  81) and (m3 < 103)) then exit:=true;
      if ((m3 > 273) and (m3 < 295)) then initialize;
    end;
  end;
end;

begin
  initialize;
  while not (keypressed or exit) do   {Exit when any key is pressed}
  begin
    m1:=2;
    mouse(m1,m2,m3,m4);     {Hide cursor}
    m1:=3;
    mouse(m1,m2,m3,m4);     {Get mouse position}
{   if m2 <> 0 then begin gotoxy(1,1);write(m3,'  ',m4,'   '); end; }
    if ((m4<11) or (m4>175)) then checkoption else
    begin
      x:=m3 div 2;        {In medium resolution, x coord. must be divided by 2}
      y:=m4;
      if m2 <> 0 then
      begin
        if eraser then
        begin
          for i:=0 to 7 do draw(x+i,y,x+i,y+8,0);
        end
        else
        begin
           draw(oldx,oldy,x,y,color);  {Draw when mouse button is pressed}
        end;
      end;
    end;
    oldx:=x;
    oldy:=y;                {Update old x/y values}
    m1:=1;
    mouse(m1,m2,m3,m4);     {Show cursor}
  end;
    TextMode(c80);
  end.
```

End Listing



Turbo Toolbox, Version 1.0

Company: Borland International, 4113 Scotts Valley Drive, Scotts Valley, CA 95066
Operating System: CP/M-80, CP/M-86, MSDOS, PCDOS
Price: $49.95
Circle Reader Service No. 143
Reviewed by Karl R. Kachigan

The Turbo Toolbox by Borland for users of Turbo Pascal can reduce development time and improve the portability of your code. It includes three tools: Turbo-Access, Turbo-Sort, and GINST. Turbo-Access is a data-filing system using B-trees that allows you rapid access to large amounts of data. Turbo-Sort is a data-sorting system that uses the Quicksort algorithm. GINST generates a general terminal installation program much like TINST for Turbo Pascal. The only restriction on the tools is that they are compatible only with Turbo Pascal, version 2.0 (or higher).

After using all the tools, I can comfortably say that any user of Turbo Pascal should be able to use these tools and find them helpful. At first, I expected the Turbo-Access system to be too complex for all but the most advanced programmers, but I now believe that even novice programmers can manage it once they go through the sample programs. GINST will be a great help to software developers who would like their programs to use Turbo's screen control features on any terminal.

The Toolbox is one disk containing 16 files, 10 of which are "toolbox" files and 6 sample programs or data files. Source code is provided for all tools and programs except the GINST-related files. GINST is the only compiled tool, probably because it modifies the Turbo Pascal runtime code. The 102-page typeset manual is much better than the Turbo Pascal, version 2.0 manuals—I found fewer typos and errors.

At $49.95, the Turbo Toolbox is less of a bargain than Turbo Pascal, but once you use one of the tools, you probably will find it worth the price. It is available on the same operating systems as Turbo Pascal: CP/M-80, CP/M-86, MSDOS, and PCDOS. In fact, I suspect that there is little or no difference between the MSDOS and PCDOS versions; they ran comfortably on both my HP-150 and IBM PC. I didn't test the CP/M versions, but because the manual makes no distinctions in the Toolbox's performance on different systems, I assume they function much the same as the MSDOS/PCDOS version.

These tools appear to be useful for both the hobbyist and the software distributor. Borland graciously has allowed developers to distribute and/or sell programs whose object code includes source code from Turbo-Access, Turbo-Sort, and the terminal installation program generated by GINST and to paraphrase the installation section from the Turbo Pascal reference manual when writing their documentation.

Turbo-Access

I'm not sure how many times I've had to create a data base to handle a lot of information, then had to add the necessary routines to find, add, delete, edit, and list the elements. Turbo-Access, a data base system that you can incorporate into your programs, is composed of four source files: ACCESS.BOX, GETKEY.BOX, ADDKEY.BOX, and DELKEY.BOX. These let you quickly create a good-sized, custom data base.

My initial fears that only an experienced programmer need delve into Turbo-Access arose when phrases like B-trees, index files, keys, pages, and tree height popped up in the tutorial section. The manual, however, explicitly states that you don't need to know how B-trees work—just how to use them. The tutorial gives a nice overview on B-trees, an approach that uses keys to find elements in its data base.

Turbo-Access is really an indexed sequential access method (ISAM) data base. For those unfamiliar with B-trees or ISAMs, these methods organize data in a manner that separately stores a unique key (like your street address) so that accessing for a specific element requires only a few searches of the key directory. This is a fast, efficient way to access an item from a large data base. In addition to your data file, you generate an index file that includes the keys organized in whatever manner you wish to treat the data. For example, if you want to create a mailing list, you could use an alphabetical list of the last names as your key. Or you may want to categorize these addresses by state or zip code—these too could be keys.

Just what is involved in using Turbo-Access? Of the four separate source files, you always must include ACCESS.BOX; it contains all the procedures and data structures to set up and maintain the data files and index files. You may include or omit the other three files as long as ACCESS.BOX comes first. GETKEY.BOX provides the searching routines for the keys in the index files and the data in the data files. ADDKEY.BOX lets you add keys and data to the index/data files, and DELKEY.BOX lets you delete keys and data from these files.

To start using Turbo-Access, first define a record that contains all the information you wish to store per element in your data base. By structuring your record to include several

items, you can use any of those items as keys in indexing your data. The key, limited to less than 256 characters, must be a string. Once you have defined this record, decide how much data you'll need to store and what you want your keys to look like. This will help you determine the constants that Turbo-Access will need to set up its data base. The six constants are data record size, key length, page size, order, page stack size, and max tree height. You've already figured out the first two. You simply guess the page size and page stack, given typical operating ranges for these. Order is half the page size, and you can compute max tree height easily, given the above numbers. I was surprised to see how little effort was needed here after you assume the suggested ballpark figures.

Finally, determine what you want to do with your data. You probably will want a complete program to find, add, delete, edit, and list the data; the sample programs for each of these tasks should get you going. The manual suggests that you include in your data record an integer variable that you can use for status. When your program stores a data element, you can set this status value to distinguish the element later from a deleted or unused element. This is helpful for recreating an index file that is corrupt—the manual even gives an example of how to do this. Corruption of index files is the biggest problem with ISAMs, so following this suggestion is a good idea.

The sample program BTREE.PAS is a fairly extensive mailing list program. It includes the capabilities of finding, adding, editing, deleting, and listing the data, with a nice menu-driven entry mode and function selection. BTREE.PAS nicely demonstrates the power of Turbo-Access. It is more extensive in scope than the sample programs that demonstrate the separate tasks of creating an index/data file and finding, adding, deleting, and listing the data. After I used these simple examples to put together my own little data base, the BTREE.PAS example showed me some nice enhancements.

What are the limitations of Turbo-Access? First, if you need to include all of your source files, it takes a while to compile. Second, be prepared while debugging your program to know what compiler directives the B-tree modules have set. Third, although a boolean variable "ok" is set after many operations to determine if a task was successful, if the task fails, you have little indication as to why. Any error that Turbo-Access can't handle becomes a fatal error and bombs you out of your program; you are left with only an indication of the error code, record number, and filename, generated by the TAiocheck routine. I suspect you could modify this routine to create a global error number that could be tested. However, it would be nice if the error code tests available in Turbo-Sort were available here so that the programmer could handle some of the disk-related errors (e.g., disk full or write-protected).

Fourth, the MakeFile routines used to create index and data files destroy any existing files that are using the same filenames. If you want to test automatically whether index and data files already exist, you must try to open them first. Finally, should you wish to use duplicate keys (and you can instruct the Turbo-Access routines to do this), you must do additional searches to find the data you may want. Hence, unique keys will simplify your data base searches.

Turbo-Sort

Many times I've needed to sort a list of items, whether numbers or strings, in descending or ascending order. Turbo-Sort provides a simple, efficient method to do so. It uses the Quicksort algorithm, a faster approach than the time-honored bubble sort. It allows you to sort data from disk and/or memory and to send the results to disk and/or memory. Why is it so flexible? You get to write the input and output processing procedures, plus the procedure that determines the sort priority.

Turbo-Sort is composed of three parts: input, sort, and output. Borland provides the code to sort, while you provide (1) the input procedure Inp that puts the data into the sort processing memory, (2) the output procedure Outp that takes the sorted data and puts it somewhere else, and (3) the less function Less that determines if one element is less than another element. This sounds like a lot, but by copying the sample programs, you'll quickly be up and running.

The Inp procedure takes the data you wish to sort and puts it into sort memory. Turbo-Sort uses all of the existing heap space to store your data. If there is insufficient heap space (the program can sort up to 32,767 elements), Turbo-Sort does a virtual memory operation and partitions some of the data to disk. If your program uses the heap, it must allocate enough heap space for itself before you enter Turbo-Sort. The minimum amount of memory needed to sort is three times the item size (in bytes) or three times 128 bytes, whichever is greater—this isn't much space. The Inp and Outp procedures, in essence, take or get your data from the sort heap, which is managed by Turbo Sort.

The nice thing about being in control of the Less procedure is that you can sort on multiple keys, sort any type of item (numeric or alphabetic), and sort in ascending or descending order. The sample programs show descending sorts on numbers (i.e., quantity of an item, stock number, etc.). You can get ascending sorts by changing the < sign to > and recompiling. Likewise, if you wish to sort a data list in several different ways—say, by quantity then by name—you can follow the multiple key sample program and structure the Less procedure with a case statement. One thing to beware of: Turbo-Sort does require that each element of your data be at least two bytes long.

To use Turbo-Sort, you simply include the SORT.BOX file when compiling your program. Turbo-Sort forward declares the Inp and Outp procedures and the Less function. Should any errors occur, Turbo-Sort returns an error number indicating what is wrong. Of the six error codes, two give you disk-related errors indicating a read error or a file create error. This is much like the IoResult function of Pascal that lets you trap system errors. As mentioned before, I wish the Turbo-Access system had

provided this.

GINST

This tool generates a terminal installation program; it isn't the installation program itself. It needs to know the name of your program so that it can patch the proper disk file. GINST produces three files: a code file, a data file, and a message file. Using this tool is simple. GINST asks you for your program name, then the name you want for the terminal installation program. That's it. To customize your program for a new terminal, just run your terminal installation program.

When running your terminal installation program, you'll notice that it looks similar to Turbo Pascal's TINST program. You select the terminal to use from a list, then have your code updated to reflect the cursor control sequences and screen editing commands of that specific terminal. You can also modify the cursor control and screen editing sequences, add new terminal definitions to the list, and delete selections from the list.

Although I heartily approve of this new tool, I do have a few complaints. First, Borland has provided an INSTALL.DOC text file that is similar to the step-by-step installation of the Turbo Pascal manual for use in your own manual or for distribution with your software. Unfortunately, the text is riddled with typos—as if someone hastily transcribed the instructions from the manual to disk but did no spelling checks. Second, GINST will not let you abort during the prompt asking for the new name of your installation program. It would be nice if a control-C were acknowledged here.

Third, on the MSDOS/PCDOS version, I couldn't make a program and its new installation program work on both my MSDOS machine and my IBM PC. If I compiled a program on my MSDOS machine then generated an installation program, it would work fine on that machine. But when I ported both object code files and the data files to my IBM PC and tried to run the installation program, oops! no IBM PC installation menu—just the MSDOS terminal selections. However, if I recompiled only my main program on the IBM PC then ran the installation program, voila! the IBM PC monitor selections appear. Obviously, GINST inspects some part of the Turbo Pascal runtime module to determine what compiler (MSDOS or PCDOS) was used then uses the appropriate terminal selections. It sure would be nice to have one object file portable to both the IBM's screen and a normal terminal on an MSDOS machine. I suspect, with the appropriate ANSI.SYS file under PCDOS 2.0 (or higher), I could add the IBM to my MSDOS terminal list, but that's not clean. That solution also doesn't do much for me if I'm still running PCDOS 1.1. Oh, well!

Finally, it would be nice to customize the GINST terminal data file for the terminals I want before I run it and generate my custom installation program. Unfortunately, if your favorite terminal (like my HP-150) doesn't appear on GINST's data file, you must add it to every installation program you generate.

Summary

As I mentioned at the start, many of us can use these tools. In general, I found Turbo Toolbox fairly bug free and quite usable, except for the problems noted. The folks at Borland have produced another decent product.

SPSS/PC

Company: SPSS, Inc., 444 N. Michigan Ave., Chicago, IL 60611
Computer: IBM PC/XT, AT,, and compatibles
Price: $795.00
Circle Reader Service No. xx
Reviewed by Dr. Steven Anthony Sola

Even before buying my microcomputer, I dreamed of having a miniature megalith of an IBM 370 on my desk, at my command, and totally within my grasp. I wanted it all: every interesting language, every systems tool, and especially the big number-crunching programs like SPSS or BMDP. The operating systems, the languages, and the applications are here, and now SPSS/PC has arrived. But I have a new concern: How good is this micro product?

For many years now, Statistical Programs for the Social Sciences (SPSS) has been the standard package that most graduate students have used with mainframe resources to analyze their thesis or dissertation data. Although it has competition from BMDP and SAS, SPSS is more than holding its own. For example, in the corporate environment as represented by the annual Datamation survey of mainframe applications packages (March 1984), the SPSS batch system scored better than its group average in all categories. Perhaps the best test of quality is a measure of the program's reliability and validity. Can it be trusted to process the data of a thesis or a dissertation?

Through the kind cooperation of an associate, Dr. Jilisa Snyder, we were able to evaluate the SPSS/PC micro product directly against the SPSS mainframe version running at the State University of New York at Albany. First, we ran Jilisa's dissertation using an IBM PC as a terminal under the control of PC TALK III, uploading the data, interactively running the SPSS mainframe programs, and finally downloading the results. We then took samples of the most arduous statistical procedures found in the dissertation and attempted to run them using the SPSS/PC micro product. The control language required only minor adjustment, and all programs ran without mishap. Finally, we compared the output of each package using side-by-side windows

The results were spectacular and reassuring: out of over 156K of output examined, results were identical in every important detail to the mainframe version. Not only were the results accurate to their scientific precision, but they were exact even in their final imprecise digits to the mainframe version! The only exceptions occurred in the numeric representation of data plots, where the micro version rounded more accurately

Copyright © 1984 by Dr. Steven Anthony Sola. All rights reserved.

than the mainframe version. Basing our opinion on this excellent level of performance, we are reasonably assured that the SPSS/PC system is at least as valid and reliable as the SPSS mainframe version.

Of course, this conclusion must be tempered by a caveat. We assessed only a small proportion of the many possible procedures in the SPSS package, principally the multiple regression and stepwise multiple regression procedures. On the other hand, the data used was all too real! It contained numerous missing values, not only in the independent variables but also in the dependent variables. The subsamples were unequal in size and disproportionate. In short, the data was the messy and ill-behaved sort so typical of the social and behavioral sciences.

Is it possible to run all of the mainframe package on the micro? Unfortunately, no. SPSS/PC has been tailored or, more accurately, truncated. The most serious omission is any form of multivariate analysis of variance; the MANOVA procedure is missing. True, the complete univariate analysis of variance procedure ANOVA is present, but this procedure is far from sophisticated. Other procedures that compare groups are acceptable as far as they go, but MEANS, ONEWAY, and T-TEST simply can't compare with MANOVA in power and flexibility. Also missing is any form of canonical correlation, discriminant analysis, logistic regression, or time series analysis.

On the other hand, simple descriptive data analyses providing means, standard deviations, and the like, including plots, are well provided. Particularly noteworthy is the excellent implementation of cross-tabulation and hierarchical log-linear models for multi-way contingency tables. Although the provision for pearson correlation and nonparametric statistical techniques is adequate, there is no provision for rank-order correlation nor for partial correlation analysis. A hierarchical cluster analysis is nicely done, but the cluster memberships cannot be saved. This failure to save also plagues the factor scores in an otherwise exemplary procedure of FACTOR. Similarly, casewise residual scores cannot be written out in the multiple regression procedure.

Such truncation is unwarranted in this otherwise excellent product. The total number of variables is limited to 200. The total number of cases ostensibly is unlimited but, in fact, is limited by an overall maximum of 64K workspace for data. When I enquired as to why this might be so, the folks at SPSS muttered some mumbo-jumbo about the limitations of the current crop of Fortran compilers for the 8088. Come come, fellows! It should be possible to overcome this 64K curse. As an additional footnote, SPSS/PC uses the Microsoft Fortran compiler. While SPSS/PC makes use of a lot of overlays, it requires at least 320K and uses a maximum of 384K. Surely using another 256K of available storage on the IBM PC wouldn't hurt. It appears that SPSS is not taking full advantage of the IBM PC's resources, and that further optimization may lead to a better product.

Although it is possible to run SPSS/PC without an 8087 numeric coprocessor, it is silly to do so. After shelling out $795 for SPSS/PC, not spending the additional $200 or so for the coprocessor, which greatly speeds things along, just isn't sensible. A hard disk is a requirement, not only for the procedures themselves and associated files, which take up approximately 3200K, but also for the temporary files used during a program run.

Transferring data out of SPSS/PC in the form of reports is helped along by a report writer; however, the line length is limited to 132 columns, and there is no printer configuration file to customize it or any of the save commands to your printer. Transferring files between a mainframe and SPSS/PC is greatly facilitated by KERMIT, a neat little data link included with the package. KERMIT can transfer some types of system files with error checking, as well as data files. Transferring files between your favorite spreadsheet, data base manager, or word processor and SPSS/PC is smooth as long as plain vanilla ASCII is your favorite flavor. If you have more exotic tastes, you will need to know how to cook in a programming language. Unfortunately, SPSS/PC provides no recipes for interfacing with their SPSS systems files.

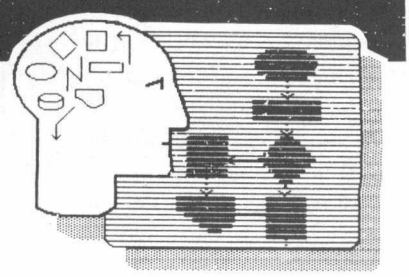
SPSS/PC does provide the most elaborate, comprehensive, well thought out, and usable documentation available anywhere, mainframe or micro! The SPSS/PC documentation is specific to the micro user, and the built-in statistics guide is easy to follow and genuinely useful. Illustrations, examples, and reference material on each command and procedure abound. The typeset documentation, which weighs in at several pounds, is well organized, indexed, and reads easily. SPSS/PC comes with a nice tutorial on disk, along with a demo program. As if this were not enough, SPSS/PC also provides instantly available on-screen help when running interactively.

Yes, SPSS/PC can run interactively, with each command and procedure executed when entered. Traditional batch processing is also available. Unfortunately, SPSS/PC is severely hampered by the lack of an on-line editor. If you make a mistake, for example, in the middle of a long command line with many parameters, well, you had better enjoy typing! It is possible to get a directory listing and to examine or erase a DOS file, but a general exit and return to the operating system doesn't exist. The impact of this becomes more apparent when you realize that you must completely restart the whole package each time you use your favorite word processor to edit some commands.

Which brings us to the topic of the built-in copy protection and its insidious necessities. Every time you restart the program, you must have the "key diskette" in drive A. This is bad enough, but the SPSS ghouls are not through with you! At unpredictable times, in the middle of a procedure, the thing looks for the key in drive A. Woe to the foolish users who do not replace the key within a few tries into drive A, for their systems will come crashing down! Whatever you used to have in drive A may have vanished into bit heaven. For the typical type

(Continued on page 93)

THE SOFTWARE DESIGNER



The Model in the Machine and the Model in the Mind

by Michael Swaine

How do you design software for people rather than for the machine or for the programmer? How do you make the model inside the machine reflect the model in the user's head? I cornered two programmers who think particularly hard about the intended users of software, and they talked about human factors in the processing of words and ideas.

Dave Winer is the president of Living Videotext; he wrote ThinkTank, the outline processor. While he devotes a lot of his time these days to meetings, budgets and strategic planning, he says, "what I am, first and foremost, is somebody who designs software." But he's a software designer who uses his own products and listens to others who use them.

Thom Hogan identifies himself here as a writer. As Editor-in-Chief of our sister magazine, *Business Software*, he's that, but he's also a programmer. It's his experience as a user of word-processing programs, though, that gave him his idea for exponential cursor movement.

Dave Winer: The first rule I try never to break is that if you don't use it yourself, it isn't going to be an interesting product. A lot of software designers don't use what they write. I guess we're fortunate in that our product, ThinkTank, is useful in a lot of different functions, so it's easy for us to say that everybody in the company uses the product. But you've got to use the product: we've modified ThinkTank many times over several years, and those modifications have been the result of heavy use of the product. Talk to any game designer about how he makes a game entertaining and he'll tell you the same thing: he plays the game himself, and he doesn't stop until it's entertaining for him.

But it's hard to put yourself in the user's frame of mind; you've got to continually cull out the things that make you too specialized. You have to work at seeing things the way a naive user sees them, and you have to know how to feel the little jolt when a program isn't right.

There's a term we use: burning brain cells. Every time the program does something that you're not expecting it to do, every time it surprises you just a little, your suspension of disbelief goes away. You're no longer thinking of the problem that you're working on, but about why the program misbehaved that way. It's that phenomenon of the software taking you away from the task at hand that we call burning brain cells.

The concept of suspension of disbelief is very important. In software just as in the movies or in fiction you have an audience, and you're trying to manipulate them, to influence their behavior; initially with a program you're trying to relax the person and invite him to try things. The suspension of disbelief develops as you draw the person in gradually: first he tries the cursor keys, say, and he finds that the cursor keys move the cursor; OK. Then what? Well, people make guesses; and you as a software designer have to try to anticipate their guesses. To a certain extent you do this on your own, but eventually you've got to try it out on real people. We found with ThinkTank that after the cursor keys, people were reaching for the spacebar, so we used the spacebar to step through the command line. So you second-guess the user, and if you do a good job, you build up a certain amount of trust on the part of the user; then over time, if it is going to be a successful experience for him, he will have suspended his disbelief. He's now no longer thinking about the software, he's thinking about the mechanics of the problem at hand. That's the goal.

One technique for getting there is not trying to force your structure on the user, but rather trying to get him to recognize his own structure in the machine. One of the reasons people like using computers is that it's neat to crawl around in there and discover things. Well it's neater if you're crawling around discovering things that you made rather than things that some guy in Cambridge made. We tried to capture a human structure in writing ThinkTank: categorizing and subcategorizing are very natural to the human mind, even though the way ThinkTank does categorizing is very different from the way humans do it. We've listened to users and in the next release incorporated features like cloning and hoisting and the use of color, all things that make the model closer to the model the user has.

Thom Hogan: I'm a writer. I spend a lot of time using word processors, and I've yet to find a word-processing program that doesn't somehow inhibit my writing and editing style. There are so many needed features of word processors that haven't been implemented yet that I sometimes wonder what to call the software I do use. This brief note describes just one concept that has yet to be implemented in any word processor.

Cursor control is out of control. I just got the latest whiz-bang word processor and I find that I can now move by character, by word, by phrase, by sentence, by paragraph, by block, by screen page, by actual page and by section. That's a *lot* of cursor keys.

Moving around in a document, especially a long one like a book or software manual, should not be a many-

fingered thing. I don't want to do nine kinds of multiple-key gymnastics, I don't want to move my hand from the home row of keys to a cursor pad and I certainly don't want to both move my hand from the home row *and* hold down multiple keys.

The cursor-control diamond in WordStar is reasonably comfortable for us touch typists; I can Control-F and Control-A to move forward and backward a word at a time almost as fast as I can type, but then I've had several books worth of practice. But WordStar's cursor controls are inadequate for moving around quickly in a large document. Here's a piece of user data: I find that I typically move screen page by screen page, then narrow in on my prey using word-by-word moves, with maybe a few up and down arrows. With a word processor that makes good use of a mouse, I can effectively narrow in on a particular spot on a screen, but it's less useful in larger moves, and there's still the issue of removing my hands from the home row.

All of which leads up to my notion of exponential cursor control. I only want six cursor control keys: four to move one character at a time, up, down, right and left; and two additional keys, a "power forward" and a "power backward" key.

My definition of a "power" key is one that moves the cursor in a fixed direction through the file, but that increases its step size with repeated presses (or if you hold the key down long enough for autorepeat to start). There are several algorithms possible for a power key:

Algorithm 1: time-dependent

```
WHILE power key held down DO
   WHILE time down < 1 unit DO
      move cursor one word
   WEND
   WHILE 1 unit < time down <
   2 units DO
      move cursor one sentence
   WEND
   WHILE 2 units < time down <
   3 units DO
      move cursor one paragraph
   WEND
   WHILE 3 units < time down <
   4 units DO
      move cursor one page
   WEND
WEND
```

Algorithm 2: context-dependent

```
flag=1
WHILE previous keypress was
power key DO
   CASE OF flag =
      1: move one word
         test for boundary
      2: move one phrase
         test for boundary
      etc.
   ENDCASE
WEND
```

The test for boundary sets the value of flag dependent on whether or not we've moved across some arbitrarily defined boundaries during the movement of the cursor:

phrase boundary: , .) ? ! ; :
sentence boundary: . ? !
paragraph boundary: <cr><lf>
page boundary: 66 lines
section boundary: section marker

With either approach, the effect is the same: the longer the power key is repeated, the greater distance it moves and the greater is its speed of movement, although the first approach requires that the user *hold* the power key down. Each approach has its advantages. The first allows you to lift your finger from the key momentarily to get your bearings and then restart at a slower cursor rate when you press the power key again. The second favors key tappers who use the rhythm of the keystrokes to reinforce what they're doing. Both approaches require programs that give priority to redisplaying the cursor and screen contents, since power keys would make the program especially sensitive to the "understeer" and "oversteer" problems that result from the display not keeping up with the cursor movement.

I've described moves in terms of units that most matter to me these days: words, sentences, paragraphs. Program editors, though, could be configured to use boundaries that make sense for the language being used. Boundaries could be set at keywords, statement boundaries, loop structures and functions, for example, if one were editing C programs. Better yet, give me an editor that lets me provide definitions for boundaries.

One last request to word-processing programmers: please keep my fingers on the keys. Use the WordStar diamond and let Control-A and Control-F be the power keys; I'll love it. I won't even object if you throw in screen-up and screen-down keys.

Micro Subscribers Please Note:

The Software Designer is a column devoted to the notion that the activity of programming engages one in a struggle with metaphors; that the only worthy goal in the struggle is to be a wielder of metaphors, a creator of metaphors, and not an unwitting tool of unexamined metaphors. That's what, I claim, distinguishes a software designer from a mere programmer: the software designer creates new metaphors for the machine.

A computer is, after all, a general purpose device, and that's a pretty ethereal concept. What does a general purpose device do? Well, it . . .

You see the point. Just to tell our friends what we are doing with what used to be our spare time, or what we do for a living, we have to create metaphors. All the metaphors are in fact just descriptions of software products: the Mac's metaphor is the desktop and VisiCalc gave us the spreadsheet metaphor. And when the metaphor is implemented, the computer, the general purpose device, acquires a specific purpose, becomes, briefly, an electronic desktop or spreadsheet. It's trivial to point out that the computer can do nothing until some programmer tells it what to do. But the computer *is* nothing until the software designer brings into the world a new metaphor, and with it a new intellectual tool.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 197.**

by Allen Holub

The topic of this month's column is sorting routines. We'll look at how the sort routine given in Kernighan & Ritchie works and talk about why you shouldn't use it. Then we'll present a general purpose Quicksort routine, modeled after the Unix routine qsort( ).

Shell Sort

Many C programmers, when they need a sort routine, grab the one out of *The C Programming Language* (p. 116). The routine works, it's short, why not use it? There are several reasons: it's not documented, it's slow, and better routines are available. Some of the slowness comes from using array indexes rather than pointers (an array index usually causes a multiply, while incrementing a pointer is an addition).

Unfortunately, the speed issue runs deeper. Just about every programming problem presents you with trade-offs, and sorting routines are worse than most in this respect. Not only are the usual considerations of code size versus efficiency a factor, but there are several possible ways to go about sorting an array, all of which behave differently. For example, with Quicksort, the time required to sort an array can change radically with the kind of data the routine is presented. You have to know how a sorting routine works before you can make an intelligent decision on whether to use it in a particular application.

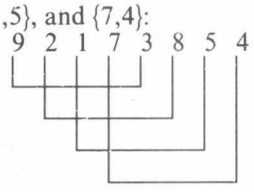
The routine in K&R is a Shell sort. I thought for years that Shell sort was named after a shell game; anyone who has sat down with the K&R code and tried to figure out what's going on can understand how I came to this conclusion. As it turns out, the algorithm is named after its inventor, Donald Shell, who developed it in 1959. A Shell sort works by breaking up the sorting problem into a series of smaller problems. For example, a set of eight objects to be sorted is broken up into four subsets of two objects, each of which is sorted separately. The eight objects are then reorganized as two sets of four objects; again each is sorted separately. Finally, the set is sorted as one set of eight objects.

The rationale behind the Shell sort is that you get the more out-of-order parts of the original set into order very quickly. The array then is easier to sort on the next pass. What characterizes a Shell sort is the reorganization into smaller sets. Although the algorithm used to actually sort the items in each subset is immaterial, the behavior of this algorithm should get better as the set it is working on gets closer to being in order. A bubble sort does this. In fact, you can look at Shell sort as an improved bubble sort.

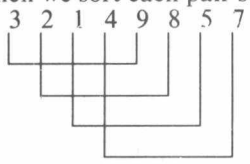
A concrete example may clarify the process. Let's start with a set of eight objects:

{9, 2, 1, 7, 3, 8, 5, 4}

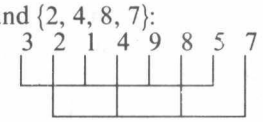
We break this up into four subsets consisting of the pairs {9,3}, {2,8}, {1,5}, and {7,4}:



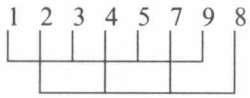
Then we sort each pair separately:



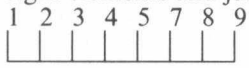
Next we break up the set into two subsets of four elements, {3, 1, 9, 5} and {2, 4, 8, 7}:



We then sort these separately:



Finally, we treat the input as one set of eight elements and just sort it:



Now, let's look at the routine in K&R (a slightly modified version appears in Listing One, page 94). The outermost for loop (line 6) determines how many elements will be in each subset; "gap" is the distance in the array between two elements of the same subset. In the above example, the distance between 9 and 3 (which compose the first subset) is 4; this would be the initial value of gap. Each time through this first for loop, we divide gap by two, which effectively doubles the number of elements in a subset.

The second and third for loops (line 7 and 8) are the actual sort routine. The algorithm is essentially a bubble sort, although the situtation is complicated by sorting all of the subsets at the same time. You can best see how the sort works by reducing the algorithm to handle only the last case (i.e., gap = = 1) and reversing the sense of the compare. This is done in Figure 1 (page 91).

In the innermost loop, the bubble sort scans along the unsorted array looking for two adjacent elements that are out of order. When it finds these elements, it exchanges them. In this way, an out-of-order element tends to "bubble" up to its correct place in the array. However, you have to do many comparisons and exchanges to get it there (worst case is an array already sorted but in reverse order; you have to make N–1 exchanges to get an out-of-place element from the right end of an N element array all the way to the left end). This inner loop must be executed enough times to guarantee that

all out-of-order elements find their way to the proper position in the array; therefore, a bubble sort will *always* use $O(n^2)$ comparisons. In the worst case, it will need almost as many exchanges. No wonder the routine is slow.

You can now see how the Shell sort strategy helps the bubble sort. It takes that pathologically out-of-place element and moves it to the front of the array in O(log N) exchanges, rather than in N–1 exchanges. You can improve the behavior of a Shell sort by fiddling with the increments between elements. This lets the various passes interact with each other in a more productive fashion. The increments should not be even multiples of each other; powers of two are actually among the worst increments. Knuth has determined that a good choice of increments is 1, 4, 13, 40, 121 .... He also says that the sequence 1, 3, 7, 15, 31, ... works well. Using this latter sequence of increments, a Shell sort will sort an N element array in $O(N^{1.2})$ time. This is better than the $O(N^2)$ time required for a bubble sort but not much. If you're interested, the gory details of this analysis are in Knuth's book *The Art of Computer Programming*, vol. 3 (Addison Wesley, 1973) pages 84f.

Quicksort

In the general case, a faster sorting routine than the Shell sort is Quicksort, developed by C. A. R. Hoare in 1962.

Quicksort, like the Shell sort, sorts an array in place; that is, it sorts by exchanging elements of an array rather than by copying. It works by selecting an arbitrary element of the array to be sorted (called the "pivot") and moving that element to its proper position in the sorted array. As it is doing this, it arranges the array so that all elements to the left of the pivot are less than the pivot and all the elements to the right are greater than the pivot. It then repeats this process recursively on the two halves of the array. It works as follows:

(1) Select an arbitrary element of the array to be sorted (the pivot). It's easiest just to lop off the last or first element of the array and use that as the pivot (we will use the last element in this example). However, if the pivot happens to be the largest or smallest element of the array, then the behavior of Quicksort degrades dramatically (Quicksort becomes Slowsort). You can avoid this worst case scenario by taking the center element as the pivot or, even better, by taking the median of three elements. This extra work will slow down the routine, but it's worth it if you expect to be sorting already sorted files.

(2) We will sort using two pointers called HIGH and LOW. To start sorting, make HIGH point at the next to

```
for( i = 1 ; i < argc ; i + + )
        for( j = i – 1; j > = 0 ; – –j )
                if( strcmp( argv[j], argv[j + 1] > 0)
                        exch( agv[j], argv[j + 1] )
```

Figure 1
Bubble Sort

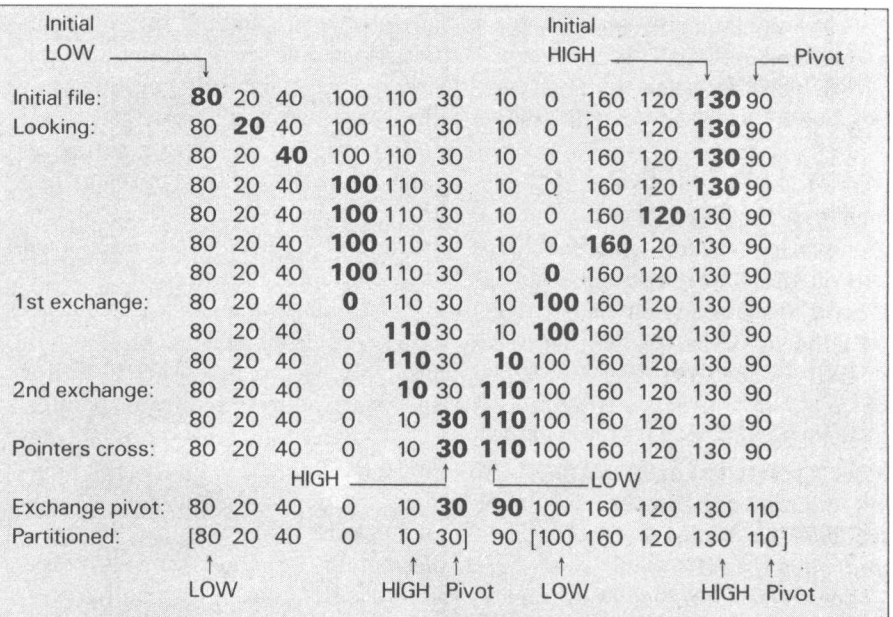


Figure 2
Quicksort

```
Given:
        typedef struct
        {
                int     key, element1, element2, etc;
        }
        ELEMENT;

        compare( e1, e2 )
        ELEMENT *e1, *e2;
        {
                return( e1–>key – e2–>key );
        }
        ELEMENT array[10];

The array can be sorted with:
        qsort( array, 10, sizeof(ELEMENT), compare );
```

Figure 3
Sorting an Array of Structures

last element of the array and LOW point to the first element (the last element is the pivot).
(3) Move LOW towards the last element of the array. Stop when HIGH and LOW cross or when the object pointed to by LOW is *greater than* the pivot.
(4) Move HIGH toward the first element of the array. Stop when HIGH and LOW cross or when the object pointed to by HIGH is *less than* the pivot.
(5) If the two pointers have not crossed, exchange the objects pointed to by HIGH and LOW.
(6) Repeat steps 3, 4, and 5 until the pointers cross.
(7) Exchange the pivot and the object pointed to by LOW. This puts the pivot into its correct place in the array.
(8) The array is now partitioned so that all elements to the left of the pivot are less than the pivot and all elements to the right of the pivot are greater than the pivot. Repeat steps 1–8 on these two partitions (not including the pivot in either partition) until the array size of all arrays is less than or equal to one.

An example of the sorting process is given in Figure 2 (page 91). The objects pointed at by HIGH and LOW are underlined. For more information see *The Art of Computer Programming*, pages 114f.

The biggest problem with Quicksort is its sensitivity to the input data. Best case performance (when you always select the median as the pivot) requires O(N log N) comparisons and O(1/6 log N) exchanges. This is quite a bit better than $O(N^2)$. The average case performance is still pretty good: O(2 1n 2) time slower than the best case. Worst case performance (when the pivot is the largest element in the array) is another matter. In this case, one partition is empty and the other contains the balance of the array (N–1). N (rather than log N) partitioning steps must be made, and the algorithm requires $O(N^2)$ time to sort the array. This time is no worse than that of the bubble sort, but it's no better either.

A general purpose Quicksort routine is presented in Listing Two (page 94). This routine, called qsort( ), works like the Unix routine of the same name. By "general purpose" I mean that you can use qsort( ) to sort arrays of *anything*. You can sort arrays of pointers, of integers, of structures—of anything. The advantages of this versatility are obvious. The disadvantage is that qsort( ) will run slower than a routine tailored to a particular application.

There are two other potential problems with this particular implementation of Quicksort: it chooses its pivot in an unintelligent way and it's recursive. If you are likely to be sorting already sorted lists, you should modify the pivot selection to take a median value rather than just grabbing the last element. Handling the recursion is trickier. I've tried to minimize the average case recursion nesting, but the worst case is still pretty bad.

In particular, if you are sorting an already sorted array of N elements, there will be N–2 levels of recursion. Each level is passed two pointer-sized local variables on the stack. Assuming a 16-bit pointer, this yields eight bytes per level. The compiler will probably need a few other bytes for the return address and frame pointer, say, 12 bytes total (this is probably an underestimate). Sorting a 1024-element array would require about 12K of stack, not even counting the space needed for the array itself. The moral is to use this routine on small arrays. It's fast but it eats memory. Best case recursion nesting will require $\log_2 N$ levels, which is better but still not good. If someone has an iterative version of Quicksort, send it to me.

The versatility of qsort( ) comes from its not making any assumptions about the object being sorted. Its calling syntax is:

```
qsort( base, nel, width, compare )
char *base;
int nel, width;
int (*compare) ( )
```

"Base" is the base address of the array being sorted; it's declared as a character pointer but actually can be a pointer to anything. "Nel" is the size of the array in elements (as compared to bytes). "Width" is the size of one element in bytes, and "compare" is a pointer to a comparison routine. This comparison should behave like strcmp( ); that is, when called with

```
(*compare) (a, b);
```

where a and b are *pointers* to two elements of the array, the comparison routine will return a negative number if a < b, zero if a = = b, and a positive number if a > b.

The main( ) routine in Listing Two (line 167) shows qsort being used to sort an array of character pointers. Figure 3 (page 91) shows it being used to sort an array of structures where the key field contains an integer sort key.

Qsort( ) itself (Listing Two, line 40) is an access routine. The actual sorting is done by rqsort( ) (line 74), which is static to the module (i.e., it isn't externally accessible). Qsort( ) first copies the two parameters that won't be modified (width and compare) into two global variables (line 63; comp and width are declared on line 20). Passing them as parameters to rqsort( ) would just increase the stack usage and make the subroutine calling overhead higher. Both variables are static, so you don't have to worry about their names conflicting with other global variables in your program. I have deviated a little from the Unix qsort( ) by defining a default comparison routine for sorting argv-like arrays of character pointers. To use the default routine, set the compare argument to 0. The actual sorting is started by calling rqsort( ) (on line 66).

Rqsort( ) is passed two parameters, "low" and "high" which point at the bottom and top elements of the array to be sorted. They have the same function as LOW and HIGH in the algorithm description. The local variable "pivot" is a pointer to the pivot (selected on line 91). Note that all the pointers are defined as character pointers. Because characters are of size 1, pointer arithmetic on a character pointer is plain arithmetic. We need to defeat pointer arithmetic because we don't know the size of the object at compile time. Adding or subtracting width from the pointer

will advance it the correct number of bytes. The comparison routine has to know what's being pointed to, but at this level we need only the size.

The two while loops on lines 95 and 97 are looking for out-of-place elements, per steps 3 and 4 of the algorithm. Step 5, exchanging LOW and HIGH, is done on lines 99-110. Again, because we don't know the object size in advance, we have to exchange the two objects one byte at a time. We look for the pointers crossing at line 113. Step 7, the pivot exchange, is done on lines 118-124. Because an exchange is an expensive operation, "Low" and the pivot are exchanged only if necessary. On the other hand, the comparison itself takes time too, so you may want to pull it out of the code.

The partitioning is done on line 131. "Low" is advanced to exclude the pivot from both partitions (it's already in the correct place in the sorted array). Finally, step 8 (sorting the two partitions) is done on lines 131-145. Several tests are made here. The test on line 132 determines which partition is smaller; the smaller partition is sorted first. This practice tends to reduce the number of recursion levels. The tests on lines 134, 136, 141, and 143 are looking for arrays with one or fewer elements in them; these arrays are already sorted. Most Quicksort algorithms put this test at the top of the algorithm, but this adds an additional level of recursion, and testing at the bottom doesn't add all that much extra code.

Conclusion

So, there are two sorting routines. Which you should use depends on the application. If you expect to sort a lot of already sorted data, the Shell sort will work in $O(N^{1.2})$ time while Quicksort needs $O(N^2)$. A Shell sort is also not the memory hog that Quicksort is. On the other hand, Quicksort is much faster on randomly sorted input, even though it eats stack space.

The qsort( ) routine given here is more versatile than the shell sort( ) routine, but it shouldn't be too hard to modify the latter to be as general purpose as qsort( ). Also, it's not hard to strip the versatility out of qsort( ) if you don't need it, improving its performance for a specific application.

I'd like to end this month with a request. If you have working C implementations of other sorting algorithms, send them to me. Perhaps we can do a follow-up of this column at some future date.

The O(N) Notation

The big O notation is often found when analyzing algorithm efficiency. It is read as "order of . . . ." O(N) is "order of N," $O(N^2)$ is "order of N squared, " and so on. "O(X) time" means that the algorithm will perform in time proportional to X. So, an algorithm that sorts an array in $O(N^2)$ time will sort the input data in time proportional to the number of input elements squared. To derive O(X), you look at the algorithm and extract the crucial operations, the ones that are performed most often. In the case of sorting algorithms, this will be some combination of comparison and exchange operations. You then figure out how many times that operation will be performed on a given set of input data. Because the main use of the big O notation is to compare algorithms, it's not particularly exact. You're just trying to express how an algorithm performs in a general sort of way. So, if an algorithm really takes $17 + N^2$ operations to do something, the 17 is ignored because it isn't particularly significant. In another example, Quicksort requires N*log N comparisons and 1/6(log N) exchanges in the general case. The 1/6 is relatively insignificant, so we usually say that Quicksort will work in O(N*log N) time in the general case.

DDJ

(Continued from page 83)

of system hang induced by these fiends involves keeping drive A running an infinite length of time. Can you use one of the bit copiers at least to provide yourself with some insurance? No, they don't appear to work. Will SPSS provide you with a backup disk? No way. Be prepared to wait about a week and to shell out more money when your key goes belly up (and they all eventually do). No, this isn't the middle of a nightmarish adventure game you've warped into—this is SPSS/PC's copy protection! (While we are on the subject of warnings: don't buy version 1.0. Version 1.1 corrects a number of significant oversights, and it will cost you an additional $100 to upgrade.)

Operationally, SPSS/PC purrs along like a kitten. It's fun and amusing to play with, and it appears immune to fatal operator errors. With an 8087 installed, it is quite frisky. Speed is not a major problem. SPSS/PC does what common sense and its manual says it will do, and it has been extensively debugged and carefully crafted. Support is provided by telephone during regular business hours (you pay the toll). Technical staff are knowledgeable, but you may have to wait 24 hours before they have time to fully answer your query. There is a mainframe-oriented users group but as yet no micro group. SPSS/PC will run on an IBM PC/XT, the AT, and on compatibles such as the Compaq, Tandy 2000, and TI Professional.

SPSS/PC is good, and it has brought me closer to my dream of having that IBM 370 on my desk. SPSS/PC's algorithms are fundamentally sound, it is finely crafted, and I would trust it to process a student's thesis. You won't find me in line to do batch processing nor cursing at the mainframe for being down again. I have my micro and I almost have it all!

DDJ

C Chest (Text begins on page 90)

Listing One

```
 1:  shell( argv, argc )
 2:  char    *argv[];
 3:  int     argc;
 4:  {
 5:          int gap, i, j;

 6:          for( gap = argc/2; gap > 0 ; gap /= 2)
 7:                  for( i = gap; i < argc; i++ )
 8:                          for( j = i-gap; j >= 0 ; j -= gap )
 9:                          {
10:                                  if( strcmp(argv[j], argv[j+gap]) <= 0 )
11:                                          break;
12:                                  exch( &argv[j], &argv[j+gap] );
13:                          }
14:  }
```

End Listing One

Listing Two

```
 1:  #include <stdio.h>

 2:  /*                                  QSORT.C
 3:   *
 4:   *                      Copyright (c) 1984 Allen I. Holub
 5:   *                            All rights reserved.
 6:   *
 7:   *      This program may be coppied for personal, non-profit use only.
 8:   *
 9:   *      Including a #define for DEBUG will make this file a stand-alone
10:   *      program which sorts argv and prints the result, along with all
11:   *      intermediate stages.  This is pretty instructive if you want to
12:   *      see how the quick sort works.
13:   */

14:  #ifdef DEBUG

15:  static int      Lev = 0 ;       /* Recursion level */
16:  static int      Maxlev = 0;     /* Maximum "       */
17:  #endif

18:  typedef int  (* PFI)();         /* pointer to a function returning int   */
19:  static PFI   Comp ;             /* Pointer to comparison routine         */
20:  static int   Width;             /* Width of an object in bytes           */

21:  /*-----------------------------------------------------------------------*/

22:  int     argvcmp( s1p, s2p )
23:  char    **s1p, **s2p;
24:  {
25:          /* Comparison routine for sorting an argv like list of pointers
26:           * to strings. Just remove one level of indirection and call
27:           * strcmp to do the comparison.
28:           */

29:  #ifdef DEBUG

30:          register int    rval;
31:          rval =  strcmp(*s1p, *s2p) ;
32:          printf("level %d: argvcmp(<%s>,<%s>) = %d\n", Lev,*s1p,*s2p,rval);
33:          return( rval );
34:  #else
35:          return( strcmp(*s1p, *s2p) );
36:  #endif
37:  }
38:  qsort( base,  nel, width, compare )
39:  char    *base;
40:  int     nel, width;
41:  int     (*compare)();
42:  {
43:          /*      Perform a quick sort on an array starting at base. The
44:           *      array is nel elements large and width is the size of a
45:           *      single element in bytes. Compare is a pointer to a
46:           *      comparison routine which will be passed pointers to two
47:           *      elements of the array. It should return a negative number
48:           *      if the left-most argument is less than the rightmost,
49:           *      0 if the two arguments are equal, a positive number if
50:           *      the left argument is greater than the right. If compare
```

 (Listing Continued, text begins on page 90)

Listing Two

```
51:             *       is 0 then the default comparison routine, argvcmp
52:             *       (which sorts an argv-like array of prointers to strings),
53:             *       is used.
54:             */

55:     #ifdef DEBUG
56:             printf("Sorting %d element array of %d byte elements at 0x%x\n",
57:                                                     nel, width, base );

58:             printf("Comparison routine at 0x%x. Unsorted list:\n", compare);
59:             ptext( nel, base );
60:     #endif

61:             Width = width;
62:             Comp  = (compare == (PFI)0) ? &argvcmp : compare ;

63:             if( nel > 1 )
64:                     rqsort( base, base + ((nel-1) * width) );

65:     #ifdef DEBUG
66:             printf("\nSort complete, list is:\n");
67:             ptext( nel, base );
68:             printf("Maximum recursion level = %d\n", Maxlev );
69:     #endif
70:     }

71:     /*-------------------------------------------------------------------------*/

72:     static          rqsort( low, high )
73:     register char   *low, *high;
74:     {
75:             /*      Workhorse function called by the access routine, qsort().
76:              *      Not externally accessible.
77:              */

78:             char            *pivot, *base;

79:             static char     *a, *b;         /* Used for exchanges, will not */
80:             static int      tmp, i;         /* need to retain their values  */
81:                                             /* during the recursion so they */
82:                                             /* can be static.               */

83:     #ifdef DEBUG
84:             printf("New pass, recursion level %d\n", Lev );
85:             if( Lev > Maxlev )
86:                     Maxlev = Lev;
87:     #endif

88:             base = low    ;         /* Remember base address of array */
89:             pivot = high  ;         /* Partition off the pivot        */
90:             high -= Width ;

91:             do
92:             {
93:                     while( low < high && (*Comp)(low, pivot)  <= 0 )
94:                             low += Width;

95:                     while( low < high && (*Comp)(high, pivot) >= 0 )
96:                             high -= Width;

97:                     if( low < high )            /* exchange  low  &  high  */
98:                     {
99:     #ifdef DEBUG
100:                            printf("lev %d: exchanging high:<%s> & low:<%s>\n",
101:                                    Lev,*((char **)high), *((char**)low));
102:    #endif
103:                            for( b = low, a = high, i = Width ; --i >= 0 ;)
104:                            {
105:                                    tmp   = *b  ;    /* Exchange *low and *high */
106:                                    *b++  = *a  ;
107:                                    *a++  = tmp ;
108:                            }
109:                    }
110:
111:            } while ( low < high );


112:    #ifdef DEBUG
113:            printf("level %d: Exchanging pivot:<%s>  & low:<%s>\n",
114:                            Lev, *((char **)pivot), *((char**)low) );
115:    #endif
```

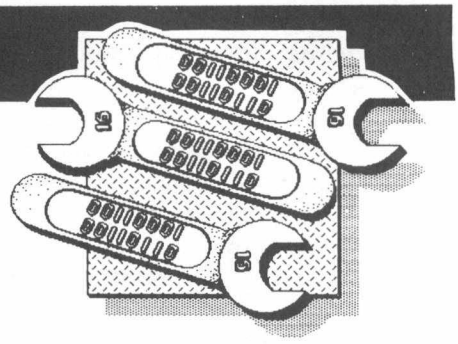
 (Listing Continued, text begins on page 90)

Listing Two

```
116:          if( low < pivot  &&  (*Comp)(low, pivot) > 0 )
117:                  for( b = low, a = pivot, i = Width ; --i >= 0 ;)
118:                  {
119:                          tmp   = *b  ;       /* Exchange *low and *pivot */
120:                          *b++  = *a  ;
121:                          *a++  = tmp ;
122:                  }
123:  #ifdef DEBUG
124:          printf("\nDone with pass, partially sorted list =\n");
125:          ptext( ((pivot - base)/Width) + 1 , base );
126:          printf("\n");
127:          ++Lev ;
128:  #endif

129:          low += Width;

130:          if( high - base  <  pivot - low )
131:          {
132:                  if( low < pivot )
133:                          rqsort( low, pivot );

134:                  if( base < high )
135:                          rqsort( base, high );
136:          }
137:          else
138:          {
139:                  if( base < high )
140:                          rqsort( base, high );

141:                  if( low < pivot )
142:                          rqsort( low, pivot );
143:          }

144:  #ifdef DEBUG
145:          --Lev ;
146:  #endif
147:  }
148:  /*------------------------------------------------------------------------*/
149:  /*      Test routine for qsort. compiled if DEBUG is #defined             */

150:  #ifdef DEBUG

151:  static ptext( argc, argv )
152:  int     argc;
153:  char    **argv;
154:  {
155:          /*      Print out argv, one element per line.
156:           */

157:          register int    i;

158:          for( i = 1; --argc >= 0 ; i++ )
159:                  printf("%2d: %s\n", i, *argv++ );
160:  }


161:  main(argc, argv)
162:  int     argc;
163:  char    **argv;
164:  {
165:          /* Test routine for qsort. Sorts argv (less the first element).
166:           */

167:          qsort( ++argv, --argc, sizeof(PFI) , 0 );
168:  }

169:  #endif
```

End Listings

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

A New Assembler for MSDOS

Philip Oliver has sent us a copy of his new product ED/ASM-86 for review. This is an interactive programming tool that combines the functions of an assembler, line editor, linker, and debugger in a single 50K program. 8087, 80186 and 80286 mnemonics are fully supported for assembly, tracing, and disassembly. Macros, structures, and conditional assembly directives are included. This is an interesting product, although I feel that its appeal would be broader if it were fully source compatible with the Microsoft Macro Assembler and if it could handle standard Microsoft Linker object files. The current price of ED/ASM-86 is $100.00; contact Oliver Computing Company at 9311 Jutland Court #D, Indianapolis, IN 46250 for more information.

TEE Filter for MSDOS 2

Mr. A. K. Head, of Melbourne, Australia, contributed a pair of MSDOS 2.x filters. We are printing the first of these, called TEE, in this month's column as Listing One (page 106). TEE, a standard filter in Unix systems, is used in a pipe to make two copies of the standard input, directing one to a file and the other to the standard output (which might also be redirected into a file). Where you normally would use a filename, here you can also use any logical device such as PRN: or CON:. For example, you could enter the command:

DIR ¦ SORT ¦ TEE C:SORTED.DIR

This would direct the output of the directory command to the SORT filter and thence to TEE, which would send one copy of the sorted listing to the disk file C:SORTED.DIR and the other copy to the standard output (console, in this case). This version of TEE doesn't support the Unix switches –i (ignore interrupts) or –a (append to previous contents of a file).

80286 Feedback

Matt Robins, of Beverly Hills, California, writes: "I am writing to you concerning two and a half errors in your November 1984 *DDJ* column.

"PCDOS 3.0 could not provide the '. . . same old horrible EDLIN that we first met in PCDOS 1.0.' In fact, as you probably know by now, the PCDOS 2.0 version of EDLIN is a significant improvement on the PCDOS 1.1 version that I started out with. It boasts 40% more commands (14 vs. 10) of the most advanced variety: Copy-line and Move-line that make programming easy. Additional capability is provided by the use of + and – signs, like old CP/M ED (which *was* a true abomination).

"What I particularly like about the new EDLIN is that it uses the DOS edit mode keys F1 through F4. I couldn't live without this capability! I've enhanced my EDLIN by assigning the arrow keys and Home, End, PgUp, and PgDn keys to ANSI sequences that execute the appropriate commands in EDLIN (to comply with the above mentioned keytop definitions) . . . .

"Your second mistake was calling the 80286 'a slightly enhanced 8086.' By now you must have read the November *BYTE* article by Paul Wells that correctly states: '. . . at the same clock speed and [when you] do not take advantage of any 80286 capabilities, the 80286 achieves 250 percent of the 8086's performance.' It does that, partly, by having five processors on board, as opposed to the 8086's three."

I agree that the 80286 in protected mode is a very different machine, but I still maintain that you don't gain much more by using an 80286 in real mode (as PCDOS 3.0 does) than you could get just by cranking up the clock on a normal 8086 and using 16-bit memory like the Compaq DeskPro.

Matt goes on to say: "Finally, a half-a-mistake (maybe only a matter of taste) is your saying that use of the multitasking, virtual memory, and memory protection will '. . . wring all of the available computing power out of the 80286.' I tend to think of computing power as throughput, and the three activities mentioned above lower the throughput, not facilitate it. Put another way, 'computing power' is benchmark timings, whereas the three activities above are memory management functions. Being a member of the 'one [or more] processors, one user' school of thought, I am interested in Unix and multitasking only for my personal use: performing several of *my* tasks simultaneously."

More Macintosh Feedback

Steve Rabalais writes: "The Macintosh contains a rich operating system that, if used right, will take a lot of the labor out of programming. Utilities for Floating Point (IEEE Standard), File Management, Transcendentals, etc., are easily accessed from Assembly Language. The Apple people supporting the development work are apparently impressed by the BIG NAME programming shops, and independent developers are left to their own wits (which is probably a good thing). I personally do not depend on phone calls to solve programming problems and prefer to 'dig it out' because this is the best way to learn. I attempted to become an Apple Certified Developer but failed to impress the Marketing and Sales people with

their requisite pile of paperwork. I purchased my Macintosh and Lisa 'over the counter' and began programming immediately. I have talked to Certified Developers that had to wait months to obtain their 'favored' hardware discounts. I do have 'INSIDE MACINTOSH' and have found it indispensable for writing Mac applications.

"Although I do not have the official Macintosh Apple Assembler, I have heard horror stories to the effect that it requires two Macs. I use an excellent, inexpensive Assembler called *MacAsm* from Mainstay that allows me to program complete applications on a 128K Macintosh with a single 400K disk drive. MacAsm is a two pass Macro Assembler and is complete with a Resource Compiler and Text Editor. I am writing a series of Macintosh CADD (computer-aided design and drafting) applications and have had no problems using this assembler. I have found it fast, efficient, and simple. I am *not* using the Apple Workshop on the Lisa to develop software because it is slow and complicated to use and Apple recently wanted more money for a complete set of the latest updates.

"I use the Workshop version I have to help learn 68000 assembly language. The Workshop contains a Pascal compiler that will dump the generated assembly code into a text file. I inspect the code, figure out how things are done, and then rewrite it (for efficiency and clarity) using MacAsm. This may sound like cheating, but I find that I am gradually depending less on the Lisa as the learning process continues. Macintosh programming in assembly language is *hard* to learn, but once you have learned, it is *easy* to write sophisticated applications. Independent Developers have made the Apple II successful and apparently will do the same for the Macintosh in spite of the roadblocks. In conclusion, I would say that the Macintosh is a superb combination of hardware and software, and a proper programming environment can be had at a reasonable price . . . . Hopefully, other programmers will read this and know that there is help in the wilderness for programming the Macintosh."

Steve also contributed a programming tip for the 68000, which appears as Listing Two (page 110) following this column.

80286 XENIX

Speaking of Unix, we have been using 80286 XENIX 3.0 for the PC/AT here at Laboratory Microsystems for about two months. Getting acquainted with Unix is an entertaining but aggravating experience. Unix devotees claim that it was designed by programmers for programmers and increases productivity enormously, but to me it looks like it grew up topsy turvy out of the late-night efforts of a legion of graduate student hackers. No doubt, back when Ken Thompson and Dennis Ritchie invented Unix, it was compact and elegant (after all, it ran on a PDP-7), but it has grown into a behemoth that needs 256K of RAM to get off the ground.

The "Adventure" game mentality is seen everywhere in the system. "You are lost in a twisty, turning maze of cryptic, poorly indexed software manuals." There is no uniformity of syntax whatsoever, the commands have obscure or arcane names (grep, ls, cat), switches are case-sensitive (for instance, a lower-case "s" switch and an upper-case "S" switch on the C compiler do vastly different things), and the screen editor *vi* makes WordStar look positively user friendly.

The documentation for XENIX is clearly derived from the original Unix manuals, even though it has been filtered through the technical writers at Microsoft and then IBM. One often comes across unexpectedly witty sayings, such as (under the SPLINE command) "a limit of 1000 points is silently enforced" or (in the terminal capabilities library) "Hazeltime = brain damage" and "Names starting with X are reserved for serious disturbances." Command names frequently have a high cuteness factor, such as "finger" (report names of active users), "whodo," and "stty sane" (used when the operating system loses its mind and starts spitting garbage out to your terminal).

None of this is meant as a criticism of the Microsoft implementation, however, because the boys in Bellevue seem to have done a pretty good job of implementing System III on the 80286. And this was *no* weekend fling, either. IBM states that "the total number of lines that migrated from the original AT&T source code to PC XENIX was well over 400,000 lines."

The performance, even with a couple of extra terminals hung onto the system, is quite reasonable—due mainly to the PC/AT's very fast hard disk. XENIX 3.0 runs the 80286 in protected mode and is very crash-proof compared to PCDOS. Any attempt by your program to access memory not specifically allocated to your program, or to execute an illegal or privileged opcode, results in its immediate termination. We have found that as you develop new software under XENIX, the friendly message "Memory fault—core dump" will be your constant companion.

A few warnings before you rush out to buy a PC/AT and XENIX. There is no operating system support for graphics display modes; there is only very primitive support for the keyboard (ALT-key combinations can't be read, and even reading the unadorned function or arrow keys must be done through writing control strings to the ANSI driver); and the C compiler apparently has bugs in the "large model" and the optimizer, as Microsoft explicitly warned us not to use those features.

Because 80286 XENIX runs in protected mode, direct access to the hardware is, for all practical purposes, nonexistent—you can forget about your carefully optimized, memory-mapped video routines and all the other code you've painstakingly created to make your PCDOS applications look flashy. The ROM BIOS isn't available either. In fact, your program can't even tell where it is running, let alone where anything else is: the segment registers contain only logical selectors, and the corresponding physical addresses are not visible. Those of us who have become spoiled by the ability on micros to reach out and twiddle bits or ports wherever we please are out of luck.

One more warning, and not the least: the operating system requires some 15 Mbytes of hard disk space. If you are planning to use both XENIX and PCDOS on your PC/AT, best to plan ahead by buying *two* hard disks.

At the UNIFORUM conference in Dallas that we attended in January, Microsoft discussed plans to upgrade 80286 XENIX for compatibility with AT&T's Unix System V. Where this will leave purchasers of IBM PC/AT XENIX is not clear. The PC/AT version is being sold and supported directly by IBM, and, as we have found out to our sorrow with other products (such as the Microsoft Assembler), new improved versions released by Microsoft hardly ever seem to migrate down the pipeline to show up in IBM distribution.

But however you slice it, the release of PC/AT XENIX is a significant event for the Unix community. This is the first time that a personal computer powerful enough to run Unix with decent performance has been available at a price the average small businessman would be willing to pay; the added prestige of the three magic initials behind it (no, I don't mean AT&T) may be sufficient to let it travel into uncharted regions where no Unix has gone before. The total installed base of Unix and Unix-derivative systems in the world is currently estimated to be less than 100,000; IBM could conceivably double or triple this in a year with the right kind of marketing and application software. It should be interesting to see what happens.

DDJ

16-Bit

Listing One

```
        name    tee
        page    55,132
        title   'TEE - tee junction for DOS pipes'
;
; TEE --- A "tee" junction for DOS 2.x pipes
;         after the fashion of the Unix function TEE
;
; By A. K. Head, 6 Duffryn Place, Melbourne, Australia 3142
;    reformatted and error handling added by Ray Duncan
;
; TEE reads from the standard input (redirectable) and
; outputs to both the standard output (redirectable)
; and a file, e.g.:   DIR | TEE C:\MYDIR\FILE.EXT | SORT
;
; The file can be CON, LPT1, etc.  If it is CON, then
; keyboard Control-S pauses the display as usual.

command equ     80h             ; buffer for command tail
buflen  equ     16384           ; buffer length, alter to taste

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII line feed
ff      equ     0ch             ; ASCII form feed
eof     equ     1ah             ; End-of-file marker
tab     equ     09h             ; ASCII tab code
blank   equ     20h             ; ASCII blank

                                ; DOS 2.x pre-defined handles
stdin   equ     0000            ; standard input file
stdout  equ     0001            ; standard output file
stderr  equ     0002            ; standard error file
stdaux  equ     0003            ; standard auxilliary file
stdprn  equ     0004            ; standard printer file

cseg    segment para public 'CODE'

        assume  cs:cseg,ds:cseg

        org     100H            ; start .COM at 100H

tee     proc    far

        mov     cl,ds:command   ; get length of command tail.
        xor     ch,ch
                                ; address of command string.
        mov     si,offset command+1
        mov     di,offset command+1
tee1:                           ; squeeze out leading spaces.
        mov     al,byte ptr [si]
        cmp     al,blank
        jbe     tee2
        mov     byte ptr [di],al
        inc     di
```

(Continued on page 108)

Listing One

```
tee2:   inc     si              ; look through command tail
        loop    tee1            ; until it's exhausted.
        mov     byte ptr [di],0 ; make ASCIIZ string.
        mov     ah,3ch          ; create output file.
        mov     cx,0            ; attribute=0.
        mov     dx,offset command+1
        int     21h
        jc      tee6            ; can't create file
        mov     handle,ax       ; save token for file.

tee3:   mov     ah,3fh          ; read standard input.
        mov     bx,stdin
        mov     cx,buflen
        lea     dx,buffer
        int     21h
        jc      tee4            ; jump, error.
        mov     nchar,ax        ; save length of read.
        cmp     ax,0            ; nothing read in?
        je      tee4            ; end of file, exit.
        mov     ah,40h          ; write to file...
        mov     bx,handle
        mov     cx,nchar
        int     21h
        jc      tee7            ; jump, write failed.
        cmp     ax,nchar
        jne     tee7            ; jump, write failed.
        mov     ah,40h          ; now write to standard output...
        mov     bx,stdout
        int     21h
        jc      tee7            ; jump, write failed.
        cmp     ax,nchar
        jne     tee7            ; jump, write failed.
        jmp     tee3            ; read again until end of file.

tee4:   mov     ah,3eh          ; close output file.
        mov     bx,handle
        int     21h

tee5:   mov     ax,4c00h        ; exit with return code=0.
        int     21h

tee6:   mov     dx,offset err1  ; print "Can't create file".
        mov     cx,err1len
        mov     ah,40h          ; display error message and exit.
        mov     bx,stderr
        int     21h
        mov     ax,4c01h        ; exit with return code=1.
        int     21h

tee7:   mov     dx,offset err2  ; print "Disk is full".
        mov     cx,err2len
        mov     ah,40h          ; display error message and exit.
        mov     bx,stderr
        int     21h
        mov     ah,3eh          ; close output file.
        mov     bx,handle
        int     21h
        mov     ax,4c02h        ; exit with return code=2.
        int     21h

tee     endp

nchar   dw      0               ; number of chars actually input.
handle  dw      0               ; token for output file.

err1    db      cr,lf
        db      'tee: Cannot create file'
        db      cr,lf
err1len equ     (this byte)-(offset err1)

err2    db      cr,lf
        db      'tee: Disk is full.'
```

(Continued on page 110)

 (Listing Continued, text begins on page 102)

Listing One

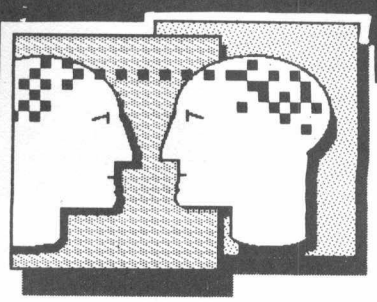
```
        db      cr,lf
err2len equ     (this byte)-(offset err2)

buffer  equ     this byte       ; data is read here from
                                ; standard input

cseg    ends

        end     tee
```

End Listing One

Listing Two

```
Programming Tip:   Jump Tables are a convenient,  efficient,
and general way of directing program flow based on a
multiple choice. Here is an implementation of a Jump Table
for the 68000.

*
*  Register D0 = choice index from 0 to N
*
start     equ       *
          add.w     d0,d0
          move.w    jtable(pc,d0.w),d0
          jmp       jtable(pc,d0.w)
jtable    equ       *
          data      /jump000-jtable
          data      /jump001-jtable
          data      /jump002-jtable
          data      /jump003-jtable
          (etc)
          (etc)
          (etc)
jump000   equ       *
          (code)
          bra       continu
jump001   equ       *
          (code)
          bra       continu
jump002   equ       *
          (code)
          bra       continu
jump003   equ       *
          (code)
          bra       continu
          (etc)
          (etc)
continu   equ       *
          (code)



End of Listing 2
```

End Listings



by Robert Blum

Bill Wilson of Pullman, Washington wrote several months ago describing a program he had written to simplify batch processing. He also included a patch to enable PIP to accept null command lines from a submit file. I am sure you will find, as I did, that both are most useful and real time-savers.

The patch described here is not approved by DRI; therefore you should first backup your distribution version of PIP and retain it for use in the unlikely event that you experience a problem. I further suggest that you exercise extreme caution as you modify PIP and install it on your system. The installation steps listed here are straightforward and shouldn't take more than a few minutes to enter; but do be mindful not to overwrite anything of value.

Listing Two (page 120) is the step by step procedure that I used to install the null line patch into PIP on my system. The first step is to verify that the PIP program being modified is version 1.5. This is the only version of PIP that I know the patch to work with successfully. To verify that you have the right version of PIP load PIP.COM with SID or DDT and display memory between 200h and 240h. At 0233h should begin the version number, which must be 1.5. Next, carefully enter the series of patches listed. And finally, save the modified program back onto disk under a new name ready for testing.

CP/M Version 2.2 Improved Batch Procedures

Batch mode automatic sequential command processing under CP/M 2.2 using SUBMIT is very inconvenient. Especially when using XSUB simply to pass a few parameters to a program. For instance, to copy a few files from one disk to another requires that ED or some other text editor first be used to build a disk file of the desired control statements. Then SUBMIT must be run to build yet another disk file of the very same control statements but in a format suitable for the CCP to use. The alternative approach is to enter each control statement from the keyboard as the copy program asks for them. Which is fine providing not too many files are being copied or you have plenty of time.

The prospect of continually being hampered by a lengthy process when using batch procedures prompted me to develop a stand-alone $$$.SUB file generator called BATCH. A source listing for BATCH written for the Z80 is given in Listing One (page 114).

The BATCH program served another very useful purpose as well. I was in the process of developing a CCP replacement in conjunction with a Computer Science student at WSU that was to have a built-in SUBMIT file generator. Thus it was important to ensure that the $$$.SUB file processor within the CCP replacement worked properly. Program BATCH was developed for this purpose, as well. As presented here it may be used to replace the combination of ED and SUBMIT for simple BATCH mode command processing.

The null line required for exiting from PIP, a line containing a CR only, also caused us a problem in the development of the CCP replacement. It is not difficult for a program like BATCH to produce a null line in a command string, but the processing of a null line by the $$$.SUB file processor caused problems. That is, it is most difficult for the $$$.SUB file processor to distinguish between a null line intended for exiting from PIP and one to designate that the end of the $$$.SUB has been found without a lot of code that uses an excessive amount of space.

A $$$.SUB file has a rather tricky construction with one command per sector stacked in reverse order. The $$$.SUB processor opens the $$$.SUB file, goes to the end of the file, reads in the last sector, decrements the record count in the $$$.SUB File Control Block (FCB), closes the $$$.SUB file, and then executes the command. In other words, the $$$.SUB file processor peels commands one at a time from the end of a $$$.SUB file. The crux of the PIP null line problem is that the $$$.SUB processor quits and erases the $$$.SUB file at the end of the file or if the character count is zero.

In order to make XSUB, PIP, and the CCP replacement function efficiently I decided to take a stab at modifying PIP. I spent an evening at home disassemling PIP on my S-100 buss Z-80 based system running under CP/M 2.2. I have a very good disassembler so the process was not very difficult. I was quite amazed by the old extra code for paper tape handling that is unused but still retained within PIP. Needless to say, I was not very impressed by what I found within PIP. However, because of the unused code, it was simple to insert some new code to solve the null line problem. Extensive testing with BATCH, XSUB, PIP, and the CCP replacement indicated that the fix worked fine. PIP would now exit on a null line or single character line. The modified version of PIP would only attempt to process an input line with two or more characters.

The next morning while shaving it dawned on me that an even simpler fix to the PIP null line problem was possi-

ble. Since it was Saturday and I did not have to go to work, I gave my new fix a try with the same results as before.

The procedure for installing the simple PIP fix using SID is shown here in Listing Two. PIP uses the buffered line function for input and does a CPI 00 at location 054Fh to check the number of characters entered. The JNZ at 0551h causes PIP to exit if and only if the character count is zero. That is, a null line is required to exit PIP. A simple fix involves changing the CPI 00 to a SUI 02 and the JNZ to a JP. PIP will now exit if the character count is less than 2. This modification will allow XSUB, PIP, CCP, etc., to all work in harmony.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 200.**

CP/M Exchange

(Text begins on page 112)

Listing One

```
;
; ** BATCH version 2.1**
;    A program to generate a $$$.SUB submit file
;     for immediate processing under CP/M 2.2 .
;
;--NOTES:
;       1) The $$$.SUB file is created on disk unit A
;          since the D. R. utility XSUB will only operate
;          on disk unit A. If BATCH is run from disk B,
;          for example, the $$$.SUB file will be created
;          on disk unit A and then processed from disk A.
;          All files referenced in the submit file created
;          by BATCH must be on the disk unit from which
;          BATCH is being run including PIP and XSUB.
;
;       2) Use DDT to Modify PIP as follows to solve the
;          null line PIP exit problem: change 053D from
;          2A to 23, 0545 from FE to D6, 0550 from 00 to
;          02, and 0551 from C2 to F2. Exit DDT by entering
;          G0, and then do a SAVE 32 PIP.COM.  The modified
;          version of PIP outputs # as an input request
;          symbol and will exit on a null or single
;          character input line.
;
;       3) Input line must not exceed input buffer length
;          which is 128 characters.
;
;       4) Example use of BATCH:
;               >BATCH<RET>
;               ->XSUB/PIP/B:=A:XDIR.COM/X/DIR A:/DIR B:<RET>
;
; ** PROGRAM DEFINITIONS **
CR      EQU     0DH             ;CARRIAGE RETURN
LF      EQU     0AH             ;LINE FEED
BDOS    EQU     005H            ;ADDR CP/M BDOS FUNCTION CALLS
;
        ORG     100H            ;PROG ORG
;
START:  LD      HL,00H
        ADD     HL,SP           ;GET CP/M STACK POINTER
        LD      (CPMSTK),HL     ;SAVE IT
        LD      SP,STACK        ;SET UP NEW STACK
        LD      DE,PROMPT       ;PTR TO SIGNON MSG
        CALL    PUTSTR          ;SEND STRING
        CALL    SETDMA          ;SET DKS BUF ADDR
        CALL    GETBUF          ;GET INPUT
        LD      HL,INPBUF+1     ;PTR NUM INP CHRS
        LD      A,(HL)          ;GET NUM INP CHRS
        LD      C,A             ;SAVE CHR CNT IN C
        DEC     A               ;CK IF ANY CHRS IN BUFF
        JP      M,BCKCPM        ;NO CHRS, BACK TO CP/M
        JP      Z,BCKCPM        ;NO CHRS, BACK TO CP/M
        LD      A,0FFH          ;GET FF=START OF INPUT
        LD      (HL),A          ;SET CHR CNT=FF
        LD      A,C             ;GET CHR CNT AGAIN
        INC     A               ;ADD ONE FOR 01 END CHR
;--ADD A TO HL TO FORM ADDR END INP STRING--
        CALL    ADATHL
        LD      (HL),01         ;INDICATE END OF INPUT
;
;--SELECT DISK UNIT A--
```

(Continued on page 116)

(Listing Continued, text begins on page 112)

Listing One

```
        CALL    INIT
;
        CALL    DELETE          ;DELETE OLD $$$.SUB
        CALL    MAKE            ;MAKE NEW $$$.SUB
; SET CURRENT RECORD=0
        LD      HL,SUBFCB+32    ;HL=PTR CUR POS IN FCB
        XOR     A
        LD      (HL),A
;--CONVERT BUFFER TO UPPER CASE CHRS--
;--SKIP ANY LEADING BLANK CHARS IN 1ST STRING--
        CALL    CNVBUF
        LD      HL,INPBUF+2     ;HL=PTR INPUT BUFF
;--COUNT CHRS IN INPUT STRING AND SET HL TO END--
COUNT   CALL    CNTCHR
;--MAIN COMMAND SCANNING LOOP (RIGHT TO LEFT)--
SCNLOP  LD      A,(HL)          ;GET CHR
        CP      0FFH            ;CHECK IF DONE
        JR      Z,DONE          ;QUIT, DONE
        LD      C,0             ;BACKUP TO PREVIOUS /
SCNLP2  DEC     HL              ;DEC CHZ PTR
        LD      A,(HL)          ;GET CHR
        INC     C               ;INC COUNT
;--FF=BEGINNING (END) OF INPUT STRING--
        CP      0FFH            ;CHECK IF LAST CMND
        JR      Z,PUTCMN        ;PUT LAST COMMAND IN SUB FILE
;--/=BEGINNING (END) OF ONE COMMAND STRING--
        CP      '/'             ;CHECK FOR / BETWEEN COMMANDS
        JR      NZ,SCNLP2       ;LOOP TILL / OR FF
;--PUT COMMAND INTO SUBMIT FILE--
;--USE CP/M DEFAULT BUFFER AREA--
PUTCMN  PUSH    HL              ;SAVE POINTER
        DEC     C               ;DEC FOR / CHR
        INC     HL              ;SKIP OVER /
        LD      A,C
        LD      (80H),A         ;PUT LENGTH IN BUFFER
        LD      DE,81H
        CALL    BLKMOV          ;MOVE COMMAND TO BUFFER
        EX      DE,HL           ;SET HL TO END OF COMMAND
;--ZERO REST OF BUFFER--
        LD      A,127
        SUB     A,C
        LD      B,A             ;SET LOOP COUNT
        LD      C,0
ZROLOP  LD      (HL),C
        INC     HL
        DJNZ    ZROLOP
;--WRITE ONE SECTOR--
        CALL    WRITE           ;WRITE BUFF TO SUB FILE
        POP     HL              ;GET POSITION POINTER AGAIN
        INC     A               ;CHECK FOR WRITE ERROR
        JR      NZ,SCNLOP       ;REPEAT TILL DONE.
;--DISK ERROR--
        LD      DE,NSPMSG
        JP      SERMSG
;
;--DONE, BACK TO CP/M--
;--DO A SYS RESET TO START SUBMIT--
DONE:   CALL    CLOSE
        JP      000H

;
;--COUNT THE CHARS IN THE COMMAND--
;  MOVE HL TO END OF COMMAND
CNTCHR: LD      C,1             ;INIT CNT=1
CNTLOP: LD      A,(HL)
        CP      0DH             ;END IF CR
        RET     Z
        CP      01H             ;END IF CHR=01H
        RET     Z
        INC     C
        INC     HL
        JR      CNTLOP
;
;--ADD A TO HL--
```

(Continued on page 118)

Listing One

```
ADATHL  ADD     A,L
        LD      L,A
        RET     NC
        INC     H
        RET
;
;--BLOCK MOVE--
;   HL=SOURCE, DE=DESTINATION, C=NUMBER OF CHARS
BLKMOV  LD      B,00H           ;ZERO B
        LDIR
        RET
;
;--CONVERT BUFFER TO UPPER CASE--
CNVBUF  LD      HL,INPBUF+2
        LD      D,H
        LD      E,L
;--SKIP LEADING BLANK CHARS--
SKPLOP  LD      A,(HL)
        CP      ' '
        JR      NZ,CNVLOP
        INC     HL
        JR      SKPLOP
CNVLOP  CALL    GETCHR
        LD      (DE),A          ;MOVE CHR
        INC     DE
        INC     HL
        CP      01H             ;END ?
        JR      NZ,CNVLOP
        RET
;
;--GET NEXT CHAR AND TO CONV UPPER CASE--
GETCHR  LD      A,(HL)          ;GET CHR
        CP      07BH            ;ABOVE z ?
        RET     NC
        CP      061H            ;BELOW a ?
        RET     C
        AND     05FH            ;LOW TO UP
        RET
;
;--GET BUFFERED LINE OF COMMANDS--
GETBUF: LD      DE,INPBUF       ;INP BUFF ADDR
        LD      C,10
        CALL    BDOS
        RET
;
;--SET DISK BUFFER ADDRESS(DMA)--
SETDMA  LD      DE,080H         ;BUFF ADDR
        LD      C,26
        CALL    BDOS
        RET
;
;--WRITE CONTENTS OF BUFF TO SUB FILM--
WRITE:  LD      DE,SUBFCB
        LD      C,21
        CALL    BDOS
        RET
;
;--CLOSE $$$.SUB--
CLOSE   LD      C,16
        LD      DE,SUBFCB
        CALL    BDOS
        RET
;
;--MAKE NEW $$$.SUB FILE--
MAKE:   LD      DE,SUBFCB
        LD      C,22            ;MAKE
        CALL    BDOS
        INC     A               ;FF=NO SPACE
        JP      Z,NOSERR        ;NO SPACE ERROR
        RET
;
;--DELETE ANY OLD  $$$.SUB FILE--
DELETE: LD      DE,SUBFCB
        LD      C,19
```

(Continued on page 120)

 (Listing Continued, text begins on page 112)

Listing One

```
        CALL    BDOS
        RET
;
;--INIT AND SELECT DISK A--
INIT:   LD      C,13            ;RESET
        CALL    BDOS
        RET
;
;--SEND STRING TO TERMINAL--
; --DE = POINTER TO STRING ADDRESS--
PUTSTR  LD      C,9
        CALL    BDOS
        RET
;
;--MESSAGES--
PROMPT  DEFB    CR,LF,'BATCH v2.1 Submit File Generator'
        DEFB    CR,LF,'for use with XSUB under CP/M 2.2  .'
        DEFB    CR,LF,'Enter Command String and then <RET> .'
        DEFB    CR,LF,'Separate Each Command with a / Symbol .'
        DEFB    CR,LF,'/X may be used to exit modified version of PIP .
        DEFB    CR,LF,'->$'
NSPMSG  DEFB    CR,LF,'Disk Full--No More Space'
        DEFB    CR,LF,07H,07H,'$'
NDSMSG  DEFB    CR,LF,'Directory Full--No More Space'
        DEFB    CR,LF,07H,07H,'$'
;
NOSERR  LD      DE,NDSMSG
;--SEND ERROR MESSAGE AND BACK TO CP/M--
SERMSG  CALL    PUTSTR
BCKCPM  LD      HL,(CPMSTK)
        LD      SP,HL           ;RESET STACK POINTER
        RET                     ;BACK TO CP/M
;
;-- FILE CONTROL BLOCK--
SUBFCB  DEFB    0H              ;DRIVE CODE
        DEFB    '$$$     SUB'   ;FILE NAME
        DEFB    0H,0H,0H,0H
        DEFS    17
;--CP/M STACK STORAGE--
CPMSTK  DEFW    00H
;-- INPUT BUFFER AREA--
INPBUF  DEFB    128             ;BUF LENGTH
BUFCNT  DEFB    0H              ;CHRS IN BUF
        DEFS    128
;--STACK AREA--
        DEFS    20              ;SIZE OF STACK
STACK   EQU     $               ;BOTTOM OF STACK
        END
```

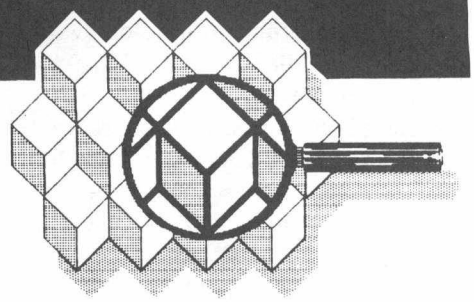
End Listing One

Listing Two

```
B>SID PIP.COM
CP/M 3 SID - Version 3.0
NEXT MSZE  PC  END
1E00 1E00 0100 C8FF
#D200,240
0200 20 20 20 43 4F 50 59 52 49 47 48 54 20 28 43 29    COPYRIGHT (C)
0210 20 31 39 37 39 2C 20 44 49 47 49 54 41 4C 20 52  1979, DIGITAL R
0220 45 53 45 41 52 43 48 2C 20 20 50 49 50 20 56 45 ESEARCH,  PIP VE
0230 52 53 20 31 2E 35 03 01 06 01 00 24 24 24 20 20 RS 1.5.....$$$
0240 20
#S053D
053D 2A 23
053E CD .
#S054F
054F FE D6
0550 00 02
0551 C2 F2
0552 5E .
#WPIPNEW.COM,100,1E00
003Ah record(s) written.
#G0
```

End Listings

OF INTEREST



by R. P. Sutherland

Hardcopy

Three articles in this issue illustrate techniques to generate various nonstandard printed characters. Despite their present utility, they will soon seem curious and quaint. The proliferation of the Macintosh and Mac-alike machines as well as the porting of Donald Knuth's TEX to micros (in tandem with the availability of laser printers) will result in easier ways of coaxing two and three dimensional graphic concepts into hardcopy. The Figure below shows a sample of output from TEX as well as a list of vendors who carry micro implementations.

Publications

Privacy Journal: An Independent Monthly on Privacy in the Computer Age contains the seeds of the antidote to *1984*. Subscriptions: $89.00 per year, $109.00 overseas. Contact Kathryn Ritter, *Privacy Journal*, P.O. Box 15300, Washington, D.C. 20003 (202) 547-2865. **Reader Service No. 101.**

Apple Access: User's Guide to Apple Computer-Related Periodical Literature for $19.95 is a reference for Apple literature. Contact Lee Webber, Stony Point Publications, Box 4467, Petaluma, CA 94953 (707) 778-8754. **Reader Service No. 103.**

The Datapro Complete Guide to Dial-Up Databases profiles more than a thousand accessible data bases in over 200 subjects. Contact Karen Morrell, Datapro Research Corpora-

"small" TEX implementations

Computer	Processor	Contact	Organization, Address
Hewlett-Packard 3000	16-bit	Lance Carnes	TexET, 163 Linden Lane, Mill Valley, CA 94941; 415-388-8853
Hewlett-Packard 1000	16-bit	John Johnson	JDJ Wordware, Box 354, Cupertino, CA 95015; 415-965-3245
DEC PDP-11/44 Plexus, Onyx IBM PC	16-bit Z8000 8086/88	Dick Gauthier	TYX, 11250 Roger Bacon Dr., Suite 16, Reston, VA 22090; 703-471-0233
Apollo	MC68000	Thom Hickey Bill Gropp Pierre Clouthier	OCLC, Box 7777, Dublin OH 43017; 614-764-6075 Dept. of Computer Science, Yale University, Box 2158, Yale Station, New Haven, CT 06520; 203-436-3761 COS Information, 6272 Notre Dame West, Montreal, H4C 1V4, P.Q.; 514-935-4222
Hewlett-Packard 9836	MC68000	Jim Crumly	Hewlett-Packard, Box 15, Boise, ID 83707; 208-376-6000 x2869
Sun Microsystems	MC68000	Jim Sterken Rich Furuta	Textset, Box 7993, Ann Arbor, MI 48107; 313-996-3566 University of Washington, Computer Science, FR-35, Seattle, WA 98195; 206-543-7798
Cyb	MC68000	Norman Naugle	Mathematics Dept., Texas A&M University, College Station, TX 77843; 409-845-3104
Apple MacIntosh, Lisa	MC68000	Barry Smith Dave Kellern	Kellerman & Smith, 2343 SE 45th Av., Portland, OR 97215; 503-232-4799
Masscomp	MC68000	Bart Childs	Dept. of Computer Science, Texas A&M University, College Station, TX 77843; 409-845-5470
Synapse	MC68000	Dick Wallenstein	Comcon, 5 Underwood Ct., Delran, NJ 08075; 609-764-1720
PERQ/ICL		Jaap van 't Ooster	Océ, St. Urbanusweg 43, 5900 MA Venlo, Holland
IBM PC, XT, AT	8088, 80286	Lance Carnes	TexET, 163 Linden Lane, Mill Valley, CA 94941; 415-388-8853
IBM XT, AT	8088, 80286	David Fuchs	Dept. of Computer Science, Stanford University, Stanford, CA 94305
IBM XT, AT	8088, 80286	Ronny Bar-Gadda	446 College Av., Palo Alto, CA 94306; 415-326-1275

Figure

tion, 1805 Underwood Boulevard, Delran, NJ 08705 (800) 257-9406 (in New Jersey 609/764-0100). Price: $145.00. **Reader Service No. 105.**

Frank Gaude's **Z-System Newsletter** for ZCPR3 users announces a Z-system electronic bulletin board at (213) 670-9465. Echelon, Inc., 101 First Street, Los Altos, CA 94022. **Reader Service No. 107.**

MSX World: The Newsletter Dedicated to the World of MSX Computers is published by MSX World, 39 W. 32nd Street, Suite 800, New York, NY 10001. **Reader Service No. 109.**

C

A demonstration of the **Safe Runtime Analyzer** (Catalytix Corporation) is available at (617) 497-4829: 1200 baud, Unix System III, login is safec, password is catalytix. The demonstration allows one to upload C programs for analysis by the Safe C Runtime Analyzer, twenty-four hours a day. Contact Janice Tosi, Catalytix Corporation, 55 Wheeler Street, Cambridge MA 02138 (617) 497-2160. **Reader Service No. 111.**

The **C lib**, a C function library, provides over 200 routines including functions to convert floating point numbers from MicroSoft to 8087 NDP format and vice versa. Price: $145.00 from Vance Info Systems, 2818 Clay Street, San Francisco, CA 94115 (415) 922-6539. Contact: Carla Radosta. **Reader Service No. 113.**

A **mathematics library** compatible with most C compilers is available for $100.00 from Micro International, Box 47, Route 36, East Fairfield, VT 05448 (802) 827-3827. Contact: Bernard Hussels. **Reader Service No. 115.**

MSDOS

A special version of Microsoft's *MSDOS* 2.11 operating system is available for S-100 computers. **SB-86**, from Lifeboat Associates, requires a CompuPro System Support I card with G086 EPROM, and at least 64K of 24-bit addressable RAM. The installation routine supports both interrupt-driven and non-interrupt-driven console I/O. The price of SB-86 is $275.00, available from Lifeboat Associates, 1651 Third Avenue, New York, NY 10128 (212) 860-0300. Contact: Doug Barth. **Reader Service No. 117.**

MasterForth Version 1.0 provides Forth-83 for the IBM PC family. MasterForth includes an 8088 macro-assembler and full interface to MSDOS 2.1. Price $125.00 from MicroMotion, 12077 Wilshire Boulevard #506, Los Angeles, CA 90025 (213) 821-4340. Contact: Linda Kahn. **Reader Service No. 119.**

MJ, from Mother Jones' Son's Software Corporation, is a utility that co-exists with MSDOS and permits one to open a window and reassign a key up to a thousand keystrokes. Also, MJ allows one to speed up the screen cursor in five increments and to expand the keystroke buffer from sixteen characters to over a thousand. MJ is not copy protected but if you abuse your backup privilege, you forfeit your soul to the author of the program. Price: $30.00 for object program, or $70.00 for object program and assembly language source code. Mother Jones' Son's Software Corporation, 6310 Caballaro Boulevard, Buena Park, CA 90620 (714) 522-7762. **Reader Service No. 121.**

Microcomputer owners who depreciate equipment for tax reasons are often approached by the IRS and asked to account for every minute spent on the computer. **TaxLog** is a product that keeps track of how much time one spends on various activities. TaxLog calculates the percentage of tax deductible use and prints a choice of reports that meet tax code requirements. Taxlog is part of Bellsoft's "Pop-Up" line of desk tools. TaxLog costs $39.95 from Bellsoft, 2820 Northrup Way, Bellevue, WA 98004 (206) 828-7282. Contact: Richard Leeds. **Reader Service No. 123.**

Macintosh

The **Public Domain Exchange** announces over a hundred public domain programs for the Macintosh. A listing of programs is available for $2.00. An introductory set of the three most popular disks and a cata-

log is $28.00. The Public Domain Exchange, 673 Hermitage Lane, San Jose, CA 95134 (408) 942-0309. Contact: Judy Rosenthal. **Reader Service No. 125.**

PortaAPL is now available for the Macintosh (512K). Price: $275.00. Portable Software, 60 Aberdeen Avenue, Cambridge, MA 02138 (617) 547-2918. Contact: Richard Smith. **Reader Service No. 127.**

Hard Disks

For a year now we have been trying to wear out a **Davong** 10 megabyte hard disk by driving our Davong Multilink system (our in-house LAN) 24 hours a day. So far it has been so reliable that we have become careless about backing anything up! Davong now has an 86-megabyte hard disk for the IBM PC. Davong's hard disk for the Macintosh is called **Mac Disk**. Mac Disks range from 10 to 43 megabytes and come with software that allows the user to configure and maintain multiple volumes. Contact John Houthton, Davong Systems, Inc., 217 Humboldt Court, Sunnyvale, CA 94089 (408) 734-4900. **Reader Service No. 129.**

Data Technology's **TeamMate** line of hard disks includes a Kodak-based 3.3 megabyte 5 1/4 inch floppy. I found the TeamMate hard disk easy to install and reliable. The Kodak floppy can be used to back up the hard disk or it can be used as an additional volume. The 3.3 megabyte floppy makes sense as a replacement to the AT's 1.2 megabyte drive because it more than doubles the capacity and it can write 48 TPI floppies more reliably than the AT's drive. Data Technology Corporation, 2775 Northwestern Parkway, Santa Clara. CA 95051 (408) 496-0434. Contact: Steve Roberts. **Reader Service No. 131.**

Single Board Computers

ForthCard, a single board computer on the STD bus with Forth kernel built in, features on-board EPROM programming and supports compilation of code from another computer. ForthCard is $499.00 in single quantities. An OEM ForthCard is also available. Contact: Ken Arnold, Hi-Tech Equipment Corporation, 9560 Black Mountain Road, San Diego, CA 92126 (619) 566-1892. **Reader Service No. 133.**

Motel Computers Ltd. has a 68000/Z80 based single board computer called **CYPHER**. Motel Computers Limited, 174 Betty Ann Drive, Willowdale, Ontario M2N 1X6 (416) 221-2340. Contact: Victor Pierobon. **Reader Service No. 135.**

Sundries

Maynard Electronics, Inc., offers a board that enables an IBM PC to run programs in DOS 3.0 that take advantage of extended memory. There are no software drivers required and the company claims the system runs as fast as the AT. Maynard Electronics, 430 Semoran Boulevard, Suite 207, Casselberry, FL 32707 (305) 331-6402. Contact: David Knapp. **Reader ServiceNo. 141.**

The **Public Domain Users Group** has expanded to include Commodore 64 users. For information and listings of the library disks, send SASE to Public Domain Users Group, P.O. Box 1442, Orange Park, FL 32067. **Reader Service No. 137.**

CommuniTree Group's **First Edition** (for IBM PC) is a program that lets people create dial-up information and conferencing services. First Edition can be used on PBX and PABX systems. The San Francisco CommuniTree Group provides an on-line newsletter and update service. "The FIG Tree," in Hayward, California, is a network for Forth programmers (415) 538-3580. First Edition is priced at $250.00. CommuniTree Group, 1150 Bryant Street, San Francisco, CA 94103 (415) 861-TREE. **Reader Service No. 139.**

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 201.**

Dr. Dobb's Journal

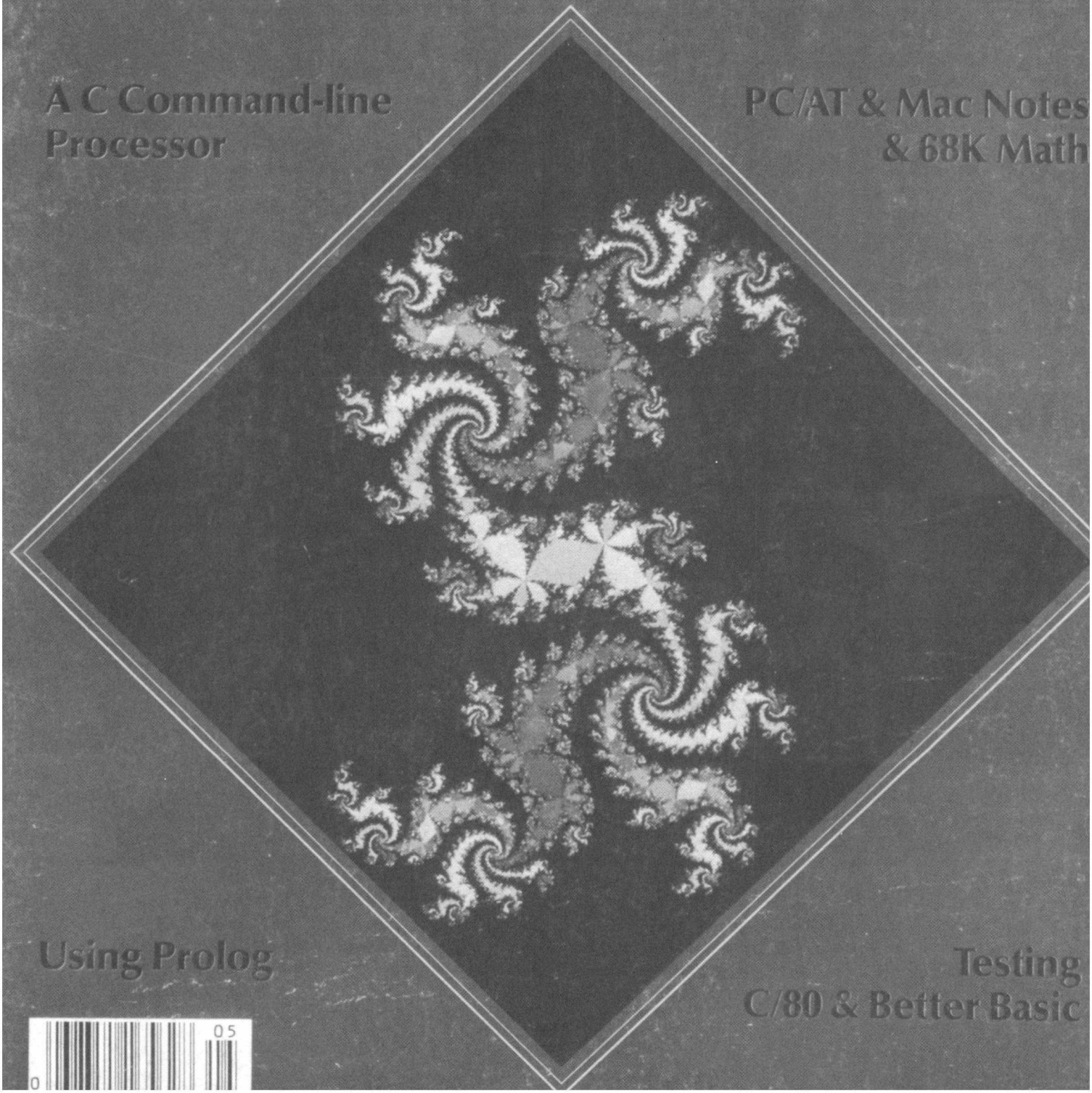
SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

#103 May 1985 $2.95 (3.50 Cana

Graphics Algorithms

A C Command-line Processor

PC/AT & Mac Notes & 68K Math



Using Prolog

Testing C/80 & Better Basic

0 05

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Randy Sutherland*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Rhoda Simmons*
Typesetter *Jean Aring*

Production
Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover *Benoit Mandelbrot*

Advertising
Advertising Director *Stephen Friedman*
Advertising Sales *Walter Andrzejewski, Shawn Horst, Beth Dudas, Michele Beaty*
DDJ Classifieds *Alice Abrams*
Advertising Coordinators *Lisa Boudreau, Jay Horvath*

Circulation
Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Snee*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Kathleen Boyd*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank. For subscription problems, call: outside of CA 1-800-321-3333; within CA 1-619-485-6535.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

May 1985
Volume 10, Issue 5

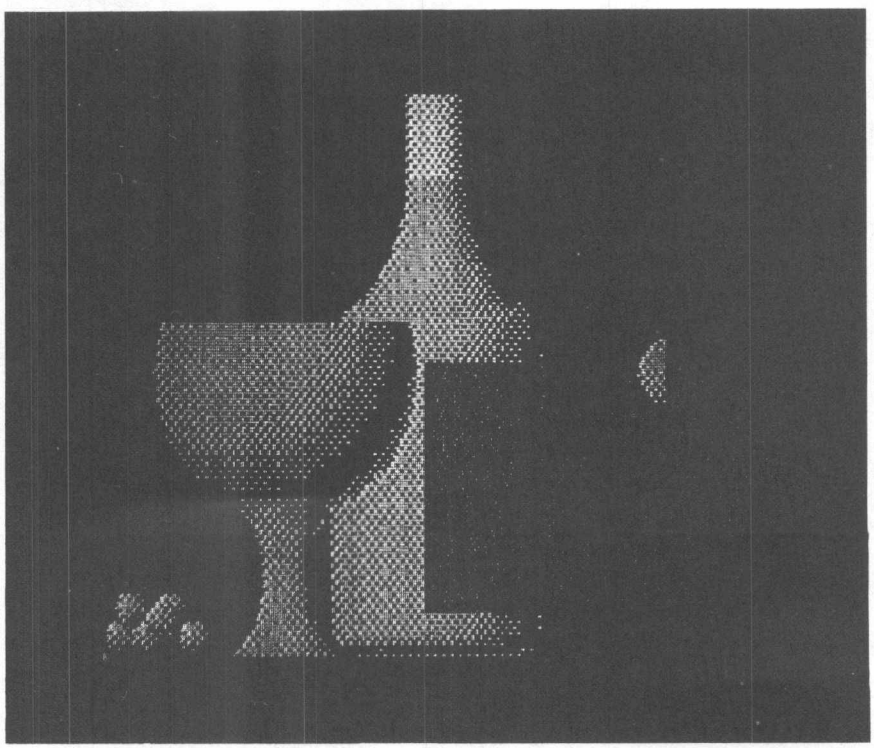
CONTENTS

Cover Artist

Our cover "Fractal Dragon," is the creation of Benoit Mandelbrot, Professor of the Practice of Mathematics at Harvard University. The most conspicuous characteristic of this dragon is that the eye and the imagination distinguish in it, in addition to the overall shape, an infinity of smaller features of every size down to the infinitesimal. For geometric shapes of this kind, Professor Mandelbrot coined the term "fractal." "Fractal Dragon" is a portion of "attractor" that occurs in a problem of dynamics. The cover illustration is a reproduction of Plate C5 of Mandelbrot's book *The Fractal Geometry of Nature* (Freeman 1982), drawn using computer programs by Mark R. Laff and V. Alan Norton. Copyright 1982 by B. B. Mandelbrot.

Referees Who Assisted With This Issue

Ted Carnevale, State University of New York at Stony Brook
Allen Holub, *DDJ* Contributing Editor
Robert Tripp, Editor-in-Chief of the late *MICRO*
James Woomer, Commodore-64 Guru and Gazebo Builder



See Richard Rylander's "Solid Shape Drawing on the Commodore 64" page 50.

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

Using Decision Variables in Graphics Primitives — 40
by Tom Hogan
An algorithm that uses the Decision Variable Method for plotting ellipses. **(Reader Ballot No.193).**

Solid Shape Drawing on the Commodore 64 — 50
by Richard Rylander
Some new approaches to computer-generated images on small systems. Combining elementary shapes, using the polygon mesh technique, creating shading effects. **(Reader Ballot No. 194).**

A Compiler Written in Prolog — 84
by G. A. Edgar
A compiler for VALGOL I written in Micro-PROLOG. **(Reader Ballot No. 195).**

COLUMNS

Dr. Dobb's Clinic — 16
by D. E. Cortesi
A wood blocks program for the Atari; keyboards with N-key rollover; DOS services and device drivers. **(Reader Ballot No. 190).**

Realizable Fantasies — 22
by D. E. Cortesi
The Resident Intern reflects on hackers, radicals, the microcomputer revolution and the role that *DDJ* will play in fostering that revolution. "Down with Cybercrud!" **Reader Ballot No. 191).**

C Chest — 28
by Allen Holub
A general purpose command line processor. **(Reader Ballot No. 192).**

16-Bit Software Toolbox — 118
by Ray Duncan
68000 square root routine; MacFeedback; MSDOS device drivers; Microsoft Assembler Bug of the Month. **(Reader Ballot No. 196)**

DEPARTMENTS

Editorial — 6

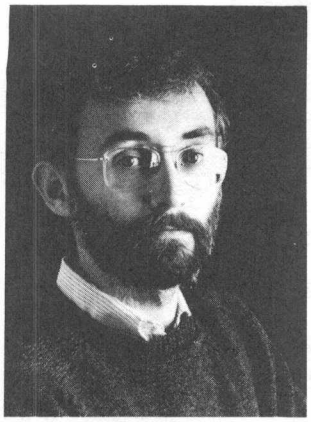
Letters — 8

Reviews — 98
Software Toolworks' C/80 and C/80 Mathpack; AMPRO's Little Board and Bookshelf Computer; New Generation Systems' Diskmaker I; Summit Software's BetterBASIC.

DDJ Classifieds — 97

Advertiser Index — 128

EDITORIAL



I want to comment on what others have been saying about two subjects germane to this or any issue of *DDJ*: graphics and copy-protected disks.

Frank Gaude regards the iconic interface of the Mac and its imitators as a regression to preliterate cave-scratchings. In his *Z-News* he says, "Long ago we used hieroglyphs to communicate. Then after a struggle an alphabet developed. Now we go back to pictures: icons. How many glyphs can we remember?"

In the macaphony of the current iconolatry this cautionary message is worthy of attention, but I suggest that it's not the whole story. The function of software is to realize metaphors. Icons are capable of doing that well but are no help in stepping outside existing metaphors to invent new ones; no help, that is, in developing new software. I would agree that icons are mnemonic for users and superfluous for programmers and let it go at that, but for the fact that I have recently seen a prototype of a system that takes icons a step further, a system in which actions can be assigned user-created visual tokens, and can be combined in ways that visually and kinaesthetically convey the form of the combination. Active, growing icons.

Furthermore, while the Mac icons are static tokens rather than elements of a true language, that is not true of all computer graphic aids to communication. French researchers are currently developing the means for transmitting sign language over telephone lines at commercial data rates. Their idea is to extract from a sequence of video frames just the essential components of a gesture and pass those along. This does not, I should make clear, mean matching the gesture to some template from a database of acceptable gestures; rather it means producing a real-time cartoon version of the gesturing party. The cartoons, subject to idiomatic and dialectical variations, are truly linguistic graphic elements, which is just what icons aren't.

As for copy-protected disks . . . .

". . . imagine yourself buying a new car. You have narrowed your choice to two cars, identical in all respects except that one will travel 100 mph and the other has a governor on it and cannot go past 55. Which car would you buy?"—Michael D. Brown, Central Point Software, producer of Copy II.

"People can rest assured that...we will always offer versions of our products that are not copy-protected."—Philippe Kahn, Borland International.

"Companies like Multimate and Borland have the right idea. Lotus, Ashton-Tate and MicroPro are the bad guys."—Edward Messerly, SF regional administrator of the GSA.

". . . a growing number of publishers are dropping traditional copy-protection methods, which they conclude have created ill-will and inconvenience for users . . . . Among the publishers who have abandoned copy-protection schemes is MicroPro . . . . Many software publishers, however, hope to see new protection methods . . . ."—Time/Life Access:Apple.

"These people are nutty."—Edward Messerly.

Yes. Something is nutty when a vendor can brag about also selling non-broken goods. It's really simple, isn't it? Vendors of software who want to stay vendors of software must find ways to profit from their efforts without breaking the tools they sell. The ability to copy files and disks is a *facility I bought* with my hardware and operating system. To try to sell me a program designed to subvert my system and hamper its operation is to try to enlist me in vandalism against my own property. Even granting that copy-crippling is a morally defensible policy, how could anyone believe that it was commercially viable in the long run? Nutty.

Michael Swaine



Anorexia Cured

Dear *DDJ*:
I read with some interest the article 'Fatten Your Mac' that appeared in your January [1985, #99] issue and forthwith did a fattening on my Macintosh. There are a number of points that I would like to add to the article, however. Firstly, a fine supplier of the 256K DRAM chips was omitted. I refer to Microprocessors Unlimited, of Beggs, Oklahoma, (918) 267-4961. I was able to phone them late Friday, and the chips they sent out on Monday arrived here (in Seattle) on Tuesday. I was quite pleased with the price and the service. They accept Visa, and they even sent a brief note explaining proper handling of the chips.

Secondly, it is *not* necessary, though it may be desirable, to destroy the 64K DRAM chips. I desoldered the original chips with an Ungar model 2000 desoldering tool, and was able to remove the solder, let the board cool, then pry the chips from the motherboard, all without clipping a single lead. This damaged two printed circuit traces (pulled them off in the prying process); as the article recommends, *there is a safer way to proceed*. My technique cannot be advised in the absence of an excellent desoldering tool, or for those not confident in their abilitites to make printed circuit repairs. The article had so fired my enthusiasm that I yanked out the 64K DRAMs days before the 256K DRAM circuits arrived. I was able to verify the repairs, and the removed memory chips, by installing sockets, plugging the 64K DRAMs into those sockets, and using the Macintosh. The nature of the lifted printed circuit traces (they showed copper color on the ICs to which the traces stuck) made detection easy, and no faults were missed. The board worked on the first try.

Thirdly, the multiplexer can be installed either piggybacked on another chip, or can be placed on a separate board, attached on the seven solder pads referred to in the article. Those seven pads include all logic signals and power required for the multiplexer. I installed wire-wrap pins in those pads, made a small board for the multiplexer from .100-inch hole-grid breadboard material, and pushed the small board over the wire-wrap pins, wire-wrapping the connections atop it to hold it in place. I found this less tricky than the piggyback soldering job would have been, and no leads of the multiplexer needed to be bent.

Fourthly, the multiplexer recommended is 74F253 or 74AS253; while these will do quite well, so will the 74F153 or 74AS153; which are slightly less expensive; the only difference is the nature of the DISABLE function, which (in this application) is always unused. All pin connections are identical to the '253.

I found that the multiple-layer board, having extra copper in the interior layers, was rather more difficult than most to desolder; I had to apply more heat (use a higher temperature on the desoldering iron) than usual, to complete the solder removal within a safe time. I did not notice any fragility of the printed circuit (no more than most, at any rate), and my procedure was, as I have mentioned, rather stressful. At one point, I fumbled, and dropped the board onto the concrete floor—there was no damage.

Prices were better than I expected, about $220 total cost for the chips, and that figure is dropping almost daily. Time required was almost exactly four hours. Since Apple charges about six hundred dollars more for the 512K Mac, it hardly pays to buy one; the fine folks in Cupertino are willing, in effect, to pay me handsomely for those hours, if I convert my 128K instead.

Instead of 'Macintosh' and 'Fat Mac', I will be using the terms 'Macintosh' and 'Anorexic Mac' in the future.

Sincerely grateful,
John Whitmore
FM-15 Physics Dept.,
University of Washington
Seattle, WA 98195

Going PUBlic

Dear *DDJ*:
PUBlic files have been used successfully on a wide variety of CP/M computers since the publication of our November 1984 [*DDJ*, #97] article "CP/M 2.2 goes PUBlic." However, we want again to caution readers that PUBPATCH must be installed on an *unmodified* CP/M 2.2 BDOS!

Several users have discovered that their Heath Co. CP/M 2.2.03 has been altered with a patch at 06E1 and 0DEE–0DF3 (relative to the start of the BDOS). The Heath patch causes the BDOS to stop building the disk group-allocation vector when it encounters the first directory entry having deleted data (E5) codes in both bytes 0 and 1. If PUBPATCH is then installed in this BDOS, the symptoms will be files that disappear from the end of the disk directory!

Heath users with this problem should patch location 06E2h back to its original value '06D2h' (relative) before installing PUBPATCH.

RELATIVE ADDRESS	06E1'
ORIGINAL 2.2 BDOS	JZ 06D2'
HEATH 2.2.03 BDOS	JZ 0EEE'

Two readers have reported problems with PUBLIC.ASM, used to set or reset the PUBlic file attribute bit. Updated code can be found in [Listing One (page 13)]. The first fix corrects a bug in v. 1.0 that prevents PUBLIC from setting the attribute for a very large file. The second takes care of a bug in the standard CCP (but not in ZCPR), which fails to restore the default dma address before continuing to execute a SUBMIT file.

No changes are required to PUBPATCH itself.

Sincerely,
Bridger Mitchell, Derek McKay
Plu*Perfect Systems
Box 1494
Idyllwild, CA 92349

Turbo Bug

Dear *DDJ*:
I have been using Borland International's excellent Turbo Pascal compiler for several months now (I started with version 1) and I think it is by far the best implementation of any language I have every seen. The quick compiler and built-in editor have reduced the pain of correcting typos far enough to let me appreciate the virtues of Pascal. In the process I have discovered two things about the language that may be of interest to others. I am including listings of three short demo programs.

First, the "initialized constants" mentioned briefly in both the reference manual and the Turbo Tutor manual are really initialized static variables. This is clear from a close reading of either manual, but it is never stated in so many words, nor do any of the example programs demonstrate it. Compiling and running the [program in Listing Two (page 13)] will demonstrate that the typed constant I in the procedure ShowStatic behaves exactly like a static variable in C.

Compiling and running the program [in Listing Three (page 14)] will demonstrate a problem I discovered while trying to process the files produced by a linker cross reference utility. The Trunc and Round functions in both the MSDOS and CP/M versions of Turbo Pascal produce a run time error if you attempt to convert a real value of −32768 (8000H). Since this is a legal integer value, and right in the middle of the range of hex numbers, I think this has to be called a bug.

Listing Four (page 14) shows one way of programming around this problem. It works just fine when using real numbers and integers to represent hex numbers between 0 and FFFFH, but it would not be very useful in an application that requires both positive and negative real numbers.

Sincerely,
William Ames
8720 Topanga Cyn. Blvd. #8
Canoga Park, CA 91304

Borland assures us that this bug will be fixed in Version 3.0 of Turbo Pascal.—Ed.

Fast Hartley Transform

Dear *DDJ*,
We were pleased to see Ron Ullmans's letter in the December 1984, issue of *Dr.Dobb's Journal* regarding the Hartley Transform and the accompanying listing of a software program. We, too, consider its advantage to be great.

Your readers may wish to know Stanford University has proprietary rights on this technology, developed by Professor Ronald N. Bracewell. Patent rights are pending.

Use of the software without a license may be an infringement of those proprietary rights. However, a license under reasonable terms to the technology is available from Stanford.

Incidentally, we have called this development the "Bracewell" transform, in recognition of Professor Bracewell's fundamental contribution.

Sincerely,
Lisa Kuuttila
Consulting Associate
Office of Technology Licensing
Stanford University

DDJ

Letters (Text begins on page 8)

Listing One

```
; UPDATE PATCHES FOR PUBLIC.ASM

vers    equ     1$2             ;update PUBLIC.ASM version number

; fix 1.  Insert the following code in the 'SAMEXT' routine
;         immediately preceeding its 'ret' instruction.

        rnz             ; v 1.1
        inx     h       ; extent is 0, check overflow (s2) ext.
        inx     h
        mov     a,m
        ani     7Fh
;
        ret             ; end of SAMEXT routine


; fix 2.  Insert the following macro statement at the label 'xit:'
;

xit:    dobdos  dmafn,80h       ;restore default dma for SUBMIT under ccp.
                                ; v 1.2
        lspd    ustack
        ret
```

End Listing One

Listing Two

```
TURBO PASCAL Program Lister, Copyright 1983 Borland International
Listing of: STATIC.PAS


program TestStaticVariables;
{ This program demonatrates that the "typed constants" of Turbo Pascal }
{ are really static variables. Each time the main loop calls ShowStatic }
{ the typed constant I is incremented by 1. When ShowStatic is called  }
{ again the incremented value is written to the CRT. }
var
  Ch : Char;
  I  : integer;

procedure ShowStatic;
{ This prints and then increments the "constant" I }
const
  I : integer = 0;

begin
  Writeln(I);
  I := I + 1;
  end;

function UseHeap(L : integer) : integer;
{ This is included to demonstrate that the typed constant above is not }
{ distroyed when another procedure uses the heap or stack. }
var
  I : integer;
  J : ^integer;

begin
  I := -L;
  new(J);
  J^ :=  L;
  L := L + 1;
  if L < 5 then
    L := UseHeap(L);
  Dispose(J);
  UseHeap := L
  end;
```

(Continued on next page)

Listing Two

```
begin
  repeat
    read(Kbd, Ch);
    I := UseHeap(0);
    ShowStatic;
    Until Ch = #13;
  end.
```

End Listing Two

Listing Three

```
TURBO PASCAL Program Lister, Copyright 1983 Borland International
Listing of: CNVRT1.PAS


program RealToInteger1;
{ This program can be used to demonstrate a problem in }
{ converting real numbers to integers. }
var
  Nmbr : real;
  I    : integer;

begin
  Readln(Nmbr);
   { if Nmbr = 32768.0 the following produces a runtime error }
    if (Nmbr ) 65535.0) or (Nmbr ( 0) then
    { I am only interested in the range of 4 digit Hex numbers }
      Writeln(' overflow error')
    else begin
      if Nmbr ( 32768.0 then
        I := Trunc(Nmbr)
      else
        I := Trunc(Nmbr - 65536.0);
      Writeln(I);
      end;
    end.
```

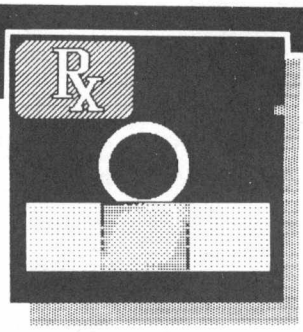
End Listing Three

Listing Four

```
TURBO PASCAL Program Lister, Copyright 1983 Borland International
Listing of: CNVRT2.PAS

program RealToInteger2;
{ this shows one way around the bug demostrated in version 1 }
var
  Nmbr : real;
  I    : integer;

begin
  Readln(Nmbr);
   { 32768.0 is converted correctly by the following code }
    if (Nmbr ) 65535.0) or (Nmbr ( 0) then
    { I am only interested in the range of 4 digit Hex numbers }
      Writeln(' overflow error')
    else begin
      if Nmbr ( 32768.0 then
        I := Trunc(Nmbr)
      else
        begin
          I := Trunc(Nmbr - 65535.0);
          I := I - 1;
        end;
      Writeln(I);
      end;
    end.
```

End Listing Four

DR. DOBB'S CLINIC



by D. E. Cortesi

Welcome, *Micro*ans

It seems that *DDJ* has picked up a batch of new readers, the erstwhile subscribers to the late, lamented *Micro*. Let's recap the purpose of this column for their benefit. It's supposed to be "a place for the display of techniques and discoveries." That's what we said the first time it appeared, in May 1981. Also, "We'd like to tell of unobvious uses for standard utilities. We want to uncover errors in documentation, to warn people away from pitfalls, and to show off those '*Eureka!*' moments that make systems work rewarding."

For a long time, most of the contributions to the Clinic involved CP/M and the Z80. Nowadays, we hear more from MSDOS/8086 users. Well, stand back, Zilog lovers; make room, Intel-lects! Now we have among us people who understand *real* computers like the 6502, the 6809, and the 68000 and *real* operating systems like OS9, SOS, and PRODOS. We can barely wait for the neat contributions they'll be sending in. Soon.

Wood Blocks

As bait for the *Micro* readers (hey, folks, we really like reader contributions here), we present our wood blocks program for the Atari. It's a little thing (see Listing at right) that we ran up one day while experimenting with the start, select, and option keys. These three keys control the three lowest bits of a byte in the Atari address space. The program samples that byte, and each time the combination of operated keys changes, it emits a heavily damped *klonk* at a different pitch. Play it by mashing down all three keys and then sort of working your fingers around.

The simulation of damped tones is pretty good. Experiment with the damping factor and with the expression that generates a note value from the key combination.

Roll Over, Keyboard

We've an interesting letter here from Charles Davis, who lives in a town with the delightful name of Flower Mound, Texas. Besides submitting some throughput timings for an Atari system (at last!), he's been studying keyboards:

"Perhaps you could survey another area of concern. I am either a very good typist or a very sloppy one! I need a keyboard with what used to be called 'N-key rollover.' These are hard to find. The current generation of computers and terminals does not seem to supply this feature. Most marketing and sales people . . . are unaware of the feature and therefore the sales literature does not mention it, even when it is present.

"A survey of the popular computers and terminals with reference to rollover would be interesting to me and, I think, to many of your readers. My procedure for checking rollover is to depress several keys in sequence, holding each until all are depressed, then I lift them all in the same order. I observe what happens on the screen. There seem to be three types of response:

- the first N keys (type 1)
- the last N keys (type 2)
- only the first and last keys (type 3)

The major variable is how big N is. Rarely must I find a helper with more fingers. One terminal manufacturer admitted that their terminals had N-key rollover and that N was supposed to be 5, but some key sequences did not work right so they would only guarantee N to be 3. This points up the weakness of my procedure: I do not do an exhaustive check and some problem characters may be missed. Without knowledge of the key matrix and the scanning algorithm, it would take too long to find the problems."

Davis sent along the results from some terminals he'd tested (see Table 1 on page 17). However, our first at-

```
10 REM WOODBLOX FOR ATARI, D.E. CORTESI
20 KEYBYTE=53279
  : DAMPING=0.29
30 SAMPLE=PEEK(KEYBYTE)
  : IF SAMPLE=BUTTONS THEN GOTO 30
40 BUTTONS=SAMPLE
  : PITCH=12*(8-SAMPLE)
  : VOLUME=15
50 SOUND 0,PITCH,12,VOLUME
  : VOLUME=VOLUME-VOLUME*DAMPING
  : IF VOLUME >= 1 THEN GOTO 50
60 SOUND 0,0,0,0
  : GOTO 30
```

Terminal	Response	Rollover
ADDS Regent 20	type 1	N > 10
ADDS Viewpoint	type 1	N > 10
Appollo DN600 Workstation	type 1	N > 10
Atari computers	type 3	n.a.
CIT 101E	type 1	N = 2
Daisy Logician Workstation	type 1	N > 10
Hazeltine 1410	type 3	n.a.
Heath/Zenith 19 and 29	type 4	N = 2
HP-85	type 3	n.a.
HP-9816	type 1	N > 10
IBM PC/XT	type 1	N > 10
Perkin-Elmer Bantam 550	type 1	N = 3
Soroc IQ-120	type 3	n.a.
Televideo 950	type 1	N = 2
Wyse 50	type 1	N = 3

Table 1

Charles Davis' survey of the number of keys that can be "rolled over" on different terminals. Types are: 1, first N keys; 2, last N keys; 3, first and last only; 4, first N keys then garbage.

Command Code	Type of Driver	Operation
0	either	initialize
1	block	media (sic) check
2	block	build Bios Parameter Block
3	either	IOCTL input call
4	either	input
5	character	nondestructive input no wait
6	character	input status check
7	character	input flush
8	either	output
9	either	output with verify
10	character	output status
11	character	output flush
12	either	IOCTL output call
13	either	notification of file-open
14	either	notification of file-close
15	either	IOCTL query: removable disk?

Table 3

The command codes to which an MSDOS device driver must respond, omitted from IBM's DOS *Technical Reference* manuals. Codes 13, 14 and 15 are new with PCDOS 3.0.–Codes 3, 12 and 15 are passed only to drivers that indicate IOCTL support in their Device Header Attribute word.

tempt to test a keyboard produced a new result that you should watch out for:

- the first N keys plus a garbage key (type 4)

This is the H19 problem we reported here in August 1982; it results from a lack of blocking diodes in the key matrix. Heath hasn't learned anything since, either. On our Heath/Zenith 29, holding L, I, and then S produces "LIQS" on the screen. We have to be very precise typing the word "list" or it comes out "liqst." Worse, some combinations produce false function keys. The combination E, I, then P generates a false down-arrow (escape-B) sequence, for instance. So the H/Z29 is an N=2 machine, even though for some key combinations it will pass Davis' test as a type 1 with much higher N. Incidentally, the good combinations occur on the home row. That suggests that you might apply Davis' test on the top or bottom rows.

Charting DOS

We've been boning up on MSDOS and PCDOS lately, in preparation for writing a book and we've compiled the chart of DOS services that appears in Table 2 (page 19) as a study aid. For each possible Int 21 service it shows the request number in decimal and hex, the releases that support it, and a capsule description of the service. The services are organized into groups by function. The groups and the services within the groups are generally in ascending order by request number, but functional relationships are given precedence.

The idea is that you can look for the service you need by scanning a small group of related ones. The capsule descriptions are just enough to differentiate between similar functions. Having found the service you need, you can access its detailed description in your DOS manual through its request number.

Our study of PCDOS has been impeded no small amount by the poor quality of the IBM DOS manuals, especially those for DOS 3.0. For one example, the documentation on service 56/38h (get or set country-dependent information) is so badly done that we honestly cannot make sense of it. If you are a reader who can write a comprehensible, accurate description of the DOS 3 version of service 38h, please do so and send it to the Clinic.

For another example, the chapter on device drivers omits any specification of the command codes to which a device driver is supposed to respond. In the DOS 2.1 manual you could still cull the essential information from the comments of a sample program. In DOS 3.0, the sample program has been dropped from the manual. The book says just enough that you can infer that command codes 3, 4, 8, 9, and 12 specify input or output, but there is no way you can tell which code calls for what action!

Fortunately, the missing information can be found in Microsoft's version of the technical manual, from which we constructed Table 3 (page 17). Although some command codes are specific to character or block devices, it seems wisest for all drivers to expect all command codes. They can treat the unwanted ones as null operations, or they can reject them with the "unknown command" error code.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

Service number in decimal and hex	Appears in which releases	Functional group	Capsule description
0/00	1/2/3	program cntrl	end program (requires PSP *CS)
49/31	−/2/3	program cntrl	end, stay resident, AL=ret, DX=size
76/4C	−/2/3	program cntrl	end program, AL=ret
38/26	1/2/3	program cntrl	make daughter PSP at (DX:00)
75/4B	−/2/3	program cntrl	(AL=0) load, run daughter from path *DS:DX
77/4D	−/2/3	program cntrl	get AH=term code, AL=ret of daughter
75/4B	−/2/3	program cntrl	(AL=3) load DS:DX —>path as overlay
98/62	−/−/3	program cntrl	get BX=paraddr of my PSP
48/30	−/2/3	system info	get DOS version number
42/2A	1/+/3	system info	get CX=year, DH=mo, DL=day (2,3:AL=dow)
44/2C	1/2/3	system info	get time as CH:CL:DH.DL
51/33	−/2/3	system info	(AL=00) get Break switch in DL
53/35	−/2/3	system info	get int(AL) vector to ES+BX
56/38	−/2/+	system info	(AL=00) get country-dependent info
72/48	−/2/3	system info	(BX=FFFF) get size available RAM to BX
89/59	−/−/3	system info	get extended error to AX, BH, BL, CH
37/25	1/2/3	system control	set int(AL) vector to (DS+DX)
43/2B	1/2/3	system control	set date from CX, DX
45/2D	1/2/3	system control	set time from CH:CL:DH.DL
51/33	−/2/3	system control	(AL=01) set Break switch from DL.0
56/38	−/−/3	system control	(??) set country-dependent info
72/48	−/2/3	system control	allocate BX paragraphs
73/49	−/2/3	system control	return RAM from ES=paraddr
74/4A	−/2/3	system control	resize ES=paraddr to BX paras
1/01	1/2/3	kbd/handle 0	get key: wait, echo, break-check
6/06	1/2/3	kbd/handle 0	(DL=FF) get key: noecho, nowait, nobreak
7/07	1/2/3	kbd/handle 0	get key: wait, noecho, nobreak
8/08	1/2/3	kbd/handle 0	get key: wait, noecho, break
10/0A	1/2/3	kbd/handle 0	buffered line input with editing
11/0B	1/2/3	kbd/handle 0	test if key available (break)
12/0C	1/2/3	kbd/handle 0	flush buffered keys, then do (AL)
2/02	1/2/3	scrn/handle 1	put DL with editing and break-check
6/06	1/2/3	scrn/handle 1	(DL < > FF) put DL, noedit, nobreak
9/09	1/2/3	scrn/handle 1	iterate DOA 2 over *DS:DX to $
3/03	1/2/3	aux/handle 3	get byte from COM1
4/04	1/2/3	aux/handle 3	put byte to COM1
5/05	1/2/3	prt/handle 4	put byte to LPT1
25/19	1/2/3	disk info	get AL=default drive
47/2F	−/2/3	disk info	get Disk Transfer Address to ES:BX
27/1B	1/+/3	disk info	get FAT *DS:BX (2,3: only 1st byte of FAT) . .
28/1C	−/2/3	disk info	. . and CX=bytes/sec AL=sec/unit DX=units/disk
54/36	−/2/3	disk info	BX=free units, CX/DX/AL as for 1C
84/54	−/2/3	disk info	get write-verify switch to AL.0

Table 2
Chart of DOS Services

(Continued on next page)

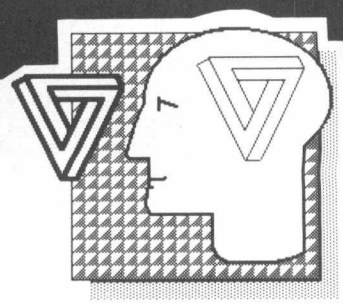
(Continued)

13/0D	1/2/3	disk control	reset disk system
14/0E	1/2/3	disk control	select drive, return count of drives
26/1A	1/2/3	disk control	set Disk Transfer Address = DS:DX
46/2E	−/2/3	disk control	set write-verify switch from AL.0
15/0F	1/2/3	fcb control	open file from FCB
16/10	1/2/3	fcb control	close file from FCB
17/11	1/2/3	fcb control	scan current directory for first match
18/12	1/2/3	fcb control	scan current directory for next match
19/13	1/2/3	fcb control	erase files matching FCB
22/16	1/2/3	fcb control	make new file from FCB
23/17	1/2/3	fcb control	rename file from FCB
35/23	1/2/3	fcb control	get size of file to fcb
36/24	1/2/3	fcb control	set relrecno from current file position
40/28	1/2/3	fcb control	(CX = 0) trunc or stretch file to size
41/29	1/+/3	fcb control	parse filespec *DS:SI into fcb *ES:DI
20/14	1/2/3	fcb i/o	read next record to DTA
21/15	1/2/3	fcb i/o	write DTA to next record
33/21	1/2/3	fcb i/o	read record at direct address
34/22	1/2/3	fcb i/o	write record to direct address
39/27	1/2/3	fcb i/o	block read CX records to DTA
40/28	1/2/3	fcb i/o	(CX > 0) block write CX records from DTA
57/39	−/2/3	directories	make directory for path *DS:DX
58/3A	−/2/3	directories	erase directory at end of path *DS:DX
59/3B	−/2/3	directories	set current path as path *DS:DX
71/47	−/2/3	directories	get current path of DL=drv to DS:DI
63/3F	−/2/3	file i/o	read CX bytes from BX=handle to DS:DX
64/40	−/2/3	file i/o	write CX bytes to BX=handle from DS:DX
60/3C	−/2/3	file control	create/trunc file (path *DS:DX, CX=attr)
91/5B	−/−/3	file control	create new or fail (path *DS:DX, CX=attr)
90/5A	−/−/3	file control	create unique file (path *DS:DX, CX=attr)
61/3D	−/2/+	file control	open file (path *DS:DX, AL= access type)
62/3E	−/2/3	file control	close file (BX=handle)
65/41	−/2/3	file control	erase file (path *DS:DX)
66/42	−/2/3	file control	seek (BX=handle, AL=method, CX+DX=offset)
67/43	−/2/3	file control	(AL=0) get CX=attr of file path *DS:DX
67/43	−/2/3	file control	(AL=1) set attr of file path *DS:DX to CX
69/45	−/2/3	file control	create a duplicate file handle
70/46	−/2/3	file control	force a handle to be a duplicate
78/4E	−/2/3	file control	search 1st match to path *DS:DX
79/4F	−/2/3	file control	search next match to path *DS:DX
86/56	−/2/3	file control	rename/move path *DS:DX to path *ES:DI
87/57	−/2/3	file control	(AL=0) get CX=time, DX=date of path *DS:DX
87/57	−/2/3	file control	(AL=1) set CX=time, DX=date of path *DS:DX
92/5C	−/−/3	file control	(AL=0) at CX+DX in BX=hdl, lock SI+DI bytes
92/5C	−/−/3	file control	(AL=1) release bytes locked by prior call

Table 2
Chart of DOS Services

REALIZABLE FANTASIES

Dr. Dobb is a Subversive



by D. E. Cortesi

The publishers of DDJ *recently threw a party to celebrate the 100th issue of the magazine. It was a friendly affair, with lots of sincere speeches about how important* DDJ *has been and about how it still has a great role to play. There was a lot of talk about "hackers:" what they accomplished; where* DDJ *stood in relation to them; where they might have gone now.*
they were expressed, the sentiments seemed incomplete to me. Something was being overlooked, but I couldn't figure out what it was until I was halfway home. Then I lay awake working out the speech that ought to have been given, but wasn't. And this is it . . . .

We've heard a lot about "hackers" in the last few minutes. I would like to protest the use of that word. I understand why it's being used. It's because nobody can think of the right word to describe the people who founded *DDJ* and the people who read and profited from it in its earliest days. We know they weren't part of the establishment; we know they didn't play a conventional role in the society of their day. We have, perhaps, an uneasy feeling that we aren't quite like them and that they wouldn't find us very interesting—even though perhaps we *were* them just a few years ago.

So we scratch around in our vocabularies, searching for the right word to describe these departed people, and the best we can come up with is "hacker." Well, it's the wrong word.

The right word is . . . *revolutionary. Subversive. Bomb-throwing anarchist.*

To me, the hacker state is a frame of mind in which one is turned inward on oneself and the machine: one's spiritual third eye rolls up into the head. The people who birthed *DDJ* and personal computing in general were quite the opposite. Their eyes were turned outward on society—and twinkled with mischief.

They were bent, quite consciously, on overthrowing the established order. It was to be a nonviolent overthrow, one conducted if possible with whimsy and good humor, but it was to be an overthrow, a toppling, nonetheless.

The central tenet of this jolly revolution was (and still is) the belief that *if we put enough computers in the hands of enough people, wonderful changes will take place.* What the wonderful changes are . . . well, that varies with the year and the person you're talking to. But that they will be wonderful, as well as radical in the deepest and best sense of the word, no computer subversive doubts. I certainly don't.

The founder of the movement and its only theorist, its Marx or Che Guevara if you like, is Ted Nelson. His book, *Computer Lib*, was written in reaction to computers, their employers, and their professional priesthood as those things existed in the early 70s. It was a protest: "Either computer systems are going to go on inconveniencing our lives, or they are going to be turned around to make life better," and then, in caps, a battle cry: "COMPUTER POWER TO THE PEOPLE! DOWN WITH CYBERCRUD!"

And to make sure we didn't miss the point, there was the symbol of the clenched fist on the cover. I hope I'm not the only one here who remembers when the word *Lib* and the upraised fist were potent, dangerous symbols. That Nelson borrowed both for his manifesto indicates how serious he was.

I don't know if the People's Computer Company admitted to subversive intentions in public or not. Unfortunately for me, I was out of the country during its glory days. I was aware of micros and an early subscriber to *DDJ*, but I was only a fellow traveler to the revolution, not a revolutionary myself.

But whether the intentions were admitted or not, I'm positive that they were there. No one who's spent any time with Bob Albrecht, or with Ramon Zamorra, who founded ComputerTown USA!, could have any doubt that these people are motivated by a deep love of mischief. Nothing pleases them better than a well-toppled apple cart or a well-rocked boat.

Another revolution that I missed out on was the birth of the acid culture in the early 60s. But I suspect that the spirit of People's Computer Company had a lot in common with the spirit of Ken Kesey's Merry Pranksters, and with that of the Diggers and the various communards of the 60s. They all felt that they'd stolen the fire of the gods and that they'd damn well better set as many blazes as they could before the gods came to take it back.

After all, what could possibly be more subversive than training children to operate the tools of the establishment, *without* at the same time inculcating the establishment's line of thinking? But that was what Albrecht, Allison, *et al.*, were doing. I don't know if this is what was specifically in their minds or not. But consider. You have all had the following experience. You program something in BASIC; you find that the program doesn't work because of the inevitable bugs; then, after you pick them off, you find that it *still* doesn't work because the limited accuracy of binary floating point has turned your nu-

merical results into oatmeal. Nobody who has been through that can ever look at their bank statement in quite the same way again. Or their phone bill. Or, if there is a live connection between one side of their brain and the other, can they ever feel quite the same about a proposal for an antiballistic missile system as they might once have done before their personal encounter with computers in the raw.

Albrecht and company used timesharing terminals to start indoctrinating children and susceptible adults, but then the micro came along. The earliest micros were pitiful, but to the eyes of a computer subversive they appeared to be the sword of liberation. A computer you could hold in your hand!

For a parallel, try to imagine that Gutenberg had never invented his printing press; that the only way for books to be reproduced was for them to be handwritten by scribes. Of course, only the wealthy and the government can afford to pay scribes and buy vellum, so the only thoughts that circulate in book form are the ones acceptable to the powers that be. Then somebody invents the typewriter: a cheaper, faster way of being a scribe, one that anybody can learn. So you start a counterculture book factory, teaching people to use typewriters to duplicate unapproved literary works.

And then somebody walks in the door with the latest invention, *the Xerox copier*. And just that simply, your revolution has been won; now it's just a matter of mopping up.

In the case of the computer revolution, the mopping up is still going on. The micro *has*, in fact, hurt the computer bureaucrats, the people who would try to make you swallow an inconvenience on the grounds that the computer wouldn't let them be helpful. These days they find it a lot harder to say "the computer won't let me" because so many clients are likely to answer "Why not? My computer would." There really is a lot less these days of what Nelson called cybercrud, and we can credit that to the wide popular exposure to micros.

But there is a lot more to be done. We've spread computers all over the place and still have no millenium, nor anything remotely like it. There's been no great liberation of minds or spirits. The revolution has been assimilated, just as the automotive, sexual, drug, and free speech revolutions were assimilated before it, and the feminist revolution is being assimilated now: they come, they pass, things are different but not very much so. Surely by now the wonderful thing should have happened. Surely by now somebody should have written the program that will set us all free? But no.

One reason for this failure is that the computer did not turn out to be a sword that could be placed in anyone's hand. It isn't a sword at all, nor anything like a simple edged or pointed weapon. For complexity it's more like a whole battalion of soldiers. If you tell some poor shmuck to start wielding one, you put him in the position of a private promoted instantly to general and asked to take over management of a battle.

To change metaphors nimbly in midstream, we have made the belated discovery that the computer priesthood that we deposed was not entirely a group of fakes; they had more than mumbo jumbo going for them. True, they'd made public access to the machines difficult, or allowed it to be difficult. But now that we have our own machines, we find out that there were good reasons for that. It turns out that quite possibly the most difficult thing you can do with a computer is to make it be truly helpful. If you are in a hurry for an answer, you will find it a lot easier to program the machine to take its input in rigid, unforgiving formats, and to give cryptic, insulting error messages, than to make it flexible and friendly and forgiving.

In fact, we find that in putting on its happy face, the machine uses up so much capacity that it has precious little left in which to do any useful work.

Partly this is a real effect. It truly is difficult to make a computer be helpful, and there are deep, unsolved problems in the way of making that happen. But in part, I think that the revolutionaries were a bit hasty in exiling those priests. They have a genuine science, computer science, that deals with the elegant and efficient application of machine power. All too little of that science is being applied today. The result is that we have machines with 512K of storage that can't keep up with the demands of a single user. Damn it, we used to support twenty APL users on a single 360 model 40 with 128K; we considered a 370/158 with 512K a big machine, suitable for use by a middle-sized company with half a dozen programmers online to TSO or VM/370. The *big* macro assembler for the IBM 370 needs a 128K partition to run in. That's the assembler program plus all the tables and work areas and buffers it uses. The macro assembler for the IBM PC can barely load *itself* into 128K, and it can't assemble anything of useful size until you give it 256K. In short, there has been a major loss of efficiency in moving from the software that the priests built for the corporate mainframes to the software that we are putting on personal computers today.

As a good revolutionary, I must believe that computer *use* is for everybody, and I really do. But programming the computers so they can *be used* is almost certainly a job for specialists.

The question that confronts us now is: whose specialists are they going to be? Will they be the tools of the new computing establishment, working for the fiscal advantage of a company like IBM or AT&T or Microsoft, or for the political advantage of a nation like Japan or France?

Or will they be grass-roots benefactors like so many who supplied the software we have now—and who often supplied it through the pages of *DDJ*? I would like it to be the latter. In fact, I propose to you that what *DDJ* has always done is to support and train and encourage our grass-roots systems programmers. And that is still the best, and most truly radical, thing that it can do in the future. I'm delighted to see that its editors agree with me. Proof that they do is in their publishing Richard Stallman's "GNU Manifesto" (*DDJ* #101, March 1985), an outrageous, revolutionary challenge quite in the old vein.

My own best writing for *DDJ* has been inspired by a species of the Robin Hood ethic: a desire to steal ideas from the well-funded and put them into the heads of the unfunded. I enjoyed writing my recent article on Prolog for just that reason. Prolog and the ideas behind it had been the property of a few academics, and these ideas were rapidly being co-opted by the Japanese for their Fifth Generation. But *DDJ* gave me a chance to snatch those same notions and spread them around everywhere, making them accessible to everybody who can grasp them. Of course the real revolutionaries are the people in London who publish Micro-PROLOG and the ones in this country who are selling a version of DEC-10 Prolog for the IBM PC. The ideas wouldn't be much use without the cheap software that implements them, and that wouldn't be of any use without the cheap micros to run it on.

So these processes all feed back on themselves and get richer and deeper. But the continuation of this revolution, at least its continuation *as* a revolution and not as a captive toy of the corporations and the advertisers, depends on getting ever more sophisticated ideas and tools into the hands of creative people, so that they can keep putting ever more sophisticated programs back into ours.

Notice how the ante goes up on every round. The level of sophistication is escalated every time through the loop. The Mac's internal toolkit, and its competitive followers Windows and GEM, are vastly harder to use well than the old CP/M BDOS functions. Graphics, AI, and speech recognition, in their different ways, require the exercise of all the algorithmic tools of computer science. And it takes very abstruse, very mathematical tools to analyze these massive, sophisticated programs and prevent them from overflowing the capacities of personal hardware, which really are quite limited. The experience of mainframers has been that your hardware is *always* an order of magnitude too small for the software you'd like to run. There's no indication that this law has been repealed for micros. We will be squeezing quarts into pint bottles forever, and it takes sound theoretical insight to do that well.

I should add that the rule that the capacity of the hardware always falls short of the needs of the software just *might* be repealed by highly parallel, multiple-micro systems. Two things about such systems: first, they are right on the fringe of computer science and the academics haven't the foggiest idea how to build good software for them; and second, they are ideally suited for hobbyist-level, garage experiments.

But this is why the face and contents of *DDJ* have been changing. Whether the editors and contributors knew it or not, they've been trying to prepare themselves to discuss and understand computers at the more sophisticated level permitted by the second generation of micros. But there's been no change in who is doing the discussing, nor in their essentially subversive motive, which is to snatch the tools of the establishment and apply them in the public domain.

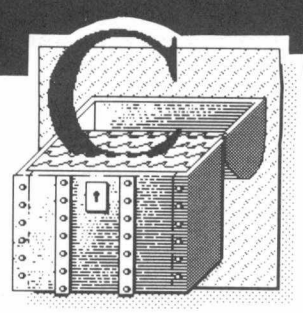
It's that mischievous, subversive motive that has kept me writing for *DDJ*. That, and the belief I still hold that, somewhere out there, fingers on the keyboard of an Apple or Osborne, is a fifteen-year-old kid who is capable of writing the program that will turn the world upside down. The proper role of *DDJ* is to make sure that that kid has free, unfettered access to the ideas and the software tools that she needs in order to write it.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 191**.

C CHEST

Command Line Processing



by Allen Holub

A command line switch is an argument usually preceded by a '−' that modifies the way in which a program works. Just about every C program I write uses command line switches, and all these programs have to process these switches one way or another. For a long time, I wrote a different little subroutine for each program, modifying the routine to deal with the idiosyncrasies of a particular program. After writing the same subroutine about a million times (and making the same off-by-one error with argc every time), I thought that there *had* to be a better way. All my processing routines were essentially the same in structure; only the names of the command line switches were different. A general purpose command line processor was clearly what I needed.

Looking around in the literature for something that did the job, I found the "Argum" package, which appeared in Anthony Skjellum's C/Unix column in *DDJ* (#70, August 1982). The program was well written and did what I needed, but it had several problems. First of all, it did a lot more than I needed and the extra functions added extra code. Because this package was going to be included in every program I wrote, a reduction in scope seemed to be a good idea. Another, related, problem was the internal tables used by Argum. They were created at runtime (as compared to compile time). This added both extra code and extra execution time to any program that used Argum. Finally, Argum had no convenient way to deal with errors on the command line. In view of these problems, I decided to write my own processor, a description of which follows. This is by far the most frequently used subroutine in my standard library.

Getargs( )

The command line processor is actually a package of subroutines. Access to these routines is through a single procedure, called getargs( ). The routines are table driven. In every program you have to declare a table in which the various command line switches are described. Getargs( ) removes the switches from the command line as it works. This means that position-sensitive arguments (e.g., a switch that applies to all files that follow it on the command line, but not to any files that precede it) are not supported. I don't use this sort of switch very often, and I got tired of having to skip past arguments that had already been processed to get to a filename. The arguments are evaluated left to right, so interaction between PROC type switches (see below) is possible.

When getargs( ) finds an error on the command line, it prints to stderr a list of all legal switches, along with a brief description of what those switches do and their default values. After it prints the error message, getargs( ) terminates the program with an exit(1) call. This closes any open files and returns to the operating system.

Command Line Switch Formats

Command line switches all must take the form:

```
−<character> [ <number> ¦
   <string> ]
```

That is, all arguments start with a minus sign; each switch is identified by a single letter that follows the minus sign immediately (no intervening spaces); the letter may be followed by an optional number or string, again with no spaces between the character that serves as the switch identifier and the corresponding number or string. Switches may be combined, provided that the argument types allow this combination. For example, the command line:

```
program −a −b123 −c
```

can also be written as:

```
program −ab123c
```

However, if getargs( ) expects a string to follow the switch identifier character, then it assumes that the rest of the argument is part of the string. You have to be careful when combining string type command line switches with other types.

Using Getargs( )

To use getargs( ) you must do two things: (1) set up a table to tell the routine what kind of command line switches to expect; (2) call the routine itself somewhere early in the main( ) module. The file getargs.h (Listing One, page 32) contains the #defines and typedefs needed to create the command switch descriptor table. This table is an array of structures:

```
typedef struct
{
    unsigned    arg : 7     ;
    unsigned    type : 4    ;
    int         *variable   ;
    char        *errmsg     ;
}
ARG;
```

The arg field is a single character that identifies the switch on the command line. In the following descriptions this character is represented as <switch>. The type field may take any one of five values, all of which are

#defined in getargs.h. The behavior of getargs( ) will vary according to the value.

INTEGER switches take the form

−<switch> <number>

Numbers preceded by 0x are hex, by 0 without the x, octal. All others are decimal. The number is terminated by any character not legal in the indicated radix. Any characters that follow are assumed to be additional switches. The int sized variable pointed to by the variable field of the ARG structure will be set to the value of the <number>. If the <number> isn't typed on the command line, *variable will be set to 0.

BOOLEAN switches will cause some action based on the presence or absence of the indicated switch on the command line. If the switch is present, then the int pointed to by the variable field is set to 1, otherwise *variable is not modified.

CHARACTER switches take the form

−<switch> <character>

When the switch is found on the command line, then *variable is set to the character immediately following the <switch> character.

STRING switches take the form

−<switch> <string>

When the switch is found on the command line, the *character pointer* pointed to by the variable field is set to point at a string consisting of all characters following (but not including) the <switch> character up to the end of the current argument (not to the end of the command line). In the case of combined switches in a single argument, the STRING argument must be the last one because all following characters will be considered part of the <string>. When defining a STRING switch in the table, be sure to cast the variable into an *integer pointer*. See Listing Three (page 38), line 15, for an example.

PROC switches take the form

−<switch> <anything>

This works like the STRING switch in that all characters following <switch> up to the end of the current argument will be part of the <anything>. However, the variable field is a pointer to a subroutine that is called indirectly as soon as the switch is encountered on the command line. A pointer to <anything> is passed to this subroutine as a single argument. An example of such a subroutine is given in Listing Three, line 19. It is the responsibility of the called subroutine to parse <anything> as appropriate.

The errmsg field is used to print an error message if an undefined switch is found. An example of the format is shown in the Figure (page 30). This message was generated when the command line "argtest −x" was given to argtest.

Listing Three (if you haven't realized it by now) is an example of how to use getargs( ). Lines 5-9 are declarations of the objects that are used in the variable fields of the switch descriptor table. The initial values of these variables remain unchanged if the switch is not encountered on the command line at runtime. The table itself, Argtab, is declared on lines 10-17. The main( ) routine calls getargs( ) on line 35. The rest of main( ) just prints the command line, both before and after the getargs( ) call, so you can see how getargs( ) strips processed switches out of argv.

Getargs( ) itself starts on line 98 of Listing Two (page 32). It is passed four arguments: argv, argc, a pointer to the switch descriptor table (tabp), and the size of this table in elements (not bytes). The routine returns a new value of argc, and argv is compressed, i.e., all entries containing command line switches are removed from it. The for loop on line 112 processes argv one element at a time. Nargv points into argv and strips processed switches. Nargv is initialized to point at argv[1] because argv[0] can't possibly contain a command line switch (it holds the program name). If no leading minus sign is found in the argument, *argv is copied to *nargv and both pointers are advanced. In addition, nargc (new argc) is incremented. If the argument does begin with a minus sign, it is processed as a command line switch. In this case *argv is not copied to *nargv, and argv, but not nargv, is advanced. This effectively eliminates the argument containing the switch from the argv array.

Findarg( ) (Listing Two, line 44) is used by getargs( ) to see if an argument is in the table. It performs a linear search. Because the table is usually fairly small, it seemed as if the extra code needed for a binary search wasn't justified. Findarg( ) takes three arguments: c is the switch identifier character being searched for; tabp and tabsize are the same parameters as were passed to getargs( ).

Setarg( ) (Listing Two, line 9) is called when a command line switch is found in the table. Argp is a pointer into the table, as returned from findarg( ). Linep is a pointer into the argv entry that is being processed. Casts were used to make this routine as transportable as possible.

Pr_usage( ) (Listing Two, line 57) is used to print the error message shown in the Figure when an illegal command line switch is found. It just spins through the table, printing the

```
Illegal argument <x>. Legal arguments are:

-b          boolean argument        (value is FALSE)
-c<c>       character argument      (value is .)
-n<num>     integer argument        (value is 0)
-s<str>     string argument         (value is <doo wha>)
-p<str>     procedure argument
```

Figure
Error Output from Program in Listing Three

current contents of the object pointed to by the variable field as well as the message given in the errmsg field.

The final routine in the package is stoi( ) (for string to integer, Listing Four, page 38), which is used to process INTEGER switches. Stoi( ) is a fancy version of the standard library routine atoi( ) (described on page 58 of Kernighan & Ritchie). Stoi( ) can handle numbers in base 8, 10, or 16. If code size is a real issue, you may want to remove the code that processes octal numbers (Listing Four, lines 45-53). Another major difference between stoi( ) and atoi( ) is that stoi( ) is passed a *pointer* to a character pointer. That is, it is passed the *address* of the pointer, which in turn points into the string to be processed. This extra level of indirection lets you modify the character pointer itself to point past the end of the number being processed. If you use atoi( ), you have to go through the string twice, once to extract the number and once more to skip past the characters that represent that number.

Conclusion

Several additions could be made to getargs( ), at the cost of an increase in complexity and code size.

- A common command line function is to open or close a file in a particular mode and to abort if the file can't be opened. At present I'm doing this with a PROC type command line switch, but you could add another argument type, whose variable field points to a FILE pointer. Alternately, variable could point to a structure that included the FILE pointer, an open mode, a pointer to an error processing routine, and a default file name.
- Another nice feature would be to mark an argument if its presence is required on the command line. You could accomplish this by adding additional types (i.e., REQ_INTEGER), or by adding another field to the ARG structure. An error message would be printed if the argument wasn't found on the command line.
- Getargs( ) depends on initializers to set default argument values. If your compiler doesn't support initializers, you can add a default value field to the ARG structure.
- I have violated my smallness rule by using stoi( ) to process numeric arguments. If you expect only decimal numbers, you may want to replace stoi( ) with atoi( ). However, stoi( ) is a useful routine in its own right.

I'm sure that you can think of other bells and whistles. I've been using getargs( ) for a of couple years now and am satisfied with it in its existing state. Defining the way the command line looks and then getting switches from it is now a painless process. **DDJ**

C Chest (Text begins on page 28)

Listing One

```
 1:  /*      Getargs.h       Typedefs and defines needed for getargs
 2:   */
 3:  #define INTEGER         0
 4:  #define BOOLEAN         1
 5:  #define CHARACTER       2
 6:  #define STRING          3
 7:  #define PROC            4

 8:  typedef struct
 9:  {
10:          unsigned        arg  : 7  ;     /* Command line switch          */
11:          unsigned        type : 4  ;     /* variable type                */
12:          int             *variable ;     /* pointer to variable          */
13:          char            *errmsg   ;     /* pointer to error message     */
14:  }
15:  ARG;
```

End Listing One

Listing Two

```
 1:  /*      GETARGS.C       Command line argument processor for C programs
 2:   *
 3:   *          (C) Copyright 1985, Allen I. Holub.  All rights reserved.
 4:   *        This program may be copied for personal, non-profit use only.
 5:   */

 6:  #include <stdio.h>
 7:  #include <getargs.h>

 8:  typedef int      (*PFI)();

 9:  static char      *setarg( argp, linep )
10:  ARG              *argp;
11:  char             *linep;
12:  {
13:          /*      Set an argument. argp points at the argument table entry
14:           *      corresponding to *linep. Return linep, updated to point
15:           *      past the argument being set.
16:           */

17:          ++linep;

18:          switch( argp->type )
```

```
19:         {
20:         case INTEGER:
21:                 *argp->variable = stoi( &linep );
22:                 break;

23:         case BOOLEAN:
24:                 *argp->variable = 1;
25:                 break;

26:         case CHARACTER:
27:                 *argp->variable = *linep++ ;
28:                 break;

29:         case STRING:
30:                 *(char **)argp->variable = linep ;
31:                 linep = "";
32:                 break;

33:         case PROC:
34:                 (* (PFI)(argp->variable) )( linep );
35:                 linep = "";
36:                 break;

37:         default:
38:                 fprintf(stderr,"INTERNAL ERROR: BAD ARGUMENT TYPE\n");
39:                 break;
40:         }
41:         return( linep );
42: }

43: /*---------------------------------------------------------------------*/

44: static ARG      *findarg( c, tabp, tabsize )
45: int             c, tabsize;
46: ARG             *tabp;
47: {
48:         /*      Return pointer to argument table entry corresponding
49:          *      to c (or 0 if c isn't in table).
50:          */
51:
52:         for(; --tabsize >= 0 ; tabp++ )
53:                 if( tabp->arg == c )
54:                         return tabp;

55:         return 0;
56: }
57: static pr_usage( tabp, tabsize )
58: ARG     *tabp;
59: int     tabsize;
60: {
61:         /*      Print the argtab in the form:
62:          *              -<arg> <errmsg>     (value is <*variable>)
63:          */

64:         for(; --tabsize >= 0 ;  tabp++ )
65:         {
66:                 switch( tabp->type )
67:                 {
68:                 case INTEGER:
69:                         fprintf(stderr, "-%c<num> %-40s (value is ",
70:                                                 tabp->arg, tabp->errmsg);
71:                         fprintf(stderr, "%-5d)\n", *(tabp->variable) );
72:                         break;

73:                 case BOOLEAN:
74:                         fprintf(stderr,"-%c      %-40s (value is ",
75:                                                 tabp->arg, tabp->errmsg);
76:                         fprintf(stderr, "%-5s)\n", *(tabp->variable)
77:                                                 ? "TRUE": "FALSE");
78:                         break;

79:                 case CHARACTER:
80:                         fprintf(stderr, "-%c<c>   %-40s (value is ",
81:                                                 tabp->arg, tabp->errmsg);
82:                         fprintf(stderr, "%-5c)\n", *(tabp->variable) );
83:                         break;

84:                 case STRING:
85:                         fprintf(stderr, "-%c<str> %-40s (value is ",
86:                                                 tabp->arg, tabp->errmsg);
```

(Continued on page 36)

 (Listing Continued, text begins on page 28)

Listing Two

```
87:                             fprintf(stderr, "<%s>)\n",
88:                                                     *(char **)tabp->variable);
89:                             break;

90:                     case PROC:
91:                             fprintf(stderr, "-%c<str> %-40s\n",
92:                                                     tabp->arg, tabp->errmsg);
93:                             break;
94:                     }
95:             }
96:     }
97:     #define ERRMSG  "Illegal argument <%c>.   Legal arguments are:\n\n"

98:     int     getargs(argc, argv, tabp, tabsize )
99:     int     argc, tabsize ;
100:    char    **argv ;
101:    ARG     *tabp  ;
102:    {
103:            /* Process command line arguments. Stripping all command line
104:             * switches out of argv. Return a new argc. If an error is found
105:             * exit(1) is called (getargs won't return) and a usage message
106:             * is printed showing all arguments in the table.
107:             */

108:            register int    nargc       ;
109:            register char   **nargv, *p ;
110:            register ARG    *argp       ;

111:            nargc = 1 ;
112:            for(nargv = ++argv ; --argc > 0 ; argv++ )
113:            {
114:                    if( **argv != '-' )
115:                    {
116:                            *nargv++ = *argv ;
117:                            nargc++;
118:                    }
119:                    else
120:                    {
121:                            p = (*argv) + 1 ;

122:                            while( *p )
123:                            {
124:                                    if(argp = findarg(*p, tabp, tabsize))
125:                                            p = setarg( argp, p );
126:                                    else
127:                                    {
128:                                            fprintf(stderr, ERRMSG, *p );
129:                                            pr_usage( tabp, tabsize );
130:                                            exit( 1 );
131:                                    }
132:                            }
133:                    }
134:            }
135:            return nargc ;
136:    }
```

End Listing Two

Listing Three

```
1:  /*      ARGTEST.C       Test program for getargs.
2:   */

3:  #include <stdio.h>
4:  #include "getargs.h"

5:  int     boolarg = 0;                    /* Variables used by argtab */
6:  int     chararg = '.';
7:  int     intarg  = 0;
8:  char    *strarg  = "doo wha" ;
9:  extern  proc();

10: ARG     Argtab[]=
```

(Continued on page 38)

 (Listing Continued, text begins on page 28)

Listing Three

```
11:  {
12:          { 'b',  BOOLEAN,   &boolarg,       "boolean argument"          },
13:          { 'c',  CHARACTER, &chararg,       "character argument"        },
14:          { 'n',  INTEGER,   &intarg,        "integer argument"          },
15:          { 's',  STRING,    (int *)&strarg, "string argument"           },
16:          { 'p',  PROC,      (int *)&proc,   "procedure argument"        }
17:  };

18:  #define TABSIZE ( sizeof(Argtab) / sizeof(ARG) )

19:  proc( str )
20:  char    *str;
21:  {
22:          printf("Inside procedure called by -p command line switch, ");
23:          printf("string = <%s>\n", str );
24:  }

25:  main(argc, argv)
26:  int     argc;
27:  char    **argv;
28:  {
29:          register int    i;

30:          printf("Argc == %d. ", argc);
31:          printf("Cmd line: argtest ");
32:          for( i = 1 ; i < argc ; printf("%s ", argv[i++]) )
33:                  ;
34:          printf("\n");

35:          argc = getargs( argc, argv, Argtab, TABSIZE) ;

36:          printf("Argc == %d. ", argc);
37:          printf("Cmd line: argtest ");
38:          for( i = 1 ; i < argc ; printf("%s ", argv[i++]) )
39:                  ;
40:          printf("\n");
41:  }
```

End Listing Three

Listing Four

```
 1:  /* STOI.C       More powerful version of atoi.
 2:   *
 3:   *      Copyright (C) 1985 by Allen  Holub.  All rights reserved.
 4:   *      This program may be copied for personal, non-profit use only.
 5:   */

 6:  #define islower(c)      ( 'a' <= (c) && (c) <= 'z' )
 7:  #define toupper(c)      ( islower(c) ? (c) - ('a' - 'A') : (c) )

 8:  int             stoi(instr)
 9:  register char   **instr;
10:  {
11:          /*      Convert string to integer. If string starts with 0x it is
12:           *      interpreted as a hex number, else if it starts with a 0 it
13:           *      is octal, else it is decimal. Conversion stops on encounteri
14:           *      the first character which is not a digit in the indicated
15:           *      radix. *instr is updated to point past the end of the number
16:           */

17:          register int    num  = 0 ;
18:          register char   *str     ;
19:          int             sign = 1 ;

20:          str = *instr;

21:          while(*str == ' '  ||  *str == '\t'  ||  *str == '\n' )
22:                  str++ ;

23:          if( *str == '-' )
24:          {
25:                  sign = -1 ;
26:                  str++;
```

```
27:         }

28:         if(*str == '0')
29:         {
30:                 ++str;
31:                 if (*str == 'x'  ||  *str == 'X')
32:                 {
33:                         str++;
34:                         while(  ('0'<= *str && *str <= '9') ||
35:                                 ('a'<= *str && *str <= 'f') ||
36:                                 ('A'<= *str && *str <= 'F')  )
37:                         {
38:                                 num *= 16;
39:                                 num += ('0'<= *str && *str <= '9') ?
40:                                         *str - '0'                 :
41:                                         toupper(*str) - 'A' + 10   ;
42:                                 str++;
43:                         }
44:                 }
45:                 else
46:                 {
47:                         while( '0' <= *str  &&  *str <= '7' )
48:                         {
49:                                 num *= 8;
50:                                 num += *str++ - '0' ;
51:                         }
52:                 }
53:         }
54:         else
55:         {
56:                 while( '0' <= *str  &&  *str <= '9' )
57:                 {
58:                         num *= 10;
59:                         num += *str++ - '0' ;
60:                 }
61:         }

62:         *instr = str;
63:         return( num * sign );
64: }
```

End Listings

Using Decision Variables in Graphics Primitives

by Tom Hogan

"Graphics algorithms" is not a listing you'll find in the* Collected Algorithms of the ACM*, but graphics programming presents unique problems to the software developer.

Picture a vast subterranean cavern packed with robed, grey-bearded men. A small platform stands at one end. Billows of purple smoke appear and out of the mist steps a tall man with white hair. He speaks and his voice fills the cavern.

"Welcome, brethren. Tonight we shall consider how to guarde our Realme of Graphix from the uninitiated who would learn our Mysteries and divulge them. We will speake of Master Bresenham, who revealed to us the Secrete of ye Mystik Circle."

Does the idea of graphics wizards jealously guarding their secrets seem strange? I can tell you that it didn't seem so to me when I searched the literature for a routine that would map ellipses and couldn't find one.

Light in the Tunnel

Thanks to a tip, I learned about *Fundamentals of Interactive Computer Graphics* by James D. Foley and Andries Van Dam (Addison Wesley, 1982). It contains an algorithm for plotting a circle using a decision variable (based on Bresenham's algorithm). I used the decision variable method (hereafter referred to as the DV method) to derive the algorithm for plotting ellipses that is found in Listing One (page 45).

The DV Method Generalized

Although the DV method was used only to plot an ellipse, it can be used to plot other conic sections as well. In fact, the DV method can be used to generate any well-behaved curve that can be expressed as $G(x, y) = 0$.

The DV Method Compared

Speed is important in graphics applications. The DV method is fast because it adds, shifts, subtracts, and multiplies integers, taking much less time than would be required by floating-point calculations.

An obvious method for plotting an ellipse is to calculate each point independently by incrementing x in unit steps and calculating y using square roots. This method is slow and generates a curve that is nonuniform when the slope of its tangent line falls below -1 (see Figure 1, page 41).

In contrast to the preceding method, the DV method does not calculate points independently. Instead, each succeeding point is mapped with reference to the point previously mapped. Furthermore, this method guarantees that succeeding points are adjacent. Therefore, the generated curve is more uniform than the curve generated by calculating points independently (see Figure 2, page 41).

Reduce the Number of Sample Points

Eight adjacent points surround *any previously* mapped point. In order to use the DV method to map the curve, you must reduce the number of sample points from eight to two. The sign of the decision variable can only be posi-

---

Tom Hogan, C Source, 12801 Frost Road, Kansas City, MO 64138.

tive or negative. The two possibilities correspond to the two sample points.

Take the case of an ellipse mapped only in the first quadrant. Points 1, 2, 3, 4, and 6 (see Figure 3, page 42) can be discarded immediately when mapping from Point A to C (see Figure 4, page 42), because x never decreases and y never increases as the ellipse is mapped in that direction. Points 1 through 3 would require y to increase, while points 1, 4, and 6 would require x to decrease.

This reduces the choice to three points: 5, 7, and 8. Which two are chosen as the set of sample points depends on the slopes of the line segments joining them and the last point plotted: the slope of the tangent line must fall within the range of the slopes of the two line segments for the region plotted. The slopes of the line segments of any two points must approach the slope of the tangent line as closely as possible. Thus, a set comprising Points 5 and 7 would be excluded because it would violate this requirement in the region close to B in Figure 4. There remain the possible combinations 5 and 8 and 7 and 8. Points 5 and 8, corresponding to Points S and T in Figure 5 (page 42), can be said to comprise set A. Points 7 and 8, corresponding to Points T and U in Figure 5, can be said to comprise set B.

Changing Sets of Sample Points

From Point A to B (Figure 4), set A is used because the slope of the tangent falls within the range delimited by the slopes of $\overline{PS}$ and $\overline{PT}$ (Figure 5). The decision variable is initialized to its value at Point A (see Equation 9 or 10 in the Table on page 44). The quantities to be added to it are based on the sample points in set A (Equations 6 and 8).

When the slope of the tangent line reaches -1 (at Point B in Figure 4), set B is used (see Figure 6, page 42). The value of the decision variable is recalculated (Equation 11) at Point B for set B. The quantities that are added to the decision variable are now based on set B (Equations 13 and 15).

Expansion on General Applications

Please note a key feature of the DV method: the choice of the sample points depends on the slope of the tangent line and the direction of its change. A curve can be broken up into sections for plotting, based on the range of the slope of the tangent line in each section. The method is useful for a broad spectrum of curves.

Example: The Ellipse

The initial value of y depends on the position of the major axis. If it is vertical, y is R_m, half the length of the major axis. If horizontal, y equals R_m multiplied by the aspect ratio. The positions of the major and minor axes depend on the aspect ratio. If it is less than one, then the major axis is horizontal. If greater than one, it is vertical.

Long variables were used in this routine instead of integers or doubles in order to provide the widest range of values for the aspect ratio and R_m while retaining reasonable plotting speed. Integers limited the ellipse's size too much, while doubles were too slow.

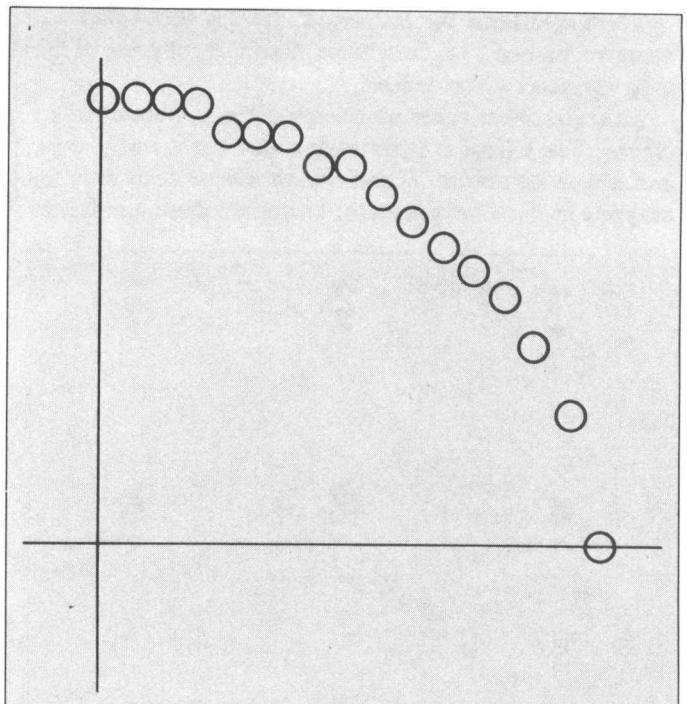


Figure 1
Circle generated by calculating points independently.

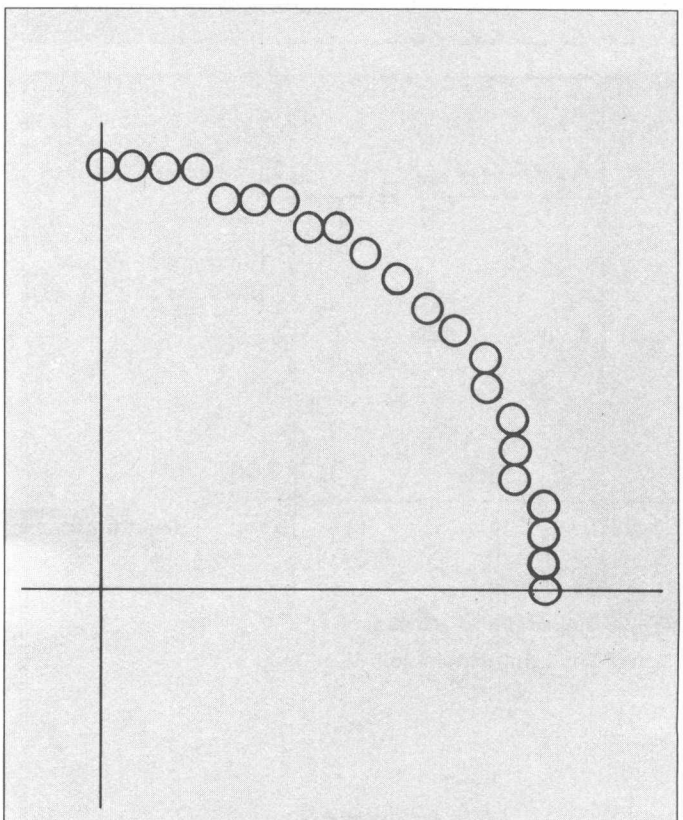


Figure 2
Circle generated by the Decision Variable Method.

The lower limit for the aspect ratio is 0.004 and R_m must not exceed 154. Otherwise, the values of some of the long variables will overflow.

This algorithm takes advantage of the symmetry of an ellipse. The ellipse is symmetric about the x and y axes, and about its center. Therefore, an ellipse need only be mapped in the first quadrant. The other three quadrants are easily plotted.

Clipping is done by the routine in Listing Two (page 48). The algorithm assumes medium resolution. DDJ

(Listings begin on page 45)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 193.**

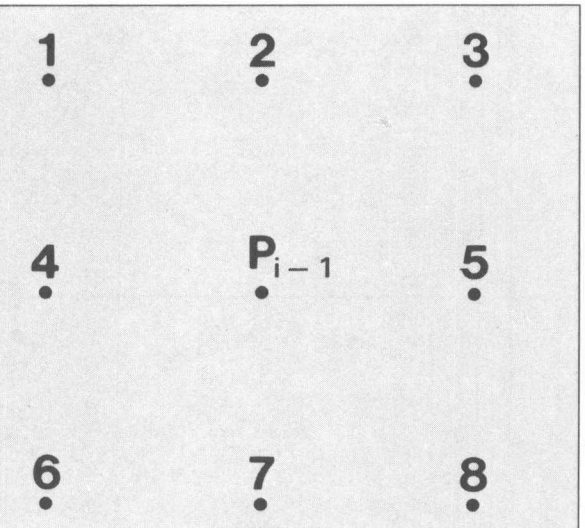


Figure 3
Last point plotted with eight adjacent points.

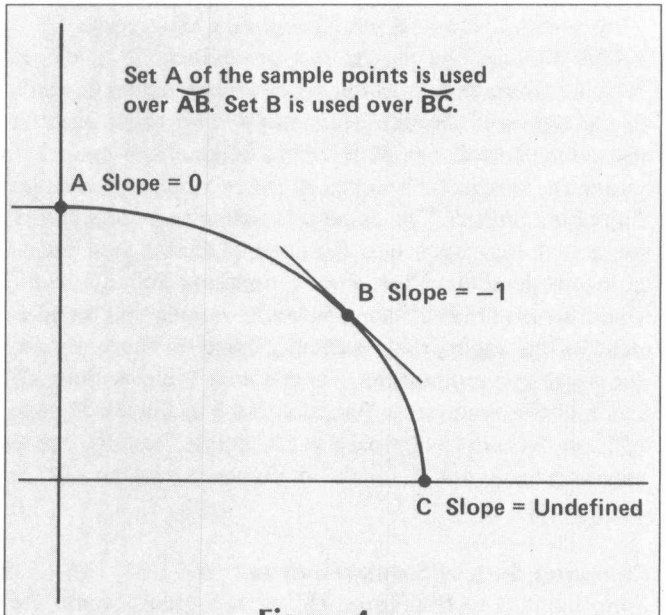


Figure 4
First quadrant of an ellipse.

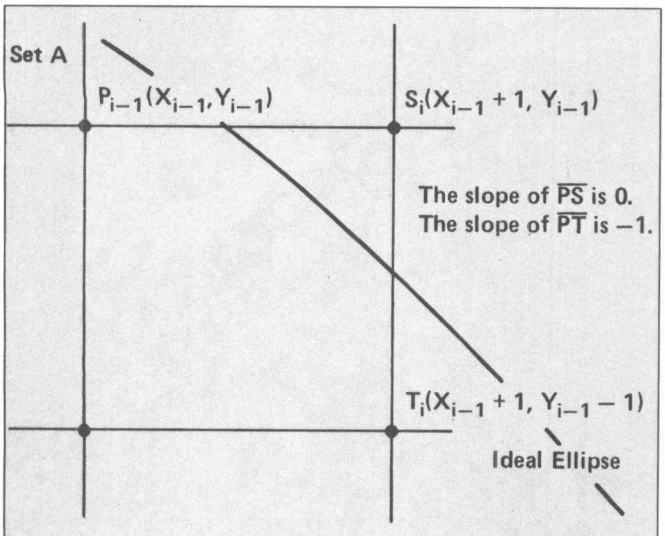


Figure 5
Last point plotted and sample points in the region where the slope of the tangent line is greater than −1.

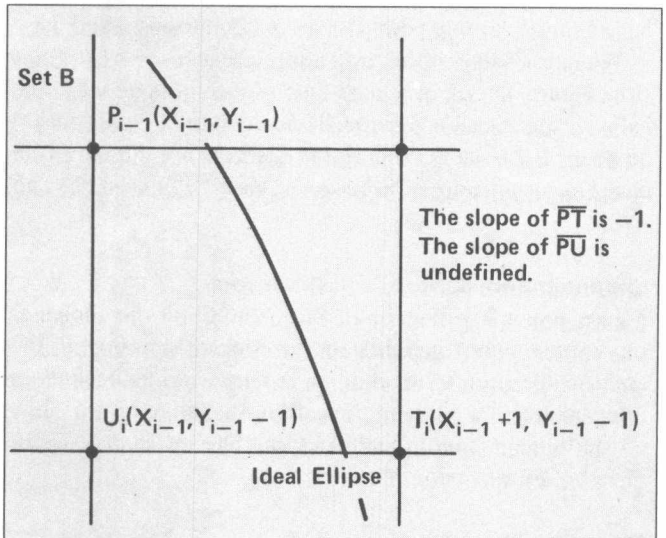


Figure 6
Last point plotted and sample points in the region where the slope of the tangent line is less than or equal to −1.

The well known equation of an ellipse is:

$$\alpha Y^2 + \beta X^2 - \alpha\beta = 0. \qquad \text{Eq. 1.}$$

Therefore define an error term $D(P_i)$ such that at any sample point P_i along the curve:

$$D(P_i) = \alpha Y^2 + \beta X^2 - \alpha\beta. \qquad \text{Eq. 2}$$

Define two points S_i and T_i relative to P_{i-1} such that P_i will be selected from one of these two points (see Figure 5). Define a decision variable d_i such that:

$$d_i = D(S_i) + D(T_i). \qquad \text{Eq. 3}$$

Notice that if the curve passes midway between S_i and T_i, $d_i = 0$. Further notice that:

1) if the curve passes above the midpoint, $d_i < 0$, and
2) if the curve passes below the midpoint, $d_i > 0$.

Therefore:

1) if $d_i < 0$, we choose S_i, and
2) if $d_i >= 0$, we choose T_i.

The quantity to add to d_i, depending on whether S_i or T_i is chosen, must be determined. The value of d_i for S_i and T_i must first be determined:

$$\begin{aligned} d_i &= D(S_i) + D(T_i). \\ d_i &= \alpha Y_{i-1}^2 + \beta(X_{i-1} + 1)^2 - \alpha\beta + \alpha(Y_{i-1} - 1)^2 \\ &\quad + \beta(X_{i-1} + 1)^2 - \alpha\beta. \\ d_i &= \alpha Y_{i-1}^2 + \alpha(Y_{i-1} - 1)^2 + 2\beta(X_{i-1} + 1)^2 \\ &\quad - 2\alpha\beta. \end{aligned} \qquad \text{Eq. 4}$$

Determine d_{i+1} if S_i is chosen:

$$\begin{aligned} d_{i+1} &= \alpha Y_{i-1}^2 + \beta(X_{i-1} + 2)^2 - \alpha\beta + \alpha(Y_{i-1} - 1)^2 \\ &\quad + \beta(X_{i-1} + 2)^2 - \alpha\beta. \\ d_{i+1} &= \alpha Y_{i-1}^2 + \alpha(Y_{i-1} - 1)^2 + 2\beta(X_{i-1} + 2)^2 \\ &\quad - 2\alpha\beta. \end{aligned} \qquad \text{Eq. 5}$$

Define Δ_S, the difference between d_{i+1} and d_i if S_i is chosen:

$$\begin{aligned} \Delta_S &= d_{i+1} - d_i. \\ \Delta_S &= 2\beta(X_{i-1} + 2)^2 - 2\beta(X_{i-1} + 1)^2. \\ \Delta_S &= 2\beta(2X_{i-1} + 3). \\ \Delta_S &= 4\beta X_{i-1} + 6\beta. \end{aligned} \qquad \text{Eq. 6}$$

Now calculate the quantity to be added to d_i if T_i is chosen:

$$\begin{aligned} d_{i+1} &= \alpha(Y_{i-1} - 1)^2 + \beta(X_{i-1} + 2)^2 - \alpha\beta \\ &\quad + \alpha(Y_{i-1} - 2)^2 + \beta(X_{i-1} + 2)^2 - \alpha\beta. \\ d_{i+1} &= \alpha(Y_{i-1} - 1)^2 + \alpha(Y_{i-1} - 2)^2 \\ &\quad + 2\beta(X_{i-1} + 2)^2 - 2\alpha\beta. \end{aligned} \qquad \text{Eq. 7}$$

Define Δ_T, the difference between d_{i+1} and d_i when T_i is chosen:

$$\begin{aligned} \Delta_T &= d_{i+1} - d_i. \\ \Delta_T &= 2\beta(X_{i-1} + 2)^2 - 2\beta(X_{i-1} + 1)^2 \\ &\quad + \alpha(Y_{i-1} - 2)^2 - \alpha Y_{i-1}^2. \\ \Delta_T &= 4\beta X_{i-1} + 6\beta - 4\alpha Y_{i-1} + 4\alpha. \end{aligned} \qquad \text{Eq. 8}$$

Now the value of d_i for the initial point, which is located on the Y-axis, must be determined. R_m is half the length of the major axis. If the Y-axis is the major axis, the initial point is the point $P(0, R_m)$. Therefore:

$$d_i = 2\alpha R_m^2 - 2\alpha R_m + \alpha + 2\beta - 2\alpha\beta. \qquad \text{Eq. 9}$$

If the X-axis is the major axis, the initial point is the point $P(0, (\beta/\alpha) \times R_m)$. Therefore:

$$d_i = (2\beta^2/\alpha) \times R_m^2 - 2\beta R_m + \alpha + 2\beta - 2\alpha\beta. \qquad \text{Eq. 10}$$

Consider now the slope of a line tangent to the ellipse. Beginning with the original point, it can be guaranteed that the slope of the tangent line will always decrease while the ellipse is being mapped in the first quadrant. Therefore, once the slope of the tangent line reaches -1, it will never be greater than -1. Also, it may be seen that from the point where the slope of the tangent line reaches -1 throughout the rest of the first quadrant, for any point P_{i-1} mapped, T_i will be closer than S_i. At some point, the ellipse will diverge enough from the tangent line of slope -1 for a point U_i (see Figure 6) to be closer to the ellipse than T_i. It will be sufficient to check the new set of points (T_i and U_i) from the point where the tangent line's slope becomes -1. d_i must be recalculated, as must Δ_T and Δ_U. Calculate d_i:

$$\begin{aligned} d_i &= D(T_i) + D(U_i). \\ d_i &= \alpha(Y_{i-1} - 1)^2 + \beta(X_{i-1} + 1)^2 - \alpha\beta \\ &\quad + \alpha(Y_{i-1} - 1)^2 + \beta X_{i-1}^2 - \alpha\beta. \\ d_i &= 2\alpha(Y_{i-1} - 1)^2 + \beta X_{i-1}^2 \\ &\quad + \beta(X_{i-1} + 1)^2 - 2\alpha\beta. \end{aligned} \qquad \text{Eq. 11}$$

Determine d_{i+1} if T_i is chosen:

$$\begin{aligned} d_{i+1} &= 2\alpha(Y_{i+1} - 2)^2 + \beta(X_{i-1} + 1)^2 \\ &\quad + \beta(X_{i-1} + 2)^2 - 2\alpha\beta. \end{aligned} \qquad \text{Eq. 12}$$

Define Δ_T as the difference between d_i and d_{i+1} if T_i is chosen:

Table
Derivation of the algorithm to map an ellipse.

$$\Delta_T = d_{i+1} - d_i.$$
$$= 2\alpha(-2Y_{i-1} + 3) + \beta(4X_{i-1} + 4). \quad \text{Eq. 13}$$

Determine d_{i+1} if U_i is chosen:

$$d_{i+1} = 2\alpha(Y_{i-1} - 2)^2 + \beta X_{i-1}^2$$
$$+ \beta(X_{i-1} + 1)^2 - 2\alpha\beta. \quad \text{Eq. 14}$$

Define Δ_U as the difference between d_{i+1} and d_i if U_i is chosen:

$$\Delta_U = d_{i+1} - d_i.$$
$$= 2\alpha(-2Y_{i-1} + 3). \quad \text{Eq. 15}$$

The point where the slope becomes less than -1 must be determined. Finding the first derivative of the ellipse produces the equation:

$$\alpha YY' = -\beta X. \quad \text{Eq. 16}$$

When the slope is required to be less than -1, the relation

$$\alpha Y < \beta X \quad \text{Eq. 17}$$

determines the point at which this occurs. This concludes the derivation of the algorithm for mapping an ellipse.

Decision Variables

(Text begins on page 40)

Listing One

```
/*   compiled 11/28/84 under Lattice C Compiler ver. 2.13
 *
 *   ellipse.c
 *     by Tom Hogan
 *   Version 2.0
 *   Copyright 1984
 *   by C Source Inc.
 *   All rights reserved.
 *   Permission is granted
 *   for unlimited
 *   personal, non-
 *   commercial use only.
 *
 *   For answers to questions,
 *   please write to:
 *
 *   Tom Hogan
 *   C Source Inc.
 *   12801 Frost Road
 *   Kansas City, MO 64138
 *
 *   If you would rather call,
 *   our number is (816) 353-8808.
 *
 *   Please call between 9 AM
 *   and 5 PM CST, Monday thru
 *   Friday.  Calls can only
 *   be returned on a collect basis.
 */

#define ERROR  0
#define SUCCESS 1

#define MAX_COL  319
#define MAX_ROW  199
#define MIN_COL  0
#define MIN_ROW  0

          /* a macro used in the error-checking code */

#define inrange(a,x,b)    ((a) <= (x) && (x) <= (b))

     /* the bounds of the following array can be set for different   *
      * graphics modes; this array is used in the assembly algorithm *
      * Listing 2 to write individual pixels.                        */
```

(Continued on next page)

Decision Variables (Listing Continued, text begins on page 40)

Listing One

```
int CRT_BNDS[] = { MAX_COL, MAX_ROW, MIN_COL, MIN_ROW };

          /* r_sub_m is half the length of the major axis */

ellipse(x, y, r_sub_m, color, aspect)
     int x, y, r_sub_m, color;
     double aspect;
{
int two_x, two_y, or_color;
int col, row, rel_x, rel_y;  /* rel_x and rel_y are relative */
double square_aspect;        /* to the center                */
long temp, beta, alpha, two_alpha, four_alpha, two_beta, four_beta, d;

                    /* d is the decision variable */

                    /*   error checking follows   */

if ((aspect < 0.004) || !inrange(0, color, 15) || !inrange(1, r_sub_m,
     154)) {
     printf("ELLIPSE(%d, %d, %d, %d, %f) - BAD ARG\n", x, y, r_sub_m,
          color, aspect);
     return ERROR; }
square_aspect = aspect * aspect;      /* initialize the beginning row  */
if (aspect < 1.0) {                   /* and set the values of         */
     alpha = r_sub_m * r_sub_m;       /* constants                     */
     beta = alpha * square_aspect;
     row = y + r_sub_m * aspect; }
else {
     beta = r_sub_m * r_sub_m;
     alpha = beta / square_aspect;
     row = y + r_sub_m; }
if (alpha == 0L) alpha = 1L;    /* compensates for very small ellipses */
if (beta == 0L) beta = 1L;
or_color = 0xc00 | color; /* sets the high byte for a screen interrupt */
col = x;                  /*     initializes the beginning column      */
two_x = x << 1;
two_y = y << 1;
rel_y = row - y;
two_alpha = alpha << 1;
four_alpha = alpha << 2;
four_beta = beta << 2;
two_beta = beta << 1;

           /* initialize the decision variable -- Eq. 9 or 10 */

d = two_alpha * ((rel_y - 1) * rel_y) + alpha + two_beta * (1 - alpha);

          /* when the slope <= -1, choose the new set of points */
          /*    -- Eq. 17                                       */

 while (alpha * (rel_y = row - y) > beta * (rel_x = col - x)) {
      write_pix(col, row, two_x - col, two_y - row, or_color);
      if (d >= 0) {
           d += four_alpha * (1 - rel_y);     /* Eq. 6; also first    */
           row--; }                           /* half of Eq. 8;       */
      d += two_beta * (3 + (rel_x << 1));     /* second half of Eq. 8 */
      col++; }

 /* initialize the decision variable for the rest of the ellipse */
 /*    -- Eq. 11                                                 */

 d = two_beta * (rel_x * (rel_x + 1) + two_alpha * (rel_y * (rel_y - 2)
         + 1) + (1 - two_alpha) * beta;

 while ((rel_y = row - y) + 1) {
      write_pix(col, row, two_x - col, two_y - row, or_color);
      if (d <= 0) {                          /* col - x == rel_x     */
           d += four_beta * (1 + col - x);   /* Eq. 15; also first   */
           col++; }                          /* half of Eq. 13;      */
      row--;
```

(Continued on page 48)

Decision Variables (Listing Continued, text begins on page 40)

Listing One

```
    d += two_alpha * (3 - (rel_y << 1)); }  /* second half of Eq. 13 */
return SUCCESS; }

        /*  write_pix writes four points to the screen, taking *
         *  advantage of the ellipse's four-way symmetry       */

write_pix(col, row, neg_col, neg_row, or_color)
    int col, row, neg_col, neg_row, or_color;
{
_pixasm(col, neg_row, or_color);        /* symmetry -- y axis */
_pixasm(neg_col, neg_row, or_color);    /* symmetry -- center */
_pixasm(neg_col, row, or_color);        /* symmetry -- x axis */
_pixasm(col, row, or_color); }          /* mapped point       */
```

End Listing One

Listing Two

8086 Assembly function that clips against the screen bounds

```
                    ;       _pixasm(x, y, type_color)
                    ;           int x, y, type_color
                    ;       type  - get or read pix in hi byte
                    ;       color - if set pix, in lo byte
                    ;
DGROUP  GROUP   DATA
DATA    SEGMENT WORD PUBLIC 'DATA'
        ASSUME  DS:DGROUP

        EXTRN CRT_BNDS:WORD

PGROUP  GROUP   PROG
PROG    SEGMENT BYTE PUBLIC 'PROG'
        ASSUME  CS:PGROUP

        PUBLIC _pixasm

_pixasm proc near
          push  bp              ; set return address
          mov   bp, sp

          mov   cx, [bp+4]      ; set x (column)
          cmp   cx, CRT_BNDS    ; check for max x value
          jg    BYE
          cmp   cx, CRT_BNDS+4  ; check for min x value
          jl    BYE
          mov   dx, [bp+6]      ; set y (row)
          cmp   dx, CRT_BNDS+2  ; check for max y value
          jg    BYE
          cmp   dx, CRT_BNDS+6  ; check for min y value
          jl    BYE
          mov   ax, [bp+8]      ; set color (al) / fcn number (ah)
          int   10h             ; do video interrupt
          sub   ah, ah
BYE:      pop   bp              ; reset frame pointer
          ret                   ; bye!

_pixasm endp
PROG ends
DATA ends
end
```

End Listings

Solid Shape Drawing on the Commodore 64

by Richard Rylander

Computer graphics is one of the fastest growing areas of computer software and hardware development. The old saying that "one picture is worth a thousand words" couldn't be truer when you consider the effectiveness of a pie chart or bar graph versus columns of numbers. Even the simplest of home computers is capable of at least this basic form of computer graphics. But the term *computer graphics* usually brings to mind more exciting images than just mundane business applications.

Special visual effects for motion pictures and some television advertisements are now being done with computers to create scenes of "enhanced reality." We see spacecraft making impossible manuevers or shimmering corporate logos "flying by" in ways that would be difficult to simulate with conventional photographic techniques. The enormous amount of data and processing necessary to produce such images has made the use of "super computers" essential. Even with the largest computers working full time on the task, only a few feet of film are produced per day. For home computer owners who want to create their own computer-generated masterpiece, however, the task is not hopeless.

The Apple Macintosh represents a major step in bringing powerful, easy-to-use graphics capabilities to the masses. The "masses," however, may still find the Macintosh pricey, especially if they just want to play around a bit with MacPaint. Low-cost computers, such as the Commodore 64, while lacking the slick graphic interaction of the Mac, are selling for such ridiculously low prices ($150 at the time of writing this article) that they are hard to pass up. They have the potential for some fairly sophisticated graphics, but so far most of the commercially available software has done little more than add the usual "point plotting" and "line drawing" commands.

This article describes a software package that will allow you to create your own computer-generated scenes, such as "Coffee and Donuts" in Figure 1 (page 55) and "Goblets" in Figure 2 (page 55). You can construct images from combinations of elementary shapes (spheres, cylinders, and toroids in various orientations) or by the "polygon mesh" technique typically used to render more complex objects. You can produce drawings with realistic shading effects and in a variety of styles to make some surprisingly detailed pictures with a minimum of effort. The package includes demonstration programs to illustrate how you use each graphic function and style option.

In keeping with the "running light without overbyte" theme of *DDJ*, the entire graphics package fits in 3K of RAM. The program sits in an area of

Pumping pixels requires tight, efficient code. Here are halftone shading, backlighting and other visual effects in a tight little package.

Richard Rylander, 179 N. McKnight Rd. Apt. 203, St. Paul, MN 55119.

RAM that is inaccessible to BASIC (a 4K block following the BASIC ROM) so that you don't lose any useful program space. Commodore supplies a "DOS Wedge" program that occupies the last 1K of this block, and one *of my objectives* in the design of the graphics package was to maintain compatibility with this useful utility.

The compactness of the code required me to leave out some "bells and whistles," but I've made no sacrifices to execution speed or user-interface ease. The main omission is error trapping—the software performs no checking to ensure that you use only "legal" point coordinates and so on. This may make program development a little more difficult, but once you have written and debugged a program properly, error checking becomes extraneous and only slows program execution.

We must address several general tasks in developing a shape-drawing program. The first is, of course, how to calculate the apparent brightness for all points on visible surfaces of objects. Next, how do we display these different brightness levels on a high-resolution display that is basically on or off? Finally, a "general" task, but detailed in nature, is creating the specific software tools that will let us do the required calculations and bit-map manipulations.

The program itself is written in machine language, because even with our simplifying restrictions, the plotting of each pixel still involves considerable computation. The program uses integer arithmetic throughout, and we utilize all the symmetry available to eliminate redundant calculations. The main program actually consists of five separate subprograms to break the process into manageable pieces and to provide a library of separate utilities that you may find useful in other programs. Before getting into the problem of shade calculation, I'll present the first subprogram, *which is* a collection of integer arithmetic utilities that all the later subprograms need.

Integer Arithmetic Utilities

Listing One (page 61) provides the source code for a set of four integer arithmetic routines. Because we will draw our shaded shapes on a high-resolution display of 320 x 200 points, we need only single-precision arguments for most functions and double-precision results for some intermediate values.

The first two routines, multiplying two single-precision numbers to yield a double-precision product and dividing a double-precision dividend by a double-precision divisor to yield a double-precision quotient, are fairly standard routines that should need no description other than the comments in the code. The multiplication routine also contains a special case to treat a single-precision number as a signed integer, which is then squared.

The integer-square-root routine deserves some special comment. Have you ever tried the BASIC square-root function? If you have, you know it takes about 52 milliseconds. At this rate, plotting the 64,000 pixels of a 320 x 200 screen would take about 3,328 seconds, or nearly an hour, if we need a square root per point! Of course, we don't need a square root for each point, but it is an

essential part of many shade calculations. Obviously, anything we can do to speed up this particular function will have a great effect on the overall program speed.

The BASIC square-root routine is slow because BASIC is written to save space, not time. Virtually all small BASICs find a square root by first taking the log of the argument, dividing that result by 2, and then exponentiating. Because LOG and EXP functions are already available, why use any more space than necessary to derive SQR?

One way to speed up this function is to use Newton's method, an iterative procedure that can give full floating-point precision in less time. Because we are interested only in integer results, we can do much better yet, though.

The method we actually use is essentially as fast as doing a single-divide operation. We construct the root, bit by bit, in a manner similar to the way we would take a decimal square root by hand using pencil and paper (does anyone remember how to do that in this age of electronic calculators?). The process is even easier for a binary root because the guesses are, of course, only 1 or 0. The procedure is almost identical to that for division: we guess a partial root bit (as we would guess the next quotient bit) and, in effect, compare the square of the partial quotient to the argument to see if we keep the 1 guessed or change it to a 0 and restore the argument. For more details about this procedure, I recommend section 17.3, "Binary Square Roots—Restoring Method," of the book *The Logic of Computer Arithmetic* by Ivan Flores (Englewood Cliffs, NJ: Prentice-Hall).

The last integer arithmetic routine is a fast procedure for generating pseudo-random bytes. An old and popular method for generating pseudo-random numbers is the so called congruential method, in which you derive each successive random number from the previous random number by multiplying by a suitably chosen constant and taking the product modulo P, where P is a large prime (see, for example, "A Better Random Number Generator," by H. Cem Kaner and John R. Vokey in the June 1984 issue of *MICRO*). Again, though, because we are interested only in generating random bytes, we can use a much faster routine.

The method we use is to store two previous random bytes, exclusive OR them bit by bit, and rotate the result to the right by one bit to generate a new random byte. Because it needs no multiply or divide operations, provides a fairly uniform distribution of random bytes, and has a long period, this process is fast. Note that even though we are only generating random bytes (0-255), the pattern or sequence of successive bytes doesn't repeat until after 35,805 calls to the random-number-generator routine (for the initial parameters used in the program). The long period is necessary to ensure that we won't produce any unwanted secondary patterns in what should be "randomly" shaded images—the human eye is adept at picking out such correlations.

Now that some low-level number crunching is out of the way, we can concentrate on the graphics aspects of the main program.

Graphics Utilities

Listing Two (page 62) provides the source code for the elementary graphics functions of displaying and clearing the bit-map, filling the color map (determining dot/background colors), and plotting (and unplotting) individual points both directly and with the plot or unplot decision weighted by a shade value and a particular shade style function. The PLOT (and UNPLOT), CLEAR, COLOR, GRFON (display graphics screen), and GROFF (return to text screen) routines are specific to the Commodore 64. To adapt this program for other 6502-based computers, you need to rewrite these routines, but the rest of the graphics package (save the final user interface to BASIC) is machine independent.

The SHADE routine is simple and straightforward. We set up an 8 x 8 threshold matrix as a linear array of values 0 to 63. The six bits we need to address an element of this threshold array are determined by masking off the lower three bits of the absolute X and Y screen coordinates (we take X modulo 8 and Y modulo 8) with the Y bits shifted to become the upper three bits. This effectively repeats the threshold pattern over the entire bit-map area. For each point within an object to be drawn, we calculate a shade value normalized to the range 0-63 and then compare it to the threshold value at that screen point to decide whether to plot or unplot.

If we have, for example, a large area where the shade value is constant at 31 (a 50 percent gray), then within each 8 × 8-pixel character cell block, half of the threshold values will be greater than or equal to the shade value, turning on half the pixels. The order in which the values 0 to 63 are arranged in the threshold table determines how these on pixels are distributed. A common pattern for these so called ordered-dither matrices is a recursive arrangement such as that shown in Table 1 (page 53). If that matrix is subdivided into quarters, sixteenths, and so on, each submatrix has the same general pattern, with offsets between submatrices.

The matrix in Table 1 (classic Bayer ordered dither) keeps the on and off pixels spread as far apart as possible for any shade. This keeps the spatial frequencies in the shade "texture" as high as possible for a particular shade. A disadvantage, though, is that texture changes then accompany shade changes, making shade quantization more apparent. Another disadvantage is that many color monitors have a hard time coping with the very high frequency on-off-on sequence of pixels, distorting shade values when isolated pixels get lost.

As an aside, a convenient algorithm to generate the ordered-dither matrix in Table 1 (or such a matrix of any size) is found in Figure 7 (page 59).

The threshold matrix we actually use in the program is shown in Table 2 (page 53). Here we follow a recursive scheme as we start turning on pixels until eight nuclei have been established. As shades increase, new pixels are added around the edges of these nuclei, simulating the dot-growth behavior seen in normal printing halftones. Except in the ex-

treme highlight and shadow regions, the shade texture remains fairly constant. Also, the clustering of on or off pixels is much less demanding on the display bandwidth. But take your pick—try filling the threshold matrix with your own patterns.

The RSHADE routine shades by comparing the shade value to a pseudo-random byte shifted right twice to match the 0-63 range. This scheme also tends to average out the tone errors generated as each pixel is turned only on or off (though we want an intermediate shade of gray) by dithering the threshold value randomly at each pixel. Both shading schemes produce sharp edges because each pixel is plotted independently. Abrupt shade changes are then followed faithfully.

The SCALE routine provides an opportunity to mention a general rule you should follow when you try to write fast programs: If at all possible, avoid division; and if you must divide, try to make it by a power of 2 (so that you can use right shifts). The SCALE routine helps correct some of the geometric distortion that otherwise results from plotting objects on the Commodore 64's rectangular bit map. While the monitor display has an aspect ratio of approximately 4:3, the bit-map aspect ratio is 320:200 or 8:5. A simple way to keep sphere outlines circular (instead of the usual egglike appearance) is to work in pseudoscreen coordinates of 320 × 240, giving the bitmap the same 4:3 aspect ratio as the screen display. The SCALE routine then converts 0-239 YPLOT values to a 0-199 range.

An obvious way to do this is to first multiply by 5 and then divide by 6. Or you might save a multiplication by first dividing a copy of YPLOT by 6 and subtracting that from the original YPLOT. A *much* faster way is to multiply the single-precision YPLOT by 213 and then take just the upper byte of the double-precision product (effectively dividing by 256) as the scaled YPLOT value. This gives us the proper range of absolute screen Y values and "rounds" the scaling as well.

This method of scaling means that you effectively replot every sixth Y line of pixels, but the routine's simplicity more than makes up for plotting 20 percent more points. Scaling is left as an option because some printers (such as the Commodore 1525) have 1:1 dot densities. By not scaling, the screen display is distorted but a hard-copy printout will have the proper geometry. On the other hand, you can set up printers, such as the Epson RX-80, with the appropriate horizontal and vertical dot densities to produce a 4:3 aspect ratio screen dump. Scaling here corrects both the screen and the hard-copy output.

The routine PLTSHD is a higher level routine that simply checks a couple of flags to see what kind of shading we want and whether to scale or not. It then calls the appropriate shading routine. To plot a shade-weighted pixel from BASIC, POKE a shade value (0-63) into VALUE; POKE the absolute X and (optionally scaled) Y values into XPLOT, XPLOT+1, and YPLOT; set up the flags HTORRN (HALFTONE or RANDOM) and NOSCAL; then SYS to PLTSHD. This seems like a lot of work to plot a single point, but we really won't be plotting shade-weighted points by hand—the later shape-drawing routines take care of this.

Line and Facet Drawing

Listing Three (page 64) completes the elementary graphics functions by adding line-drawing and shaded-facet-drawing routines. We draw lines using a modified form of Bresenham's algorithm, a DDA (Digital Differential Analyzer) technique that keeps the actual plotted points within one half-screen unit of the true line. Regardless of the order in which we specify the line's endpoints, the program sorts them so that lines are always drawn from left to right (the X position is incremented or unchanged at each step). We then need only determine which is greater, the change in X or the change in Y, and whether the Y coordinate difference is positive or negative. The program checks

0	32	8	40	2	34	10	42
48	16	56	24	50	18	58	26
12	44	4	36	14	46	6	38
60	28	52	20	62	30	54	22
3	35	11	43	1	33	9	41
51	19	59	27	49	17	57	25
15	47	7	39	13	45	5	37
63	31	55	23	61	29	53	21

Table 1
Recursive Arrangement of Ordered-Dither Matrix

0	8	53	61	2	10	55	63
16	24	37	45	18	26	39	47
49	57	4	12	51	59	6	14
33	41	20	28	35	43	22	30
3	11	54	62	1	9	52	60
19	27	38	46	17	25	36	44
50	58	7	15	48	56	5	13
34	42	23	31	32	40	21	29

Table 2
Threshold Matrix

the scale flag to see if the endpoints should be adjusted (in their Y coordinates) before plotting commences. This keeps a common coordinate system for drawing shaded shapes (spheres and so on) and lines in the same image. A flag MODE determines whether the line is drawn by setting or clearing pixels ("black" or "white" lines).

The program's last elementary function is to draw shaded triangular facets. A powerful and flexible technique for rendering objects is by means of a polygon mesh. Natural polyhedral objects (for example, cubes and pyramids) are obvious candidates for this method, but you can also approximate curved surfaces by a connected mesh of planar polygonal sections. As in piecewise-linear approximations of curves, the finer the segmentation, the better the approximation—although at the price of greater computational overhead. The coordinates of the polygon vertices then constitute a data base that you can manipulate easily for rotational transformations, perspective transformations, and so on. Triangular sections are the simplest to handle and the most general because you can break any higher-order polygon down into triangular sections.

As in the line-drawing routine, the program sorts the endpoints of the facets into a left-to-right order and checks the scale flag to maintain a consistent coordinate system. At each X position across the facet, a top and bottom Y coordinate pair is determined by a simple proportionality between the X difference and Y difference for the particular triangular sides. Although this involves both multiplication and division operations, it is not the innermost loop here (drawing the shaded line segment between Y pairs is). It also does not give more than one Y value per X position, as would a DDA method (for lines with slope greater than 1 in magnitude).

The maximum X difference is restricted to less than 256 (single precision), though the absolute X coordinates for the triangle vertices can be anywhere on the screen. This is not too severe a constraint because most polygon meshes are made up of small sections, and, if a larger triangle is essential, you can break it into smaller triangles that meet this condition. The occasional inconvenience this might cause is worth the increase in speed and shortened program.

We leave the shade value used in drawing facets as a parameter that the calling program specifies. Although adding "surface normal" calculation and shade value computation based on some illumination model (and requiring the Z coordinates then to be specified for the vertices) is not difficult, the same shade value is used for all points of the facet, making shade determination by a BASIC program practical. Also, leaving shade calculations up to BASIC adds the flexibility that we can use any shading model. The curved surface-drawing routines I describe later do include shade determination, but here each point can potentially have a different shade value, putting it in the innermost loop. For this reason, the curved surfaces must have a fixed shading model.

An interesting use for the BASIC-specified shade values is drawing "white" facets by setting the shade to 64. This doesn't seem particularly useful, but when you draw "wireframe" polyhedra, it makes a simple, hidden-line removal scheme possible. If the facets of an object are sorted so that they are drawn from those furthest to those nearest the observer, with "black" lines added around the edges of each facet, then foreground facets that partially obscure background facets erase the lines in their interior. We can set a flag EDGES so that, after the program has drawn a shaded facet, it can add lines automatically to outline and emphasize the edges. The MODE flag again determines how the lines are drawn (set or cleared).

Now that we have a toolbox of arithmetic and primitive-graphics routines, we can concentrate on the main subprogram for drawing shaded, curved surfaces rapidly.

Determining Shade Values for Curved Surfaces

Many recent advances in computer image synthesis have dealt specifically with how different surfaces reflect, refract, scatter, or absorb light. Increasingly complex models for the visual appearance of various surface textures have led to ever more realistic images, but increasing computational overhead accompanies these models. We'll employ the simplest shading scheme to start with and then suggest how you might modify it to add "realism," while still keeping the computations manageable.

If we restrict ourselves to simple, symmetrical objects, we can greatly simplify the problem of calculating surface brightness. This is not a severe restriction in itself, as you can break most "complex" objects down into combinations of simple elemental shapes. We will have to limit ourselves to "normal" (that is, head-on) views of the objects, because once we allow objects to be rotated, they lose most of their symmetry. Also, calculation time (particularly that for hidden surface removal) increases enormously.

We use the Lambertian (diffuse reflector) model to determine the surface brightness. We assume that light comes from a single, specific direction and then find the cosine of the angle between this reference vector and a surface normal vector. This gives us shade values from 1, where the surface faces the light source, to 0 at the terminator, where light just grazes the surface; and negative values where the surface is turned away from the light. Here is our first option: We can clip the brightness values at 0 so that areas facing away from the light have zero brightness, as an object lit by a single source (in deep space) would appear. Or, we can use the absolute value of the cosine so that the object appears to be lit by two identical sources on opposite sides. Either way is simple, so we allow for both cases by using a flag to determine what kind of illumination scheme we want to use for a whole scene or any individual shape making up the scene.

Although this simple model is valid for diffusely reflecting surfaces, it does not account for any contributions from specular surface reflections, secondary light sources (other

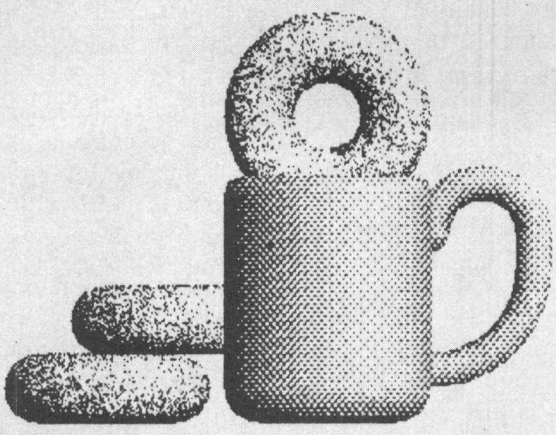


Figure 1

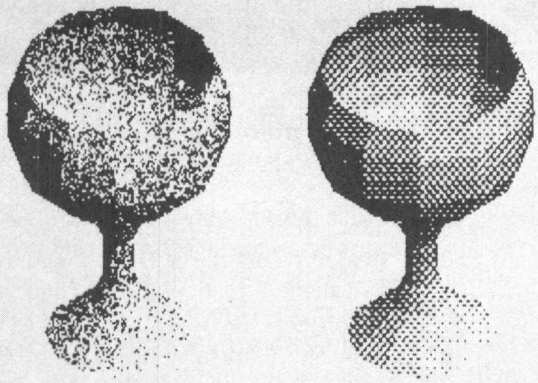


Figure 2

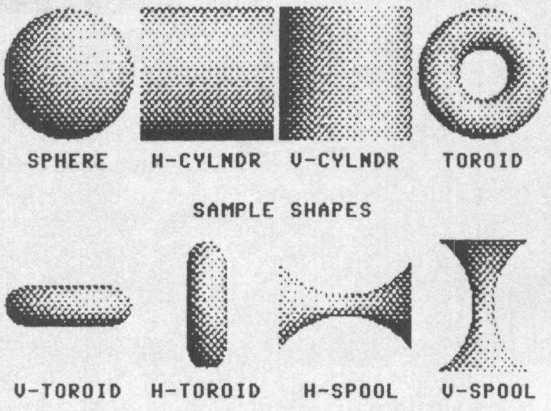


Figure 3

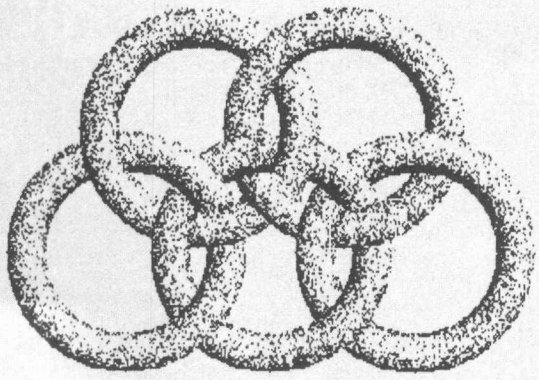


Figure 4

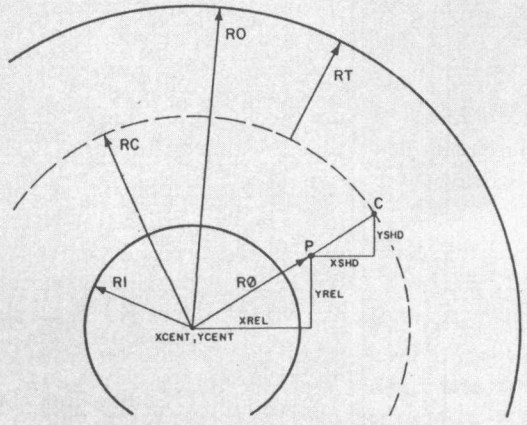


Figure 5

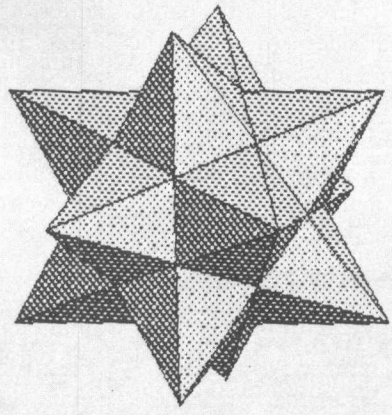


Figure 6

than the "opposite face" light) or ambient (nondirectional) light. These additional subtleties are not difficult to implement, but for the level of detail and dynamic range (number of gray levels) in our halftone display, they really aren't worth the effort.

Object Geometry

Calculating the surface normal vectors for points on an arbitrary surface can be difficult, but this is why we have restricted ourselves to objects that can be made up of combinations of the eight simple shapes shown in Figure 3 (page 55). All the basic shapes have at least fourfold symmetry; that is, once we have found the surface normal for a point (X, Y) in one quadrant of the object, we can easily find the surface normals for the other three quadrants by complementing appropriate components. Also, by restricting ourselves to shapes with relatively simple surfaces, we can determine the required surface normal vectors without resorting to trigonometric functions, using instead some simple geometric considerations.

Another time-saver is to choose an illumination reference vector to best take advantage of symmetries in the objects. Unfortunately, the "best" choice for computational purposes—a light source directly behind the viewer—gives a rather flat-looking image. Much more pleasing shading results from illumination coming from "over the shoulder." The slight assymmetry that results gives a stronger sense of depth to the objects.

The coordinate system describing objects on the screen has its X axis increasing horizontally to the right, the Y axis increasing vertically upward, and the Z axis increasing out of the screen toward the viewer (a conventional right-handed coordinate system). The illumination reference vector used here has X, Y, and Z components of (1,1,2), representing light coming from over the right shoulder. The symmetry in X and Y makes for easier shade calculations and gives a similar looking scene if you rotate a hard-copy output 90 degrees to fit objects that are taller than they are wide into the display. Lighting in the "portrait" (as opposed to "landscape"—unrotated) mode then comes from over the left shoulder.

Drawing a Shaded Sphere

Spheres are particularly easy objects to deal with because a radial vector from the center to a point on the surface is in the same direction as a surface normal from that point. The only real problem we must deal with is one of normalization. We will *always* deal with coordinates relative to the "local center of curvature" of objects. For spheres, this is just their geometric center.

Consider a point on the surface that extends out of the screen from relative coordinates (X,Y). We find the corresponding Z value using the equation for a sphere:

$$Z = SQR(R*R - X*X - Y*Y)$$

This calculated Z then forms the third member of the surface normal vector. To calculate the cosine of the angle between this vector and the illumination reference vector, we must first normalize both to have unit length. The normal we have just found has, of course, a length of R, so we need only divide each component by the radius of the sphere. The illumination vector (1,1,2) has a length of SQR(6), so we normalize this vector to unit length by dividing each component by SQR(6).

Now that we have two vectors of unit length but different directions, we find the cosine of the angle between them by taking their inner product, which is nothing more than summing the products of the X, Y, and Z coordinate pairs—like so:

$$COSINE = (1*X + 1*Y + 2*Z) / (R*SQR(6))$$

To use integer arithmetic throughout, we don't really want brightness expressed as a fraction between 0 and 1, but rather scaled to a range of integers from 0 to 63. We choose a maximum shade value of 63 to match the 64 gray levels that can be approximated by an 8×8 threshold matrix (described earlier in the "Graphics Utilities" section). We scale the pseudo-random bytes (used in the RSHADE routine) to the same range simply by shifting them right twice. So, for a surface point (X, Y, Z) on a sphere of radius R, the appropriate SHADE integer (0-63) is:

$$SHADE = 26*(X + Y + 2*Z)/R$$

The factor 26 properly accounts for the SQR (6) in the denominator of the previous equation.

As stated earlier, all the primitive shapes considered here have at least fourfold symmetry. We need compute the Z component only once for each ± X and ± Y quartet of points (relative to the geometric center of the object). The sphere and top-view toroid actually have n-fold symmetry, which the Cartesian grid of screen pixels reduces to eightfold symmetry. That is, in addition to changing signs of X and Y, we can exchange the X and Y coordinates to find another surface point with the same Z value.

Shape-Drawing Routines

Listing Four (page 68) is a set of routines for drawing the eight simple shapes shown in Figure 5. The routines resemble some form of compiled BASIC in that they use no special bit manipulations or unusual addressing methods, but just have appropriately ordered calls to the lower-level routines set up earlier. Comment statements precede each shape-drawing routine, giving an equivalent BASIC routine. This seems to be the easiest way to explain each routine because most readers are familiar with BASIC program methods. I thus give special comment to only a few of the routines here.

GETVAL is a shade-normalization routine that also checks for normal or backlit illumination via the flag BAKLIT. The byte pair at TONE contains the inner (dot) product of the illumination vector with the local surface normal vector. GETVAL then effectively does the division by SQR(6)*R and multiplication by 63 to put VALUE into the proper range, clipping at zero or taking the absolute value, as necessary.

PTPLOT does all the dirty work needed to plot shaded points for four-

fold symmetrical objects. Provision is also made here for clipping the object at independent levels up, down, left, and right of the object's center. The clipping feature is needed to blend various primitive objects into more complex ones with smooth transitions at the seams. Clipping also allows us to create some intricate "weaving" effects by redrawing portions of overlapping shapes as in Figure 4, "Linked Toroids," (page 55).

The flag NOROT determines whether the X and Y coordinates will be exchanged. This is useful both for drawing the eightfold symmetrical objects (top-view toroid and sphere) and for rotating the fourfold symmetrical objects 90 degrees so that a single drawing routine can produce two orientations of an object. The calculations are basically those for the sphere, but we'll see later that we can put other objects, such as toroids, into forms that look like spheres (at least locally).

GETZ calculates the effective Z coordinate for a spherelike surface, given the X and Y coordinates relative to the local center of curvature. Note that this routine needs the SQRT function, and because it is in general called for each quartet of points plotted, our fast SQRT makes a big difference in execution speed.

SPHERE follows the previously outlined algorithm and takes full advantage of the available symmetry. The radius is POKEd into RADIUS, and clipping distances relative to the sphere center are POKEd into CLIPL (left), CLIPR (right), CLIPU (up), and CLIPD (down).

CYLNDR draws a cylinder with sizes POKEd into RADIUS and HLEN (half-length). The flag NOROT determines whether the cylinder will be drawn with a horizontal or a vertical axis. The routine is simple, using PTPLOT for the main shade-calculation tasks. Actually, we could write a much faster routine to take advantage of the fact that we can use the same shade value for all points along lines parallel to the cylinder axis. This would require considerably more space, however.

The peculiar manipulations of relative coordinates for plotting toroids in various orientations are easier to explain with the help of the diagram in Figure 5 (appropriate for a top-view toroid) (page 55).

Consider a point P at coordinates (XREL, YREL) relative to the center of the object. We can take the local center of curvature of the toroid to be at the point C, which lies on the intersection of the line passing through the center of the toroid and a point beneath P (where Z=0) with a circle of radius RC (the "average" radius of the toroid is RC=(RO+RI) / 2). We determine the surface normal by the relative X, Y, and Z displacements from the point C.

The byte R0 is used for temporary storage of the radial distance from the toroid center to the point beneath P (where Z=0):

```
R0 = SQR(XREL*XREL
   + YREL*YREL)
```

We can determine the relative X and Y displacements, XSHD and YSHD, respectively, of P from C easily by using similar triangles:

```
XSHD = XREL*(1 - RC/R0)
YSHD = YREL*(1 - RC/R0)
```

We find the Z coordinate via the Pythagorean theorem, using the radius of the "ring" portion, RT, as the hypotenuse. RT is then also the length of the normal vector and is the value POKEd into RADIUS for normalization in the shade calculations.

The routine TOROID follows this algorithm, taking advantage of the eightfold symmetry in the object. EDGTOR and SPOOL use similar center of curvature coordinates to draw other orientations of a toroid. I have given the BASIC equivalents but leave it to readers to draw the appropriate geometric constructions if they desire more details.

Using the Shape-Drawing Routines—Interface to BASIC

The last module, completing our graphics package, is a convenient interface to BASIC. Listing Five (page 76) is a collection of routines to pass parameters easily to the graphics routines from the BASIC programs that control image generation. These routines are specific to the Commodore 64 in that they use some of the routines in the BASIC ROM to interpret statements in a BASIC program or entered in the direct command mode.

Graphics commands are given in the form:

```
SYS(FNCTN),PARAM1,PARAM2,
   [OPTIONAL PARAMETERS]
```

where FNCTN is the address of the graphic function you desire (or a variable that has been assigned the address value) and PARAM1 and PARAM2 are parameters (such as the center coordinates of an object) that the function requires, followed by any optional parameters.

The parameters can be either literal numerics or expressions that are evaluated for the desired values. We do not have to put the function address in parentheses, but it helps to make the program more readable. Separating parameters by commas is required. If you desire an optional parameter (such as specifying a new radius for sphere drawing), you add it by following the last required parameter by a comma and then the optional value or expression.

The use of optional parameters is more easily explained with the sequence of BASIC statements found in Figure 8 (page 59):

For all the shape-drawing functions, you must specify the X and Y center coordinates, but shape sizes are optional parameters. The last values specified become the new default sizes, allowing you to draw copies of similar objects (such as a bunch of grapes) just by setting the new center coordinates. All shapes but the sphere take two size parameters, and you must specify both when you desire a change (even when only one parameter takes on a new value). All size parameters (radii and cylinder half-lengths) must be less than 256, not a severe limitation considering the screen is only 200 pixels high ("240" with scaling).

The commands to clear the bit map and initialize the color map each take a single optional parameter, but the default values remain unchanged. Unless you specify alternate bytes, the bit

map is filled with 0s (cleared) and the color map is filled with 1s (for black dots on a white background). The most useful alternate byte with which to fill the bit map is 255, setting the entire background. Because the shading process clears, as well as sets, appropriate points, objects still appear normal but against a black background (particularly effective when you use the "back light" style).

The byte used to fill the color map is made up of background and foreground nibbles. To initialize the color map for a different color combination than black on white, use:

```
SYS(52001),16*DC+BC
```

where DC is the dot-color number and BC is the background-color number (*as given in* any reference to programming the Commodore 64).

The addresses for all the graphic functions and the parameters they take, along with locations to be POKEd with clipping values and flags for various drawing styles, are summarized in Table 3 (page 60). The best way to see how these shape-drawing routines are used is to examine the demonstration programs in Listing Six (page 78), "SHAPES DEMO," and Listing Seven (page 80), "STELLATION." All the different style options are exercized there. It is useful to save an abbreviated version of SHAPES DEMO, consisting of just the lines up to 340 and the subroutines following line 1620. This skeleton program can then form a base, providing all the POKE and SYS locations you need and to which you can add your own image-generation control program.

Auxiliary Programs

SHAPES DEMO and STELLATION make use of two auxiliary programs that, while not strictly needed to produce graphic images, provide utilities to both enhance the image itself and speed the drawing process. Listing Eight (page 80) is a routine for sorting an integer array indirectly through a key array (to determine drawing order for facets in a polygon mesh quickly). Listing Nine (page 82) allows you to add text to graphic

```
10   REM ORDERED DITHER ARRAY—RECURSIVE FILL
20   I=0:J=0:K=1:N=0:P=3
30   REM 'P' IS ORDER OF ARRAY, SIZE IS (2 ^ P) x (2 ^ P)
40   P=2 ^ P:DIM A(P - 1,P - 1)
50   GOSUB 100
60   END
100  IF K=P THEN A(I,J)=N:N=N+1:K=K/2:RETURN
110  K=2*K:GOSUB 100
120  I=I+K:J=J+K:K=2*K:GOSUB 100
130  I=I-K:K=2×K:GOSUB 100
140  I=I+K:J=J - K:K=2*K:GOSUB 100
150  I=I - K:K=K/2:RETURN
```

Figure 7

```
...  (SET UP GRAPHICS, STYLES, ETC)
200  SP=52119:REM ADDRESS OF SPHERE FUNCTION
210  SYS(SP),80,75,30:REM DRAW A SPHERE OF RADIUS 30 AT X=80, Y=75
220  SYS(SP),300,50 :REM DRAW ANOTHER SPHERE AT X=300, y=50
230  REM SINCE NO RADIUS IS SPECIFIED, LAST VALUE IS USED AS DEFAULT
240  SYS(SP),200,150,40:REM NEW SPHERE RADIUS (40) BECOMES DEFAULT
```

Figure 8

images in a variety of styles.

The code for these routines is short enough to be tucked in behind Commodore's DOS Wedge program so that, again, you don't lose any BASIC program space but maintain full *compatibility* with the wedge. If you are willing to give up the convenience of the wedge, however, you can easily relocate these routines to the end of the "interface" subprogram to keep the graphics package in one piece.

The KEYSORT routine works with two 1-dimensional integer arrays of the same size. One is filled with some "priority" parameter (such as the "average Z" coordinates for the facets in a polygon mesh), and the other becomes a "key" array whose elements index those in the "priority" array in increasing order.

To use the routine, POKE the maximum element index of the arrays into location 140, the address of the first (0th) element of the key array into locations 251 (low byte) and 252 (high byte), and the base address of the priority array into 253 and 254. You can find these array base addresses easily because the Commodore 64 places the address of the "current BASIC variable" into locations 71 and 72. Setting the 0th ele-

BIT-MAP SCREEN : 40960-48959 ($A000-$BF3F)

COLOR MAP : 33792-34791 ($8400-$87E7)

CLIPPING BOUNDS (relative to object center):

```
  893  ($037D)   : LEFT BOUND
  894  ($037E)   : RIGHT BOUND
  895  ($037F)   : UPPER BOUND
  896  ($0380)   : LOWER BOUND

53280  ($D020)   : BORDER COLOR [0-15], POKE with 1 to match white screen
```

STYLE FLAGS:

```
  838  ($0346)   : SHADE STYLE [0=random, 1=halftone]
  839  ($0347)   : SCALE FLAG [0=normal (1:1), 1=scaled (3:4)]
  868  ($0364)   : EDGES FLAG [0=normal, 1=add lines to facet edges]
  871  ($0367)   : EDGE/LINE MODE [0=draw, 1=erase]
  898  ($0382)   : LIGHTING STYLE [0=normal single source, 1=backlit]
```

FUNCTION LOCATIONS—CALL WITH "SYS(FNCTN), PARAMETERS." OPTIONAL PARAMETERS IN SQUARE BRACKETS, DEFAULT VALUES ARE USED IF NOT SPECIFIED

```
49378  ($C0E2)   : SWITCH TO GRAPHICS MODE
49411  ($C103)   : RETURN TO TEXT SCREEN

51979  ($CB0B)   : CLEAR[,CLEAR BYTE] FILL BIT MAP WITH CLEAR BYTE [DEFAULT = 0]
52001  ($CB21)   : COLOR[,COLOR BYTE] FILL COLOR MAP WITH COLOR BYTE

52023  ($CB37)   : PLOT,X,Y SET POINT AT (X,Y)
52026  ($CB3A)   : UNPLOT,X,Y CLEAR POINT AT (X,Y)

52049  ($CB51)   : LINE,X1,Y1,X2,Y2 DRAW A LINE BETWEEN (X1,Y1) AND (X2,Y2)

52052  ($CB54)   : FACET,X1,Y1,X2,Y2,X3,Y3,VA DRAW A SHADED TRIANGULAR FACET BETWEEN COORDINATE
                   PAIRS (X1,Y1), (X2,Y2) AND (X3,Y3) USING "VA" SHADE VALUE (0:BLACK TO 64:WHITE)
```

CURVED SURFACES:

```
52119  ($CB97)   : SPHERE,X,Y[,R] DRAW A SPHERE CENTERED AT (X,Y) WITH RADIUS R
52141  ($CBAD)   : TOROID,X,Y[,RI,RO] DRAW A TOP-VIEW TOROID WITH INNER RADIUS RI AND OUTER RADIUS RO
52150  ($CBB6)   : VCYL,X,Y[,R,HL] DRAW A CYLINDER WITH AXIS VERTICAL, RADIUS R AND HALF-LENGTH HL
52153  ($CBB9)   : HCYL,X,Y[,R,HL] DRAW A CYLINDER WITH AXIS HORIZONTAL
52186  ($CBDA)   : VTOR,X,Y[,RI,RO] DRAW AN EDGE-VIEW TOROID WITH AXIS VERTICAL
52189  ($CBDD)   : HTOR,X,Y[,RI,RO] DRAW AN EDGE-VEIW TOROID WITH AXIS HORIZONTAL
52203  ($CBEB)   : VSPOOL,X,Y[,RI,RO] DRAW AN INSIDE-VIEW TOROID ("SPOOL") WITH AXIS VERTICAL
52206  ($CBEE)   : HSPOOL,X,Y[,RI,RO] DRAW AN INSIDE-VIEW TOROID ("SPOOL") WITH AXIS HORIZONTAL
```

Table 3
Graphics Addresses

ment of an array equal to itself merely establishes it as the current variable. Then you can transfer the contents of 71 and 72 to the proper pointer at 251/252 or 253/254. Be careful to use literal numerics for "251" and so on, because you don't want to change the current variable. After setting up the pointers, SYS to the start of the KEYSORT routine. Another caution to observe if you use KEYSORT for other applications is *not* to create any new variables between finding the array base addresses and invoking KEYSORT, because arrays will be pushed up in memory to keep them at the end of BASIC's variable storage.

KEYSORT is set up to handle arrays with up to 256 elements each. If needed, you can sort larger arrays by breaking them into smaller arrays, sorting with KEYSORT, and then collating the results. Merging a few already sorted arrays is a fairly simple operation that you can do in a single pass, extending the usefulness of this routine.

STELLATION uses KEYSORT (lines 840–880) to determine the drawing order for the facets of a small stellated dodecahedron (Figure 6, page 55). This is a polyhedron that is not strictly convex (some internal dihedral angles are greater than 180 degrees), causing some foreground facets to partially obscure those in the background. The Painter's algorithm is used to handle hidden surface removal. Each newly drawn facet completely overwrites the background (clearing, as well as setting, points) so that drawing facets from back to front gives a proper rendering of the object.

The stellated dodecahedron of Figure 6 usually has only 30 visible or partially visible facets. Sorting with BASIC would be satisfactory here, but more complex polygon mesh objects can easily contain hundreds of visible and partially visible facets. You can really appreciate KEYSORT in those cases. Although it uses the simple and inefficient bubble-sort algorithm, being implemented in machine code lets KEYSORT run circles around any BASIC sorting alternative.

The final routine, in Listing Nine, WRITE, allows you to add text to the graphic display. The primary reason for implementing this as a machine code routine is that the bit map for the graphic image is located in the shadow RAM beneath the BASIC ROM (again saving useful BASIC program memory space). We can transfer bit images from the character ROM to the bit map just by POKEing values to the bit map, but we gain more flexibility if we first read the bit-map bytes to exclusive OR them, AND them, and so on with character image. Reading the shadow RAM requires that the BASIC ROM be switched out (through the bank switching used in the Commodore 64), something a BASIC program cannot do (and hope to continue execution).

The subroutine following line 1780 in SHAPES DEMO illustrates how to print text in a variety of styles on the bit-map image. Although the program actually uses only one style, the subroutine allows for five possibilities. Normal ("black" on "white") characters and reverse characters can overwrite the background, black characters can be ORed with the background, white characters ANDed with the background, or black characters exclusive ORed with the background. The last possibility gives characters that "change phase" across black/white edges in the image.

Conclusion

The graphics package described here we hope presents some new approaches to computer-generated images on small systems. At the same time, I'm sure I've missed some obvious tricks that would speed up the programs or reduce their size. I will be interested to see responses to this article and shortcuts or enhancements that might be added. Although the price/performance ratio of small graphics system continues to improve (especially with the development of custom LSI graphics chips), there are yet many unexplored software approaches to the problem. **DDJ**

Drawing on the C-64 (Text begins on page 50)

Listing One

```
00001  0000            ; INTEGER ARITHMETIC ROUTINES
00002  0000            ;
00003  0000            ; RICHARD L. RYLANDER    8/12/84
00004  0000            ;
00005  0000            ; REVISED 10/29/84 TO ADD FULL DOUBLE
00006  0000            ; PRECISION ARGUMENTS IN DIVIDE ROUTINE
00007  0000            ;
00008  0000            ;****************************************
00009  0000            ;
00010  0000            ; USE PAGE ZERO LOCATIONS WHERE POSSIBLE FOR
00011  0000            ; ITERATIVE PROCEDURE WORK SPACE
00012  0000            ;
00013  0000            MLPCND =$AC             ; MULTIPLICAND
00014  0000            MLPLIER =$AD            ; MULTIPLIER
00015  0000            PROD   =$AE             ; PRODUCT
00016  0000            ;
00017  0000            DVDND  =$FD             ; DIVIDEND/QUOTIENT
00018  0000            DVSOR  =$FB             ; DIVISOR
00019  0000            RMNDR  =$B4             ; REMAINDER
00020  0000            ;
00021  0000            RADCND =$AC             ; RADICAND
00022  0000            ROOT   =$033C           ; SQUARE ROOT
00023  0000            ;
00024  0000            TEMP   =$FB
00025  0000            ;
00026  0000            ; SET UP SEED VALUES FOR PSEUDO RANDOM NUMBERS
00027  0000                    *=$C000
00028  C000  FF        RNDM   .BYTE $FF,$55
00028  C001  55
00029  C002  00        RTEMP  .BYTE $00,$00
00029  C003  00
00030  C004            ;
00031  C004            ;
00032  C004            ;****************************************
00033  C004            ;
00034  C004            ; MULTIPLY SINGLE PRECISION MULTIPLICAND
00035  C004            ; BY SINGLE PRECISION MULTIPLIER GIVING
00036  C004            ; DOUBLE PRECISION PRODUCT (ENTER AT "MULT")
00037  C004            ;
00038  C004            ; SPECIAL CASE: ENTER AT "SQUARE" TO FIND
00039  C004            ; SQUARE OF SIGNED 8-BIT NUMBER
00040  C004            ;
00041  C004  A5 AC     SQUARE LDA MLPCND       ; ENTRY TO SQUARE
00042  C006  10 07            BPL POSITV      ; USE ABSOLUTE VALUE
00043  C008  38                SEC             ; NEGATE IF NEEDED
00044  C009  A9 00             LDA #$00
00045  C00B  E5 AC            SBC MLPCND
00046  C00D  85 AC            STA MLPCND
00047  C00F  85 AD     POSITV STA MLPLER
00048  C011  A9 00     MULT   LDA #$00         ; ENTRY TO MULTIPLY
00049  C013  A2 08             LDX #$08
00050  C015  46 AD     MLOOP  LSR MLPLER
00051  C017  90 03             BCC NOADD
00052  C019  18                CLC
00053  C01A  65 AC             ADC MLPCND
00054  C01C  6A        NOADD  ROR A
00055  C01D  66 AE             ROR PROD
00056  C01F  CA                DEX
00057  C020  D0 F3             BNE MLOOP
00058  C022  85 AF             STA PROD+1
00059  C024  60                RTS
00060  C025            ;
00061  C025            ;****************************************
```

(Continued on next page)

Drawing on the C-64

(Listing Continued, text begins on page 50)

Listing One

```
00062  C025                ;
00063  C025                ; DIVIDE DOUBLE PRECISION DIVIDEND
00064  C025                ; BY DOUBLE PRECISION DIVISOR GIVING
00065  C025                ; DOUBLE PRECISION QUOTIENT
00066  C025                ;
00067  C025                ; DIVIDEND IS REPLACED BY QUOTIENT
00068  C025                ; IN THE PROCESS
00069  C025                ;
00070  C025                ; QUOTIENT IS ROUNDED TO NEAREST INTEGER
00071  C025                ;
00072  C025  A9 00         DIVIDE LDA #$00
00073  C027  85 B4                STA RMNDR
00074  C029  85 B5                STA RMNDR+1
00075  C02B  A2 10                LDX #$10
00076  C02D  26 FD         DLOOP  ROL DVDND
00077  C02F  26 FE                ROL DVDND+1
00078  C031  26 B4                ROL RMNDR
00079  C033  26 B5                ROL RMNDR+1
00080  C035  38                   SEC
00081  C036  A5 B4                LDA RMNDR
00082  C038  E5 FB                SBC DVSOR
00083  C03A  AB                   TAY
00084  C03B  A5 B5                LDA RMNDR+1
00085  C03D  E5 FC                SBC DVSOR+1
00086  C03F  90 04                BCC DECCNT
00087  C041  84 B4                STY RMNDR
00088  C043  85 B5                STA RMNDR+1
00089  C045  CA            DECCNT DEX
00090  C046  D0 E5                BNE DLOOP
00091  C048  26 FD                ROL DVDND          ; CHECK IF REMAINDER
00092  C04A  26 FE                ROL DVDND+1        ; IS >= 1/2 OF DIVIDEND
00093  C04C  06 B4                ASL RMNDR          ; FOR ROUNDING
00094  C04E  26 B5                ROL RMNDR+1
00095  C050  B0 0B                BCS ROUND
00096  C052  38                   SEC
00097  C053  A5 FB                LDA DVSOR
00098  C055  E5 B4                SBC RMNDR
00099  C057  A5 FC                LDA DVSOR+1
00100  C059  E5 B5                SBC RMNDR+1
00101  C05B  B0 06                BCS NOCHNG
00102  C05D  E6 FD         ROUND  INC DVDND
00103  C05F  D0 02                BNE NOCHNG
00104  C061  E6 FE                INC DVDND+1
00105  C063  60            NOCHNG RTS
00106  C064                ;
00107  C064                ;*****************************************
00108  C064                ;
00109  C064                ; TAKE INTEGER SQUARE ROOT OF A
00110  C064                ; DOUBLE PRECISION RADICAND GIVING
00111  C064                ; SINGLE PRECISION ROOT ( <= REAL ROOT )
00112  C064                ;
00113  C064  A2 08         SQRT   LDX #$08
00114  C066  A9 00                LDA #$00
00115  C068  8D 3C 03             STA ROOT
00116  C06B  8D 3D 03             STA ROOT+1
00117  C06E  85 FB                STA TEMP
00118  C070  85 FC                STA TEMP+1
00119  C072  0E 3C 03      SQRT1  ASL ROOT
00120  C075  2E 3D 03             ROL ROOT+1
00121  C078  EE 3C 03             INC ROOT           ; ASSUME CURRENT LSB OF
00122  C07B  D0 03                BNE NEXT1          ; ROOT WILL BE 1
00123  C07D  EE 3D 03             INC ROOT+1
00124  C080  06 AC         NEXT1  ASL RADCND         ; SHIFT RADICAND LEFT
00125  C082  26 AD                ROL RADCND+1       ; TWICE INTO TEMP
00126  C084  26 FB                ROL TEMP
00127  C086  26 FC                ROL TEMP+1
00128  C088  06 AC                ASL RADCND
00129  C08A  26 AD                ROL RADCND+1
00130  C08C  26 FB                ROL TEMP
00131  C08E  26 FC                ROL TEMP+1
00132  C090  38                   SEC                ; SUBTRACT ROOT ESTIMATE
00133  C091  A5 FB                LDA TEMP           ; FROM TEMP
00134  C093  ED 3C 03             SBC ROOT
00135  C096  AB                   TAY
00136  C097  A5 FC                LDA TEMP+1
00137  C099  ED 3D 03             SBC ROOT+1
00138  C09C  90 12                BCC RESTOR
00139  C09E  85 FC                STA TEMP+1         ; SUBTRACTION OK
00140  C0A0  84 FB                STY TEMP
00141  C0A2  EE 3C 03             INC ROOT
00142  C0A5  D0 03                BNE NEXT2
00143  C0A7  EE 3D 03             INC ROOT+1
00144  C0AA  CA            NEXT2  DEX
00145  C0AB  D0 C5                BNE SQRT1
00146  C0AD  4C C1 C0             JMP FINI
00147  C0B0  38            RESTOR SEC                ; IGNORE SUBTRACTION
00148  C0B1  AD 3C 03             LDA ROOT           ; AND RESET LSB OF ROOT
00149  C0B4  E9 01                SBC #$01
00150  C0B6  8D 3C 03             STA ROOT
00151  C0B9  B0 03                BCS NEXT3
00152  C0BB  CE 3D 03             DEC ROOT+1
00153  C0BE  CA            NEXT3  DEX
00154  C0BF  D0 B1                BNE SQRT1
00155  C0C1  6E 3D 03      FINI   ROR ROOT+1         ; FINAL /2 TO NORMALIZE
00156  C0C4  6E 3C 03             ROR ROOT
00157  C0C7  60                   RTS
00158  C0C8                ;
00159  C0C8                ;*****************************************
00160  C0C8                ;
00161  C0C8                ; GENERATE PSEUDO-RANDOM BYTES
00162  C0C8                ; EXIT WITH P-R BYTE IN ACCUM.
00163  C0C8                ;
00164  C0C8  AD 00 C0      RANDOM LDA RNDM
00165  C0CB  8D 02 C0             STA RTEMP
00166  C0CE  4D 01 C0             EOR RNDM+1
00167  C0D1  2E 03 C0             ROL RTEMP+1        ; RTEMP+1 PRESERVES
00168  C0D4  6A                   ROR A              ; CARRY BIT FOR CYCLING
00169  C0D5  6E 03 C0             ROR RTEMP+1        ; RANDOM NUMBERS
00170  C0D8  8D 00 C0             STA RNDM
00171  C0DB  AD 02 C0             LDA RTEMP
00172  C0DE  8D 01 C0             STA RNDM+1
00173  C0E1  60                   RTS
00174  C0E2                .END

ERRORS = 00000

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 DECCNT   C045    DIVIDE   C025    DLOOP   C02D    DVDND   00FD
 DVSOR    00FB    FINI     C0C1    MLOOP   C015    MLPCND  00AC
 MLPLER   00AD    MULT     C011    NEXT1   C080    NEXT2   C0AA
 NEXT3    C0BE    NOADD    C01C    NOCHNG  C063    POSITV  C00F
 PROD     00AE    RADCND   00AC    RANDOM  C0CB    RESTOR  C0B0
 RMNDR    00B4    RNDM     C000    ROOT    033C    ROUND   C05D
 RTEMP    C002    SQRT     C064    SQRT1   C072    SQUARE  C004
 TEMP     00FB

END OF ASSEMBLY
```

End Listing One

Listing Two

```
00001  0000                ; GRAPHICS UTILITIES
00002  0000                ;
00003  0000                ; RICHARD L. RYLANDER   11/4/84
00004  0000                ;
00005  0000                ; LOAD ARITHMETIC UTILITIES FIRST
00006  0000                ;
00007  0000                RAM=$033E
00008  0000                ORIGIN=$C0E2
00009  0000                ;
00010  0000                MLPCND=$AC              ; MULTIPLICAND (S)
00011  0000                MLPLER=$AD              ; MULTIPLIER (S)
00012  0000                PROD=$AE                ; PRODUCT (D)
00013  0000                MULT=$C011              ; CALL FOR MULTIPLY
00014  0000                ;
00015  0000                RNDM=$C000              ; RANDOM NUMBER
00016  0000                RANDOM=$C0C8            ; CALL FOR RANDOM
00017  0000                ; NOTE - A CALL TO 'RANDOM' LEAVES A RANDOM BYTE
00018  0000                ; IN THE ACCUMULATOR
00019  0000                ;
00020  0000                      *=RAM
00021  033E                PLTFLG *=*+1            ; PLOT/UNPLOT FLAG
00022  033F                XPLT   *=*+2            ; ABSOLUTE PLOT COORD
00023  0341                YPLT   *=*+1            ; ABSOLUTE PLOT COORD
00024  0342                VIC1   *=*+1            ; REGISTER STORAGE
00025  0343                VIC2   *=*+1            ; REGISTER STORAGE
00026  0344                VALUE  *=*+2            ; FINAL NORMALIZED SHADE VALUE
00027  0346                HTORRN *=*+1            ; SHADE FLAG, 1=HALFTONE
00028  0347                NOSCAL *=*+1            ; SCALE FLAG, 1=NO SCALE
00029  0348                TEMP   *=*+2            ; TEMPORARY STORAGE
00030  034A                ;
00031  034A                      *=ORIGIN
00032  C0E2                ;
00033  C0E2                ;*****************************************
00034  C0E2                ;
00035  C0E2                ; TURN ON BIT MAP GRAPHICS MODE,
00036  C0E2                ; SAVING REGISTER VALUES FOR
00037  C0E2                ; RETURN TO TEXT MODE LATER.
00038  C0E2                ;
00039  C0E2  AD 11 D0      GRFON  LDA $D011
00040  C0E5  09 20                ORA #$20
00041  C0E7  8D 11 D0             STA $D011
00042  C0EA  AD 00 DD             LDA $DD00
00043  C0ED  8D 42 03             STA VIC1
00044  C0F0  29 FC                AND #$FC
00045  C0F2  09 01                ORA #$01
00046  C0F4  8D 00 DD             STA $DD00
00047  C0F7  AD 18 D0             LDA $D018
00048  C0FA  8D 43 03             STA VIC2
00049  C0FD  A9 19                LDA #$19
00050  C0FF  8D 18 D0             STA $D018
00051  C102  60                   RTS
00052  C103                ;
00053  C103                ;*****************************************
00054  C103                ;
00055  C103                ; RETURN TO TEXT SCREEN
00056  C103                ;
00057  C103  AD 11 D0      GRFOFF LDA $D011
00058  C106  29 DF                AND #$DF
00059  C108  8D 11 D0             STA $D011
00060  C10B  AD 42 03             LDA VIC1
00061  C10E  8D 00 DD             STA $DD00
00062  C111  AD 43 03             LDA VIC2
00063  C114  8D 18 D0             STA $D018
00064  C117  60                   RTS
00065  C118                ;
00066  C118                ;*****************************************
00067  C118                ;
00068  C118                ; FILL COLOR MAP FOR BLACK DOTS ON WHITE
00069  C118                ;
00070  C118  A9 01         COLOR  LDA #$01         ; POKE NEW COLORS HERE
00071  C11A  A2 00                LDX #0
00072  C11C  9D 00 84      COL1   STA $8400,X
00073  C11F  9D 00 85             STA $8500,X
00074  C122  9D 00 86             STA $8600,X
00075  C125  9D 00 87             STA $8700,X
00076  C128  CA                   DEX
00077  C129  D0 F1                BNE COL1
00078  C12B  60                   RTS
00079  C12C                ;
00080  C12C                ;*****************************************
00081  C12C                ;
00082  C12C                ; CLEAR HI-RES GRAPHICS SCREEN
00083  C12C                ;
00084  C12C  A9 A0         CLEAR  LDA #$A0
00085  C12E  85 FC                STA $FC
00086  C130  A0 00                LDY #0
00087  C132  84 FB                STY $FB
00088  C134  A9 00                LDA #0          ; CLEAR BYTE
00089  C136  A2 20                LDX #$20
00090  C138  91 FB         CLRLP  STA ($FB),Y
00091  C13A  C8                   INY
00092  C13B  D0 FB                BNE CLRLP
00093  C13D  E6 FC                INC $FC
00094  C13F  CA                   DEX
00095  C140  D0 F6                BNE CLRLP
00096  C142  60                   RTS
00097  C143                ;
00098  C143                ;*****************************************
00099  C143                ;
00100  C143                ; PLOT AND UNPLOT POINTS ON HI-RES GRAPHICS
00101  C143                ; SCREEN. ABSOLUTE X AND Y SCREEN COORDINATES
00102  C143                ; ARE POKED INTO XPLT, XPLT+1, AND YPLT
00103  C143                ;
00104  C143  A9 00         PLOT   LDA #0
00105  C145  2C                   .BYTE $2C
00106  C146  A9 80         UNPLOT LDA #$80
00107  C148  8D 3E 03             STA PLTFLG
00108  C14B  A5 01                LDA $01         ; BASIC ROM OUT
00109  C14D  29 FE                AND #$FE
00110  C14F  85 01                STA $01
00111  C151  38                   SEC             ; INVERT Y COORDINATE TO
00112  C152  A9 C7                LDA #$C7        ; PUT ORIGIN IN LOWER LEFT
00113  C154  ED 41 03             SBC YPLT        ; CORNER OF SCREEN
00114  C157  AA                   TAX             ; (199.-YPLT)
00115  C158  4A                   LSR A
00116  C159  4A                   LSR A
00117  C15A  4A                   LSR A
00118  C15B  A8                   TAY
00119  C15C  B9 AB C1             LDA TABLE1,Y
00120  C15F  85 FB                STA $FB
00121  C161  B9 C4 C1             LDA TABLE2,Y
00122  C164  85 FC                STA $FC
```

(Continued on page 64)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Two

```
00123  C166  8A              TXA
00124  C167  29 07           AND #$07
00125  C169  18              CLC
00126  C16A  65 FB           ADC $FB
00127  C16C  85 FB           STA $FB
00128  C16E  AD 3F 03        LDA XPLT
00129  C171  29 F8           AND #$F8
00130  C173  65 FB           ADC $FB
00131  C175  85 FB           STA $FB
00132  C177  AD 40 03        LDA XPLT+1
00133  C17A  65 FC           ADC $FC
00134  C17C  85 FC           STA $FC
00135  C17E  A9 A0           LDA #$A0
00136  C180  65 FC           ADC $FC
00137  C182  85 FC           STA $FC
00138  C184  AD 3F 03        LDA XPLT
00139  C187  29 07           AND #$07
00140  C189  49 07           EOR #$07
00141  C18B  AA              TAX
00142  C18C  A9 01           LDA #$01
00143  C18E  CA        FLOTLP DEX
00144  C18F  30 03           BMI PLOT2
00145  C191  0A              ASL A
00146  C192  D0 FA           BNE PLOTLP
00147  C194  A0 00     PLOT2 LDY #0
00148  C196  2C 3E 03        BIT PLTFLG
00149  C199  10 05           BPL NOPLOT
00150  C19B  49 FF           EOR #$FF
00151  C19D  31 FB           AND ($FB),Y
00152  C19F  2C              .BYTE $2C
00153  C1A0  11 FB     NOPLOT ORA ($FB),Y
00154  C1A2  91 FB           STA ($FB),Y
00155  C1A4  A5 01           LDA $01          ; BASIC ROM RESTORED
00156  C1A6  09 01           ORA #$01
00157  C1A8  85 01           STA $01
00158  C1AA  60              RTS
00159  C1AB            ;
00160  C1AB  00        TABLE1 .BYTE $00,$40,$80,$C0
00160  C1AC  40
00160  C1AD  80
00160  C1AE  C0
00161  C1AF  00              .BYTE $00,$40,$80,$C0
00161  C1B0  40
00161  C1B1  80
00161  C1B2  C0
00162  C1B3  00              .BYTE $00,$40,$80,$C0
00162  C1B4  40
00162  C1B5  80
00162  C1B6  C0
00163  C1B7  00              .BYTE $00,$40,$80,$C0
00163  C1B8  40
00163  C1B9  80
00163  C1BA  C0
00164  C1BB  00              .BYTE $00,$40,$80,$C0
00164  C1BC  40
00164  C1BD  80
00164  C1BE  C0
00165  C1BF  00              .BYTE $00,$40,$80,$C0,$00
00165  C1C0  40
00165  C1C1  80
00165  C1C2  C0
00165  C1C3  00
00166  C1C4            ;
00167  C1C4  00        TABLE2 .BYTE $00,$01,$02,$03
00167  C1C5  01
00167  C1C6  02
00167  C1C7  03
00168  C1C8  05              .BYTE $05,$06,$07,$08
00168  C1C9  06
00168  C1CA  07
00168  C1CB  08
00169  C1CC  0A              .BYTE $0A,$0B,$0C,$0D
00169  C1CD  0B
00169  C1CE  0C
00169  C1CF  0D
00170  C1D0  0F              .BYTE $0F,$10,$11,$12
00170  C1D1  10
00170  C1D2  11
00170  C1D3  12
00171  C1D4  14              .BYTE $14,$15,$16,$17
00171  C1D5  15
00171  C1D6  16
00171  C1D7  17
00172  C1D8  19              .BYTE $19,$1A,$1B,$1C,$1E
00172  C1D9  1A
00172  C1DA  1B
00172  C1DB  1C
00172  C1DC  1E
00173  C1DD            ;
00174  C1DD            ;*****************************************
00175  C1DD            ;
00176  C1DD            ; SHADING BY HYBRID DITHER/DOT-GROWTH
00177  C1DD            ;
00178  C1DD  AD 3F 03  SHADE  LDA XPLT         ; USE BITS -----***
00179  C1E0  29 07            AND #$07         ; OF 'X' SCREEN COORD
00180  C1E2  8D 48 03         STA TEMP
00181  C1E5  AD 41 03         LDA YPLT         ; AND BITS -----***
00182  C1E8  29 07            AND #$07         ; OF 'Y' SCREEN COORD
00183  C1EA  0A               ASL A            ; SHIFTED INTO --***---
00184  C1EB  0A               ASL A            ; POSITION TO DETERMINE
00185  C1EC  0A               ASL A            ; 6-BIT OFFSET IN
00186  C1ED  0D 48 03         ORA TEMP         ; THRESHOLD TABLE
00187  C1F0  AA               TAX
00188  C1F1  BD 2F C2         LDA THRESH,X     ; SCREEN-POSITION-WEIGHTED
00189  C1F4  CD 44 03         CMP VALUE        ; THRESHOLD VALUE
00190  C1F7  10 03            BPL GREATR
00191  C1F9  4C 46 C1         JMP UNPLOT
00192  C1FC  4C 43 C1  GREATR JMP PLOT
00193  C1FF            ;
00194  C1FF            ;*****************************************
00195  C1FF            ;
00196  C1FF            ; SHADING BY RANDOM HALFTONE
00197  C1FF            ;
00198  C1FF  20 C8 C0  RSHADE JSR RANDOM
00199  C202  4A               LSR A            ; REDUCE RANDOM BYTE
00200  C203  4A               LSR A            ; TO 6 BITS FOR SHADE
00201  C204  CD 44 03         CMP VALUE        ; VALUE COMPARISON
00202  C207  10 03            BPL MORE
00203  C209  4C 46 C1         JMP UNPLOT
00204  C20C  4C 43 C1  MORE   JMP PLOT
00205  C20F            ;
00206  C20F            ;*****************************************
00207  C20F            ;
00208  C20F            ; PLOT A POINT WEIGHTED BY SHADING SCHEME
00209  C20F            ; AND SHADE VALUE
00210  C20F            ; CHECK 'NOSCAL' FLAG FOR SCALING OF Y COORD
00211  C20F            ; CHECK 'HTORRN' FLAG FOR TYPE OF SHADING
00212  C20F            ;
00213  C20F  AD 47 03  PLTSHD LDA NOSCAL
00214  C212  F0 10            BEQ NORM
00215  C214            ;
00216  C214            ; SCALE Y FROM 0-239 PSEUDO-COORDINATES
00217  C214            ; TO 0-199 TRUE SCREEN COORDINATES BY
00218  C214            ;    Y = (Y+1)*213/256
00219  C214            ;
00220  C214  AC 41 03  SCALE  LDY YPLT
00221  C217  C8               INY
00222  C218  84 AD            STY MLPLER
00223  C21A  A9 D5            LDA #$D5         ; 213.
00224  C21C  85 AC            STA MLPCND
00225  C21E  20 11 C0         JSR MULT         ; RETURN WITH HIGH BYTE
00226  C221  8D 41 03         STA YPLT         ; IN ACCUMULATOR
00227  C224  AD 46 03  NORM   LDA HTORRN
00228  C227  F0 03            BEQ RPLT
00229  C229  4C DD C1         JMP SHADE
00230  C22C  4C FF C1  RPLT   JMP RSHADE
00231  C22F            ;
00232  C22F            ;
00233  C22F            ;
00234  C22F  00        THRESH .BYTE $00,$08,$35,$3D
00234  C230  08
00234  C231  35
00234  C232  3D
00235  C233  02              .BYTE $02,$0A,$37,$3F
00235  C234  0A
00235  C235  37
00235  C236  3F
00236  C237  10              .BYTE $10,$18,$25,$2D
00236  C238  18
00236  C239  25
00236  C23A  2D
00237  C23B  12              .BYTE $12,$1A,$27,$2F
00237  C23C  1A
00237  C23D  27
00237  C23E  2F
00238  C23F  31              .BYTE $31,$39,$04,$0C
00238  C240  39
00238  C241  04
00238  C242  0C
00239  C243  33              .BYTE $33,$3B,$06,$0E
00239  C244  3B
00239  C245  06
00239  C246  0E
00240  C247  21              .BYTE $21,$29,$14,$1C
00240  C248  29
00240  C249  14
00240  C24A  1C
00241  C24B  23              .BYTE $23,$2B,$16,$1E
00241  C24C  2B
00241  C24D  16
00241  C24E  1E
00242  C24F  03              .BYTE $03,$0B,$36,$3E
00242  C250  0B
00242  C251  36
00242  C252  3E
00243  C253  01              .BYTE $01,$09,$34,$3C
00243  C254  09
00243  C255  34
00243  C256  3C
00244  C257  13              .BYTE $13,$1B,$26,$2E
00244  C258  1B
00244  C259  26
00244  C25A  2E
00245  C25B  11              .BYTE $11,$19,$24,$2C
00245  C25C  19
00245  C25D  24
00245  C25E  2C
00246  C25F  32              .BYTE $32,$3A,$07,$0F
00246  C260  3A
00246  C261  07
00246  C262  0F
00247  C263  30              .BYTE $30,$38,$05,$0D
00247  C264  38
00247  C265  05
00247  C266  0D
00248  C267  22              .BYTE $22,$2A,$17,$1F
00248  C268  2A
00248  C269  17
00248  C26A  1F
00249  C26B  20              .BYTE $20,$28,$15,$1D
00249  C26C  28
00249  C26D  15
00249  C26E  1D
00250  C26F                  .END

ERRORS = 00000


SYMBOL TABLE

SYMBOL VALUE
 CLEAR  C12C    CLRLP  C138    COL1   C11C    COLOR  C118
 GREATR C1FC    GRFOFF C103    GRFON  C0E2    HTORRN 0346
 MLPCND 00AC    MLPLER 00AD    MORE   C20C    MULT   C011
 NOPLOT C1A0    NORM   C224    NOSCAL 0347    ORIGIN C0E2
 PLOT   C143    PLOT2  C194    PLOTLP C1BE    PLTFLG 033E
 PLTSHD C20F    PROD   00AE    RAM    033E    RANDOM C0C8
 RNDM   C000    RPLT   C22C    RSHADE C1FF    SCALE  C214
 SHADE  C1DD    TABLE1 C1AB    TABLE2 C1C4    TEMP   0348
 THRESH C22F    UNFLOT C146    VALUE  0344    VIC1   0342
 VIC2   0343    XPLT   033F    YPLT   0341

END OF ASSEMBLY
```

End Listing Two

Listing Three

```
00001  0000            ; FACET - DRAW SHADED TRIANGULAR FACETS
00002  0000            ; AND STRAIGHT LINES.
00003  0000            ;
00004  0000            ; RICHARD L. RYLANDER   11/4/84
00005  0000            ;
00006  0000            ; LOAD "ARITH.HEX" AND "GRAPH.HEX"
00007  0000            ; BEFORE USING
00008  0000            ;
00009  0000            ORIGIN = $C26F
00010  0000            RAM    = $034A
00011  0000            ;
00012  0000            XPLT   = $033F
00013  0000            YPLT   = $0341
00014  0000            NORM   = $C224
00015  0000            NOSCAL = $0347
00016  0000            PLOT   = $C143
00017  0000            UNPLOT = $C146
```

(Continued on page 66)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Three

```
00018  0000             ;
00019  0000             MLPCND = $AC
00020  0000             MLPLER = $AD
00021  0000             PROD   = $AE
00022  0000             MULT   = $C011
00023  0000             ;
00024  0000             DVDND  = $FD
00025  0000             DVSOR  = $FB
00026  0000             QUOT   = $FD
00027  0000             DIVIDE = $C025
00028  0000             ;
00029  0000                    *=RAM
00030  034A             ;
00031  034A             XMIN   *=*+2
00032  034C             YMIN   *=*+1
00033  034D             XMID   *=*+2
00034  034F             YMID   *=*+1
00035  0350             XMAX   *=*+2
00036  0352             YMAX   *=*+1
00037  0353             YTOP   *=*+1
00038  0354             YBOT   *=*+1
00039  0355             YBASE  *=*+1
00040  0356             DLTAX1 *=*+2
00041  0358             DLTAX2 *=*+1
00042  0359             DLTAX3 *=*+1
00043  035A             DELTAX *=*+1
00044  035B             DLTAY1 *=*+1
00045  035C             DLTAY2 *=*+1
00046  035D             DLTAY3 *=*+1
00047  035E             DELTAY *=*+1
00048  035F             XDIFF  *=*+1
00049  0360             FLAG1  *=*+1
00050  0361             FLAG2  *=*+1
00051  0362             FLAG3  *=*+1
00052  0363             FLAG   *=*+1
00053  0364             EDGES  *=*+1
00054  0365             ERROR  *=*+2
00055  0367             MODE   *=*+1
00056  0368             COUNT  *=*+2
00057  036A             ;
00058  036A             ;
00059  036A                    *=ORIGIN
00060  C26F             ;
00061  C26F             ;*****************************************
00062  C26F             ;
00063  C26F             ; SCALE ALL Y COORDINATES FROM 0..239
00064  C26F             ; PSUEDO-COORDINATE RANGE TO 0..199
00065  C26F             ; TRUE SCREEN COORDINATE RANGE
00066  C26F             ;
00067  C26F A0 06       SCALE  LDY #6
00068  C271 A9 D5              LDA #$D5
00069  C273 85 AC              STA MLPCND
00070  C275 B9 4C 03    SCLP   LDA YMIN,Y
00071  C278 85 AD              STA MLPLER
00072  C27A 20 11 C0           JSR MULT
00073  C27D 99 4C 03           STA YMIN,Y
00074  C280 88                 DEY
00075  C281 88                 DEY
00076  C282 88                 DEY
00077  C283 10 F0              BPL SCLP
00078  C285 60                 RTS
00079  C286             ;
00080  C286             ;
00081  C286             ;*****************************************
00082  C286             ;
00083  C286             ; EXCHANGE 'MIN' AND 'MID' COORDINATES
00084  C286             ;
00085  C286 A0 02       SWAP12 LDY #2
00086  C288 B9 4A 03    LOOP1  LDA XMIN,Y
00087  C28B 48                 PHA
00088  C28C B9 4D 03           LDA XMID,Y
00089  C28F 99 4A 03           STA XMIN,Y
00090  C292 68                 PLA
00091  C293 99 4D 03           STA XMID,Y
00092  C296 88                 DEY
00093  C297 10 EF              BPL LOOP1
00094  C299 60                 RTS
00095  C29A             ;
00096  C29A             ;*****************************************
00097  C29A             ;
00098  C29A             ; EXCHANGE 'MID' AND 'MAX' COORDINATES
00099  C29A             ;
00100  C29A A0 02       SWAP23 LDY #2
00101  C29C B9 4D 03    LOOP2  LDA XMID,Y
00102  C29F 48                 PHA
00103  C2A0 B9 50 03           LDA XMAX,Y
00104  C2A3 99 4D 03           STA XMID,Y
00105  C2A6 68                 PLA
00106  C2A7 99 50 03           STA XMAX,Y
00107  C2AA 88                 DEY
00108  C2AB 10 EF              BPL LOOP2
00109  C2AD 60                 RTS
00110  C2AE             ;
00111  C2AE             ;*****************************************
00112  C2AE             ;
00113  C2AE             ; SORT COORDINATES ACCORDING TO X COMPONENTS
00114  C2AE             ;
00115  C2AE A2 02       SORTX  LDX #2
00116  C2B0 38          SORTLP SEC
00117  C2B1 AD 4D 03           LDA XMID
00118  C2B4 ED 4A 03           SBC XMIN
00119  C2B7 AD 4E 03           LDA XMID+1
00120  C2BA ED 4B 03           SBC XMIN+1
00121  C2BD B0 03              BCS NOSWP1
00122  C2BF 20 B6 C2           JSR SWAP12
00123  C2C2 CA          NOSWP1 DEX
00124  C2C3 F0 15              BEQ SORTED
00125  C2C5 38                 SEC
00126  C2C6 AD 50 03           LDA XMAX
00127  C2C9 ED 4D 03           SBC XMID
00128  C2CC AD 51 03           LDA XMAX+1
00129  C2CF ED 4E 03           SBC XMID+1
00130  C2D2 B0 DC              BCS SORTLP
00131  C2D4 20 9A C2           JSR SWAP23
00132  C2D7 4C B0 C2           JMP SORTLP
00133  C2DA 60          SORTED RTS
00134  C2DB             ;
00135  C2DB             ;*****************************************
00136  C2DB             ;
00137  C2DB             ; DRAW A LINE BETWEEN XMIN,YMIN AND XMID,YMID
00138  C2DB             ; USING FAST DDA (DIGITAL DIFFERENTIAL ANALYZER)
00139  C2DB             ; TECHNIQUE
00140  C2DB             ;
00141  C2DB A9 02       LINE   LDA #2          ; ENSURE XMAX IS
00142  C2DD 8D 51 03           STA XMAX+1      ; LARGEST BEFORE
00143  C2E0 20 AE C2           JSR SORTX       ; ORDERING 'MIN' AND 'MID'
00144  C2E3 AD 47 03           LDA NOSCAL
00145  C2E6 F0 03              BEQ OUTLN
00146  C2E8 20 6F C2           JSR SCALE
00147  C2EB 20 6F C4    OUTLN  JSR FINDXY      ; ENTRY POINT TO
00148  C2EE AD 4A 03           LDA XMIN        ; OUTLINE FACETS
00149  C2F1 8D 3F 03           STA XPLT
00150  C2F4 AD 4B 03           LDA XMIN+1
00151  C2F7 8D 40 03           STA XPLT+1
00152  C2FA AD 4C 03           LDA YMIN
00153  C2FD 8D 41 03           STA YPLT
00154  C300 AD 57 03           LDA DLTAX1+1    ; CHECK FOR DX>DY
00155  C303 D0 7D              BNE STEPX
00156  C305 38                 SEC
00157  C306 AD 56 03           LDA DLTAX1
00158  C309 ED 5B 03           SBC DLTAY1
00159  C30C B0 74              BCS STEPX
00160  C30E AD 5B 03    STEPY  LDA DLTAY1
00161  C311 8D 65 03           STA ERROR
00162  C314 8D 68 03           STA COUNT
00163  C317 4E 65 03           LSR ERROR
00164  C31A 38                 SEC
00165  C31B AD 56 03           LDA DLTAX1
00166  C31E ED 65 03           SBC ERROR
00167  C321 8D 65 03           STA ERROR
00168  C324 AD 57 03           LDA DLTAX1+1
00169  C327 E9 00              SBC #0
00170  C329 8D 66 03           STA ERROR+1
00171  C32C EE 68 03           INC COUNT
00172  C32F AD 67 03    LNLP1  LDA MODE        ; 0 = DRAW, 1 = ERASE
00173  C332 D0 06              BNE ERASE1
00174  C334 20 43 C1           JSR PLOT
00175  C337 4C 3D C3           JMP SK1
00176  C33A 20 46 C1    ERASE1 JSR UNPLOT
00177  C33D AD 60 03    SK1    LDA FLAG1       ; 0 = POSITIVE SLOPE
00178  C340 D0 05              BNE NSLOPE
00179  C342 EE 41 03           INC YPLT
00180  C345 D0 03              BNE SK2         ; ALWAYS BRANCH
00181  C347 CE 41 03    NSLOPE DEC YPLT
00182  C34A 2C 66 03    SK2    BIT ERROR+1
00183  C34D 30 1A              BMI SK3
00184  C34F EE 3F 03           INC XPLT
00185  C352 D0 03              BNE NOINC1
00186  C354 EE 40 03           INC XPLT+1
00187  C357 38          NOINC1 SEC
00188  C358 AD 65 03           LDA ERROR
00189  C35B ED 5B 03           SBC DLTAY1
00190  C35E 8D 65 03           STA ERROR
00191  C361 AD 66 03           LDA ERROR+1
00192  C364 E9 00              SBC #0
00193  C366 8D 66 03           STA ERROR+1
00194  C369 18          SK3    CLC
00195  C36A AD 65 03           LDA ERROR
00196  C36D 6D 56 03           ADC DLTAX1
00197  C370 8D 65 03           STA ERROR
00198  C373 AD 66 03           LDA ERROR+1
00199  C376 6D 57 03           ADC DLTAX1+1
00200  C379 8D 66 03           STA ERROR+1
00201  C37C CE 68 03           DEC COUNT
00202  C37F D0 AE              BNE LNLP1
00203  C381 60                 RTS
00204  C382             ;
00205  C382 AD 56 03    STEPX  LDA DLTAX1
00206  C385 8D 65 03           STA ERROR
00207  C388 8D 68 03           STA COUNT
00208  C38B AD 57 03           LDA DLTAX1+1
00209  C38E 8D 66 03           STA ERROR+1
00210  C391 8D 69 03           STA COUNT+1
00211  C394 4E 66 03           LSR ERROR+1
00212  C397 6E 65 03           ROR ERROR
00213  C39A 38                 SEC
00214  C39B AD 5B 03           LDA DLTAY1
00215  C39E ED 65 03           SBC ERROR
00216  C3A1 8D 65 03           STA ERROR
00217  C3A4 A9 00              LDA #0
00218  C3A6 ED 66 03           SBC ERROR+1
00219  C3A9 8D 66 03           STA ERROR+1
00220  C3AC AD 67 03    LNLP2  LDA MODE
00221  C3AF D0 06              BNE ERASE2
00222  C3B1 20 43 C1           JSR PLOT
00223  C3B4 4C BA C3           JMP SKP1
00224  C3B7 20 46 C1    ERASE2 JSR UNPLOT
00225  C3BA EE 3F 03    SKP1   INC XPLT
00226  C3BD D0 03              BNE NOINC2
00227  C3BF EE 40 03           INC XPLT+1
00228  C3C2 2C 66 03    NOINC2 BIT ERROR+1
00229  C3C5 30 20              BMI SKP3
00230  C3C7 AD 60 03           LDA FLAG1
00231  C3CA D0 05              BNE NGSLP
00232  C3CC EE 41 03           INC YPLT
00233  C3CF D0 03              BNE SKP2        ; ALWAYS BRANCH
00234  C3D1 CE 41 03    NGSLP  DEC YPLT
00235  C3D4 38          SKP2   SEC
00236  C3D5 AD 65 03           LDA ERROR
00237  C3D8 ED 56 03           SBC DLTAX1
00238  C3DB 8D 65 03           STA ERROR
00239  C3DE AD 66 03           LDA ERROR+1
00240  C3E1 ED 57 03           SBC DLTAX1+1
00241  C3E4 8D 66 03           STA ERROR+1
00242  C3E7 18          SKP3   CLC
00243  C3E8 AD 65 03           LDA ERROR
00244  C3EB 6D 5B 03           ADC DLTAY1
00245  C3EE 8D 65 03           STA ERROR
00246  C3F1 AD 66 03           LDA ERROR+1
00247  C3F4 69 00              ADC #0
00248  C3F6 8D 66 03           STA ERROR+1
00249  C3F9 38                 SEC
00250  C3FA AD 68 03           LDA COUNT
00251  C3FD E9 01              SBC #1
00252  C3FF 8D 68 03           STA COUNT
00253  C402 B0 03              BCS TEST
00254  C404 CE 69 03           DEC COUNT+1
00255  C407 2C 69 03    TEST   BIT COUNT+1
00256  C40A 10 A0              BPL LNLP2
00257  C40C 60                 RTS
00258  C40D             ;
00259  C40D             ;*****************************************
00260  C40D             ; DRAW A SHADED VERTICAL LINE AT
00261  C40D             ; XPLT FROM YTOP TO YBOT
00262  C40D             ;
00263  C40D 38          VLINE  SEC             ; MAKE SURE YTOP>YBOT
00264  C40E AD 53 03           LDA YTOP
00265  C411 ED 54 03           SBC YBOT
00266  C414 B0 0E              BCS DRAW
00267  C416 AD 53 03           LDA YTOP
00268  C419 48                 PHA
00269  C41A AD 54 03           LDA YBOT
00270  C41D 8D 53 03           STA YTOP
00271  C420 68                 PLA
00272  C421 8D 54 03           STA YBOT
```

(Continued on page 68)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Three

```
00273  C424  AD 53 03   DRAW   LDA YTOP
00274  C427  8D 41 03          STA YPLT
00275  C42A  20 24 C2          JSR NORM        ; PLOT A SHADE-WEIGHTED
00276  C42D  AD 53 03          LDA YTOP        ; PIXEL CHECKING ONLY
00277  C430  CD 54 03          CMP YBOT        ; FOR SHADE STYLE
00278  C433  F0 06             BEQ DONE
00279  C435  CE 53 03          DEC YTOP
00280  C438  4C 24 C4          JMP DRAW
00281  C43B  60                DONE   RTS
00282  C43C                    ;
00283  C43C                    ;****************************************
00284  C43C                    ;
00285  C43C                    ; FIND ENDPOINTS FOR VERTICAL LINES
00286  C43C                    ; BETWEEN FACET EDGES
00287  C43C                    ;
00288  C43C  AD 5F 03   ENDPTS LDA XDIFF
00289  C43F  85 AC             STA MLPCND
00290  C441  AD 5E 03          LDA DELTAY
00291  C444  85 AD             STA MLPLER
00292  C446  20 11 C0          JSR MULT
00293  C449  85 FE             STA DVDND+1
00294  C44B  A5 AE             LDA PROD
00295  C44D  85 FD             STA DVDND
00296  C44F  A9 00             LDA #0
00297  C451  85 FC             STA DVSOR+1
00298  C453  AD 5A 03          LDA DELTAX
00299  C456  85 FB             STA DVSOR
00300  C458  20 25 C0          JSR DIVIDE
00301  C45B  AD 63 03          LDA FLAG
00302  C45E  D0 08             BNE NEGSLP
00303  C460  18                CLC
00304  C461  AD 55 03          LDA YBASE
00305  C464  65 FD             ADC QUOT
00306  C466  90 06             BCC SKIP2
00307  C468  38                NEGSLP SEC
00308  C469  AD 55 03          LDA YBASE
00309  C46C  E5 FD             SBC QUOT
00310  C46E  60                SKIP2  RTS
00311  C46F                    ;
00312  C46F                    ;****************************************
00313  C46F                    ;
00314  C46F                    ; FIND COORDINATE DIFFERENCES
00315  C46F                    ;
00316  C46F                    ; ALL "DELTA X" VALUES POSITIVE,
00317  C46F                    ; SINGLE PRECISION (JUST LOWER BYTE)
00318  C46F                    ;
00319  C46F  38                FINDXY SEC
00320  C470  AD 4D 03          LDA XMID
00321  C473  ED 4A 03          SBC XMIN
00322  C476  8D 56 03          STA DLTAX1
00323  C479  AD 4E 03          LDA XMID+1
00324  C47C  ED 4B 03          SBC XMIN+1
00325  C47F  8D 57 03          STA DLTAX1+1
00326  C482  38                SEC
00327  C483  AD 50 03          LDA XMAX
00328  C486  ED 4D 03          SBC XMID
00329  C489  8D 58 03          STA DLTAX2
00330  C48C  38                SEC
00331  C48D  AD 50 03          LDA XMAX
00332  C490  ED 4A 03          SBC XMIN
00333  C493  8D 59 03          STA DLTAX3
00334  C496                    ;
00335  C496                    ; USE ABS(DELTA Y) VALUES,
00336  C496                    ; FLAGS INDICATE SLOPE OF LIMIT LINES
00337  C496                    ;
00338  C496  A9 00             LDA #$00
00339  C498  8D 60 03          STA FLAG1
00340  C49B  8D 61 03          STA FLAG2
00341  C49E  8D 62 03          STA FLAG3
00342  C4A1  38                SEC
00343  C4A2  AD 4F 03          LDA YMID
00344  C4A5  ED 4C 03          SBC YMIN
00345  C4A8  B0 09             BCS STORE1
00346  C4AA  EE 60 03          INC FLAG1
00347  C4AD  AD 4C 03          LDA YMIN
00348  C4B0  ED 4F 03          SBC YMID
00349  C4B3  8D 5B 03   STORE1 STA DLTAY1
00350  C4B6  38                SEC
00351  C4B7  AD 52 03          LDA YMAX
00352  C4BA  ED 4F 03          SBC YMID
00353  C4BD  B0 09             BCS STORE2
00354  C4BF  EE 61 03          INC FLAG2
00355  C4C2  AD 4F 03          LDA YMID
00356  C4C5  ED 52 03          SBC YMAX
00357  C4C8  8D 5C 03   STORE2 STA DLTAY2
00358  C4CB  38                SEC
00359  C4CC  AD 52 03          LDA YMAX
00360  C4CF  ED 4C 03          SBC YMIN
00361  C4D2  B0 09             BCS STORE3
00362  C4D4  EE 62 03          INC FLAG3
00363  C4D7  AD 4C 03          LDA YMIN
00364  C4DA  ED 52 03          SBC YMAX
00365  C4DD  8D 5D 03   STORE3 STA DLTAY3
00366  C4E0  60                RTS
00367  C4E1                    ;
00368  C4E1                    ;****************************************
00369  C4E1                    ;
00370  C4E1                    ; DRAW A SHADED TRIANGULAR FACET
00371  C4E1                    ;
00372  C4E1  20 AE C2   FACET  JSR SORTX
00373  C4E4  AD 47 03          LDA NOSCAL
00374  C4E7  F0 03             BEQ YSOK
00375  C4E9  20 6F C2          JSR SCALE
00376  C4EC  20 6F C4   YSOK   JSR FINDXY
00377  C4EF  AD 4A 03          LDA XMIN
00378  C4F2  8D 3F 03          STA XPLT
00379  C4F5  AD 4B 03          LDA XMIN+1
00380  C4F8  8D 40 03          STA XPLT+1
00381  C4FB  38                FCETLP SEC
00382  C4FC  AD 3F 03          LDA XPLT
00383  C4FF  ED 4A 03          SBC XMIN
00384  C502  8D 5F 03          STA XDIFF
00385  C505  AD 56 03          LDA DLTAX1
00386  C508  F0 53             BEQ CONT
00387  C50A  8D 5A 03          STA DELTAX
00388  C50D  AD 5B 03          LDA DLTAY1
00389  C510  8D 5E 03          STA DELTAY
00390  C513  AD 60 03          LDA FLAG1
00391  C516  8D 63 03          STA FLAG
00392  C519  AD 4C 03          LDA YMIN
00393  C51C  8D 55 03          STA YBASE
00394  C51F  20 3C C4          JSR ENDPTS
00395  C522  8D 53 03          STA YTOP
00396  C525  AD 59 03          LDA DLTAX3
00397  C528  F0 33             BEQ CONT
00398  C52A  8D 5A 03          STA DELTAX
00399  C52D  AD 5D 03          LDA DLTAY3
00400  C530  8D 5E 03          STA DELTAY
00401  C533  AD 62 03          LDA FLAG3
00402  C536  8D 63 03          STA FLAG
00403  C539  20 3C C4          JSR ENDPTS
00404  C53C  8D 54 03          STA YBOT
00405  C53F  20 0D C4          JSR VLINE
00406  C542  AD 40 03          LDA XPLT+1
00407  C545  CD 4E 03          CMP XMID+1
00408  C548  D0 08             BNE NEXTX1
00409  C54A  AD 3F 03          LDA XPLT
00410  C54D  CD 4D 03          CMP XMID
00411  C550  F0 0B             BEQ CONT
00412  C552  EE 3F 03   NEXTX1 INC XPLT
00413  C555  D0 03             BNE SKIP3
00414  C557  EE 40 03          INC XPLT+1
00415  C55A  4C FB C4   SKIP3  JMP FCETLP
00416  C55D  38                CONT   SEC
00417  C55E  AD 3F 03          LDA XPLT
00418  C561  ED 4A 03          SBC XMIN
00419  C564  8D 5F 03          STA XDIFF
00420  C567  AD 59 03          LDA DLTAX3
00421  C56A  F0 63             BEQ FINI
00422  C56C  8D 5A 03          STA DELTAX
00423  C56F  AD 5D 03          LDA DLTAY3
00424  C572  8D 5E 03          STA DELTAY
00425  C575  AD 62 03          LDA FLAG3
00426  C578  8D 63 03          STA FLAG
00427  C57B  AD 4C 03          LDA YMIN
00428  C57E  8D 55 03          STA YBASE
00429  C581  20 3C C4          JSR ENDPTS
00430  C584  8D 54 03          STA YBOT
00431  C587  38                SEC
00432  C588  AD 3F 03          LDA XPLT
00433  C58B  ED 4D 03          SBC XMID
00434  C58E  8D 5F 03          STA XDIFF
00435  C591  AD 58 03          LDA DLTAX2
00436  C594  F0 39             BEQ FINI
00437  C596  8D 5A 03          STA DELTAX
00438  C599  AD 5C 03          LDA DLTAY2
00439  C59C  8D 5E 03          STA DELTAY
00440  C59F  AD 61 03          LDA FLAG2
00441  C5A2  8D 63 03          STA FLAG
00442  C5A5  AD 4F 03          LDA YMID
00443  C5A8  8D 55 03          STA YBASE
00444  C5AB  20 3C C4          JSR ENDPTS
00445  C5AE  8D 53 03          STA YTOP
00446  C5B1  20 0D C4          JSR VLINE
00447  C5B4  AD 40 03          LDA XPLT+1
00448  C5B7  CD 51 03          CMP XMAX+1
00449  C5BA  D0 08             BNE NEXTX2
00450  C5BC  AD 3F 03          LDA XPLT
00451  C5BF  CD 50 03          CMP XMAX
00452  C5C2  F0 0B             BEQ FINI
00453  C5C4  EE 3F 03   NEXTX2 INC XPLT
00454  C5C7  D0 03             BNE SKIP4
00455  C5C9  EE 40 03          INC XPLT+1
00456  C5CC  4C 5D C5   SKIP4  JMP CONT
00457  C5CF  AD 64 03   FINI   LDA EDGES
00458  C5D2  F0 15             BEQ FINISH
00459  C5D4  20 EB C2          JSR OUTLN
00460  C5D7  20 9A C2          JSR SWAP23
00461  C5DA  20 EB C2          JSR OUTLN
00462  C5DD  20 B6 C2          JSR SWAP12
00463  C5E0  20 9A C2          JSR SWAP23
00464  C5E3  20 B6 C2          JSR SWAP12
00465  C5E6  20 EB C2          JSR OUTLN
00466  C5E9  60                FINISH RTS
00467  C5EA                    .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 CONT    C55D   COUNT   0368   DELTAX  035A   DELTAY  035E
 DIVIDE  C025   DLTAX1  0356   DLTAX2  0358   DLTAX3  0359
 DLTAY1  035B   DLTAY2  035C   DLTAY3  035D   DONE    C43B
 DRAW    C424   DVDND   00FD   DVSOR   00FB   EDGES   0364
 ENDPTS  C43C   ERASE1  C33A   ERASE2  C3B7   ERROR   0365
 FACET   C4E1   FCETLP  C4FB   FINDXY  C46F   FINI    C5CF
 FINISH  C5E9   FLAG    0363   FLAG1   0360   FLAG2   0361
 FLAG3   0362   LINE    C2DB   LNLP1   C32F   LNLP2   C3AC
 LOOP1   C288   LOOP2   C29C   MLPCND  00AC   MLPLER  00AD
 MODE    0367   MULT    C011   NEGSLP  C468   NEXTX1  C552
 NEXTX2  C5C4   NGSLP   C3D1   NOINC1  C357   NOINC2  C3C2
 NORM    C224   NOSCAL  0347   NOSWP1  C2C2   NSLOPE  C347
 ORIGIN  C26F   OUTLN   C2EB   PLOT    C143   PROD    00AE
 QUOT    00FD   RAM     034A   SCALE   C26F   SCLP    C275
 SK1     C33D   SK2     C34A   SK3     C369   SKIP2   C46E
 SKIP3   C55A   SKIP4   C5CC   SKP1    C3BA   SKP2    C3D4
 SKP3    C3E7   SORTED  C2DA   SORTLP  C2B0   SORTX   C2AE
 STEPX   C382   STEPY   C30E   STORE1  C4B3   STORE2  C4C8
 STORE3  C4DD   SWAP12  C2B6   SWAP23  C29A   TEST    C407

 SYMBOL TABLE

 SYMBOL VALUE
  UNPLOT   C146   VLINE   C40D   XDIFF   035F   XMAX   0350
  XMID     034D   XMIN    034A   XPLT    033F   YBASE  0355
  YBOT     0354   YMAX    0352   YMID    034F   YMIN   034C
  YPLT     0341   YSOK    C4EC   YTOP    0353

 END OF ASSEMBLY
```

End Listing Three

Listing Four

```
00001  0000                    ; PRIMITIVE SOLID SHAPE DRAWING
00002  0000                    ;
00003  0000                    ; RICHARD L. RYLANDER   11/7/84
00004  0000                    ;
00005  0000                    ; LOAD ARITHMETIC AND GRAPHIC UTILITIES FIRST
00006  0000                    ;
00007  0000                    ;****************************************
00008  0000                    RAM=$036A
00009  0000                    ORIGIN=$C5EA
00010  0000                    ;
00011  0000                    MLPCND=$AC              ; MULTIPLICAND (S)
00012  0000                    MLPLER=$AD              ; MULTIPLIER (S)
00013  0000                    PROD=$AE                ; PRODUCT (D)
00014  0000                    MULT=$C011              ; CALL FOR MULTIPLY
00015  0000                    ;
00016  0000                    DVDND=$FD               ; DIVIDEND (D)
```

(Continued on page 70)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Four

```
00017  0000                DVSOR=$FB               ; DIVISOR (D)
00018  0000                DUOT=$FD                ; QUOTIENT (D)
00019  0000                DIVIDE=$C025            ; CALL FOR DIVIDE
00020  0000                ;
00021  0000                ARG=$AC                 ; ARGUMENT (S)
00022  0000                SQR=$AE                 ; SQUARE OF ARG (D)
00023  0000                SQUARE=$C004            ; CALL FOR SQUARE
00024  0000                ;
00025  0000                RADCND=$AC              ; RADICAND (D)
00026  0000                ROOT=$033C              ; SQUARE ROOT (S)
00027  0000                SQRT=$C064              ; CALL FOR SQRT
00028  0000                ;
00029  0000                RNDM=$C000              ; RANDOM NUMBER
00030  0000                RANDOM=$C0C8            ; CALL FOR RANDOM
00031  0000                ; NOTE - A CALL TO 'RANDOM' LEAVES A RANDOM BYTE
00032  0000                ; IN THE ACCUMULATOR
00033  0000                ;
00034  0000                XPLT=$033F
00035  0000                YPLT=$0341
00036  0000                NORM=$C224
00037  0000                PLTSHD=$C20F
00038  0000                VALUE=$0344             ; FINAL NORMALIZED SHADE VALUE
00039  0000                HTORRN=$0346            ; SHADE FLAG, 1=HALFTONE
00040  0000                NOSCAL=$0347            ; SCALE FLAG, 1=NO SCALE
00041  0000                ;
00042  0000                      *=RAM
00043  036A                XCENT   *=*+2           ; CENTER COORD
00044  036C                XREL    *=*+1           ; RELATIVE (TO CENTER)
00045  036D                XSHD    *=*+2           ; USED IN SHADE CALC
00046  036F                YCENT   *=*+1           ; CENTER COORD
00047  0370                YREL    *=*+1           ; RELATIVE (TO CENTER)
00048  0371                YSHD    *=*+2           ; USED IN SHADE CALC
00049  0373                ZREL    *=*+2           ; RELATIVE (TO CENTER)
00050  0375                ZWX     *=*+2           ; Z WITH X (+ OR -)
00051  0377                ;
00052  0377                RADIUS  *=*+2           ; LOCAL RADIUS OF SURFACE
00053  0379                TONE    *=*+2           ; USED IN SHADE CALC
00054  037B                TNTMP   *=*+2           ; USED IN SHADE CALC
00055  037D                ;
00056  037D                CLIPL   *=*+1           ; LEFT CLIPPING BOUND
00057  037E                CLIPR   *=*+1           ; RIGHT CLIPPING BOUND
00058  037F                CLIPU   *=*+1           ; UP CLIPPING BOUND
00059  0380                CLIPD   *=*+1           ; DOWN CLIPPING BOUND
00060  0381                ;
00061  0381                HEMI    *=*+1           ; PLOTTING HEMISPHERE
00062  0382                ;
00063  0382                BAKLIT  *=*+1           ; BACKLIT FLAG
00064  0383                HVFLAG  *=*+1           ; HORIZONTAL/VERTICAL FLAG
00065  0384                TEMP    *=*+2           ; TEMPORARY STORAGE
00066  0386                CNTX    *=*+1           ; LOOP COUNTER
00067  0387                CNTY    *=*+1           ; LOOP COUNTER
00068  0388                MAX     *=*+1           ; LOOP LIMIT
00069  0389                ;
00070  0389                HLEN    *=*+1           ; HALF LENGTH OF CYLINDERS
00071  038A                RS      *=*+2           ; SQUARE OF TOROID RADIUS
00072  038C                RT      *=*+1           ; TOROID (RING) RADIUS
00073  038D                RC      *=*+1           ; CENTER RADIUS OF TOROID
00074  038E                RO      *=*+1           ; OUTER RADIUS OF TOROID
00075  038F                RI      *=*+1           ; INNER RADIUS OF TOROID
00076  0390                XSQR    *=*+2
00077  0392                XMAX    *=*+1
00078  0393                ;
00079  0393                RO=HLEN
00080  0393                ;
00081  0393                      *=ORIGIN
00082  C5EA                ;****************************************
00083  C5EA                ;
00084  C5EA                ; DIVIDE WITH SINGLE PRECISION DIVISOR
00085  C5EA                ; (USED OFTEN IN SHAPE ROUTINES)
00086  C5EA                ;
00087  C5EA  A9 00         SDIV    LDA #0
00088  C5EC  85 FC                 STA DVSOR+1
00089  C5EE  4C 25 C0              JMP DIVIDE
00090  C5F1                ;
00091  C5F1                ;****************************************
00092  C5F1                ;
00093  C5F1                ; CALCULATE SHADE VALUE (0-63) BY
00094  C5F1                ; MULTIPLYING 'TONE' BY 26 THEN
00095  C5F1                ; DIVIDE RESULT BY RADIUS OF SURFACE
00096  C5F1                ;
00097  C5F1  2C 7A 03      GETVAL  BIT TONE+1
00098  C5F4  10 12                 BPL CNTNU       ; IF 'TONE'<0, THEN
00099  C5F6  AD 82 03              LDA BAKLIT      ; MAKE VALUE 0 OR ABS(TONE)
00100  C5F9  D0 04                 BNE NEGATE      ; DEPENDING ON BAKLIT FLAG
00101  C5FB  8D 44 03              STA VALUE
00102  C5FE  60                   RTS
00103  C5FF  38            NEGATE  SEC
00104  C600  A9 00                 LDA #0
00105  C602  ED 79 03              SBC TONE
00106  C605  8D 79 03              STA TONE
00107  C608  AD 79 03      CNTNU   LDA TONE
00108  C60B  85 AC                 STA MLPCND
00109  C60D  A9 1A                 LDA #$1A
00110  C60F  85 AD                 STA MLPLER
00111  C611  20 11 C0              JSR MULT
00112  C614  85 FE                 STA DVDND+1
00113  C616  A5 AE                 LDA PROD
00114  C618  85 FD                 STA DVDND
00115  C61A  AD 77 03              LDA RADIUS
00116  C61D  85 FB                 STA DVSOR
00117  C61F  20 EA C5              JSR SDIV
00118  C622  A5 FD                 LDA DUOT
00119  C624  8D 44 03              STA VALUE
00120  C627  60                   RTS
00121  C628                ;
00122  C628                ;****************************************
00123  C628                ;
00124  C628                ; POINT PLOTTING BY QUADRANTS USING
00125  C628                ; THE FOUR-FOLD SYMMETRY OF SIMPLE OBJECTS
00126  C628                ;
00127  C628                ; DEPENDING ON STATUS OF 'HVFLAG', EXCHANGE
00128  C628                ; X AND Y COORDINATES TO ROTATE OBJECTS 90 DEG
00129  C628                ; SINGLE SHAPE ROUTINE CAN THEN BE USED TO
00130  C628                ; DRAW 'HORIZONTAL' OR 'VERTICAL' VERSIONS
00131  C628                ; OF AN OBJECT
00132  C628                ;
00133  C628                ; THE FOLOWING IS A 'BASIC SUBROUTINE'
00134  C628                ; EQUIVALENT TO EXPLAIN ITS OPERATION
00135  C628                ;
00136  C628                ; NOTE THAT LABELS ARE USED IN PLACE OF
00137  C628                ; LINE NUMBERS
00138  C628                ;
00139  C628                ; 'PTPLOT' IF HVFLAG<0 THEN GOTO 'NOROT'
00140  C628                ;          (STACK)=XREL:XREL=YREL:YREL=(STACK)
00141  C628                ;          (STACK)=XSHD:XSHD=YSHD:YSHD=(STACK)
00142  C628                ; 'NOROT'  GOSUB 'GETZ'
00143  C628                ;          REM CALCULATE 2*Z FROM X,Y AND RADIUS
00144  C628                ;          HEMI = 1
00145  C628                ;          IF XREL>CLIPL THEN GOTO 'RHEMI'
00146  C628                ;          ZWX=2*Z-XSHD
00147  C628                ;          XPLT=XCENT-XREL:REM LEFT HEMISPHERE
00148  C628                ; 'CHCLUP' IF YREL>CLIPU THEN GOTO 'DHEMI'
00149  C628                ;          TONE=ZWX+YSHD
00150  C628                ;          GOSUB 'GETVAL':REM NORMALIZE SHADE VAL
00151  C628                ;          YPLT=YCENT+YREL
00152  C628                ;          GOSUB 'PLTSHD':REM PLOT OR UNPLOT
00153  C628                ;          REM   POINTS WEIGHTED BY SHADE VALUE
00154  C628                ; 'DHEMI'  IF YREL>CLIPD THEN GOTO 'RHEMI'
00155  C628                ;          TONE=ZWX-YSHD
00156  C628                ;          GOSUB 'GETVAL'
00157  C628                ;          YPLT=YCENT-YREL
00158  C628                ;          GOSUB 'PLTSHD'
00159  C628                ; 'RHEMI'  IF HEMI=0 THEN RETURN
00160  C628                ;          HEMI=0
00161  C628                ;          IF XREL>CLIPR THEN RETURN
00162  C628                ;          ZWX=2*Z+XSHD
00163  C628                ;          XPLT=XCENT+XREL
00164  C628                ;          GOSUB 'CHCLUP'
00165  C628                ;          RETURN
00166  C628                ;
00167  C628  2C 83 03      PTPLOT  BIT HVFLAG
00168  C62B  10 2D                 BPL NOROT
00169  C62D  AD 6C 03              LDA XREL
00170  C630  48                   PHA
00171  C631  48                   PHA
00172  C632  AD 70 03              LDA YREL
00173  C635  8D 6C 03              STA XREL
00174  C638  68                   PLA
00175  C639  8D 70 03              STA YREL
00176  C63C  AD 6D 03              LDA XSHD
00177  C63F  48                   PHA
00178  C640  48                   PHA
00179  C641  AD 71 03              LDA YSHD
00180  C644  8D 6D 03              STA XSHD
00181  C647  68                   PLA
00182  C648  8D 71 03              STA YSHD
00183  C64B  AD 6E 03              LDA XSHD+1
00184  C64E  48                   PHA
00185  C64F  48                   PHA
00186  C650  AD 72 03              LDA YSHD+1
00187  C653  8D 6E 03              STA XSHD+1
00188  C656  68                   PLA
00189  C657  8D 72 03              STA YSHD+1
00190  C65A  20 45 C7      NOROT   JSR GETZ
00191  C65D  A9 01         PTPLT2  LDA #$01
00192  C65F  8D B1 03              STA HEMI
00193  C662  38                   SEC
00194  C663  AD 7D 03              LDA CLIPL       ; CHECK LEFT HEMISPHERE
00195  C666  CD 6C 03              CMP XREL
00196  C669  90 7D                 BCC RHEMI
00197  C66B  38                   SEC
00198  C66C  AD 3C 03              LDA ROOT
00199  C66F  ED 6D 03              SBC XSHD
00200  C672  8D 75 03              STA ZWX
00201  C675  AD 3D 03              LDA ROOT+1
00202  C678  ED 6E 03              SBC XSHD+1
00203  C67B  8D 76 03              STA ZWX+1
00204  C67E  38                   SEC
00205  C67F  AD 6A 03              LDA XCENT
00206  C682  ED 6C 03              SBC XREL
00207  C685  8D 3F 03              STA XPLT
00208  C688  AD 6B 03              LDA XCENT+1
00209  C68B  E9 00                 SBC #$00
00210  C68D  8D 40 03              STA XPLT+1
00211  C690                ;
00212  C690  38            CHCLUP  SEC
00213  C691  AD 7F 03              LDA CLIPU       ; CHECK FOR UP CLIPPING
00214  C694  CD 70 03              CMP YREL
00215  C697  90 23                 BCC DHEMI
00216  C699  18                   CLC
00217  C69A  AD 75 03              LDA ZWX
00218  C69D  6D 71 03              ADC YSHD
00219  C6A0  8D 79 03              STA TONE
00220  C6A3  AD 76 03              LDA ZWX+1
00221  C6A6  6D 72 03              ADC YSHD+1
00222  C6A9  8D 7A 03              STA TONE+1
00223  C6AC  20 F1 C5              JSR GETVAL
00224  C6AF  18                   CLC
00225  C6B0  AD 6F 03              LDA YCENT
00226  C6B3  6D 70 03              ADC YREL
00227  C6B6  8D 41 03              STA YPLT
00228  C6B9  20 0F C2              JSR PLTSHD
00229  C6BC                ;
00230  C6BC  38            DHEMI   SEC
00231  C6BD  AD 80 03              LDA CLIPD       ; CHECK FOR DOWN CLIPPING
00232  C6C0  CD 70 03              CMP YREL
00233  C6C3  90 23                 BCC RHEMI
00234  C6C5  38                   SEC
00235  C6C6  AD 75 03              LDA ZWX
00236  C6C9  ED 71 03              SBC YSHD
00237  C6CC  8D 79 03              STA TONE
00238  C6CF  AD 76 03              LDA ZWX+1
00239  C6D2  ED 72 03              SBC YSHD+1
00240  C6D5  8D 7A 03              STA TONE+1
00241  C6D8  20 F1 C5              JSR GETVAL
00242  C6DB  38                   SEC
00243  C6DC  AD 6F 03              LDA YCENT
00244  C6DF  ED 70 03              SBC YREL
00245  C6E2  8D 41 03              STA YPLT
00246  C6E5  20 0F C2              JSR PLTSHD
00247  C6E8                ;
00248  C6E8  AD B1 03      RHEMI   LDA HEMI
00249  C6EB  F0 34                 BEQ PLDONE
00250  C6ED  CE B1 03              DEC HEMI
00251  C6F0  38                   SEC
00252  C6F1  AD 7E 03              LDA CLIPR       ; CHECK FOR RIGHT CLIPPING
00253  C6F4  CD 6C 03              CMP XREL
00254  C6F7  90 28                 BCC PLDONE
00255  C6F9  18                   CLC
00256  C6FA  AD 3C 03              LDA ROOT
00257  C6FD  6D 6D 03              ADC XSHD
00258  C700  8D 75 03              STA ZWX
00259  C703  AD 3D 03              LDA ROOT+1
00260  C706  6D 6E 03              ADC XSHD+1
00261  C709  8D 76 03              STA ZWX+1
00262  C70C  18                   CLC
00263  C70D  AD 6A 03              LDA XCENT
00264  C710  6D 6C 03              ADC XREL
00265  C713  8D 3F 03              STA XPLT
00266  C716  AD 6B 03              LDA XCENT+1
00267  C719  69 00                 ADC #$00
00268  C71B  8D 40 03              STA XPLT+1
00269  C71E  4C 90 C6              JMP CHCLUP
00270  C721  2C 83 03      PLDONE  BIT HVFLAG
00271  C724  10 1E                 BPL NORSTR
```

(Continued on page 72)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Four

```
00272  C726  AD 6E 03          LDA XSHD+1      ; RESTORE COORDS
00273  C729  8D 72 03          STA YSHD+1
00274  C72C  68                PLA
00275  C72D  8D 6E 03          STA XSHD+1
00276  C730  AD 6D 03          LDA XSHD
00277  C733  8D 71 03          STA YSHD
00278  C736  68                PLA
00279  C737  8D 6D 03          STA XSHD
00280  C73A  AD 6C 03          LDA XREL
00281  C73D  8D 70 03          STA YREL
00282  C740  68                PLA
00283  C741  8D 6C 03          STA XREL
00284  C744  60          NORSTR RTS
00285  C745                    ;
00286  C745                    ;****************************************
00287  C745                    ;
00288  C745                    ; CALCULATE Z FROM LOCAL X,Y BY
00289  C745                    ; PYTHAGOREAN SUM
00290  C745                    ;
00291  C745  AD 77 03    GETZ  LDA RADIUS
00292  C748  85 AC             STA ARG
00293  C74A  20 04 C0          JSR SQUARE
00294  C74D  8D 7C 03          STA TNTMP+1
00295  C750  A5 AE             LDA SQR
00296  C752  8D 7B 03          STA TNTMP
00297  C755  AD 6D 03          LDA XSHD
00298  C758  85 AC             STA ARG
00299  C75A  20 04 C0          JSR SQUARE
00300  C75D  38                SEC
00301  C75E  AD 7B 03          LDA TNTMP
00302  C761  E5 AE             SBC SQR
00303  C763  8D 7B 03          STA TNTMP
00304  C766  AD 7C 03          LDA TNTMP+1
00305  C769  E5 AF             SBC SQR+1
00306  C76B  8D 7C 03          STA TNTMP+1
00307  C76E  AD 71 03          LDA YSHD
00308  C771  85 AC             STA ARG
00309  C773  20 04 C0          JSR SQUARE
00310  C776  38                SEC
00311  C777  AD 7B 03          LDA TNTMP
00312  C77A  E5 AE             SBC SQR
00313  C77C  85 AC             STA RADCND
00314  C77E  AD 7C 03          LDA TNTMP+1
00315  C781  E5 AF             SBC SQR+1
00316  C783  85 AD             STA RADCND+1
00317  C785  30 0A             BMI ZEROOT
00318  C787  20 64 C0          JSR SQRT
00319  C78A  0E 3C 03          ASL ROOT
00320  C78D  2E 3D 03          ROL ROOT+1
00321  C790  60                RTS
00322  C791  A9 00       ZEROOT LDA #$00
00323  C793  8D 3C 03          STA ROOT
00324  C796  8D 3D 03          STA ROOT+1
00325  C799  60                RTS
00326  C79A                    ;
00327  C79A                    ;****************************************
00328  C79A                    ;
00329  C79A                    ; SET UP PARAMETERS FOR TOROIDS
00330  C79A                    ;
00331  C79A                    ; RT=(RO-RI)/2   RS=RT+RT   RC=RT+RI
00332  C79A                    ;
00333  C79A  AD BE 03    TPARM LDA RO
00334  C79D  38                SEC
00335  C79E  ED BF 03          SBC RI
00336  C7A1  4A                LSR A
00337  C7A2  8D BC 03          STA RT
00338  C7A5  8D 77 03          STA RADIUS
00339  C7A8  18                CLC
00340  C7A9  6D BF 03          ADC RI
00341  C7AC  8D BD 03          STA RC
00342  C7AF  AD BC 03          LDA RT
00343  C7B2  85 AC             STA ARG
00344  C7B4  20 04 C0          JSR SQUARE
00345  C7B7  A5 AE             LDA SQR
00346  C7B9  8D BA 03          STA RS
00347  C7BC  A5 AF             LDA SQR+1
00348  C7BE  8D BB 03          STA RS+1
00349  C7C1  A9 00             LDA #0
00350  C7C3  8D B6 03          STA CNTX
00351  C7C6  60                RTS
00352  C7C7                    ;
00353  C7C7                    ;****************************************
00354  C7C7                    ;
00355  C7C7                    ; DRAW A SHADED SPHERE
00356  C7C7                    ;
00357  C7C7                    ; 'BASIC SUBROUTINE' EQUIVALENT
00358  C7C7                    ;
00359  C7C7                    ; 'SPHERE' FOR CNTX=0 TO RADIUS/SQR(2)
00360  C7C7                    ;          XREL=CNTX:XSHD=CNTX
00361  C7C7                    ;          FOR CNTY=CNTX TO SQR(RAD*RAD-CNTX*CNTX)
00362  C7C7                    ;          YREL=CNTY:YSHD=CNTY
00363  C7C7                    ;          HVFLAG=0
00364  C7C7                    ;          GOSUB 'PTPLOT'
00365  C7C7                    ;          REM EXCHANGE X & Y TO USE 8-FOLD SYM
00366  C7C7                    ;          HVFLAG=-128
00367  C7C7                    ;
00368  C7C7                    ;          GOSUB 'PTPLOT'
00369  C7C7                    ;          NEXT CNTY
00370  C7C7                    ;          NEXT CNTX
00371  C7C7                    ;          RETURN
00372  C7C7                    ;
00373  C7C7                    ;
00374  C7C7  AD 77 03    SPHERE LDA RADIUS
00375  C7CA  85 AC             STA ARG
00376  C7CC  20 04 C0          JSR SQUARE
00377  C7CF  06 AE             ASL SQR
00378  C7D1  26 AF             ROL SQR+1
00379  C7D3  A5 AE             LDA SQR
00380  C7D5  85 AC             STA RADCND
00381  C7D7  A5 AF             LDA SQR+1
00382  C7D9  85 AD             STA RADCND+1
00383  C7DB  20 64 C0          JSR SQRT
00384  C7DE  4E 3D 03          LSR ROOT+1
00385  C7E1  6E 3C 03          ROR ROOT
00386  C7E4  AD 3C 03          LDA ROOT
00387  C7E7  8D 92 03          STA XMAX
00388  C7EA  A9 00             LDA #$00
00389  C7EC  8D B6 03          STA CNTX
00390  C7EF  8D 6E 03          STA XSHD+1
00391  C7F2  8D 72 03          STA YSHD+1
00392  C7F5  AD 77 03          LDA RADIUS
00393  C7F8  85 AC             STA ARG
00394  C7FA  20 04 C0          JSR SQUARE
00395  C7FD  8D B5 03          STA TEMP+1
00396  C800  A5 AE             LDA SQR
00397  C802  8D B4 03          STA TEMP
00398  C805  AD B6 03    LOOPX LDA CNTX
00399  C808  8D B7 03          STA CNTY
00400  C80B  85 AC             STA ARG
00401  C80D  8D 6C 03          STA XREL
00402  C810  8D 6D 03          STA XSHD
00403  C813  20 04 C0          JSR SQUARE
00404  C816  38                SEC
00405  C817  AD B4 03          LDA TEMP
00406  C81A  E5 AE             SBC SQR
00407  C81C  85 AC             STA RADCND
00408  C81E  AD B5 03          LDA TEMP+1
00409  C821  E5 AF             SBC SQR+1
00410  C823  85 AD             STA RADCND+1
00411  C825  20 64 C0          JSR SQRT
00412  C828  AD 3C 03          LDA ROOT
00413  C82B  8D B8 03          STA MAX
00414  C82E  AD B7 03    LOOPY LDA CNTY
00415  C831  8D 70 03          STA YREL
00416  C834  8D 71 03          STA YSHD
00417  C837  A9 00             LDA #0
00418  C839  8D B3 03          STA HVFLAG
00419  C83C  20 2B C6          JSR PTPLOT
00420  C83F  A9 80             LDA #$80
00421  C841  8D B3 03          STA HVFLAG
00422  C844  20 2B C6          JSR PTPLOT
00423  C847  AD B7 03          LDA CNTY
00424  C84A  CD B8 03          CMP MAX
00425  C84D  F0 06             BEQ DONEY
00426  C84F  EE B7 03          INC CNTY
00427  C852  4C 2E C8          JMP LOOPY
00428  C855  AD B6 03    DONEY LDA CNTX
00429  C858  CD 92 03          CMP XMAX
00430  C85B  F0 06             BEQ DONE
00431  C85D  EE B6 03          INC CNTX
00432  C860  4C 05 C8          JMP LOOPX
00433  C863  60          DONE  RTS
00434  C864                    ;
00435  C864                    ;****************************************
00436  C864                    ;
00437  C864                    ; DRAW SHADED CYLINDERS
00438  C864                    ;
00439  C864                    ; 'BASIC SUBROUTINE' EQUIVALENT
00440  C864                    ;
00441  C864                    ; 'CYLNDR' XSHD=0
00442  C864                    ;          FOR YREL=RADIUS TO 0
00443  C864                    ;          YSHD=YREL
00444  C864                    ;          FOR XREL=HLEN TO 0
00445  C864                    ;          GOSUB 'PTPLOT'
00446  C864                    ;          NEXT XREL
00447  C864                    ;          NEXT YREL
00448  C864                    ;          RETURN
00449  C864                    ;
00450  C864  A9 00       CYLNDR LDA #0
00451  C866  8D 6D 03          STA XSHD
00452  C869  8D 6E 03          STA XSHD+1
00453  C86C  8D 72 03          STA YSHD+1
00454  C86F  AD 77 03          LDA RADIUS
00455  C872  8D 70 03          STA YREL
00456  C875  AD B9 03    CYLOOP LDA HLEN
00457  C878  8D 6C 03          STA XREL
00458  C87B  AD 70 03          LDA YREL
00459  C87E  8D 71 03          STA YSHD
00460  C881  20 2B C6    CXLOOP JSR PTPLOT
00461  C884  CE 6C 03          DEC XREL
00462  C887  10 F8             BPL CXLOOP
00463  C889  CE 70 03          DEC YREL
00464  C88C  10 E7             BPL CYLOOP
00465  C88E  60                RTS
00466  C88F                    ;
00467  C88F                    ;****************************************
00468  C88F                    ;
00469  C88F                    ; DRAW EDGE-VIEW TOROIDS
00470  C88F                    ;
00471  C88F                    ; 'BASIC SUBROUTINE' EQUIVALENT
00472  C88F                    ;
00473  C88F                    ; 'EDGTOR' GOSUB 'TPARM':REM SET UP RADII
00474  C88F                    ;          FOR CNTX=0 TO RT
00475  C88F                    ;          XREL=CNTX:XSHD=CNTX
00476  C88F                    ;          R0=SQR(RT*RT-CNTX*CNTX)
00477  C88F                    ;          FOR CNTY=0 TO R0+RC
00478  C88F                    ;          YREL=CNTY
00479  C88F                    ;          YSHD=(R0*CNTY)/(R0+RC)
00480  C88F                    ;          GOSUB 'PTPLOT'
00481  C88F                    ;          NEXT CNTY
00482  C88F                    ;          NEXT CNTX
00483  C88F                    ;          RETURN
00484  C88F                    ;
00485  C88F  20 9A C7    EDGTOR JSR TPARM
00486  C892  A9 00             LDA #$00
00487  C894  8D 6E 03          STA XSHD+1
00488  C897  8D 72 03          STA YSHD+1
00489  C89A  AD B6 03    LOOPX4 LDA CNTX
00490  C89D  8D 6C 03          STA XREL
00491  C8A0  8D 6D 03          STA XSHD
00492  C8A3  85 AC             STA ARG
00493  C8A5  20 04 C0          JSR SQUARE
00494  C8A8  38                SEC
00495  C8A9  AD BA 03          LDA RS
00496  C8AC  E5 AE             SBC SQR
00497  C8AE  85 AC             STA RADCND
00498  C8B0  AD BB 03          LDA RS+1
00499  C8B3  E5 AF             SBC SQR+1
00500  C8B5  85 AD             STA RADCND+1
00501  C8B7  20 64 C0          JSR SQRT
00502  C8BA  AD 3C 03          LDA ROOT
00503  C8BD  8D B9 03          STA R0
00504  C8C0  18                CLC
00505  C8C1  6D BD 03          ADC RC
00506  C8C4  8D B8 03          STA MAX
00507  C8C7  A9 00             LDA #$00
00508  C8C9  8D B7 03          STA CNTY
00509  C8CC  AD B7 03    LOOPY4 LDA CNTY
00510  C8CF  8D 70 03          STA YREL
00511  C8D2  85 AD             STA MLPLER
00512  C8D4  AD B9 03          LDA R0
00513  C8D7  85 AC             STA MLPCND
00514  C8D9  20 11 C0          JSR MULT
00515  C8DC  85 FE             STA DVDND+1
00516  C8DE  A5 AE             LDA PROD
00517  C8E0  85 FD             STA DVDND
00518  C8E2  AD B8 03          LDA MAX
00519  C8E5  85 FB             STA DVSOR
00520  C8E7  20 EA C5          JSR SDIV
00521  C8EA  A5 FD             LDA QUOT
00522  C8EC  8D 71 03          STA YSHD
00523  C8EF  20 2B C6          JSR PTPLOT
00524  C8F2  AD B7 03          LDA CNTY
00525  C8F5  CD B8 03          CMP MAX
00526  C8F8  F0 06             BEQ DONE4
00527  C8FA  EE B7 03          INC CNTY
```

(Continued on page 74)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Four

```
00528  C8FD  4C CC C8          JMP LOOPY4
00529  C900  AD 86 03  DONE4   LDA CNTX
00530  C903  CD 8C 03          CMP RT
00531  C906  F0 06             BEQ DONEHT
00532  C908  EE 86 03          INC CNTX
00533  C90B  4C 9A C8          JMP LOOPX4
00534  C90E  60          DONEHT RTS
00535  C90F                    ;
00536  C90F                    ;****************************************
00537  C90F                    ;
00538  C90F                    ; DRAW A SHADED, TOP-VIEW TOROID
00539  C90F                    ;
00540  C90F                    ; 'BASIC SUBROUTINE' EQUIVALENT
00541  C90F                    ;
00542  C90F                    ; 'TOROID' GOSUB 'TPARM'
00543  C90F                    ;          FOR CNTX=0 TO RO/SQR(2)
00544  C90F                    ;          REM 8-FOLD SYMMETRY USED
00545  C90F                    ;          XREL=CNTX
00546  C90F                    ;          MAX=SQR(RO*RO-CNTX*CNTX)
00547  C90F                    ;          IF CNTX>RI THEN GOTO 'GRTR'
00548  C90F                    ;          CNTY=SQR(RI*RI-CNTY*CNTY)
00549  C90F                    ;          GOTO 'LLPY1'
00550  C90F                    ; 'GRTR'   CNTY=CNTX
00551  C90F                    ; 'LLPY1'  YREL=CNTY
00552  C90F                    ;          R0=SQR(CNTY*CNTY+CNTX*CNTX)
00553  C90F                    ;          XSHD=CNTX-(CNTX*RC)/R0
00554  C90F                    ;          YSHD=CNTY-(CNTY*RC)/R0
00555  C90F                    ;          HVFLAG=0:GOSUB 'PTPLOT'
00556  C90F                    ;          HVFLAG=-128:GOSUB 'PTPLOT'
00557  C90F                    ;          IF CNTY=MAX THEN GOTO 'DDNY1'
00558  C90F                    ;          CNTY=CNTY+1
00559  C90F                    ;          GOTO 'LLPY1'
00560  C90F                    ; 'DDNY1'  NEXT CNTX
00561  C90F                    ;          RETURN
00562  C90F                    ;
00563  C90F  20 9A C7  TOROID JSR TPARM
00564  C912  AD 8E 03          LDA RO
00565  C915  85 AC             STA ARG
00566  C917  20 04 C0          JSR SQUARE
00567  C91A  06 AE             ASL SQR
00568  C91C  26 AF             ROL SQR+1
00569  C91E  A5 AE             LDA SQR
00570  C920  85 AC             STA RADCND
00571  C922  A5 AF             LDA SQR+1
00572  C924  85 AD             STA RADCND+1
00573  C926  20 64 C0          JSR SQRT
00574  C929  4E 3D 03          LSR ROOT+1
00575  C92C  6E 3C 03          ROR ROOT
00576  C92F  AD 3C 03          LDA ROOT
00577  C932  8D 92 03          STA XMAX
00578  C935  AD 86 03  LLPX1   LDA CNTX
00579  C938  8D 6C 03          STA XREL
00580  C93B  85 AC             STA ARG
00581  C93D  20 04 C0          JSR SQUARE
00582  C940  8D 91 03          STA XSQR+1
00583  C943  A5 AE             LDA SQR
00584  C945  8D 90 03          STA XSQR
00585  C948  AD 8E 03          LDA RO
00586  C94B  85 AC             STA ARG
00587  C94D  20 04 C0          JSR SQUARE
00588  C950  38                SEC
00589  C951  A5 AE             LDA SQR
00590  C953  ED 90 03          SBC XSQR
00591  C956  85 AC             STA RADCND
00592  C958  A5 AF             LDA SQR+1
00593  C95A  ED 91 03          SBC XSQR+1
00594  C95D  85 AD             STA RADCND+1
00595  C95F  20 64 C0          JSR SQRT
00596  C962  AD 3C 03          LDA ROOT
00597  C965  8D 8B 03          STA MAX
00598  C968  38                SEC
00599  C969  AD 8F 03          LDA RI
00600  C96C  ED 86 03          SBC CNTX
00601  C96F  90 23             BCC GRTR
00602  C971  AD 8F 03          LDA RI
00603  C974  85 AC             STA ARG
00604  C976  20 04 C0          JSR SQUARE
00605  C979  38                SEC
00606  C97A  A5 AE             LDA SQR
00607  C97C  ED 90 03          SBC XSQR
00608  C97F  85 AC             STA RADCND
00609  C981  A5 AF             LDA SQR+1
00610  C983  ED 91 03          SBC XSQR+1
00611  C986  85 AD             STA RADCND+1
00612  C988  20 64 C0          JSR SQRT
00613  C98B  AD 3C 03          LDA ROOT
00614  C98E  8D 87 03          STA CNTY
00615  C991  4C 9A C9          JMP LLPY1
00616  C994  AD 86 03  GRTR    LDA CNTX
00617  C997  8D 87 03          STA CNTY
00618  C99A  AD 87 03  LLPY1   LDA CNTY
00619  C99D  8D 70 03          STA YREL
00620  C9A0  85 AC             STA ARG
00621  C9A2  20 04 C0          JSR SQUARE
00622  C9A5  18                CLC
00623  C9A6  A5 AE             LDA SQR
00624  C9A8  6D 90 03          ADC XSQR
00625  C9AB  85 AC             STA RADCND
00626  C9AD  A5 AF             LDA SQR+1
00627  C9AF  6D 91 03          ADC XSQR+1
00628  C9B2  85 AD             STA RADCND+1
00629  C9B4  20 64 C0          JSR SQRT
00630  C9B7  AD 3C 03          LDA ROOT
00631  C9BA  8D B9 03          STA R0
00632  C9BD  85 FB             STA DVSOR
00633  C9BF  AD 86 03          LDA CNTX
00634  C9C2  85 AD             STA MLPLER
00635  C9C4  AD 8D 03          LDA RC
00636  C9C7  85 AC             STA MLPCND
00637  C9C9  20 11 C0          JSR MULT
00638  C9CC  85 FE             STA DVDND+1
00639  C9CE  A5 AE             LDA PROD
00640  C9D0  85 FD             STA DVDND
00641  C9D2  20 EA C5          JSR SDIV
00642  C9D5  38                SEC
00643  C9D6  AD 86 03          LDA CNTX
00644  C9D9  E5 FD             SBC QUOT
00645  C9DB  8D 6D 03          STA XSHD
00646  C9DE  A9 00             LDA #$00
00647  C9E0  E5 FE             SBC QUOT+1
00648  C9E2  8D 6E 03          STA XSHD+1
00649  C9E5  AD 87 03          LDA CNTY
00650  C9E8  85 AD             STA MLPLER
00651  C9EA  AD 8D 03          LDA RC
00652  C9ED  85 AC             STA MLPCND
00653  C9EF  20 11 C0          JSR MULT
00654  C9F2  85 FE             STA DVDND+1
00655  C9F4  A5 AE             LDA PROD
00656  C9F6  85 FD             STA DVDND
00657  C9F8  AD 89 03          LDA R0
00658  C9FB  85 FB             STA DVSOR
00659  C9FD  20 EA C5          JSR SDIV
00660  CA00  38                SEC
00661  CA01  AD 87 03          LDA CNTY
00662  CA04  E5 FD             SBC QUOT
00663  CA06  8D 71 03          STA YSHD
00664  CA09  A9 00             LDA #$00
00665  CA0B  8D 83 03          STA HVFLAG
00666  CA0E  E5 FE             SBC QUOT+1
00667  CA10  8D 72 03          STA YSHD+1
00668  CA13  20 28 C6          JSR PTPLOT
00669  CA16  A9 80             LDA #$80
00670  CA18  8D 83 03          STA HVFLAG
00671  CA1B  20 28 C6          JSR PTPLOT
00672  CA1E  AD 87 03          LDA CNTY
00673  CA21  CD 8B 03          CMP MAX
00674  CA24  F0 06             BEQ DDNY1
00675  CA26  EE 87 03          INC CNTY
00676  CA29  4C 9A C9          JMP LLPY1
00677  CA2C  AD 86 03  DDNY1   LDA CNTX
00678  CA2F  CD 92 03          CMP XMAX
00679  CA32  F0 06             BEQ DUNTOR
00680  CA34  EE 86 03          INC CNTX
00681  CA37  4C 35 C9          JMP LLPX1
00682  CA3A  60        DUNTOR RTS
00683  CA3B                    ;
00684  CA3B                    ;****************************************
00685  CA3B                    ;
00686  CA3B                    ; DRAW "INSIDE VIEW" TOROIDS
00687  CA3B                    ;
00688  CA3B                    ; 'BASIC SUBROUTINE' EQUIVALENT
00689  CA3B                    ;
00690  CA3B                    ; 'SPOOL'  GOSUB 'TPARM'
00691  CA3B                    ;          FOR CNTX=0 TO RT
00692  CA3B                    ;          XREL=CNTX:XSHD=CNTX
00693  CA3B                    ;          MAX=RC-SQR(RS-CNTX*CNTX)
00694  CA3B                    ;          FOR CNTY=0 TO MAX
00695  CA3B                    ;          YREL=CNTY
00696  CA3B                    ;          YSHD=(RC*CNTY/MAX)-CNTY
00697  CA3B                    ;          GOSUB 'PTPLOT'
00698  CA3B                    ;          NEXT CNTY
00699  CA3B                    ;          NEXT CNTX
00700  CA3B                    ;          RETURN
00701  CA3B                    ;
00702  CA3B  20 9A C7  SPOOL   JSR TPARM
00703  CA3E  AD 86 03  LLPX2   LDA CNTX
00704  CA41  8D 6C 03          STA XREL
00705  CA44  85 AC             STA ARG
00706  CA46  38                SEC
00707  CA47  A9 00             LDA #$00
00708  CA49  ED 86 03          SBC CNTX
00709  CA4C  8D 6D 03          STA XSHD
00710  CA4F  A9 00             LDA #$00
00711  CA51  E9 00             SBC #$00
00712  CA53  8D 6E 03          STA XSHD+1
00713  CA56  20 04 C0          JSR SQUARE
00714  CA59  38                SEC
00715  CA5A  AD BA 03          LDA RS
00716  CA5D  E5 AE             SBC SQR
00717  CA5F  85 AC             STA RADCND
00718  CA61  AD BB 03          LDA RS+1
00719  CA64  E5 AF             SBC SQR+1
00720  CA66  85 AD             STA RADCND+1
00721  CA68  20 64 C0          JSR SQRT
00722  CA6B  38                SEC
00723  CA6C  AD 8D 03          LDA RC
00724  CA6F  ED 3C 03          SBC ROOT
00725  CA72  8D 8B 03          STA MAX
00726  CA75  A9 00             LDA #$00
00727  CA77  8D 87 03          STA CNTY
00728  CA7A  AD 87 03  LLPY2   LDA CNTY
00729  CA7D  8D 70 03          STA YREL
00730  CA80  85 AD             STA MLPLER
00731  CA82  AD 8D 03          LDA RC
00732  CA85  85 AC             STA MLPCND
00733  CA87  20 11 C0          JSR MULT
00734  CA8A  85 FE             STA DVDND+1
00735  CA8C  A5 AE             LDA PROD
00736  CA8E  85 FD             STA DVDND
00737  CA90  AD 8B 03          LDA MAX
00738  CA93  85 FB             STA DVSOR
00739  CA95  20 EA C5          JSR SDIV
00740  CA98  A5 FD             LDA QUOT
00741  CA9A  38                SEC
00742  CA9B  ED 87 03          SBC CNTY
00743  CA9E  8D 71 03          STA YSHD
00744  CAA1  A5 FE             LDA QUOT+1
00745  CAA3  E9 00             SBC #$00
00746  CAA5  8D 72 03          STA YSHD+1
00747  CAA8  20 28 C6          JSR PTPLOT
00748  CAAB  AD 87 03          LDA CNTY
00749  CAAE  CD 8B 03          CMP MAX
00750  CAB1  F0 06             BEQ DDNY2
00751  CAB3  EE 87 03          INC CNTY
00752  CAB6  4C 7A CA          JMP LLPY2
00753  CAB9  AD 86 03  DDNY2   LDA CNTX
00754  CABC  CD 8C 03          CMP RT
00755  CABF  F0 06             BEQ DUNHSP
00756  CAC1  EE 86 03          INC CNTX
00757  CAC4  4C 3E CA          JMP LLPX2
00758  CAC7  60        DUNHSP RTS
00759  CAC8                    .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 ARG      00AC   BAKLIT   0382   CHCLUP   C690   CLIPD    0380
 CLIPL    037D   CLIPR    037E   CLIPU    037F   CNTNU    C600

SYMBOL TABLE

SYMBOL VALUE
 CNTX     0386   CNTY     0387   CXLOOP   C881   CYLNDR   C864
 CYLOOP   C875   DDNY1    CA2C   DDNY2    CAB9   DHEMI    C6BC
 DIVIDE   C025   DONE     C863   DONE4    C900   DONEHT   C90E
 DONEY    C855   DUNHSP   CAC7   DUNTOR   CA3A   DVDND    00FD
 DVSOR    00FB   EDGTOR   C88F   GETVAL   C5F1   GETZ     C745
 GRTR     C994   HEMI     0381   HLEN     0389   HTORRN   0346
 HVFLAG   0383   LLPX1    C935   LLPX2    CA3E   LLPY1    C99A
 LLPY2    CA7A   LOOPX    C805   LOOPX4   C89A   LOOPY    C82E
 LOOPY4   C8CC   MAX      0388   MLPCND   00AC   MLPLER   00AD
 MULT     C811   NEGATE   C5FF   NORM     C224   NOROT    C65A
```

End Listing Four

(Listing Five begins on page 76)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Four

```
NORSTR  C744    NOSCAL  0347    ORIGIN  C5EA    PLDONE  C721
PLTSHD  C20F    PROD    00AE    PTPLOT  C628    PTPLT2  C65D
QUOT    00FD    R0      0389    RADCND  00AC    RADIUS  0377
RAM     036A    RANDOM  C0C8    RC      038D    RHEMI   C6E8
RI      038F    RNDM    C000    RO      038E    ROOT    033C
RS      038A    RT      038C    SDIV    C5EA    SPHERE  C7C7
SPOOL   CA3B    SQR     00AE    SQRT    C064    SQUARE  C004
TEMP    0384    TNTMP   037B    TONE    0379    TOROID  C90F
TPARM   C79A    VALUE   0344    XCENT   036A    XMAX    0392
XPLT    033F    XREL    036C    XSHD    036D    XSQR    0390
YCENT   036F    YPLT    0341    YREL    0370    YSHD    0371
ZEROOT  C791    ZREL    0373    ZWX     0375

END OF ASSEMBLY
```

Listing Five

```
00001  0000                ; INTERFACE - EASY PARAMETER SETTING FOR SHAPE
00002  0000                ; DRAWING ROUTINES FROM BASIC.
00003  0000                ;
00004  0000                ; RICHARD L. RYLANDER  11/23/84
00005  0000                ;
00006  0000                ;****************************************
00007  0000                ORIGIN=$CAC8
00008  0000                RAM    =$0393
00009  0000                ;
00010  0000                ; PARAMETER LOCATIONS FOR VARIOUS SHAPES
00011  0000                ;
00012  0000                XCENT  =$036A
00013  0000                YCENT  =$036F
00014  0000                XPLOT  =$033F
00015  0000                YPLOT  =$0341
00016  0000                XMIN   =$034A
00017  0000                YMIN   =$034C
00018  0000                XMID   =$034D
00019  0000                YMID   =$034F
00020  0000                XMAX   =$0350
00021  0000                YMAX   =$0352
00022  0000                RADIUS =$0377
00023  0000                HLEN   =$0389
00024  0000                RI     =$038F
00025  0000                RO     =$038E
00026  0000                ;
00027  0000                HVFLAG =$0383
00028  0000                VALUE  =$0344
00029  0000                PLTFLG =$033E
00030  0000                ;
00031  0000                DEFLAG =$FB
00032  0000                ;
00033  0000                ;****************************************
00034  0000                ;
00035  0000                ; FUNCTION LOCATIONS
00036  0000                ;
00037  0000                GRFON  =$C0E2          ; SWITCH TO GRAPHICS MODE
00038  0000                GRFOFF =$C103          ; RETURN TO TEXT DISPLAY
00039  0000                ;
00040  0000                CLEARR=$C12C           ; CLEAR BITMAP
00041  0000                CLRBYT=$C135           ; CLEAR (FILL) BYTE
00042  0000                COLORR=$C118           ; LOAD COLOR MAP
00043  0000                COLBYT=$C119           ; COLOR BYTE
00044  0000                ;
00045  0000                PLOTR  =$C14B          ; POINT PLOT ROUTINE
00046  0000                LINER  =$C2DB          ; DRAW A LINE
00047  0000                FACETR =$C4E1          ; DRAW A SHADED FACET
00048  0000                ;
00049  0000                ;****************************************
00050  0000                ;
00051  0000                ; SHADED SHAPE DRAWING ROUTINES
00052  0000                ;
00053  0000                SPHERR=$C7C7           ; SPHERE
00054  0000                CYLNDR=$CB64           ; CYLINDER
00055  0000                TORUSR=$C90F           ; TOP-VIEW TOROID
00056  0000                EDGTOR=$CB8F           ; EDGE-VIEW TOROID
00057  0000                SPOOLR=$CA3B           ; INSIDE-VIEW TOROID
00058  0000                ;
00059  0000                ;****************************************
00060  0000                ;
00061  0000                ; BASIC ROM ROUTINES
00062  0000                ;
00063  0000                CHKCOM=$AEFD           ; CHECK FOR COMMA
00064  0000                EVAEXP=$AD9E           ; EVALUATE EXPRESSION
00065  0000                FLTFIX=$B1AA           ; CONVERT TO FIXED
00066  0000                ;
00067  0000                **RAM
00068  0393                LINFAC **+1            ; LINE OR FACET FLAG
00069  0394                ;
00070  0394                **ORIGIN
00071  CAC8                ;
00072  CAC8                ;****************************************
00073  CAC8                ;
00074  CAC8                ; GET PARAMETERS FROM BASIC CALLING STATEMENT
00075  CAC8                ; OF THE FORM:
00076  CAC8                ; SYS(FNCTN),PARAM1,PARAM2,PARAM3[OPT]
00077  CAC8                ; WHERE THE THIRD PARAMETER (FOR EXAMPLE)
00078  CAC8                ; MAY BE OPTIONAL (A DEFAULT VALUE IS USED
00079  CAC8                ; IF THE PARAMETER IS NOT SPECIFIED)
00080  CAC8                ;
00081  CAC8  20 FD AE      GETNUM JSR CHKCOM      ; LOOK FOR COMMA
00082  CACB  20 9E AD             JSR EVAEXP      ; EVALUATE EXPRESSION
00083  CACE  20 AA B1             JSR FLTFIX      ; CHANGE TO INTEGER WITH
00084  CAD1                ; HIGH BYTE IN "A" AND LOW BYTE IN "Y"
00085  CAD1  60                   RTS
00086  CAD2                ;
00087  CAD2                ; CHECK FOR ADDITIONAL (OPTIONAL) PARAMETERS
00088  CAD2                ;
00089  CAD2  A9 2C         PCHECK LDA #$2C        ; "," COMMA
00090  CAD4  A0 00                LDY #0
00091  CAD6  84 FB                STY DEFLAG
00092  CAD8  D1 7A                CMP ($7A),Y
00093  CADA  D0 03                BNE NOMORE      ; NO COMMA - USE DEFAULT
00094  CADC  4C 73 00             JMP $0073
00095  CADF  A0 80         NOMORE LDY #$80
00096  CAE1  84 FB               STY DEFLAG
00097  CAE3  60                   RTS
00098  CAE4                ;
00099  CAE4                ; GET TWO ADDITIONAL PARAMETERS FOR TOROIDS
00100  CAE4                ;
00101  CAE4  20 D2 CA      GETTWO JSR PCHECK
00102  CAE7  24 FB                BIT DEFLAG
00103  CAE9  30 0F                BMI DFAULT
00104  CAEB  20 9E AD             JSR EVAEXP
00105  CAEE  20 AA B1             JSR FLTFIX
00106  CAF1  8C 8F 03             STY RI
00107  CAF4  20 C8 CA             JSR GETNUM
00108  CAF7  8C 8E 03             STY RO
00109  CAFA  60            DFAULT RTS
00110  CAFB                ;
00111  CAFB                ;****************************************
00112  CAFB                ;
00113  CAFB                ; SET CENTER COORDINATES
00114  CAFB                ;
00115  CAFB  20 C8 CA      CENTER JSR GETNUM
00116  CAFE  8C 6A 03             STY XCENT
00117  CB01  8D 6B 03             STA XCENT+1
00118  CB04  20 C8 CA             JSR GETNUM
00119  CB07  8C 6F 03             STY YCENT
00120  CB0A  60                   RTS
00121  CB0B                ;
00122  CB0B                ;****************************************
00123  CB0B                ;
00124  CB0B                ; CLEAR THE BITMAP, FILLING WITH (OPTIONAL)
00125  CB0B                ; FILL VALUE SPECIFIED OR WITH (DEFAULT) "0"
00126  CB0B                ;
00127  CB0B  20 D2 CA      CLEAR2 JSR PCHECK
00128  CB0E  24 FB                BIT DEFLAG
00129  CB10  30 07                BMI DEFCLR
00130  CB12  20 9E AD             JSR EVAEXP
00131  CB15  20 AA B1             JSR FLTFIX
00132  CB18  2C                  .BYTE #2C
00133  CB19  A0 00         DEFCLR LDY #0
00134  CB1B  8C 35 C1             STY CLRBYT
00135  CB1E  4C 2C C1             JMP CLEARR
00136  CB21                ;
00137  CB21                ;****************************************
00138  CB21                ;
00139  CB21                ; FILL COLOR MAP WITH (OPTIONAL) COLOR BYTE
00140  CB21                ; SPECIFIED OR WITH (DEFAULT) "$01"
00141  CB21                ; (BLACK DOTS ON WHITE BACKGROUND)
00142  CB21                ;
00143  CB21  20 D2 CA      COLOR2 JSR PCHECK
00144  CB24  24 FB                BIT DEFLAG
00145  CB26  30 07                BMI DEFCOL
00146  CB28  20 9E AD             JSR EVAEXP
00147  CB2B  20 AA B1             JSR FLTFIX
00148  CB2E  2C                   .BYTE $2C
00149  CB2F  A0 01         DEFCOL LDY #$01
00150  CB31  8C 19 C1             STY COLBYT
00151  CB34  4C 18 C1             JMP COLORR
00152  CB37                ;
00153  CB37                ;****************************************
00154  CB37                ;
00155  CB37                ; PLOT OR UNPLOT POINTS
00156  CB37                ;
00157  CB37  A9 00         PLOT2  LDA #0
00158  CB39  2C                   .BYTE #2C
00159  CB3A  A9 80         UNPLT2 LDA #$80
00160  CB3C  8D 3E 03             STA PLTFLG
00161  CB3F  20 C8 CA             JSR GETNUM
00162  CB42  8C 3F 03             STY XPLOT
00163  CB45  8D 40 03             STA XPLOT+1
00164  CB48  20 C8 CA             JSR GETNUM
00165  CB4B  8C 41 03             STY YPLOT
00166  CB4E  4C 4B C1             JMP PLOTR
00167  CB51                ;
00168  CB51                ;****************************************
00169  CB51                ;
00170  CB51                ; DRAW LINES BETWEEN (X1,Y1) AND (X2,Y2)
00171  CB51                ; OR SHADED FACETS BETWEEN THREE POINTS
00172  CB51                ; (X1,Y1), (X2,Y2) AND (X3,Y3)
00173  CB51                ;
00174  CB51  A9 00         LINE2  LDA #0
00175  CB53  2C                   .BYTE #2C
00176  CB54  A9 80         FACET2 LDA #$80
00177  CB56  8D 93 03             STA LINFAC
00178  CB59  20 C8 CA             JSR GETNUM
00179  CB5C  8C 4A 03             STY XMIN
00180  CB5F  8D 4B 03             STA XMIN+1
00181  CB62  20 C8 CA             JSR GETNUM
00182  CB65  8C 4C 03             STY YMIN
00183  CB68  20 C8 CA             JSR GETNUM
00184  CB6B  8C 4D 03             STY XMID
00185  CB6E  8D 4E 03             STA XMID+1
00186  CB71  20 C8 CA             JSR GETNUM
00187  CB74  8C 4F 03             STY YMID
00188  CB77  2C 93 03             BIT LINFAC
00189  CB7A  10 18                BPL LDRAW
00190  CB7C  20 C8 CA             JSR GETNUM
00191  CB7F  8C 50 03             STY XMAX
00192  CB82  8D 51 03             STA XMAX+1
00193  CB85  20 C8 CA             JSR GETNUM
00194  CB88  8C 52 03             STY YMAX
00195  CB8B  20 C8 CA             JSR GETNUM
00196  CB8E  8C 44 03             STY VALUE
00197  CB91  4C E1 C4             JMP FACETR
00198  CB94  4C DB C2      LDRAW  JMP LINER
00199  CB97                ;
00200  CB97                ;****************************************
00201  CB97                ;
00202  CB97                ; DRAW A SPHERE CENTERED AT (XCENT,YCENT)
00203  CB97                ; DEFAULT RADIUS IS LAST VALUE USED
00204  CB97                ;
00205  CB97  20 FB CA      SPHER2 JSR CENTER
00206  CB9A  20 D2 CA             JSR PCHECK
00207  CB9D  24 FB                BIT DEFLAG
00208  CB9F  30 09                BMI SKIP1
00209  CBA1  20 9E AD             JSR EVAEXP
00210  CBA4  20 AA B1             JSR FLTFIX
00211  CBA7  8C 77 03             STY RADIUS
00212  CBAA  4C C7 C7      SKIP1  JMP SPHERR
00213  CBAD                ;
00214  CBAD                ;****************************************
00215  CBAD                ;
00216  CBAD                ; DRAW A TOP-VIEW TOROID AT (XCENT,YCENT)
00217  CBAD                ; DEFAULT INNER AND OUTER RADII ARE LAST USED
00218  CBAD                ;
00219  CBAD  20 FB CA      TORUS2 JSR CENTER
00220  CBB0  20 E4 CA             JSR GETTWO
00221  CBB3  4C 0F C9             JMP TORUSR
00222  CBB6                ;
00223  CBB6                ;****************************************
00224  CBB6                ;
00225  CBB6                ; DRAW CYLINDERS WITH AXES HORIZONTAL OR
00226  CBB6                ; VERTICAL.  DEFAULT RADIUS AND "HALF-LENGTH"
00227  CBB6                ; ARE LAST VALUES USED.
00228  CBB6                ;
00229  CBB6  A9 80         VCYL2  LDA #$80
00230  CBB8  2C                   .BYTE #2C
00231  CBB9  A9 00         HCYL2  LDA #0
00232  CBBB  8D 83 03             STA HVFLAG
00233  CBBE  20 FB CA             JSR CENTER
00234  CBC1  20 D2 CA             JSR PCHECK
00235  CBC4  24 FB                BIT DEFLAG
00236  CBC6  30 0F                BMI SKIP2
00237  CBC8  20 9E AD             JSR EVAEXP
```

(Continued on page 78)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Five

```
00238  CBCB  20 AA B1          JSR FLTFIX
00239  CBCE  8C 77 03          STY RADIUS
00240  CBD1  20 C8 CA          JSR GETNUM
00241  CBD4  8C 89 03          STY HLEN
00242  CBD7  4C 64 CB    SKIP2 JMP CYLNDR
00243  CBDA                    ;
00244  CBDA                    ;**************************************
00245  CBDA                    ;
00246  CBDA                    ; DRAW EDGE-VIEW TOROIDS WITH AXES HORIZONTAL
00247  CBDA                    ; OR VERTICAL
00248  CBDA                    ; INNNER AND OUTER RADII ARE OPTIONAL
00249  CBDA                    ;
00250  CBDA  A9 80       VTOR2 LDA ##80
00251  CBDC  2C                .BYTE #2C
00252  CBDD  A9 00       HTOR2 LDA #0
00253  CBDF  8D 83 03          STA HVFLAG
00254  CBE2  20 FB CA          JSR CENTER
00255  CBE5  20 E4 CA          JSR GETTWO
00256  CBE8  4C 8F CB          JMP EDGTOR
00257  CBEB                    ;
00258  CBEB                    ;**************************************
00259  CBEB                    ;
00260  CBEB                    ; DRAW INSIDE-VIEW TOROIDS, "SPOOLS",
00261  CBEB                    ; WITH AXES HORIZONTAL OR VERTICAL
00262  CBEB                    ; INNER AND OUTER RADII ARE OPTIONAL
00263  CBEB                    ;
00264  CBEB  A9 80       VSPL2 LDA ##80
00265  CBED  2C                .BYTE #2C
00266  CBEE  A9 00       HSPL2 LDA #0
00267  CBF0  8D 83 03          STA HVFLAG
00268  CBF3  20 FB CA          JSR CENTER
00269  CBF6  20 E4 CA          JSR GETTWO
00270  CBF9  4C 3B CA          JMP SPOOLR
00271  CBFC                .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 CENTER   CAFB     CHKCOM   AEFD     CLEAR2   CB0B     CLEARR   C12C
 CLRBYT   C135     COLBYT   C119     COLOR2   CB21     COLORR   C118
 CYLNDR   CB64     DEFCLR   CB19     DEFCOL   CB2F     DEFLAG   00FB
 DFAULT   CAFA     EDGTOR   CB8F     EVAEXP   AD9E     FACET2   CB54
 FACETR   C4E1     FLTFIX   B1AA     GETNUM   CAC8     GETTWO   CAE4
 GRFOFF   C103     GRFON    C0E2     HCYL2    CBB9     HLEN     0389
 HSPL2    CBEE     HTOR2    CBDD     HVFLAG   0383     LDRAW    CB94
 LINE2    CB51     LINER    C2DB     LINFAC   0393     NOMORE   CADF
 ORIGIN   CAC8     PCHECK   CAD2     PLOT2    CB37     PLOTR    C14B
 PLTFLG   033E     RADIUS   0377     RAM      0393     RI       038F
 RO       038E     SKIP1    CBAA     SKIP2    CBD7     SPHER2   CB97
 SPHERR   C7C7     SPOOLR   CA3B     TORUS2   CBAD     TORUSR   C90F
 UNFLT2   CB3A     VALUE    0344     VCYL2    CBB6     VSPL2    CBEB
 VTOR2    CBDA     XCENT    036A     XMAX     0350     XMID     034D
 XMIN     034A     XPLOT    033F     YCENT    036F     YMAX     0352
 YMID     034F     YMIN     034C     YPLOT    0341

END OF ASSEMBLY
```

End Listing Five

Listing Six

```
10 REM         SHAPES DEMO
20 :
30 REM      RICHARD L. RYLANDER     11/23/84 (REVISED 1/20/85 TO ADD LABELING)
40 :
50 GR=49378    :REM GRAPHICS MODE
60 TX=49411    :REM TEXT MODE
70 :
80 LB=893      :REM LEFT BOUND
90 RB=894      :REM RIGHT BOUND
100 UB=895     :REM UP BOUND
110 DB=896     :REM DOWN BOUND
120 :
130 REM   FLAGS FOR VARIOUS DRAWING MODES
140 :
150 SH=838     :REM SHADE STYLE - 0=RANDOM, 1=HALFTONE
160 SC=839     :REM SCALING - 0=NORMAL (1:1), 1=SCALED (3:4) FOR SCREEN DISPLAY
170 LT=898     :REM LIGHTING - 0=NORMAL SINGLE-SOURCE, 1=BACKLIT ILLUMINATION
180 :
190 BO=53280   :REM  BORDER COLOR
200 :
210 REM   FUNCTION LOCATIONS
220 :
230 CL=51979   :REM CLEAR BITMAP AREA
240 CO=52001   :REM FILL COLOR MAP
250 :
260 SP=52119   :REM  SPHERE
270 TR=52141   :REM  TOP-VIEW TOROID
280 VC=52150   :REM  CYLINDER (AXIS VERTICAL)
290 HC=52153   :REM  CYLINDER (AXIS HORIZONTAL)
300 VT=52186   :REM  EDGE-VIEW TOROID (AXIS VERTICAL)
310 HT=52189   :REM  EDGE-VIEW TOROID (AXIS HORIZONTAL)
320 VS=52203   :REM  INSIDE-VIEW TOROID ("SPOOL") (AXIS VERTICAL)
330 HS=52206   :REM  INSIDE-VIEW TOROID ("SPOOL") (AXIS HORIZONTAL)
340 :
350 REM  DRAW SAMPLES OF PRIMITIVE SHAPES WITH LABEL BENEATH EACH
360 :
370 POKE SH,1 :REM HALFTONE SHADING
380 POKE SC,1 :REM USE SCREEN SCALING (FOR UNDISTORTED SPHERES, ETC.)
390 POKE LT,0 :REM USE NORMAL (NO BACKLIGHT) ILLUMINATION
400 SYS(CL)   :REM CLEAR SCREEN (DEFAULT - FILL BITMAP WITH 0'S)
410 SYS(CO),16*11+1  :REM COLOR COMBINATION - DARK GRAY (11) DOTS ON WHITE (1)
412 REM    LOOKS BETTER THAN DEFAULT COLOR COMBINATION OF BLACK DOTS ON WHITE
414 REM    ON MOST COLOR MONITORS
420 POKE BO,1 :REM WHITE BORDER TO MATCH BACKGROUND
430 SYS(GR)   :REM ENTER BITMAPPED GRAPHICS MODE
432 RW=12:CM=14:MD=1:A#="SAMPLE SHAPES":GOSUB 1900:REM  PUT TITLE ON IMAGE
434 X1=110:Y1=120:X2=210:Y2=120:BC=1:DC=7:GOSUB 1700:REM  COLOR TITLE TEXT
440 POKE LB,38:POKE RB,38:POKE UB,38:POKE DB,38       :REM INITIALIZE CLIPPING
450 SYS(SP),40,199,38
455 RW=9:CM=2:MD=1:A#="SPHERE":GOSUB1900
460 SYS(HC),120,199,38,38
465 CM=11:A#="H-CYLNDR":GOSUB1900
470 SYS(VC),200,199 :REM NO SIZE PARAMETERS GIVEN, DEFAULT (PREVIOUS) ARE USED
475 CM=21:A#="V-CYLNDR":GOSUB1900
480 SYS(TR),280,199,15,38
485 CM=32:A#="TOROID":GOSUB1900
490 SYS(VT),40,64
495 RW=23:CM=1:A#="V-TOROID":GOSUB1900
500 SYS(HT),120,64
505 CM=11:A#="H-TOROID":GOSUB1900
510 SYS(HS),200,64,5,100
515 CM=22:A#="H-SPOOL":GOSUB1900
520 SYS(VS),280,64
525 CM=32:A#="V-SPOOL":GOSUB1900
530 POKE 198,0:WAIT 198,1:POKE 198,0
540 REM  WAIT FOR A KEY TO BE PRESSED
550 :
560 REM  DRAW TWO "GOBLETS" ONE WITH HALFTONE, THE OTHER RANDOM SHADING.
570 :
580 SYS(CL):SYS(CO),16*11+1
582 RW=14:CM=15:A#="COMPARISON":GOSUB 1900:RW=15:CM=19:A#="OF":GOSUB 1900
584 RW=16:CM=14:A#="SHADE STYLES":GOSUB 1900:RW=18:CM=14:A#="<-- HALFTONE"
586 GOSUB 1900:RW=20:CM=15:A#="RANDOM -->":GOSUB 1900
590 POKE LB,255:POKE RB,255:POKE UB,49:POKE DB,255: REM CLIP AT SCREEN TOP
600 SYS(SP),80,190,80
610 POKE UB,51:POKE DB,51: REM CLIP AT JUNCTION WITH SPHERE FOR SMOOTH SEAM
620 SYS(VS),80,69,10,130
630 POKE DB,9:POKE UB,8
640 SYS(VT),80,9,25,45
650 POKE SH,0  :REM SWITCH TO RANDOM SHADING
660 POKE LB,255:POKE RB,255:POKE UB,49:POKE DB,255
670 SYS(SP),240,190,80
680 POKE UB,51:POKE DB,51
690 SYS(VS),240,69,10,130
700 POKE DB,9:POKE UB,8
710 SYS(VT),240,9,25,45
720 POKE 198,0:WAIT 198,1:POKE 198,0
740 :
750 REM  DRAW "WINE" SCENE
760 POKE LT,1 :REM BACKLIT ILLUMINATION
770 POKE SH,1 :REM HALFTONE SHADING FOR MOST (RANDOM "LABEL" ON BOTTLE)
780 SYS(CO):SYS(CL),255: REM FILL BITMAP WITH 1'S ("SET" BACKGROUND)
790 POKE BO,0 : REM BLACK BORDER TO MATCH BACKGROUND
792 RW=0:CM=0:MD=2:A#="DRAWING  WITH":GOSUB 1900:RW=1:CM=2:A#="BACKLIGHT"
794 GOSUB 1900:RW=2:CM=0:A#="AGAINST A SET":GOSUB 1900
796 RW=3:CM=2:A#="BACKGROUND":GOSUB 1900
798 RW=1:CM=26:A#="COLORS  ADDED":GOSUB 1900
800 RW=2:CM=25:A#="TO SELECT AREAS":GOSUB 1900
810 REM  DRAW BOTTLE
820 POKE UB,0:POKE DB,255:POKE LB,255:POKE RB,255
830 SYS(VT),150,10,30,50
840 POKE UB,255:SYS(VC),150,70,50,60
850 POKE DB,0:SYS(VT),150,130,6,50
860 POKE DB,55:POKE UB,0:SYS(VS),150,204,15,181
870 POKE UB,255:SYS(VC),150,221,16,17
880 :
890 REM  DRAW WINE GLASS
900 POKE UB,20:SYS(SP),80,120,60
910 POKE UB,35:POKE DB,34:SYS(VS),80,34,10,110
920 :
930 REM  DRAW SOME GRAPES
940 SYS(SP),8,8,8: REM OTHER "GRAPES" WILL BE SAME RADIUS - JUST GIVE POSITIONS
950 SYS(SP),20,8:SYS(SP),40,8:SYS(SP),12,20:SYS(SP),30,20:SYS(SP),25,16
960 :
970 REM  DRAW AN APPLE BY A PAIR OF EDGE-VIEW TOROIDS AND A SPHERE SECTION
980 POKE UB,255:POKE DB,255:POKE LB,255:POKE RB,59
990 SYS(VT),260,29,0,50:SYS(VT),260,79
1000 POKE UB,43:POKE DB,43:SYS(SP),260,54,60
1010 REM   PUT STEM ON APPLE
1020 POKE RB,0:POKE DB,0:SYS(TR),272,104,10,15
1030 REM   ADD A LEAF BY A SPHERE SECTION
1040 POKE DB,255:POKE RB,0:SYS(SP),256,119,15
1050 REM  ADD A RANDOM SHADED "LABEL" TO THE BOTTLE
1060 POKE UB,255:POKE RB,255:POKE LB,6
1070 POKE,SH,0:SYS(VC),150,72,50,48
1080 :
1090 REM  ADD COLORS TO "WINE" SCENE
1100 SYS(CO),12:REM REPLACE WHITE "HOLES" (BACKGROUND) WITH MEDIUM GRAY
1102 X1=0:Y1=200:X2=100:Y2=239:DC=0:BC=0:GOSUB 1700:REM HIDE TEXT IN CORNERS
1104 X1=180:Y1=200:X2=319:Y2=239:DC=0:BC=0:GOSUB 1700
1110 X1=200:Y1=1:X2=315:Y2=100:DC=0:BC=2:GOSUB 1700
1120 X1=240:Y1=110:X2=255:Y2=150:BC=5:GOSUB 1700
1130 X1=260:Y1=110:X2=270:Y2=135:BC=9:GOSUB 1700
1140 X1=1:Y1=1:X2=48:Y2=30:BC=4:GOSUB 1700
1150 X1=140:Y1=205:X2=180:Y2=235:BC=7:GOSUB 1700
1160 X1=145:Y1=25:X2=195:Y2=115:BC=6:GOSUB 1700
1170 POKE 198,0:WAIT 198,1:POKE 198,0
1190 :
1200 REM   "COFFEE AND DONUTS"
1210 POKE SH,0 :REM RANDOM SHADING ON DONUTS
1220 SYS(CO),16*11+1:SYS(CL):POKE BO,1:REM WHITE BORDER TO MATCH BACKGROUND
1230 POKE LB,255:POKE RB,255:POKE UB,255:POKE DB,255 :REM NO INITIAL CLIPPING
1240 SYS(VT),60,20,20,60
1250 POKE RB,29:SYS(VT),99,60:POKE RB,255
1260 SYS(TR),188,180
1270 REM ADD COFFEE CUP (HALFTONE SHADING)
1280 POKE SH,1:POKE UB,0:SYS(VT),188,20:POKE UB,255
1290 POKE DB,0:SYS(TR),278,110,20,40
1300 POKE DB,255:POKE UB,0:POKE LB,0
1310 SYS(TR),248,90,50,70:POKE LB,255:SYS(SP),248,110,10:POKE UB,255
1320 SYS(VC),308,100,10,10
1330 SYS(VC),188,77,60,57
1340 POKE DB,0:SYS(VT),188,134,40,60
1350 REM  ADD COLOR - FIRST MAKE SCREEN BROWN DOTS ON WHITE THEN MAKE CUP GREEN
1360 SYS(CO),1+16*9 :REM 1=WHITE BACKGROUND, 9=BROWN DOTS
1370 X1=130:Y1=1:X2=319:Y2=136:BC=1:DC=5:GOSUB 1700
1380 X1=250:Y1=144:X2=319:Y2=144:GOSUB 1700
1390 POKE 198,0:WAIT 198,1:POKE 198,0
1410 :
1420 REM  DRAW "LINKED" TOROIDS BY REDRAWING OVERLAP AREAS USING CLIPPING
1430 POKE LT,0 :REM BLUE DOTS ON WHITE, NO BACKLIGHT
1440 SYS(CL):SYS(CO),1+16*6: REM 1=WHITE BACKGROUND, 6=BLUE DOTS
1450 POKE UB,255:POKE DB,255:POKE LB,255:POKE RB,255: REM NO INITIAL CLIPPING
1460 POKE SH,0   :REM RANDOM SHADING
1470 SYS(TR),244,84,48,70
1480 SYS(TR),160,84:SYS(TR),76,84
1490 SYS(TR),118,156:SYS(TR),202,156
1500 REM  ADD PROPER OVERLAP SECTIONS
1510 POKE RB,0:POKE DB,0:SYS(TR),160,84:POKE RB,255:POKE LB,0
1520 SYS(TR),76,84:POKE DB,255:POKE UB,0:SYS(TR),118,156
1530 POKE LB,255:POKE RB,0:SYS(TR),202,156:POKE LB,27
1540 POKE DB,0:POKE UB,255:SYS(TR),160,84
1550 POKE DB,0:POKE UB,255:SYS(TR),160,84
1560 POKE LB,255:POKE UB,27:SYS(TR),244,84
1570 POKE LB,0:POKE RB,27:POKE UB,255:SYS(TR),244,84
1580 POKE 198,0
1590 GET A#:IF A#= "" THEN 1590
1600 SYS(TX):POKE BO,14:REM RETURN TO TEXT MODE
1610 END
1620 :
1630 REM SUBROUTINE FOR ADDING COLOR TO DIFFERENT SCREEN AREAS
1640 REM REMEMBER THAT COLOR BOUNDARIES MUST CORRESPOND TO CHARACTER BOUNDARIES
1650 REM DEFINE A RECTANGULAR AREA BY LOWER-LEFT AND UPPER-RIGHT COORDINATES
1660 REM (X1,Y1)=LOWER-LEFT POINT, (X2,Y2)=UPPER-RIGHT POINT
1670 REM THE CORNER POINTS CAN BE ARBITRARY BUT COLOR CHANGE WILL BE IN THE
1680 REM SMALLEST CHARACTER-CELL BOUNDED RECTANGLE THAT INCLUDES THOSE POINTS
1690 REM  Y-COORDINATES MUST BE "UNSCALED" IF SCALE FLAG IS SET:
1700 IF PEEK(SC)THEN Y1=(Y1+1)*213/256:Y2=(Y2+1)*213/256
1710 REM COLOR TO BE POKED IN IS GIVEN BY VARIABLE CC="COMPOSITE COLOR"
1720 REM WHERE CC=16*DC + BC [DC=DOT COLOR, BC=BACKGROUND COLOR NUMBERS]
1730 CC=16*DC+BC
1740 FOR IX=INT(X1/8) TO INT(X2/8)
1750 FOR IY=INT(Y1/8) TO INT(Y2/8)
```

(Continued on page 80)

Drawing on the C-64 (Listing Continued, text begins on page 50)

Listing Six

```
1760 POKE 34752+IX-40*IY,CC
1770 NEXT:NEXT:RETURN
1780 :
1790 REM SUBROUTINE TO ADD TEXT TO GRAPHIC SCREEN.
1800 REM "RW" AND "CM" ARE THE ROW (0-24) AND COLUMN (0-39) COORDINATES FOR THE
1810 REM FIRST LETTER OF THE TEXT STRING TO BE PRINTED.
1820 REM THE TEXT STRING ITSELF IS ASSIGNED TO "A$".
1830 REM "MD" IS THE MODE FOR THE PRINTING. FIVE MODES ARE ALLOWED:
1840 REM  1 - NORMAL ("BLACK" LETTERS ON "WHITE" BACKGROUND)
1850 REM  2 - REVERSED ("WHITE" LETTERS ON "BLACK" BACKGROUND)
1860 REM  3 - SET ("BLACK") LETTERS "OR'ED" WITH BACKGROUND
1870 REM  4 - UNSET ("WHITE") LETTERS "AND'ED" WITH BACKGROUND
1880 REM  5 - SET LETTERS "XOR'ED" WITH BACKGROUND
1890 :
1900 SB=40952:TB=54272:IF(MD AND 1)THEN TB=53248:REM SCREEN AND TEXT BASE ADDR'S
1910 OS=320*RW+8*CM:REM  OFFSET FROM CHARACTER SCREEN BASE
1920 POKE 56334,PEEK(56334)AND 254:REM DISABLE IRQ TIMER
1930 POKE 1,PEEK(1)AND 251:REM SWITCH CHARACTER ROM IN
1940 L=LEN(A$):FOR N=1 TO L:N8=N8+N*8+OS+SB
1950 X=ASC(MID$(A$,N,1)):IF X>63 THEN X=X-64
1960 TC=TB+8*X
1970 ON MD GOTO 1980,1980,1990,2000,2010
1980 POKE 53231,36:GOTO 2020
1990 POKE 53231,17:GOTO 2020
2000 POKE 53231,49:GOTO 2020
2010 POKE 53231,81
2020 POKE252,N8/256:POKE251,N8-256*INT(N8/256)
2030 POKE254,TC/256:POKE253,TC-256*INT(TC/256)
2040 SYS(53221):NEXT
2050 POKE 1,PEEK(1)OR 4:POKE 56334,PEEK(56334)OR 1:REM RESTORE TO NORMAL
2060 RETURN
```

READY.

End Listing Six

Listing Seven

```
10 REM "STELLATION" DRAW A SMALL STELLATED DODECAHEDRON IN VARIOUS ORIENTATIONS
20 REM  AND STYLES    RICHARD L. RYLANDER      12/5/84
30 :
40 GR=49378     :REM GRAPHICS MODE
50 TX=49411     :REM TEXT MODE
60 BO=53280     :REM BORDER COLOR
70 :
80 REM  INITIALIZE A FEW STYLE PARAMETERS
90 POKE 839,1   :REM SCALE (3:4) FOR UNDISTORTED SCREEN DISPLAY
100 POKE 871,0 :REM FACET EDGE/LINE MODE (0=DRAW, 1=ERASE)
110 SH=838      :REM SHADE STYLE (0=RANDOM, 1=HALFTONE)
120 EG=868      :REM EDGES FLAG (0=NORMAL, 1=ADD LINES TO FACET EDGES)
130 :
140 REM   FUNCTION LOCATIONS
150 CL=51979    :REM CLEAR BITMAP
160 CO=52001    :REM FILL COLOR MAP
170 FC=52052    :REM DRAW A SHADED FACET
180 KS=53081    :REM DO KEYED SORT
190 :
200 XC=160:YC=120  :REM (SCALED) SCREEN CENTER COORDINATES
210 :
220 PRINT"{SC}     ********************************"
230 PRINT"     * SMALL STELLATED DODECAHEDRON *"
240 PRINT"     ********************************"
250 PRINT"{CD}{CD} SELECT SHADING STYLE:"
260 PRINT" R=RANDOM, H=HALFTONE"
270 INPUT"{CD} YOUR CHOICE    H{CL}{CL}{CL}";A$
280 POKE SH,0:IF A$="H" THEN POKE SH,1
290 PRINT"{CD}{CD} SELECT STYLE FOR DRAWING:"
300 PRINT"{CD} N - NORMAL":PRINT"{CD} E - EDGES EMPHASIZED"
310 PRINT"{CD} W - WIRE FRAME (HIDDEN LINES REMOVED)"
320 INPUT"{CD} YOUR CHOICE    N{CL}{CL}{CL}";A$
330 POKE EG,0:WI=0:IF A$="N" THEN 360
340 POKE EG,1:IF A$="W" THEN WI=-1
350 :
360 PRINT"{CD} READING VERTEX DATA"
370 VN=32:DIM P%(VN-1,2):REM  VN = NUMBER OF VERTICES
380 FOR N=0 TO VN-1:READ P%(N,0),P%(N,1),P%(N,2):NEXT
390 :
400 PRINT"{CD} ENTER X, Y, AND Z ANGLES FOR ROTATION"
410 PRINT"        (ANGLES IN DEGREES)"
420 INPUT X,Y,Z
430 J=3.14159265/180:X=X*J:Y=Y*J:Z=Z*J
440 X0=COS(Y)*COS(Z)-SIN(X)*SIN(Y)*SIN(Z):X1=COS(Y)*SIN(Z)+SIN(X)*SIN(Y)*COS(Z)
450 X2=-COS(X)*SIN(Y):Y0=-COS(X)*SIN(Z):Y1=COS(X)*COS(A):Y2=SIN(X)
460 Z0=SIN(Y)*COS(Z)+SIN(X)*COS(Y)*SIN(Z)
470 Z1=SIN(Y)*SIN(Z)-SIN(X)*COS(Y)*COS(Z):Z2=COS(X)*COS(Y)
480 PRINT"{CD} PERFORMING ROTATION"
490 FOR N=0 TO VN-1
500 X=P%(N,0):Y=P%(N,1):Z=P%(N,2)
510 P%(N,0)=X0*X+X1*Y+X2*Z:P%(N,1)=Y0*X+Y1*Y+Y2*Z:P%(N,2)=Z0*X+Z1*Y+Z2*Z:NEXT
520 :
530 FA=60:  REM  TOTAL NUMBER OF FACETS
540 DIM F%(FA/2,2),SH(FA/2),Z%(FA/2),K%(FA/2)
550 PRINT" READING CONNECTION DATA              "
560 VF=-1:  REM  VF = # VISIBLE FACETS
570 FOR N=1 TO FA
580 VF=VF+1
590 FOR I=0 TO 2 :READ F%(VF,I):NEXT
600 REM  CALCULATE COMPONENTS OF NORMAL VECTOR TO DETERMINE VISIBILITY
610 Z=(P%(F%(VF,2),0)-P%(F%(VF,1),0))*(P%(F%(VF,0),1)-P%(F%(VF,1),1))
620 Z=Z-(P%(F%(VF,0),0)-P%(F%(VF,1),0))*(P%(F%(VF,2),1)-P%(F%(VF,1),1))
630 IF Z<=0 THEN 720:REM  FACET NOT VISIBLE, GO ON
640 X=(P%(F%(VF,2),1)-P%(F%(VF,1),1))*(P%(F%(VF,0),2)-P%(F%(VF,1),2))
650 X=X-(P%(F%(VF,0),1)-P%(F%(VF,1),1))*(P%(F%(VF,2),2)-P%(F%(VF,1),2))
660 Y=(P%(F%(VF,2),2)-P%(F%(VF,1),2))*(P%(F%(VF,0),0)-P%(F%(VF,1),0))
670 Y=Y-(P%(F%(VF,0),2)-P%(F%(VF,1),2))*(P%(F%(VF,2),0)-P%(F%(VF,1),0))
680 NC=SQR(X*X+Y*Y+Z*Z):REM  LENGTH OF NORMAL VECTOR
690 SH(VF)=26*(2*Z+X+Y)/NC
700 SH(VF)=(SH(VF)+64)*(SH(VF)+64)/256:REM RAISED COSINE SHADING
710 GOTO 730
720 VF=VF-1
730 NEXT
740 :
750 PRINT" SCALING TO DISPLAY SIZE"
760 Y=0:FOR N=0 TO VN-1:IF ABS(P%(N,1))>Y THEN Y=ABS(P%(N,1))
770 NEXT:S=119/Y
780 FOR N=0 TO VN-1:P%(N,1)=S*P%(N,1)+YC:P%(N,0)=S*P%(N,0)+XC:NEXT
790 :
800 REM FIND AVERAGE Z FOR EACH FACET
810 FOR N=0 TO VF
820 Z%(N)=(P%(F%(N,0),2)+P%(F%(N,1),2)+P%(F%(N,2),2))/3:NEXT
830 :
840 PRINT" SORTING FACETS ACCORDING TO AVG 'Z' "
850 POKE 140,VF
860 K%(0)=K%(0):POKE 251,PEEK(71):POKE 252,PEEK(72)
870 Z%(0)=Z%(0):POKE 253,PEEK(71):POKE 254,PEEK(72)
880 SYS(KS)
890 :
900 REM DRAW FACETS (ACCORDING TO Z DEPTH SINCE NOT CONVEX)
910 SYS(GR):SYS(CO):SYS(CL):POKE BO,1
920 FOR N=0 TO VF:FA=K%(N)
930 IF WI THEN SH(FA)=64
940 X0=P%(F%(FA,0),0):Y0=P%(F%(FA,0),1):X1=P%(F%(FA,1),0):Y1=P%(F%(FA,1),1)
950 X2=P%(F%(FA,2),0):Y2=P%(F%(FA,2),1)
960 SYS(FC),X0,Y0,X1,Y1,X2,Y2,SH(FA)
970 NEXT
980 POKE198,0
990 GET A$:IF A$="" THEN 990
1000 SYS(TX):POKE BO,14:END
1010 :
1020 REM       VERTEX DATA
1030 DATA 1000,618,0, 1000,-618,0, -1000,618,0, -1000,-618,0
1040 DATA 0,1000,618, 0,1000,-618, 0,-1000,618, 0,-1000,-618
1050 DATA 618,0,1000, -618,0,1000, 618,0,-1000, -618,0,-1000
1060 DATA 618,0,236, 618,0,-236, -618,0,236, -618,0,-236
1070 DATA 236,618,0, -236,618,0, 236,-618,0, -236,-618,0
1080 DATA 0,236,618, 0,-236,618, 0,236,-618, 0,-236,-618
1090 DATA 382,382,382, 382,382,-382, 382,-382,382, 382,-382,-382
1100 DATA -382,382,382, -382,382,-382, -382,-382,382, -382,-382,-382
1110 :
1120 REM    CONNECTION DATA
1130 DATA 0,12,13, 0,13,25, 0,25,16, 0,16,24, 0,24,12
1140 DATA 1,12,26, 1,26,18, 1,18,27, 1,27,13, 1,13,12
1150 DATA 2,15,14, 2,14,28, 2,28,17, 2,17,29, 2,29,15
1160 DATA 3,14,15, 3,15,31, 3,31,19, 3,19,30, 3,30,14
1170 DATA 4,16,17, 4,17,20, 4,20,20, 4,20,24, 4,24,16
1180 DATA 5,17,16, 5,16,25, 5,25,22, 5,22,29, 5,29,17
1190 DATA 6,19,18, 6,18,26, 6,26,21, 6,21,30, 6,30,19
1200 DATA 7,18,19, 7,19,31, 7,31,23, 7,23,27, 7,27,18
1210 DATA 8,20,21, 8,21,26, 8,26,12, 8,12,24, 8,24,20
1220 DATA 9,21,20, 9,20,28, 9,28,14, 9,14,30, 9,30,21
1230 DATA 10,23,22, 10,22,25, 10,25,13, 10,13,27, 10,27,23
1240 DATA 11,22,23, 11,23,31, 11,31,15, 11,15,29, 11,29,22
```

READY.

End Listing Seven

Listing Eight

```
00001  0000                ; KEYSORT - RELOCATABLE BUBBLE SORT USING KEY ARRAY
00002  0000                ; POINTING TO INTEGER ARRAY
00003  0000                ;
00004  0000                ;  RICHARD L. RYLANDER    1/12/85
00005  0000                ;
00006  0000                ORIGIN=$CF59            ; 53081. (FOLLOWING DOS 5.1)
00007  0000                ;
00008  0000                KB      = $FB           ; 251. POINTER TO KEY ARRAY
00009  0000                ZB      = $FD           ; 253. POINTER TO DATA ARRAY
00010  0000                MAX     = $8C           ; 140. POKE WITH MAX ARRAY INDEX
00011  0000                TOP     = $AC
00012  0000                TOPDIS  = $AD
00013  0000                FLAG    = $AE
00014  0000                NXTFLG  = $61
00015  0000                CRRNT   = $62
00016  0000                REPEAT  = $64
00017  0000                ;
00018  0000                        *=ORIGIN
00019  CF59                ;
00020  CF59  A0 FF         INIT    LDY #$FF        ; INITIALIZE KEY ARRAY
00021  CF5B  C8            INLOOP  INY
00022  CF5C  98                    TYA
00023  CF5D  91 FB                 STA (KB),Y
00024  CF5F  C5 8C                 CMP MAX
00025  CF61  D0 F8                 BNE INLOOP
00026  CF63                ;
00027  CF63  85 AD         SORT    STA TOPDIS      ; 'A' HOLDS 'MAX'
00028  CF65  A5 AD         LOOP1   LDA TOPDIS
00029  CF67  85 AC                 STA TOP
00030  CF69  A2 00                 LDX #0
00031  CF6B  86 61                 STX NXTFLG
00032  CF6D  86 AE                 STX FLAG
00033  CF6F  86 64         LOOP2   STX REPEAT
00034  CF71                ;
00035  CF71                ; GET BOTH BYTES OF INTEGER POINTED TO BY
00036  CF71                ; 'KEY' ELEMENT.  RETURN WITH MSB ON STACK
00037  CF71                ; AND LSB IN THE ACCUMULATOR
00038  CF71                ;
00039  CF71  8A            GETINT  TXA
00040  CF72  A8                    TAY
00041  CF73  B1 FB                 LDA (KB),Y
00042  CF75  0A                    ASL A
00043  CF76  90 04                 BCC LOAD
00044  CF78  C6 61                 DEC NXTFLG
00045  CF7A  E6 FE                 INC ZB+1
00046  CF7C  A8            LOAD    TAY
00047  CF7D  B1 FD                 LDA (ZB),Y
00048  CF7F  48                    PHA
00049  CF80  C8                    INY
00050  CF81  B1 FD                 LDA (ZB),Y
00051  CF83  24 61                 BIT NXTFLG
00052  CF85  10 04                 BPL NODEC
00053  CF87  E6 61                 INC NXTFLG
00054  CF89  C6 FE                 DEC ZB+1
00055  CF8B  E4 64         NODEC   CPX REPEAT
00056  CF8D  D0 08                 BNE ORDER
00057  CF8F  85 62                 STA CRRNT
00058  CF91  68                    PLA
00059  CF92  85 63                 STA CRRNT+1
00060  CF94  E8                    INX
00061  CF95  D0 DA                 BNE GETINT
00062  CF97                ;
00063  CF97                ; COMPARE INTEGERS OBTAINED THROUGH KEY ARRAY
00064  CF97                ; IF 'CURRENT' >= 'NEXT' THEN SWAP KEY
00065  CF97                ; ELEMENTS, ELSE CONTINUE
00066  CF97                ;
00067  CF97  C5 62         ORDER   CMP CRRNT
00068  CF99  68                    PLA
00069  CF9A  E5 63                 SBC CRRNT+1
00070  CF9C  50 02                 BVC TEST
00071  CF9E  49 80                 EOR #$80
00072  CFA0  10 13         TEST    BPL NOSWAP
00073  CFA2  8A            SWAP    TXA
00074  CFA3  A8                    TAY
00075  CFA4  86 AD                 STX TOPDIS
00076  CFA6  B1 FB                 LDA (KB),Y
00077  CFA8  48                    PHA
00078  CFA9  88                    DEY
00079  CFAA  B1 FB                 LDA (KB),Y
00080  CFAC  C8                    INY
00081  CFAD  91 FB                 STA (KB),Y
00082  CFAF  68                    PLA
00083  CFB0  88                    DEY
00084  CFB1  91 FB                 STA (KB),Y
```

(Continued on page 82)

Drawing on the C-64

(Listing Continued, text begins on page 50)

Listing Eight

```
00085  CFB3  E6 AE           INC FLAG
00086  CFB5  E4 AC     NOSWAP CPX TOP
00087  CFB7  D0 B6           BNE LOOP2
00088  CFB9  A5 AE           LDA FLAG
00089  CFBB  D0 A8           BNE LOOP1
00090  CFBD                  ;
00091  CFBD                  ; UNPACK THE BYTE ELEMENTS OF THE 'KEY' ARRAY
00092  CFBD                  ; INTO BASIC'S NORMAL 2-BYTE INTEGER FORMAT
00093  CFBD                  ;
00094  CFBD  A6 BC     UNPACK LDX MAX
00095  CFBF  E8               INX
00096  CFC0  CA        PKLOOP DEX
00097  CFC1  8A               TXA
00098  CFC2  A8               TAY
00099  CFC3  B1 FB            LDA (KB),Y
00100  CFC5  48               PHA
00101  CFC6  8A               TXA
00102  CFC7  0A               ASL A            ; MOVE TO 2*I+1
00103  CFC8  09 01            ORA #1
00104  CFCA  90 04            BCC STORE
00105  CFCC  E6 61            INC NXTFLG
00106  CFCE  E6 FC            INC KB+1
00107  CFD0  A8        STORE  TAY
00108  CFD1  68               PLA
00109  CFD2  91 FB            STA (KB),Y
00110  CFD4  A9 00            LDA #0
00111  CFD6  88               DEY
00112  CFD7  91 FB            STA (KB),Y
00113  CFD9  A5 61            LDA NXTFLG
00114  CFDB  F0 04            BEQ OK
00115  CFDD  C6 61            DEC NXTFLG
00116  CFDF  C6 FC            DEC KB+1
00117  CFE1  8A        OK     TXA
00118  CFE2  D0 DC            BNE PKLOOP
00119  CFE4  60        DONE   RTS
00120  CFE5                  .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 CRRNT    0062   DONE     CFE4   FLAG     00AE   GETINT   CF71
 INIT     CF59   INLOOP   CF5B   KB       00FB   LOAD     CF7C
 LOOP1    CF65   LOOP2    CF6F   MAX      00BC   NODEC    CF8B
 NOSWAP   CFB5   NXTFLG   0061   OK       CFE1   ORDER    CF97
 ORIGIN   CF59   PKLOOP   CFC0   REPEAT   0064   SORT     CF63
 STORE    CFD0   SWAP     CFA2   TEST     CFA0   TOP      00AC
 TOPDIS   00AD   UNPACK   CFBD   ZB       00FD

END OF ASSEMBLY
```

End Listing Eight

Listing Nine

```
00001  0000                  ;   "WRITE"   RICHARD L. RYLANDER
00002  0000                  ;             12/30/84
00003  0000                  ; REVISED  1/19/85  - ORIGIN MOVED TO $CFE5 (53221.)
00004  0000                  ;
00005  0000                  ;  PUT TEXT CHARACTERS ON GRAPHIC SCREEN
00006  0000                  ;  (UNDER BASIC ROM) IN VARIOUS STYLES
00007  0000                  ;
00008  0000                  *=$CFE5                ; PUT CODE AFTER DOS 5.1
00009  CFE5  A5 01     WRITE  LDA $01                ; SWITCH OUT BASIC ROM
00010  CFE7  29 FE            AND #$FE
00011  CFE9  85 01            STA $01
00012  CFEB  A0 07            LDY #7
00013  CFED  B1 FD     LOOP   LDA ($FD),Y            ; READ CHARACTER BYTE
00014  CFEF  31 FB            AND ($FB),Y            ; MODIFY W/SCREEN BYTE
00015  CFF1  91 FB            STA ($FB),Y            ; STORE IN SCREEN
00016  CFF3                  ;
00017  CFF3                  ; POKE NEW LOGICAL OPERATOR TO REPLACE
00018  CFF3                  ; 'AND' (53231.) FOR DIFFERENT STYLES
00019  CFF3                  ; ORA=17.  BIT (NOP)=36.  AND=49.  EOR=81.
00020  CFF3                  ;
00021  CFF3  88               DEY
00022  CFF4  10 F7            BPL LOOP
00023  CFF6  A5 01            LDA $01                ; RESTORE BASIC ROM
00024  CFF8  09 01            ORA #1
00025  CFFA  85 01            STA $01
00026  CFFC  60               RTS
00027  CFFD                  .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 LOOP     CFED    WRITE    CFE5

END OF ASSEMBLY
```

End Listings

A Compiler Written in Prolog

by G. A. Edgar

Why would anyone write a compiler in a language like Prolog? Actually, the compiler seems to have come out rather well. I wrote it primarily as a learning exercise for Micro-PROLOG.

Micro-PROLOG

I first heard of the Prolog programming language in connection with the Japanese effort to develop the "fifth generation" in computing. At the time, I read a few things about Prolog, such as the book by Clocksin and Mellish.[1] My impression was that Prolog is an interesting language (I was right), but that it probably is not practical to implement PROLOG on a microcomputer (was I ever wrong!).

The early returns on our March special issue on Prolog are encouraging—so much so that we're offering this excursion into programming in logic.

D. E. Cortesi, in "Dr. Dobb's Clinic" for May 1984, gave a half-page description of Micro-PROLOG. I wrote for information on it and by July was running it on my CP/M Z80 computer. Micro-PROLOG is also available for MSDOS, PCDOS, CP/M-86, and Unix.

Micro-PROLOG comes with an IBM PC-size loose-leaf manual with more than 200 pages. This is a complete reference to the language but not a good tool for learning. Micro-PROLOG, however, also comes with a 400-page soft-cover tutorial, which is a good place for a beginner to start. Learning the language will take a substantial effort for a programmer experienced only in Pascal, C, or PL/I. The tutorial[2] is published separately as a book; if you are curious, it should be available in most large university libraries.

After I began to get a feel for the language, I used it to solve a few logic puzzles. For example, Lewis Carroll's logic book[3] includes some puzzles at the end as a teaser for a later book that was never published. They are much too complicated to solve simply by common sense reasoning, but I can now say definitively that "All of the Monitors are awake."

If Prolog were good only for logic problems, it would not be of wide interest. So I looked around for some other relatively easy task to use as a programming exercise. A compiler for VALGOL fit the need nicely. It improved my understanding of Micro-PROLOG and my understanding of what a compiler does.

This compiler is described below. It translates VALGOL I, also described below, into 8080 assembly language. Of course, you would learn more by undertaking such a programming project yourself than you will by examining mine, but it serves as proof that you can use Micro-PROLOG for something unexpected.

Using Micro-PROLOG

Before I describe the compiler, let me give a brief description of how Micro-PROLOG differs from other versions of Prolog.

G. A. Edgar, 107 W. Dodridge St., Columbus, OH 43202

Upper- and lower-case letters are considered different, and the hyphen is treated like a letter. Variables are X, Y, Z, x, y, z, or one of these six letters followed by a positive integer, such as X132 or z22. Other strings of letters and numbers are identifiers for constants.

Micro-PROLOG consists primarily of a pattern matcher, plus a few other built-in routines, such as a routine for I/O. The way Micro-PROLOG looks depends on which of the front ends you are using. Each front-end program is written in Prolog. If you are using the SIMPLE front end, one-place predicates can be given either in prefix form

```
alphanumeric(X)
```

or in postfix form

```
X alphanumeric
```

Two-place predicates can be given either in prefix form

```
LESS(@ X)
```

or in infix form

```
@ LESS X
```

Predicates with three or more arguments must be in prefix form. Thus, a Prolog program entered via SIMPLE tends to look like this:

```
labelused (0)
X alphanumeric if X alphabetic
X alphanumeric if X digit
X alphabetic if @ LESS X and X
    LESS [
X alphabetic if ' LESS X and X
    LESS {
```

One useful object in Micro-PROLOG is the list. The notation for a list is to enclose the items between parentheses: (A B C 1 2 3). Some of the items might be lists themselves: (A (B C) (1 2 3)).

The 16-bit versions of Micro-PROLOG include a front-end supervisor that imitates the DEC-10 Prolog:

```
labelused(0).
alphanumeric(X) :- alphabetic (X).
alphanumeric(X) :- digit(X).
```

The form in which an operator is used is declared using the types fx, xf, xfy, and so on (see Clocksin and Mellish, page 91). DEC-10 lists are built using square brackets: [a,b,c].

The standard Micro-PROLOG syntax simply uses lists. In this regard, it is somewhat like LISP. The list (labelused 0), which indicates the predicate name "labelused" applied to the argument "0," is an example of an atom. A clause is a list of atoms. You add things to the data base in this way:

```
((labelused 0))
((alphanumeric X) (alphabetic X))
((alphanumeric X) (digit X))
```

This may seem confusing at first, but after a while it becomes more or less routine.

This minimal introduction to the Micro-PROLOG interpreter should be enough for you to gain some understanding of the VALGOL compiler, but you will need a much more complete knowledge of Micro-PROLOG to understand all of the details.

VALGOL I

The language VALGOL I (very small ALGOL?) is a derivative of ALGOL-60. D. V. Schorre described it in 1964 as a sample language for the compiler-writing language META II. His paper[4] was reprinted in *Dr. Dobb's Journal* (April 1980), which is where I found it. I had worked with VALGOL I in connection with the compiler-writing language Meta4 (in a SIG/M disk), so it seemed natural to use it for the same thing in Prolog.

VALGOL I has a few peculiarities. The keywords that usually are typeset in boldface here are preceded by a period (.begin, .end, .if, .then, .else, .until, .do, .integer). This also applies to the equals sign for comparing two expressions (.=). VALGOL I has only one data type, namely .integer, a 16-bit two's-complement number. The assignment statement is the reverse from the normal order (5 =: x assigns the value 5 to x). Arithmetic allows addition, subtraction, and multiplication, but there is no unary minus sign, so if you want – 2 you must write either 0 – 2 or 65534.

A more complete description of the syntax appears below in the description of the compiler.

Using the VALGOL I Compiler

The VALGOL I compiler is in a file called VALGOL.LOG. A typical use, to compile the short VALGOL program in the file P.VAL, is shown in Listing One (page 89). The command shown is in the standard Micro-PROLOG syntax. If the SIMPLE front end is also loaded, you could use the command:

```
is("P.VAL" compile "P.ASM")
```

The compiler prints out the input file P.VAL on the console as it works. The result of the compilation in the sample is P.ASM, included here as Listing Four (page 96).

If you are familiar with the Intel mnemonics for the 8080, you may find it instructive to compare the VALGOL input with the assembly language output to see how the compiler works.

The VALGOL Definition

Take a look at the compiler in Listing Two (page 90). I have repeated part of the compiler using the SIMPLE syntax in Listing Three (page 96). SIMPLE is unsuitable for the complete listing for several reasons. In a few places in the compiler, I have used "meta-variables," which are not supported by SIMPLE. A SIMPLE listing also would not keep the comments and the formatting, making it confusing to read. So try to understand instead the list notation used for the standard syntax of Micro-PROLOG in Listing Two.

The lists preceded by a ? are to be executed immediately when they are loaded; the others are to be stored in the data base. Because the predicate symbol (/ *) is a special one that is always true, a list in the file of the form

```
?(( / * anything goes here ))
```

will serve as a comment. (This is a clumsy feature of Micro-PROLOG.)

The compiler listing is in two parts. The first part is independent of the particular syntax of VALGOL I and could be used equally well with many

other programming languages. The second, below a horizontal line, is a VALGOL-specific part. In one sense, the part above the line implements an interpreter that runs the compiler described below the line. Let's examine a portion of the VALGOL-specific part of the code.

The message line, like all these lines, becomes a statement in the Prolog data base. The predicate symbol of the statement is "message," and the single argument is the string enclosed in quotation marks. The compiler will look for a message fact in the data base and print it out when a compilation starts.

The next line is the syntax line. This tells the compiler that the primary unit of syntax is the program.

The next statement is a description of a program. First comes the keyword *.begin. When* this is matched, seven opening lines are sent to the output file. (I have put the VALGOL syntax description on the left-hand side of the page and the ASM output description on the right-hand side. A VALGOL I compiler for another processor would keep the same left-hand side and change the right-hand side.) Then an opt-declaration (optional declaration, see below) is followed by one or more statements separated by semicolons. The multiple predicate allows matching of its arguments zero or more times. Finally comes the keyword .end. When this is matched, the final JMP is inserted in the output file, followed by the four subroutines and the stack space.

The opt-declaration could be a declaration followed by a semicolon, or (if that cannot be found) it could be empty, which matches the input file automatically. Inclusion of the "empty" alternative is a way to allow for no declaration. A declaration is the keyword .integer, followed by one or more identifiers separated by commas. For each identifier, the output file reserves a two-byte storage location.

A statement is one of the following:
(1) An I/O statement, either

```
edit( expression , 'string' )!
```

which will send (expression) spaces and then the (string) to the console, or

```
print
```

which will send an end-of-line to the console.
(2) An assignment statement:

```
expression =: variable
```

(3) A loop:

```
.until expression
   .do statement
```

A value of zero for the expression is considered false, and a nonzero value is true.
(4) A conditional statement:

```
.if expression
   .then statement
   .else statement
```

The .else is not optional.
(5) A block, which consists of the keyword .begin, followed by an optional declaration, followed by zero or more statements separated by semicolons, followed by the keyword .end. The null statement

```
.begin .end
```

is allowed.

The following test for equality is allowed:

```
expression .= expression
```

This has value 1 if true and 0 if false. Expressions can be built up from variables, numbers, the operations +, –, *, and parentheses. Notice how the syntax near the end of Listing Two is arranged to take into account the precedence of these operations.

The Language-Independent Part

Now let's look at the first part of Listing Two (part of this is repeated in SIMPLE syntax as Listing Three). When Micro-PROLOG attempts to verify a compile atom, it looks first in the data base for a message fact; if it finds one, it verifies the assertion (P x), which is always true but which has the side effect of printing x on the console. The assertion PP is also always true and starts a new line at the console. The compile atom continues as it prints X –> Y, where X is the first argument (presumably the filename of the VALGOL input file) and Y is the second argument (the ASM output file).

Next the compiler deletes all facts about "labelused" (which may be left from a previous compile) and starts over with (labelused 0). It does something similar with "last-shown," which keeps track of how much of the file has been shown on the console.

Then the compiler opens the input file X and adds a fact to the data base so that later it can find the name of the input file. Similarly, it opens the output file Y. It skips white-space characters in the input file, starting at the beginning and ending at location Z1 (x in Listing Three). Then it finds the syntax fact to tell it what to start compiling with and feeds that to the Q routine, which carries out the actual compile.

Finally, the compiler closes the files and prints out a concluding message. If compilation fails for some reason, control goes to the second compile clause, and it prints out the error message.

Now look at Q. This short routine implements the recursive-descent compile, according to the code in the last half of the compiler. The first two arguments are the before and after file positions for both the input and output files. Whenever Q has to backtrack, the files automatically are rewound to the appropriate point for another try.

Notice the use of the variable Z (the third argument of Q) as the predicate symbol in a search of the data base. This is called a "meta-variable" in Micro-PROLOG: the name of the predicate to be verified is supplied as a variable, not as a constant. This powerful feature of Micro-PROLOG may not be available in other Prologs. The X1 after the vertical line, which indicates the rest of the list, is another meta-variable. The remainder of this half of the code comprises various subroutines for use in the compile.

This is a very brief description of only a small part of this compiler. A complete understanding requires a

greater familiarity with the many features and quirks of the Micro-PROLOG package.

Conclusion

The compiler shown in Listing Two is written in the standard syntax of Micro-PROLOG. As such, it should run under any version of Micro-PROLOG. I would be interested in hearing from readers who get it to work under other versions. (Of course, the output would still be in 8080 assembly language.)

This compiler runs rather slowly. The common belief is that interpreters are slow, and this is an interpreter within an interpreter. The 10-line sample compilation in Listing One takes about 30 sec on my 4 MHz Z80. Because I wrote the compiler primarily as a programming exercise, I have not been overly concerned with a remedy for the speed problem. Probably the bottleneck occurs in the input routines that process one character at a time. For a faster compiler, routines such as match, id, or number would be recoded in assembly language. The Micro-PROLOG manual has instructions on how to link such new built-in functions in assembly code with Micro-PROLOG.

References

1. W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* Springer-Verlag, 1981.
2. K. L. Clark and F. G. McCabe, *Micro-PROLOG: Programming in Logic,* Prentice-Hall, 1984.
3. Lewis Carroll, *Symbolic Logic,* 4th edition, 1896 (Reprinted by Dover Publications, 1958).
4. D. V. Schorre, "META II, a syntax-oriented compiler writing language," *Proc. 19th Nat. Conf. ACM,* 1964 (Reprinted in *Dr. Dobb's Journal,* April 1980, pp. 17–21; listings pp. 22–25).

DDJ

Compiler in Prolog (Text begins on page 84)

Listing One

Use of the compiler

```
B>PROLOG LOAD VALGOL

Micro-PROLOG 3.1 for RML 380/480Z 29/03/84
(c) 1984 Logic Programming Associates Ltd.
38517 Bytes Free

&&.?((compile "P.VAL" "P.ASM"))
VALGOL I compiler - translates VALGOL to ASM
P.VAL -> P.ASM

.begin
.integer x;
0 =: x;
.until x .= 15 .do
      .begin
      edit (1+14*x-x*x, '+');
      print;
      x+1 =: x
      .end
.end

** Compilation of "P.VAL" complete **
&.QT.

B>ASM P.BBZ

CP/M ASSEMBLER - VER 2.0
01F7
001H USE FACTOR
END OF ASSEMBLY

B>LOAD P


FIRST ADDRESS 0100
LAST  ADDRESS 01F6
BYTES READ    00B9
RECORDS WRITTEN 02

B>P

 +
               +
                        +
                                 +
                                          +
                                                   +
                                                    +
                                                     +
                                                    +
                                          +
                                 +
                        +
               +
 +

type p.val

.begin
.integer x ;
0 =: x ;
.until x .= 9 .do
         .begin
         edit ( x*x + 1 , '+' ) ;
         print ;
         x + 1 =: x
         .end
.end

A>p

 +
  +
     +
          +
                   +
                            +
                                     +
                                              +
                                                       +
                                                                +
```

End Listing One

Compiler in Prolog (Listing Continued, text begins on page 84)

Listing Two

VALGOL.LOG

```
?(( /*
   VALGOL.LOG
   Compiler for VALGOL I -- ver. 1.2

   Written in Micro-PROLOG.
   (Tested with Micro-PROLOG version 3.1 for CP/M-80)
   Translates VALGOL I to ASM-compatible 8080 assembly language.
   Uses file SEEKs for backtracking.

        written by: G. A. EDGAR     status: public domain

   0.3      September 9, 1984       M80, Z80
   1.0      September 16, 1984      output for ASM
   1.1      November 1, 1984        display input
   1.2      November 10, 1984       version for DDJ

   Usage - in SIMPLE syntax:
        &.is("FOO.VAL" compile "FOO.ASM")
         - in standard syntax:
        &.?((compile "FOO.VAL" "FOO.ASM"))
))


?(( /* ---------- Language independent part ---------------))

?(( /* compile: input X, output Y))
((compile X Y)
    (message x) (P x) (PP) (P X "->" Y) (PP) (PP)
    (KILL labelused) (ADDCL ((labelused 0)) )
    (KILL last-shown) (ADDCL ((last-shown (-1|-1))) )
    (OPEN X) (KILL infile) (ADDCL ((infile X)) )
    (erase Y) (CREATE Y) (KILL outfile) (ADDCL ((outfile Y)) )
    (skip ((0|0)|(0|0)) Z1 ) (syntax z) (Q Z1 Z z)
    (CLOSE X) (CLOSE Y)
    (PP) (PP ** Compilation of X complete **))
((compile X Y)
    (PP) (PP ** Syntax error **)
    (CLOSE X) (CLOSE Y))
((C X Y)
    (compile X Y))

?(( /* Q:  find it in the language-specific database))
((Q X Y Z)
    (Z | X1) (sequential X Y | X1))
((sequential X X))
((sequential X Y (z|Z)|y)
    (z X X1|Z) / (sequential X1 Y|y))

?(( /* out: send a line to outfile))
((out (y1|z1) (y1|z2) | X)
    (outfile Z) (SEEK Z z1) (outx Z | X)
    (outx Z "^M^J") (SEEK Z z2))
((outx Z))
((outx Z X | x)
    (W Z (X)) / (outx Z | x))

?(( /* match: the input matches string Z))
((match X Y Z)

    (STRINGOF Z1 Z) / (matchx X X2 Z1) (skip X2 Y))
((matchx X Y (x|Z))
    (inchar X X1 x1) (EQ x x1) / (matchx X1 Y Z))
((matchx X X ()))

?(( /* empty: matches automatically))
((empty X X))

?(( /* multiple: match the following zero or more times))
((multiple X Y|Z)
    (once X X1|Z) / (multiple X1 Y|Z))
((multiple X X|Z))
((once X X))
((once X Y (z|Z)|y)
    (z X X1|Z) / (once X1 Y|y))

?(( /* label: generate a new label))
((label X X Y)
    (labelused Y1) (SUM Y1 1 Y) (KILL labelused)
    (ADDCL ((labelused Y)) ))
```

Compiler in Prolog

(Listing Continued, text begins on page 84)

Listing Two

```
?(( /* string: match a string quoted within characters x))
((string X Y x Z)
    / (inchar X X2 x1) (EQ x x1) /
    (stringx X2 X3 x Z1) (skip X3 Y) (STRINGOF Z1 Z))
((stringx X Y x ())
    (inchar X Y x1) (EQ x x1) / )
((stringx X Y x (y | Z))
    (inchar X X1 y) / (stringx X1 Y x Z))

?(( /* id: match an identifier))
((id X Y Z)
    (idx X X2 Z1) (skip X2 Y) (STRINGOF Z1 Z))
((idx X Y (x|Z))
    (inchar X X1 x) (alphabetic x) / (alphanum X1 Y Z))
((alphanum X Y (x|Z))
    (inchar X X1 x) (alphanumeric x) / (alphanum X1 Y Z))
((alphanum X X ()))

?(( /* number: match a number))
((number X Y Z)
    / (numberx X X2 Z1)
    (NOT EQ Z1 ()) (skip X2 Y) (STRINGOF Z1 Z))
((numberx X Y (x|Z))
    (inchar X X1 x) (digit x) / (numberx X1 Y Z))
((numberx X X ()))

((alphanumeric x)
    (alphabetic x))
((alphanumeric x)
    (digit x))

((alphabetic x)
    (LESS "@" x)(LESS x "["))
((alphabetic x)
    (LESS "`" x)(LESS x "{"))

((digit x)
    (LESS "/" x)(LESS x ":"))

?(( /* skip: skip over characters listed as "skipchar"))
((skip X Y)
    (inchar X X1 x) (skipchar x) / (skip X1 Y))
((skip X X))

((skipchar " "))
((skipchar "^I"))
((skipchar "^J"))
((skipchar "^M"))

?(( /* inchar: input one character, show on console first time))
((inchar (y1|z1) (y2|z1) x)
    (infile Y) (SEEK Y y1) (RDCH Y x) (SEEK Y y2) (last-shown y3)
    (IF (before y3 y2)
      ((P x) (KILL last-shown) (ADDCL ((last-shown y2)) ))
      ()))

?(( /* before: compare two file positions ))
((before (y1|z1) (y1|z2))
    / (LESS z1 z2))
((before (y1|z1) (y2|z2))
    (LESS y1 y2))

?(( /* erase: erase a file, no failure))
((erase Y)
    (ERA Y) /)
((erase Y))

?(( /* --- VALGOL specifics ------------------------------ ))

((message "VALGOL 1 compiler - translates VALGOL to ASM"))
((syntax PROGRAM))

((PROGRAM
    (match ".begin")            (out    "VCPM   EQU     0")
                                (out    "VBDOS  EQU     5")
                                (out    "VTPA   EQU     256")
                                (out    "VCR    EQU     13")
                                (out    "VLF    EQU     10")
                                (out    "       ORG     VTPA")
                                (out    "       LXI     SP,VSTACK")
    (Q OPT-DECLARATION)
    (Q STATEMENT)
```

```
    (multiple
        (match ";")
        (Q STATEMENT))
    (match ".end")                  (out    "       JMP     VCPM")
                                    (out    "VMULT:")
                                    (out    "       MOV     B,H")
                                    (out    "       MOV     C,L")
                                    (out    "       XRA     A")
                                    (out    "       MOV     H,A")
                                    (out    "       MOV     L,A")
                                    (out    "       MVI     A,16")
                                    (out  "VMULT1:  PUSH    PSW")
                                    (out    "       DAD     H")
                                    (out    "       XRA     A")
                                    (out    "       MOV     A,C")
                                    (out    "       RAL")
                                    (out    "       MOV     C,A")
                                    (out    "       MOV     A,B")
                                    (out    "       RAL")
                                    (out    "       MOV     B,A")
                                    (out    "       JNC     VMULT2")
                                    (out    "       DAD     D")
                                    (out  "VMULT2:  POP     PSW")
                                    (out    "       DCR     A")
                                    (out    "       ORA     A")
                                    (out    "       JNZ     VMULT1")
                                    (out    "       RET")
                                    (out    "VEDIT:")
                                    (out   "        MOV     A,H")
                                    (out    "       ORA     L")
                                    (out    "       JZ      VEDIT1")
                                    (out    "       MVI     A,' '")
                                    (out    "       CALL    VCPMOUT")
                                    (out    "       DCX     H")
                                    (out    "       JMP     VEDIT")
                                    (out  "VEDIT1:  POP     H")
                                    (out  "VEDIT2:  MOV     A,M")
                                    (out    "       CPI     0")
                                    (out    "       INX     H")
                                    (out    "       JZ      VEDIT3")
                                    (out    "       CALL    VCPMOUT")
                                    (out    "       JMP     VEDIT2")
                                    (out  "VEDIT3:  PUSH    H")
                                    (out    "       RET")
                                    (out    "VPRINT:")
                                    (out    "       MVI     A,VCR")
                                    (out    "       CALL    VCPMOUT")
                                    (out    "       MVI     A,VLF")
                                    (out    "       CALL    VCPMOUT")
                                    (out    "       RET")
                                    (out    "VCPMOUT:")
                                    (out    "       PUSH    H")
                                    (out    "       MOV     E,A")
                                    (out    "       MVI     C,2")
                                    (out    "       CALL    VBDOS")
                                    (out    "       POP     H")
                                    (out    "       RET")
                                    (out    "       DS      60")
                                    (out  "VSTACK:  DW      0")
                                    (out    "       END") ))

((OPT-DECLARATION
    (Q DECLARATION)
    (match ";")))
((OPT-DECLARATION
    (empty)))

((DECLARATION
    (match ".integer")              (label X1)
                                    (out    "       JMP     V" X1)
    (Q ID-SEQUENCE)                 (out    "V" X1 ":") ))

((ID-SEQUENCE
    (Q IDENTIFIER)
    (multiple
        (match ",")
        (Q IDENTIFIER) )))

((IDENTIFIER
    (id Z)                        (out   Z "V:    DS      2") ))

((STATEMENT
    (Q IO-STATEMENT)))
((STATEMENT
    (Q ASSIGNMENT-STATEMENT)))
((STATEMENT
    (Q UNTIL-STATEMENT)))
((STATEMENT
    (Q CONDITIONAL-STATEMENT)))
((STATEMENT
    (Q BLOCK)))
```

```
((BLOCK
    (match ".begin")
    (Q DECL-OR-ST)
    (multiple
        (match ";")
        (Q STATEMENT) )
    (match ".end") ))
((BLOCK
    (match ".begin")
    (match ".end") ))

((DECL-OR-ST
    (Q DECLARATION)))
((DECL-OR-ST
    (Q STATEMENT)))

((IO-STATEMENT
    (match "edit")
    (match "(")
    (Q EXPRESSION)
    (match ",")
    (string "'" Z)              (out    "       CALL    VEDIT")
                                (out    "       DB      '" Z "',0")
    (match ")") ))
((IO-STATEMENT
    (match "print")             (out    "       CALL    VPRINT") ))

((CONDITIONAL-STATEMENT
    (match ".if")               (label X1) (label X2)
    (Q EXPRESSION)
    (match ".then")             (out    "       MOV     A,H")
                                (out    "       ORA     L")
                                (out    "       JZ      V" X1)
    (Q STATEMENT)
    (match ".else")             (out    "       JMP     V" X2)
                                (out    "V" X1 ":")
    (Q STATEMENT)               (out    "V" X2 ":") ))

((UNTIL-STATEMENT
    (match ".until")            (label X1) (label X2)
                                (out    "V" X1 ":")
    (Q EXPRESSION)
    (match ".do")               (out    "       MOV     A,H")
                                (out    "       ORA     L")
                                (out    "       JNZ     V" X2)
    (Q STATEMENT)               (out    "       JMP     V" X1)
                                (out    "V" X2 ":") ))

((ASSIGNMENT-STATEMENT
    (Q EXPRESSION)
    (match "=:")
    (id Z)                      (out    "       SHLD    " Z "V") ))

((EXPRESSION
    (Q EXPRESSION1)
    (Q OPT-RIGHT-SIDE)))

((OPT-RIGHT-SIDE
    (match ",=")                (label X1) (label X2)
                                (out    "       PUSH    H")
    (Q EXPRESSION1)             (out    "       POP     D")
                                (out    "       MOV     A,L")
                                (out    "       SUB     E")
                                (out    "       JNZ     V" X2)
                                (out    "       MOV     A,H")
                                (out    "       SBB     D")
                                (out    "       JNZ     V" X2)
                                (out    "       LXI     H,1")
                                (out    "       JMP     V" X1)
                                (out    "V" X2 ":")
                                (out    "       LXI     H,0")
                                (out    "V" X1 ":") ))
((OPT-RIGHT-SIDE
    (empty)))

((EXPRESSION1
    (Q TERM)
    (multiple
        (Q SIGNED-TERM))))

((SIGNED-TERM
    (match "+")                 (out    "       PUSH    H")
    (Q TERM)                    (out    "       POP     D")
                                (out    "       DAD     D")))
```

```
((SIGNED-TERM
    (match "-")
    (Q TERM)
                                (out    "       PUSH    H")
                                (out    "       POP     D")
                                (out    "       MOV     A,E")
                                (out    "       SUB     L")
                                (out    "       MOV     L,A")
                                (out    "       MOV     A,D")
                                (out    "       SBB     H")
                                (out    "       MOV     H,A")  ))

((TERM
    (Q PRIMARY)
    (multiple
        (match "*")             (out    "       PUSH    H")
        (Q PRIMARY)             (out    "       POP     D")
                                (out    "       CALL    VMULT") )))

((PRIMARY
    (id Z)                      (out    "       LHLD    " Z "V")))
((PRIMARY
    (number Z)                  (out    "       LXI     H," Z)))

 ((PRIMARY
     (match "(")
     (Q EXPRESSION)
     (match ")") ))
```

End Listing Two

Compiler in Prolog

Listing Three

(Listing Continued, text begins on page 84)

Part of VALGOL.LOG in SIMPLE syntax

```
X compile Y if
     message (Z) and
     P (Z) and
     PP and
     PP and
     KILL (labelused) and
     add ((0 labelused)) and
     KILL (last-shown) and
     add (((-1|-1) last-shown)) and
     OPEN (X) and
     KILL (infile) and
     add ((X infile)) and
     erase (Y) and
     CREATE (Y) and
     KILL (outfile) and
     add ((Y outfile)) and
     skip (((0|0) 0|0) x) and
     syntax (y) and
     QQ (Q x z y) and
     CLOSE (X) and
     CLOSE (Y) and
     PP and
     PP (*** Compilation of X complete **)
X compile Y if
     PP and
     PP (** Syntax error **) and
     CLOSE (X) and
     CLOSE (Y)
X alphanumeric if
     X alphabetic
X alphanumeric if
     X digit
X alphabetic if
     @ LESS X and
     X LESS [
X alphabetic if
     ` LESS X and
     X LESS {
X digit if
     / LESS X and
     X LESS :
```

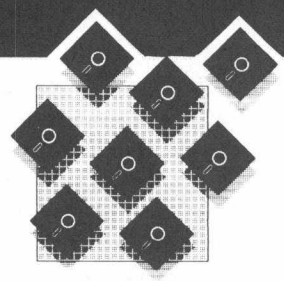
End Listing Three

Listing Four

P.ASM

```
VCPM    EQU     0
VBDOS   EQU     5
VTPA    EQU     256
VCR     EQU     13
VLF     EQU     10
        ORG     VTPA
        LXI     SP,VSTACK
        JMP     V1
xV:     DS      2
V1:
        LXI     H,0
        SHLD    xV
V2:
        LHLD    xV
        PUSH    H
        LXI     H,15
        POP     D
        MOV     A,L
        SUB     E
        JNZ     V5
        MOV     A,H
        SBB     D
        JNZ     V5
        LXI     H,1
        JMP     V4
V5:
        LXI     H,0
V4:
        MOV     A,H
        ORA     L
        JNZ     V3

        LXI     H,14
        PUSH    H
        LHLD    xV
        POP     D
        CALL    VMULT
        POP     D
        DAD     D
        PUSH    H
        LHLD    xV
        PUSH    H
        LHLD    xV
        POP     D
        CALL    VMULT
        POP     D
        MOV     A,E
        SUB     L
        MOV     L,A
        MOV     A,D
        SBB     H
        MOV     H,A
        CALL    VEDIT
        DB      '+',0
        CALL    VPRINT
        LHLD    xV
        PUSH    H
        LXI     H,1
        POP     D
        DAD     D
        SHLD    xV
        JMP     V2
V3:
        JMP     VCPM
VMULT:
        MOV     B,H
        MOV     C,L
        XRA     A
        MOV     H,A
        MOV     L,A
        MVI     A,16
VMULT1: PUSH    PSW
        DAD     H
        XRA     A
        MOV     A,C
        RAL
        MOV     C,A
        MOV     A,B
        RAL
        MOV     B,A
        JNC     VMULT2
        DAD     D
VMULT2: POP     PSW
        DCR     A
        ORA     A
        JNZ     VMULT1
        RET
VEDIT:
        MOV     A,H
        ORA     L
        JZ      VEDIT1
        MVI     A,' '
        CALL    VCPMOUT
        DCX     H
        JMP     VEDIT
VEDIT1: POP     H
VEDIT2: MOV     A,M
        CPI     0
        INX     H
        JZ      VEDIT3
        CALL    VCPMOUT
        JMP     VEDIT2
VEDIT3: PUSH    H
        RET
VPRINT:
        MVI     A,VCR
        CALL    VCPMOUT
        MVI     A,VLF
        CALL    VCPMOUT
        RET
VCPMOUT:
        PUSH    H
        MOV     E,A
        MVI     C,2
        CALL    VBDOS
        POP     H
        RET
        DS      60
VSTACK: DW      0
        END
```

End Listings



Toolworks C/80 and Toolworks C/80 Mathpack, Version 3.1

Company: Software Toolworks, 15233 Ventura Blvd., Suite 1118, Sherman Oaks, CA 91403

Computer: Heath; Kaypro II, 4, and 10; Osborne 1 and the Executive; DEC VT-180 and Rainbow; Xerox 820

Price: Compiler, $49.95; Mathpack, $29.95

***Circle Reader* Service No. 101**

Reviewed by D. C. Shoemaker

Few readers of *Dr. Dobb's Journal* are unaware of the contribution made to small computer system programmers by Ron Cain and his minimal C subset compiler. Over the years since its introduction, there have been many extensions and improvements, all building on his basic idea. One of the best and most complete versions is from Walt Bilofsky's Software Toolworks. First released in 1981, Version 3.1 is the most current and comes the closest to being a complete C compiler.

This software is a bargain at the price. The package comes in two parts, sold separately: the C compiler itself is $49.95, and the C/80 Mathpack is $29.95. There's a reason for the way the software is packaged and priced. Those just learning C probably have no need for the FLOAT and LONG data types that the Mathpack provides. Adding these features somewhat increases the size of the libraries and the assembled code. If you need them, Mathpack is a bargain-level entry into other more powerful capabilities given by the additional data types. Therefore, you don't have to pay for something you don't need, but you aren't limited to an entry-level compiler that can't do serious work.

C/80 is a complete implementation of the C language described by Brian Kernighan and Dennis Ritchie in *The C Programming Language* (Prentice-Hall, 1978) with the exception of the following features:

- FLOAT and LONG data types (available in the C/80 Mathpack)
- DOUBLE data type
- typedef
- Arguments to #define macros
- #line
- Declarations within nested blocks
- Bit fields

Whether or not these omissions are significant depends a great deal on what you want to do with the software. If you want to learn the language and get a whiff of programming in C, the C/80 package is sufficient. If you want to write some fast-running utility software and don't want to invest the time in assembly language, these omissions will cause no problems.

C/80 supports structures, statics, initialization, casts, compile time evaluation of constant expressions, and all the other C language features. It's worth mentioning some of the features that C/80 does provide, however, because they're often lacking in other subsets:

- Unix-style I/O redirection
- Conventional C I/O and string library
- Formatted and random access file I/O
- Dynamic storage allocation
- Runtime execution profile capability
- In-line assembly language capability

In addition, C/80 can generate source code compatible with Macro-80 and RMAC. Should you have neither, a serviceable assembler comes with the C/80 package. It works well and does the job with no fuss.

If you have a version of C/80 earlier than 3.0, here are the main changes in version 3.1:

- An expanded runtime library
- ROMable code
- RMAC compatibility
- A menu-configurable compiler
- \ at the end of a line for continuation
- #ifneed for selective compilation
- True alloc/free
- Command line wildcard expansion
- CP/M files now written in 128-byte records
- File 0 always the terminal

The changes from version 3.0 to 3.1 are to correct minor bugs and speed code generation. Some new compiler switches have been added, and you are now prohibited from reading from and writing to the same open file. If you have one of the earlier versions of C/80, return the original disk to Software Toolworks with $10.00, and they will give you the most recent version. I bought my first copy of C/80 in 1981 and have updated it twice.

Software Toolworks makes the C/80 package available in a variety of formats. Walt Bilofsky was one of the first to support Heath's in-house disk operating system HDOS, and he still does. However, the standard remains CP/M, and Software Toolworks provides disks for Heath, both hard and soft sector; Kaypro II, 4, and 10; Osborne I and the Osborne Executive; DEC's VT-180 and the Rainbow; the Xerox 820; and the standard 8-inch CP/M single-density format. There's

even a version for the Epson QX-10. Currently no version is offered for the IBM PC, PCjr, or any MSDOS, PCDOS, or ZDOS computer because the programs are written in 8080 code, but I expect that will change. Incidentally, I'm using the CP/M version on a Heath H120 running CP/M-85 with no trouble at all. Just order the soft-sector Heath version.

Using C/80 is simple. The hard part (as usual) is writing the original C language program. Then you call C/80 and let it compile the source into 8080 assembly language code. Let's use one of the test files on the distribution disk, HELLO.C. We do this by typing "C80 HELLO." The output file from C/80 can be modified if you wish. I'm not the greatest assembly language programmer, so I don't change the code, but someone more experienced might be able to tighten it up in some areas.

Next, call the assembler of your choice. I use the one that comes with the package. It's invoked as you might expect, by typing "AS HELLO." When the assembly is finished, you have an executable file called HELLO.COM. It will be rather large, due to the inclusion of pieces from the I/O and other libraries, but that's part of the tradeoff. The completed code executes relatively quickly. I don't have the capability to do much comparative benchmarking, but comparisons I've seen suggest that in some cases C/80 is a bit faster than BDS C and in others a bit slower. On balance they seem about equal, except that BDS C has some nonstandard features that get in the way of portability, and it costs twice what C/80 costs. BDS C doesn't provide as many formats, either.

Software Toolworks doesn't charge a license fee for the COM files created by C/80. This means that you're free to market whatever you write and compile as you see fit. Unlike some of the so-called "big boys" of the software industry, Bilofsky encourages programmers to market software using his tools.

A word about documentation: If you're a first-time C user, the manual that accompanies C/80 won't be too helpful. There are some examples in the documentation that you can play with, and several source code files on the distribution disks, but that's it for introductory material. Several good books on the market can help you get a better grasp of the language. The documentation lists eight of the most common of these, and new books appear almost monthly.

The C/80 manual is 50 pages long with a three-page index. It has 16 chapters, with most of the attention paid to a language summary, a description of the I/O and runtime libraries, and a discussion on running the compiler. The manual includes a brief discussion of the internal features of the compiler and useful tricks to bend the compiler to your will. There is also a detailed list of compiler error messages, what they mean, and what to do about them.

The C/80 Mathpack requires C/80 version 3.0 or greater. If you've been working with an earlier verson of C/80, the Mathpack is a good reason to take advantage of the Software Toolworks' update policy. Mathpack comes on one disk with a short manual. The Mathpack adds true 32-bit LONG and FLOAT data types to the C/80 compiler and runtime library. It includes a program to patch the C/80 compiler to recognize LONG and FLOAT, a runtime library to perform 32-bit arithmetic, routines to convert between ASCII and floating-point representation, an augmented printf, and a transcendental library written in C.

In practical terms, this gives the C/80 a 32-bit signed number capability in the range –2,147,483,648 to 2,147, 483, 647. Floating-point numbers have 24-bit precision, which equates to about seven decimal digits, and an exponent range of 10 to the plus or minus 38th power.

Installing the Mathpack requires only that you run the modifying program CCONFIGF.COM. A menu will appear with a range of options. Choose option N and press RETURN. This changes FLOAT and LONG from not available to available. Now type a Y in response to the question about making changes permanent, and you're done. Everything else about C/80 remains as before, except that the size of the compiled and assembled code increases a bit.

The Mathpack manual is only 10 pages long with six chapters; it spends most of its time on installation and additions to the function library. You'll have to know how to make use of FLOAT and LONG before the Mathpack will do you much good.

In summary, if you want to learn C or if you already know a bit about the language and want an inexpensive but powerful implementation for your CP/M computer, my first choice would be the Software Toolworks C/80 and an introductory book. If you want an enhanced 32-bit capable C, add the Mathpack. For under $80.00 you can't miss.

Disk Maker I

Company: New Generation Systems, 1800 Michael Faraday Drive, Suite 206, Reston, VA 22090
(703) 471-5598,
(800) 368-3359
Operating System: CP/M 2.2
Price: $1695.00
Circle Reader Service No. 103
Reviewed by Jim Kronman

Disk Maker I from New Generation Systems offers a practical solution to the virtual Tower of Babel that exists in the world of CP/M and MSDOS disk formats. Acting as a peripheral device in an S-100 system, Disk Maker I can format, read, write, and duplicate over 170 disk formats (at the time of this review—the total grows every month or two), allowing you to copy virtually any soft-sectored format and transfer disk files between various CP/M and MSDOS disk formats. An optional file transfer utility program transfers files from DecMate II and Wang OIS word processing disk formats.

The standard Disk Maker I package consists of an S-100 disk controller board capable of supporting up to four disk drives, a 48 track-per-inch (tpi), double-sided 5¼-inch disk drive with power supply in a fan-cooled cabinet, and the Disk Maker software. Other disk drives for Disk Maker are a 96 tpi 5¼-inch drive, a

135 tpi 3½-inch drive, or a 96 tpi 1.2 Mb IBM PC/AT drive (as used in the IBM PC/AT) installed in the cabinet with the 48 tpi drive. A double-sided double-density (DSDD) 8-inch drive with its power supply in a separate cabinet is also available. Software options are the word processing transfer utilities and disk drive test software.

If you don't have an S-100 system, do not stop reading: New Generation Systems also offers a stand-alone Disk Maker II system. It is a 6 MHz, Z80, 64K CP/M 2.2 system that includes one DSDD 8-inch drive and one 48 tpi, DSDD 5¼-inch drive plus the same software provided with Disk Maker I. You can add the same options described for Disk Maker I, up to a total of four drives; a 10 Mb hard disk option is also available. Having reviewed the manual for the Disk Maker II (but not having used the equipment), I am reasonably confident that my remarks about Disk Maker I will be applicable to the operation of Disk Maker II as well.

The Disk Maker I controller uses the Western Digital 1795 controller chip, allowing the software to read or write disks in all 170-plus formats shown in the Table on page 101 (if you have the appropriate drives, of course). Disk Maker I cannot accommodate some disk formats such as Apple, NorthStar, Micropolis, Commodore, and Victor; these formats either are hard-sectored or depend on special tricks in the disk controller hardware.

When you have the optional 96 tpi 5¼-inch drive installed, the software allows you to set the 96 tpi drive as a read-only drive for a 48 tpi format; this allows you to transfer files from one 5¼-inch disk to another without having to do an intermediate copy. With the two 5¼-inch drives set to the same format, rapid disk duplication is possible using the DUPE utility program. New formats are added regularly, and New Generation Systems offers each upgrade to current users for only $50.

Programs

The Disk Maker I software includes these programs:

DMINSTAL.COM—sets user-determined and system-dependent options
DMSET.COM—loads Disk Maker BIOS into the system
DMFORM.COM—formats disks
WHATDISK.COM—determines exact format of IBM, Z-100, or Morrow disk
MC.COM—Mass Copy, a file copy program
DUPE.COM—rapid track-for-track copy program with optional verify
DMCOMP.COM—rapid track-for-track compare program
TOMS.COM—CP/M < = > PCDOS file exchange utility

Various public domain utilities such as SWEEP and D normally are provided on the distribution disk as space permits. Two or more versions of DMSET and DMFORM are provided to accommodate various hardware peculiarities.

The optional software is:

DDD.COM—disk alignment program used with special disks
DEC2CPM—utility to transfer files from DecMate II word processor to CP/M
WANG2CPM—utility to transfer files from Wang OIS word processor to CP/M

Requirements

Disk Maker I is designed to operate under standard CP/M 2.2 with an 8080, 8085, or Z80 CPU. New Generation Systems recommends using it with a standard CP/M system only, but I run it with CP/M 2.2 and ZCPR, version 1, without any problems. The controller board and Disk Maker software normally use ports 98 to 9F (hex), but New Generation Systems provides software and hardware modification instructions to relocate the ports to other addresses if you have an unresolvable conflict in your system.

Although a 64K system is recommended for Disk Maker I, it will run in as little as 48K. The more disks you attach to the Disk Maker, the more memory the software requires. When you use the TOMS utility, a smaller TPA means that you may not be able to handle a full directory on an MSDOS disk; my 57K system will handle the maximum. Double-density 8-inch formats and the IBM PCAT drive require at least a 4 MHz system clock; otherwise, a 2 MHz clock should be sufficient.

Installation of the Disk Maker I hardware is as easy as plugging the disk controller board into an empty slot in your S-100 bus motherboard and connecting the disk cable between the disk cabinet and the controller board. A menu-driven installation program allows you to customize some features of the system to your hardware and personal taste; the installation process takes only a few minutes.

How It Works

Disk Maker I is simple to use. To format a disk, type DMFORM to invoke the format program, then choose a format by number from the on-screen menu. You can also invoke DMFORM by making a format choice on the command line; the program proceeds without showing you the menu. You can return to the menu if you want to change your selection. DMFORM will verify a disk after formatting if you set a permanent option with DMINSTALL or specify verification on the command line when you invoke DMFORM.

The DMSET program gives you access to the Disk Maker disk drives. You use DMSET the same way you use DMFORM: you can choose from a menu or go directly to work by specifying a format on the command line. You can set each Disk Maker drive to a different format. Every time you warm boot, the Disk Maker software displays the disk formats you have selected.

DMSET installs a special BIOS under your normal CP/M system; you can run any CP/M program that will fit in the somewhat reduced memory left for applications. Even though there is less memory, I have found it adequate for running any of my utilities, dBASE II or my text editor (which does bidirectional scrolling through a disk file). You probably will notice the memory reduction the most when you run a spreadsheet program or any other application that uses all available memory but does not scroll data to and from disk,

Sanyo 1200/50		96(DSDD) 620K	Sanyo 2000		96(SSDD) 302K
Sanyo 4050		96(DSDD) 620K	TeleVideo 1603		96(DSDD) 706K
Vector 4-S		96(DSDD) 712K	Vector VSX		96(DSDD) 710K
ALTOS		8"(SSDD) 446K	CCS	(256b)	8"(SSDD) 482K
CCS	(1024b)	8"(SSDD) 596K	Colonial Data SB80		8"(SSDD) 596K
Colonial Data SB80		8"(DSDD)1208K	Columbia 1800		8"(SSDD) 592K
CompuPro	(1024b)	8"(SSDD) 596K	CompuPro	(1024b)	8"(DSDD)1192K
CP/M SSSD Standard		8"(SSSD) 241K	Delta Products 125		8"(DSDD)1212K
Harris 1685-50MFT		8"(SSDD) 596K	Harris 1685-50MFT		8"(DSDD) 968K
Harris 1685-50MFT		8"(DSDD)1196K	IBM 3740 TO Direct		8"(SSSD) 248K
IBM 5520 WP		8"(DSDD)1132K	IBM 128b no skew		8"(SSSD) 241K
IBM 256b no skew		8"(SSDD) 482K	Insight IQ-120		8"(SSDD) 484K
Megaflex Apple		8"(SSDD) 558K	Molecular		8"(SSDD) 496K
NEC APC CP/M-86		8"(DSDD) 980K	S. D. Sales	(128b)	8"(SSDD) 464K
S. D. Sales	(256b)	8"(SSDD) 476K	Tarbell	(128b)	8"(SSDD) 472K
TRS-80 II,12/6 P&T		8"(SSDD) 596K	TRS-80 II,12/6 P&T		8"(DSDD)1210K
TRS-80 Lifeboat		8"(SSDD) 482K	TRS-80 Lifeboat		8"(SSDD) 596K
Vector 2800		8"(DSDD) 984K	Zenith Z-100		8"(SSDD) 482K
Zenith Z-100		8"(DSDD) 980K	ACT Apricot	MSDOS	3"(SSDD) 314K
HP-150	MSDOS	3"(SSDD) 258K	Intertec H-S	MSDOS	3"(SSDD) 395K
IBM PC	PCDOS1.1	(SSDD) 156K	IBM PC	PCDOS1.1	(DSDD) 315K
IBM PC	PCDOS2.0	(SSDD) 175K	IBM PC	PCDOS2.0	(DSDD) 354K
IBM PC-AT	PCDOS3.0	(DSDD)1186K	Kaypro 10	MSDOS	(DSDD) 354K
Wang	MSDOS	(DSDD) 354K	Zenith	ZDOS 1.1	(DSDD) 315K
BurroughsB25	MSDOS	96(DSDD) 610K	DEC Rainbow	MSDOS	96(SSDD) 384K
Eagle 1600	MSDOS	96(DSDD) 785K	Monroe	MSDOS	96(DSDD) 712K
NCR D-M "M"	MSDOS	96(DSDD) 790K	Otrona	MSDOS	96(DSDD) 712K
Sanyo MBC	MSDOS	96(DSDD) 792K	Tandy 2000	MSDOS	96(DSDD) 714K
Lomas Data	MSDOS	8"(SSSD) 239K	NEC APC	MSDOS	8"(DSDD)1221K
Standard	MSDOS	8"(SSSD) 246K	Zenith V1.1	ZDOS	8"(SSSD) 245K

```
NOTE:  all formats 5 1/4" 48 tpi, CP/M-80/86 unless noted as:
          3"  -  3 1/2 inch format
          8"  -  8 inch format
          96  -  96 tpi 5 1/4 inch format
       SS = Single Sided; DS = Double Sided
       SD = Single Density; DD = Double Density
```

Table
Disk Maker I Formats

the way many text editors do.

Because of the radically different structure of MSDOS files, you need a special program called TOMS (TO MSDOS) to access individual disk files on a MSDOS disk. You cannot run another program while using TOMS; if you want a CP/M application to access a MSDOS data file, you first must *transfer* the file to a CP/M format. TOMS is like a small part of the MSDOS operating system, providing the commands DIR, COPY, TYPE, and ERASE. TOMS cannot deal with subdirectories, so all files must be in the root directory on a MSDOS disk coming into the system; all files written by TOMS are in the root directory.

Performance

I have been using Disk Maker I for well over a year. The capabilities of the system are impressive, but even more impressive is the support New Generation Systems provides. I was an early user and, as you might expect, everything did not start out working perfectly. One annoying problem was caused by the routing of the data cable in the disk drive cabinet. The cable was picking up noise from the power supply, which gave me errors when reading the inner tracks of the 48 tpi drive. I solved the problem by changing the routing of the cable at the suggestion of New Generation Systems, but in the meantime, I received a replacement 48 tpi drive that had been specially aligned and certified. I have always received courteous and competent assistance anytime I have had a reason to call New Generation Systems with a problem or a question. You can expect the same treatment. (I was dealing with New Generation Systems as just another user, before I had thought of doing this review.)

The Disk Maker I hardware consists of off-the-shelf components integrated by New Generation Systems—nothing "razzle-dazzle" but obviously chosen carefully for quality and value. What makes the Disk Maker I package outstanding is the

Apricot CP/M-86	3"	(SSDD) 306K	Intertec HeadStart	3"	(SSDD) 372K
Access		(SSDD) 169K	Actrix Matrix		(DSDD) 348K
AMPRO		(SSDD) 188K	AMPRO		(DSDD) 386K
Balcones BRAVO		(DSDD) 346K	Bondwell Model 14/16		(DSDD) 346K
BYAD		(SSDD) 154K	Casio FP 1000/1100		(DSDD) 300K
Compuview CP/M-86		(SSDD) 193K	Cromemco		(SSSD) 81K
Cromemco		(SSDD) 188K	Cromemco C-10		(DSDD) 386K
Davidge 1024		(DSDD) 368K	DEC VT180		(SSDD) 169K
Digilog		(DSDD) 332K	Eagle I		(SSSD) 70K
Epson QX-10 CP/M		(DSDD) 378K	Epson QX-10 MF		(DSDD) 278K
Fujitsu Micro 16s		(DSDD) 314K	Heath H-37		(SSDD) 148K
Heath H-37		(DSDD) 304K	Heath w/CDR Systems		(DSDD) 380K
Heath w/Magnolia		(SSDD) 162K	HP 86/87/125		(DSDD) 248K
IBM PC (CP/M-86)		(SSDD) 154K	IBM PC (CP/M-86)		(DSDD) 314K
ICL PC1 10,30,31,32		(DSDD) 254K	Insight Dev. IQ-120		(SSDD) 146K
KayPro II		(SSDD) 191K	KayPro 4/10		(DSDD) 394K
Lobo Max-80		(SSDD) 164K	Lobo Max-80 35tk		(SSDD) 142K
Lobo Max-80 35tk		(DSDD) 296K	Microbee		(DSSD) 171K
Microbee		(DSDD) 386K	Molecular 9X		(DSDD) 338K
Morrow Micro Dec I/II		(SSDD) 186K	Morrow Micro Dec III		(DSDD) 384K
Morrow Micro Dec 11		(DSDD) 384K	MultiTech MIC-501		(SSDD) 168K
NCR Decision Mate V		(DSDD) 304K	NEC PC-8000 Series		(SSDD) 169K
NEC PC-8000 Series		(DSDD) 300K	Osborne		(SSSD) 90K
Osborne DD/Executive		(SSDD) 183K	Otrona Attache		(DSDD) 360K
PMC 101 (CP/M 3.0)		(DSDD) 386K	Sanyo 1000/1100/1150		(DSDD) 310K
SD Sales		(SSSD) 71K	SD Sales		(SSDD) 112K
Sharp 3540		(DSDD) 292K	SuperBrain		(SSDD) 162K
SuperBrain		(DSDD) 340K	Systel III		(DSDD) 346K
TeleVideo 802/3/6		(DSDD) 340K	TeleVid. 806 TurboDos		(DSDD) 396K
TI Professional		(SSDD) 154K	Toshiba T100		(DSDD) 254K
TRS-80 Mod I Omikron		(SSSD) 70K	TRS-80 Mod I Omikron		(SSDD) 131K
TRS-80 Mod III MM		(SSDD) 186K	TRS-80 Mod IV		(SSDD) 154K
TRS-80 IV Montezuma		(SSDD) 166K	Versa Floppy II		(DSDD) 278K
Xerox 820		(SSSD) 82K	Xerox 820		(SSDD) 155K
Zenith Z-100		(SSDD) 148K	Zenith Z-100		(DSDD) 304K
Zenith Z-100 (orig)		(DSDD) 304K	Zorba		(DSDD) 388K
ALTOS 586	96	(DSDD) 700K	AMPRO	96	(SSDD) 386K
AMPRO	96	(DSDD) 782K	Analyzer	96	(DSDD) 620K
Archives I	96	(SSDD) 386K	Archives II	96	(DSDD) 786K
Associate +	96	(DSDD) 786K	CompuPro 10	96	(DSDD) 772K
CompuStar 40	96	(DSDD) 786K	DEC Rainbow	96	(SSDD) 386K
Digilog DBS 16	96	(DSDD) 780K	Direct OA1000	96	(DSDD) 622K
Discovery	96	(DSDD) 624K	Eagle II/IV	96	(SSDD) 386K
Eagle III	96	(DSDD) 762K	Epic	96	(DSDD) 782K
Facit DTC	96	(SSDD) 308K	Heath H-37	96	(SSDD) 308K
Heath H-37	96	(DSDD) 624K	Honeywell	96	(DSDD) 632K
ICL Model 25	96	(DSDD) 628K	ICL Model 35	96	(DSDD) 778K
IMS (TurboDos)	96	(DSDD) 772K	Intel iPDS	96	(DSDD) 608K
Ithaca 525	96	(SSDD) 346K	Ithaca 525	96	(DSDD) 628K
Lobo Max-80	96	(DSDD) 698K	MACSYM 150/350	96	(SSDD) 308K
Monroe OC8820	96	(SSDD) 306K	Monroe 2000	96	(DSDD) 628K
MultiTech MIC-504	96	(DSDD) 698K	NCR Work Saver	96	(DSDD) 544K
Olympia People	96	(DSDD) 618K	OSM Zeus 4	96	(DSDD) 620K
Otrona	96	(DSDD) 770K	Philips P2000C	96	(DSDD) 628K
Philips 3000	96	(SSDD) 294K	Pied Piper	96	(DSDD) 776K

Table
Disk Maker I Formats

(Continued on next page)

software and support. The programs perform well and are easy to use. The no-nonsense operator interface was designed for users who need to get things done quickly.

Although the software is not friendly in the current sense of having elaborate menus and on-line help facilities, each user prompt and program response is clear in its meaning. With one exception, which I will discuss later, all errors are trapped cleanly and appropriate error messages appear on the screen. The user interface is appropriate for the intended users of this package: software professionals, experienced computer users, and serious hobbyists with a need to transfer files between diverse disk formats. Operation is simple enough for a novice to use the system, too.

The Disk Maker I (and Disk Maker II) documentation is well organized and appropriate for the intended users. The *User's Manual* for Disk Maker I contains over 50 pages with a good table of contents in a three-ring binder. The table of contents is adequate for using the manual: no index is provided and none is needed. The manual, prepared with a word processor, is much cleaner in appearance than most manuals prepared this way. It is relatively easy to flip through the pages and spot whatever you are looking for.

New Generation Systems has assumed that you already know how to use CP/M and your computer. The manual will say "copy this file to another disk" rather than give a blow-by-blow description of how to use the file copy utility. There is a section titled "How To Run Disk Maker Without Reading The Manual" for the impatient and for the technical-minded a section on each program called "How xxxx Works" that explains the inner workings of the program.

I have validated Disk Maker's ability to format, read, or write disks in about 20 of the 170 available formats. I have used Disk Maker primarily to move files between an S-100 system and an Osborne I (double density) or between the S-100 system and a PCDOS system. Other than the read errors I encountered (now fixed), the Disk Maker hardware and software have performed flawlessly for me.

I have encountered a few instances where it has been necessary to bulk erase a disk and format it with Disk Maker I because, if I wrote on a disk already formatted by the receiving system, that system could not read the disk.

Although Disk Maker I is easy to run, one enhancement would be welcome. The Disk Maker I software assumes you know what disk format you are dealing with. If you tell it you are feeding it a disk of format X and then give it a disk of format Y, you may or may not get an error message; the program you run could go merrily along without complaint, producing either garbage files or a disk that the target system cannot read. The Disk Maker software cannot identify positively that you have inserted the correct disk format in the drive being written to because no standard means of identification exists for a blank formatted disk; even if you determine the sector size, sectors per track, and so forth, you still cannot deduce the directory size and other characteristics needed for positive identification.

WHATDISK enables you to determine which of several similar formats a disk might be, if you already know that the disk is for IBM PCDOS, Heath/Zenith Z-100, or Morrow Micro Decision. A generalized WHATDISK program to identify any disk format would be very useful. Such a program is impossible to write, however, for the above reasons.

What You Pay

The basic Disk Maker I, which includes the controller board, one 48 tpi, DSDD 5¼-inch drive, and the Disk Maker software is $1695. The 96 tpi, DSDD drive is an additional $395. The 8-inch drive and cabinet for Disk Maker I is $849. The 3½-inch drive costs $295, and the PCAT drive is $495. The word processing transfer utilities package is $295, and the disk testing software is $150 plus $40 each for the 48 and 96 tpi Dysan test disks. You also can purchase the Disk Maker I controller and software with no drives.

The basic Disk Maker II system is $3395 for the stand-alone 6 MHz Z80B system with one DSDD 8-inch drive, one 48 tpi, DSDD 5¼-inch drive, and CP/M 2.2; you only need to add a terminal. The option prices are the same as for Disk Maker I, and a second DSDD 8-inch drive costs $600.

Are these prices reasonable? It depends on what you need. Programs for the Kaypro, IBM PC, Morrow, and other computers that allow you to read, write, and format a number of 5¼-inch formats cost less than $100, but don't ask them to recognize an 8-inch disk drive, a PCAT drive, or a 3½-inch drive. At the other end of the spectrum are systems that claim less capability than Disk Maker II and cost over $5000. Disk Maker I and Disk Maker II offer outstanding versatility and performance for the money.

Conclusion

Disk Maker I is a solid product backed by a competent and cooperative company that operates as if it wants to remain in business for a long time. If you are a software developer, software distributor, user group, large office, or serious hobbyist with the need to exchange files between computers with various disk formats, you should seriously consider Disk Maker I or Disk Maker II as an answer to your needs.

AMPRO Little Board and Bookshelf Computers

Company: AMPRO Computers, Inc., 67 East Evelyn Ave., Mountain View, CA 94041 (415) 962-0230.
Price: $349
Circle Reader Service No. 105
Reviewed by Richard Conn

The heart of the AMPRO Bookshelf Computer® is the AMPRO Little Board®, a full-featured single-board computer that measures only 5¾-inch by 7¾-inch and is designed for mounting on the back of a standard 5¼-inch minifloppy disk drive. Although it is small in size, the AMPRO Little Board is big in features: the Little Board contains all of the hardware necessary to support a conventional CP/M environment.

The Little Board is based around a 4 MHz Z80A microprocessor and the associated Zilog Z80 family of support chips. On board this small computer is:

- A 4 MHz Z80A microprocessor
- 64K of dynamic RAM
- One 4K 2732-type EPROM
- A Z80A CTC (Counter/Timer Circuit)
- A Z80 DART (Dual Asynchronous Receiver/Transmitter)
- A parallel output port (based on D-type latches)
- A Western Digital 1770 Floppy Disk Controller Chip

The AMPRO Bookshelf Computer is an AMPRO Little Board computer housed in an attractive case with one or two 5¼-inch floppies, built-in power supply, two 25-pin EIA RS-232C connectors, one Centronics connector, and a connector for tagging on up to two more 5¼-inch floppies. An ON/OFF switch and a RESET switch are available on the front panel.

The AMPRO Little Board Computer

Hardware

The Little Board has two possible memory configurations, which depend on the setting of an EPROM-enable bit in the Board Control Register. Enabling the EPROM establishes the following memory configuration:

Address (Hex)	Memory Element
0000 - 0FFF	2732 EPROM
1000 - 7FFF	2732 EPROM duplicated on each 4K group
8000 - FFFF	RAM

If the EPROM is disabled, all memory from 0 to 0FFFFH is enabled as RAM.

A Z80 DART and RS-232C line drivers provide the serial input/output ports. The DART is an asynchronous device, providing for baud rates up to 38,400 on its A channel and 9600 on its B channel in the Little Board configuration. The Little Board does not implement all of the RS-232C signals on its two serial channels. Only the following signals are provided:

- Serial Data Output (RS-232C pin 3)
- Serial Data Input (RS-232C pin 2)
- Request to Send Handshaking Output (RS-232C pin 5)
- Clear to Send Handshaking Input (RS-232C pin 20)

Both channels of the DART are wired as DCE (Data Communications Equipment), which is the complement of DTE (Data Terminal Equipment)—the wiring of the RS-232C connector on computer terminals. Notably missing from this list is the Data Carrier Detect signal. You may use the Clear to Send signal in place of Data Carrier Detect.

The parallel port is an output-only port, wired in a Centronics-standard configuration. The Z80 CTC on the Little Board provides four counter/timers for use in conjunction with the hardware and software of the computer.

The Western Digital 1770 Floppy Disk Controller chip used in the Little Board design can work with just under 400K per disk on double-sided, 48 tracks-per-inch (TPI) drives and just under 800K per disk on double-sided, 96 TPI drives.

Software

The AMPRO Little Board is supplied with one copy of CP/M 2.2, which uses the ZCPR3 Command Processor in place of the CP/M CCP. Although only the ZCPR3 Command Processor, in a less than maximum configuration, is provided with the Little Board, you can obtain a more complete ZCPR3 implementation from Echelon (see below).

The ZCPR3 configuration distributed with the Little Board provides CP/M 2.2 compatibility while simultaneously presenting the following additional features:

- Multiple commands per line, separated by semicolons, enabling command lines like DIR;ERA *.BAK;DIR
- Automatic command search path that searches through directories in the following sequence when looking for COM files: (1) current disk, current user; (2) current disk, user 0; (3) disk A, current user; (4) disk A, user 0; (5) disk A, user 15; and (6) current disk, user 15
- Four-element shell stack, so shells other than the ZCPR3 Command Processor can act as the user interface to the system
- Several built-in commands, including: GET—load file anywhere in memory; JUMP—call subroutine anywhere in memory; and LIST—print file on printer
- Extended directory references over normal CP/M: D:—reference disk (e.g., TYPE B:MYFILE.TXT); U:—reference user area (e.g., DIR 5:); and DU:—reference disk and user area (e.g.,

ERA B7:*.*)

See articles in *Dr. Dobb's, Computer Language, Byte*, and other such magazines for more information on ZCPR3. The authorized agent for ZCPR3 is Echelon, Inc., 101 First St., Los Altos, CA 94022 (415) 948-3820. The ZCPR3 RCP/M and BBS are also available to you at (415) 489-9005.

The nine standard CP/M utility programs (ED, DDT, PIP, etc.) and thirteen additional utility programs are supplied with the Little Board. Included in this list is MULTIDSK, which allows the user to dynamically select one drive to support any one of over 40 different 5¼-inch floppy disk formats, and 48TPI, which allows the user to read 48 TPI disks on a 96 TPI drive.

One nice option that AMPRO gives the customer is that of purchasing the source code to several key programs at a reasonable price. The customer may purchase the Technical Support Software package, which includes the source files for MULTIDSK, the BIOS, and the BOOT.

Pricing

The AMPRO Little Board computer costs $349. This price includes the single-board computer and all of the software listed above. An extra $50 buys the Technical Support Software package.

The AMPRO Bookshelf Computer

Hardware

As mentioned above, the AMPRO Bookshelf Computer consists of an AMPRO Little Board, one or two 5¼-inch minifloppy disk drives, a built-in power supply, two 25-pin EIA RS-232C connectors, one Centronics connector, one connector for attaching up to two external minifloppy disk drives, and ON/OFF and RESET switches on the front panel. The Bookshelf is available in the following models:

Model	Configuration
121	One double-sided, 48 TPI drive (400K)
122	Two double-sided, 48 TPI drives (800K)
141	One double-sided, 96 TPI drive (800K)
142	Two double-sided, 96 TPI drives (1600K)

Software

All of the software provided with the AMPRO Little Board is also provided with the AMPRO Bookshelf Computer. In addition, a ZCPR3 shell designed specifically for use on the AMPRO computer is provided (in object code only). FRIENDLY is a screen-oriented tool that provides a directory of files to the user of the current disk and allows him or her to perform a variety of operations on the files displayed. By simply moving a pointer to a desired file and pressing a few keys, a FRIENDLY user can copy a file, delete a file, view a file on the console, print a file on the printer, run a program, mark a file for later delete or copy with a group, perform a sequence of commands on the file based upon a preprogrammed menu, and perform many more operations.

FRIENDLY is easy to use, usually requiring only a few minutes to learn, and it removes casual computer users from the CP/M command environment, relieving them of the necessity of learning how to use CP/M in order to use the computer.

A second software package provided with the AMPRO Bookshelf is T/MAKER, a product of T/MAKER Company. T/MAKER provides a screen-oriented, memory-based editor, text formatter, electronic spreadsheet, bar graph generator, and file management system in one integrated package. T/MAKER will run under conventional CP/M, CP/M enhanced by the ZCPR3 Command Processor, or FRIENDLY. Several articles have already been published on T/MAKER, and you can obtain further information by writing to T/MAKER Company, P.O. Box 6430, Falls Church, VA 22046.

Documentation

The documentation provided with the AMPRO Little Board is the *Little Board User's Manual*, a mainly hardware manual that describes the Little Board in great detail. It is well written and clear if you have a background in digital electronics.

No documentation on CP/M or the standard CP/M programs is provided. The first pages of the manual, however, give pointers to where to obtain this documentation. Also, no documentation other than a brief overview of ZCPR3 is provided. Again, the manual gives pointers to Echelon.

If you buy the AMPRO Bookshelf Computer, you get the *Little Board User's Manual* as well as documentation on FRIENDLY and T/MAKER. The documentation on FRIENDLY and T/MAKER is oriented to computer users having some limited experience with CP/M.

General Impressions and Comments

I have used the AMPRO Bookshelf for several months now. I started with the Model 122, which had two 400K, 48 TPI drives, and am now using the Model 142, which has two 800K, 96 TPI drives. I find its clean design, speed, and reliability to be excellent. Given my background in CP/M and digital electronics, I found all of the documentation on the Little Board/Bookshelf to be readable and understandable.

The Bookshelf is a nice computer for any user, especially with the FRIENDLY shell as a front end. Although it is a good choice for the first-time computer user, as with any computer, you must overcome a learning curve before you can expect effective use of the computer.

The AMPRO BIOS performs well. The speed of the disks with this BIOS is nice: the disks are faster than most 5¼-inch disk systems I have observed and many 8-inch disk systems I have used. At 800K per disk with the Model 142 (96 TPI drives), I find the disk capacity also to be quite reasonable.

Overall, the AMPRO Bookshelf and AMPRO Little Board are good computers, but there are a few drawbacks for some applications:

(1) Lack of documentation on CP/M and ZCPR3. However, AMPRO supplies pointers to where you can obtain such documentation, at additional cost.

(2) Limitation of serial I/O ports to only two serial RS-232C ports and one parallel port. Nevertheless, if

your needs call for no more than a terminal, modem, and printer (as many popular computers are configured today), the AMPRO computers are adequate.

(3) Lack of hard disk support and expandability. The Little Board does not support hard disks or other such devices in its current form. You may buy, however, the Little Board SCSI/PLUS Adapter for $99. This adapter "piggy-backs" on the Little Board and allows you to attach a Winchester disk drive, slave processor and device boards (perhaps for I/O), and other facilities. This board, with some effort, may eliminate drawbacks 2 and 3.

I am excited about the Little Board and the Bookshelf for their applications in several environments. As a stand-alone personal computer with limited I/O resources and disk space, these computers are good choices. The SCSI/PLUS Adapter promises to expand their capabilities.

For embedded computer applications in which 4K of code is enough to run the application program, the Little Board provides a nice, inexpensive self-contained computer. The 4K 2732 EPROM can be programmed for the application, and 32K of RAM is available for data. The two RS-232C ports are there, as well as the parallel output-only port. No disks would be needed for this type of application. For embedded computer applications with disks, the AMPRO Little Board would "piggy-back" on a 5¼-inch 96 TPI minifloppy and feed off of the floppy's power supply. This would provide a very small embedded computer with 800K of disk and 64K of RAM.

As a slave computer to another computer, the AMPRO Bookshelf with its two disks could act as a background batch processor, communications controller (for a BBS), printer spooler, or other such intelligent and flexible slave device.

Overall, I like the AMPRO Little Board and AMPRO Bookshelf and intend to continue using my Bookshelf for a variety of slave computer and stand-alone applications (such as computer assistance for talks and presentations). I recommend it with some reservation to first-time computer users and with no reservation to embedded applications designers and more experienced computer users.

BetterBASIC, Version 1.1

Company: Summit Software, P.O. Box 99, Babson Park, Wellesley, MA 02157 (617) 235-0729

Computer: IBM PC or close compatible with minimum 192K and one disk drive

Price: BetterBASIC $199; 8087 Math Module $99; Runtime System $250

Circle Reader Service No. 107

Reviewed by Matthew Trask

Among the reasons that Summit Software gives for purchasing BetterBASIC are full 640K memory support, separately compiled program modules, language extensibility, window support, 8087 support, and incremental compilation. Although these reasons would be persuasive in the case of another programming language, I question whether they will induce many programmers to shell out $199, the cost of BetterBASIC, when some form of GW BASIC (aka BASICA) is provided at no charge with most IBM type computers these days and an excellent Pascal compiler can be purchased for less than fifty dollars. In order to help programmers make a decision on this purchase, I will discuss the unique features of this programming system, benchmark its performance, and examine the difficulties of converting programs from GW BASIC to BetterBASIC.

Installation

Although the package can be run right out of the box by using its default configuration, you will probably want to configure it to suit your hardware environment. B.COM is the executable portion of BetterBASIC and it uses the file B.CNF to determine which of the other modules to load for operation. Table 1 (page 110) is a listing of all the modules that are provided on the BetterBASIC disk and Table 2 (page 110) shows the default configuration. An ASCII editor can be used to add or delete modules and create new configurations. For example, users that don't have a graphics card may wish to delete GRAPHICS.IBM in order to free up memory for program use. I replaced FILE.DOS with FILE2.DOS in order to take advantage of DOS 2.x subdirectories and added SYSCALL.IBM in order to use the SHELL command and BIOS/DOS calls. You may specify the use of alternative configurations when starting the program by typing B-MYCONFIG/c where MYCONFIG is the name of the optional configuration file.

A Tandy 2000 version of the program is also available. The modules with the extension .IBM are all replaced with equivalent modules that end with .TDY.

Startup and Operation

After a sign-on message and copyright notice you will see the first major difference between BetterBASIC and GW BASIC: instead of the usual "60891 Bytes free" message, a 512K machine will display "Left:260304 bytes." All available system memory is used as BetterBASIC's workspace. This allows much larger programs to be written. BetterBASIC supports most of GW BASIC's commands, including the on-screen editing keys that are used on the IBM PC. Table 3 (page 111) contains a complete listing of GW BASIC commands that are different or not supported and a list of new commands that are found in BetterBASIC.

Every statement that you enter is checked for syntax and then compiled. If you make a mistake when entering a program line, an error message will be given immediately. For example:

```
40 PRINT (1 + 2 * 3
- - - - - - - - - - - - - - ^
" ) " Expected
```

If the statement is entered correctly, it is compiled to virtual-machine code rather than being interpreted at runtime, thus giving a large speed improvement over the GW BASIC interpreter.

As in Pascal and C, all variables are declared at the beginning of the

program. The exception to this is the AUTODEF command, which allows automatic declaration of a variable by the first assignment of a value. AUTODEF defaults to ON, but if you turn it off, the compiler will catch misspelled variable names. The data types that are supported in BetterBASIC are byte, integer, real, string (up to 32,767 characters), array, pointer, and structure. Structures are identical to Pascal's record data type and may contain any other data type. Arrays can contain byte, integer, real, and string data as well as structures and other arrays, thus allowing arrays of arrays, arrays of structures, and structures of arrays.

A BetterBASIC programming session superficially resembles one in GW BASIC in that lines are numbered and most of the command keywords are identical. When saving to disk, however, you must specify the file's extension because .BAS is not assumed by BetterBASIC. At the end of a session you can exit to DOS with the SYSTEM command or by typing BYE.

B	COM	35312	1-03-85	1:00a
B	DEF	27020	1-03-85	1:00a
B	CNF	142	1-03-85	1:00a
MATH	BCD	8112	1-03-85	1:00a
CONSOLE	IBM	21392	1-03-85	1:00a
MAIN		3840	1-03-85	1:00a
FILE	DOS	19056	1-03-85	1:00a
GRAPHICS	IBM	10512	1-03-85	1:00a
PLAY	IBM	3136	1-03-85	1:00a
EVENT	IBM	7744	1-03-85	1:00a
FILE2	DOS	21072	1-03-85	1:00a
CHAIN	MOD	2704	1-03-85	1:00a
SYSCALL	IBM	3520	1-03-85	1:00a
COMM	BAS	8484	1-03-85	1:00a
PLIST	BAS	2542	1-03-85	1:00a
SECTOR	BAS	6702	1-03-85	1:00a
STRUC	BAS	2980	1-03-85	1:00a
SHELL	CNF	175	1-03-85	1:00a
CHAIN	SIZ	11	1-03-85	1:00a
READ	ME	126734	1-03-85	1:00a
		20 File(s)	1024 bytes free	

Table 1
Directory listing of the BetterBASIC diskette showing the many modules that make up the Better BASIC system.

```
MODULES=MATH.BCD
MODULES=CONSOLE.IBM
MODULES=MAIN
MODULES=FILE.DOS
MODULES=GRAPHICS.IBM
MODULES=EVENT.IBM
MODULES=PLAY.IBM
STATUS=ON
```

Table 2
The default B.CNF configuration file.

Unique Features of BetterBASIC

The best feature of BetterBASIC is the use of true procedures with variables of local scope that exist only inside the procedure. Global variables can be used by declaring them as EXTERNAL in the procedure and providing the actual declaration at a level outside the procedure. A procedure can be declared as RECURSIVE if its name is declared as EXTERNAL inside the procedure. GOSUB and RETURN are both supported, but as you gain experience with BetterBASIC, they will probably fall by the wayside in favor of these Pascal-like, parameterized procedures. In addition to passing arguments by value or reference, you may specify optional and/or default arguments.

One particularly distinctive aspect of BetterBASIC is the ANY ARG declaration. This allows a procedure or function to determine what type of data was passed to it at runtime with the following functions:

TYPE( ) extracts the type of datum
DIM( ) extracts the dimensions of a datum
SIZE( ) extracts the size of a datum

For example, A=TYPE(FOO) will assign the following values to A:

0	if FOO is a byte
1	if FOO is an integer
2	if FOO is a real
3	if FOO is a string
4	if FOO is a structure
10 +N	if FOO is an array of above type N

If TYPE(A) returns 3 for string you can determine its allocated size (not length) with SIZE(A). If FOO is a structure, DIM(A) will return the number of fields and TYPE(FOO(N)) will return the type of field N. This flexibility of optional/ambiguous parameter passing can be a tremendous improvement over Pascal's rigid

requirements.

BetterBASIC has a concept called Procedure Families in which more than one procedure or function can have the same name with the only distinction being the parameter list. This allows BetterBASIC to select which one to use based on the type of argument that is used. As an example, the two procedures that follow can be used to give informative error messages on integer input, rather than GW BASIC's cryptic "Redo from start."

```
PROCEDURE:IntegerInput
INT ARG Value
  10 INPUT Value
  20 PRINT "Your integer is ",
     Value

PROCEDURE:IntegerInput.a
STRING ARG SomeString
  10 PRINT SomeString, "is not a
     valid integer, please try again"
```

In this example, IntegerInput.a will be used any time a valid integer is not entered when IntegerInput is called. This technique can also be used to assign additional functionality to BetterBASIC's reserved words.

Functions are similar to procedures in that they share all the same data types. The difference is that they are used in expressions to return a value that is used in the expression. The last statement of a function must be RESULT= and the value that is to be returned.

Both procedures and functions can use KEYWORD ARGuments—a concept that is similar to Pascal's user-enumerated data types. If an argument is declared as

```
KEYWORD ARG:Color
   \RED\GREEN\BLUE
```

then the procedure will accept either RED, GREEN, or BLUE as valid data

GW BASIC Statements Not Supported in BetterBASIC

BLOAD	BSAVE	CDBL	CSNG	CVD
CVI CVS	CONT	DEFDBL	DEFINT	
DEFSTR	DEF FN	DEF USR	EQV	ERASE
ERL FIELD	FRE	GET	IMP	
MERGE	MKD	MKI	MKS	MOTOR
ON PEN	ON STRIG	PEN	PMAP	PUT
RESUME	STICK	STRIG	TROFF	TRON
USR VARPTR	VIEW	WEND	WINDOW	

New BetterBASIC Statements

ANY ARG	ASH	AUTODEF	BYE
BYTE	BYTE ARG	BYTE ARRAY	BYTE ARRAY STRUC
BYTE PTR	CHECK	CODE	CODE?
COLOR BORDER	COMPRESS	CONSTANT	DEFINE WINDOW
DEL$	DO . . END DO	DO IF	DO UNTIL
DYNAMIC	EDIT proc	ERROR	EXIT
EXIT n LEVELS	EXTERNAL or EXT	FRAME WINDOW	HEADER
INPUT FROM	INS$	INTEGER or INT	INT ARG
INT ARRAY	INT ARRAY STRUC	INT FUNCTION	INT PTR
INTERRUPT	INTERRUPT PROC	KEYWORD ARG	LINE$
LIST ALL	LIST ARGS	LIST PROCS	LOWER$
MAIN	MAKE MODULE	MAKE PROGRAM	OFFSET
ON INTERRUPT	PRECISION	PROCEDURE or PROC	PUBLIC
READCHR	READCHR FROM	READLINE	READLINE FROM
REAL	REAL ARRAY	REAL ARRAY STRUC	REAL ARG
REAL FUNCTION	REAL PTR	REPEAT	REPEAT IF
RESTORE SCREEN	RESULT=	ROT	SAVE MODULE
SAVE SCREEN	SEG	SET	SH
SIZE	SPAN	STRING or STR	STR ARG
STR ARRAY	STR ARRAY STRUC	STR FUNCTION	STR PTR
STRUCTURE	UPPER$	WHILE . . DO	XREF

Table 3
A comparison of BetterBASIC and GW BASIC Keywords

for Color.

BetterBASIC has a full complement of flow-controlling block structures. In addition to the usual FOR-..NEXT, *IF..THEN*, and WHILE..REPEAT, BetterBASIC has DO UNTIL..REPEAT, DO..REPEAT, DO n TIMES..REPEAT, DO..END DO, DO..REPEAT, DO IF..REPEAT IF, and more. GOTOs are implemented in a way that could placate even Dijkstra—they cannot be used to enter or exit a control block. The only way out of a control structure is via the EXIT statement, and then only to the statement that immediately follows the block.

Variables can be declared as absolute by adding a segment and offset after the declaration. BYTE ARRAY-(4000):VideoBuffer [&hB000:0] is an absolute reference to the IBM's 2000 character monochrome video refresh buffer. This technique can be used to access the BIOS communication area in low memory for equipment determination.

The BetterBASIC language can be extended by adding your own procedures and functions to the language as keywords. The MAKE MODULE command will convert all procedures and functions currently in memory to a module that can be added to the B.CNF configuration file. Like C language function libraries, useful procedures can be distributed to other BetterBASIC users as modules with a good degree of source code protection because the source code cannot be derived from a module. Summit Software is currently selling an 8087 module and expects to offer an interface module for the SoftCraft Btrieve system sometime this spring.

Programs can be prepared for standalone execution with the MAKE PROGRAM command and the optional Runtime System. The Runtime System consists of RUN.COM, RUN.$$$, and MKEXE.COM. RUN-.COM will execute any program that has been prepared with MAKE PROGRAM and MKEXE.COM appends the RUN.$$$ runtime library to a program file for stand-alone execution.

Assembly language support is pretty straightforward with a very good example given in Appendix G of the manual. The keyword CODE is used instead of BLOAD in order to load an assembler module into memory at runtime. CODE? can be used to report the absolute load address if debugging is necessary. The assembly language procedure is then invoked by CALL procedurename(arglist) just as in GW BASIC. An assembly language module can contain one or more procedures with a Symbol Table at the beginning that defines the entry point for each procedure in the module. The excerpt from the BetterBASIC manual in the Listing (page 116) will give the flavor of assembly language interfacing.

Also worthy of note are the XREF command and the debug window. XREF generates a symbol cross-reference of all user-declared identifiers and the line numbers on which they appear. Advanced programmers may wish to use the DOS debug program to test programs that are developed in BetterBASIC. If they start with DEBUG B and then execute BetterBASIC inside DEBUG, the [Ctrl-PgUp] keys will exit to the debug prompt.

Current Version

The current version of BetterBASIC is 1.1. I received the upgrade from 1.0 to 1.1 during the review. New features supported include CHAINing and CALLing modules as overlays, communication between overlays via disk, SYStem calls for access to BIOS and DOS functions, and a SHELL command that can be used to execute programs (such as COMMAND-.COM) from within BetterBASIC. The new XMEM keyword allows data structures such as arrays up to the limits of system memory, not just 64K as in v1.0. Also, the release notes describe bug fixes that were applied to this new version.

Summit Software informs me that all registered users will be provided with free updates to v1.1, with a copy of the new documentation on the disk and an option to purchase printed documentation for $15. I also received test versions of the MATH1-.BNY and MATH2.BNY (single and double precision binary math) that are significantly faster than MATH.BCD, but less precise. By the time you read this, both modules should be included in the standard BetterBASIC system in addition to the BCD math module.

Benchmarks

I compared the performance of BetterBASIC and GW BASIC by running Rich Malloy's four BASIC benchmark programs that I downloaded from the *Byte* bulletin board. These tests separate I/O from computation to give fairly accurate representations of speed. With one exception (see Table 4, page 114), BetterBASIC was the better performer by far. The exception was in the floating point calculations, where the BCD math module was about 30% slower than GW BASIC. The new single precision binary math module ran about twice as fast.

Round-off error can be of concern in business and financial programming. Although MATH1.BNY showed no error in this benchmark (compared to GW BASIC's −1.788 E−07 error), it is probably well worth the slower speed of MATH.BCD to have guaranteed zero error in financial calculations.

The Runtime System, unlike the BASCOM compiler, will not show an improvement in execution speed. This is because BASCOM generates native 8086 code while MKEXE and RUN-COM only provide a stand-alone BetterBASIC environment to run the virtual-machine code that is produced by the MAKE PROGRAM command.

These benchmarks were run on an IBM PC with 512K of RAM, 10Mb hard disk, and a monochrome display. I have also successfully run BetterBASIC on a Leading Edge PC/XT, a PCjr, and a Hyperion PC with no difficulty.

GW BASIC Program Conversion

To put it bluntly, there is no easy conversion from GW BASIC to BetterBASIC. There is a simple procedure outlined at the end of the manual that involves editing an ASCII version of the original. The keywords SOURCE and ENDFILE are added at the start and end of the file for compatibility with BetterBASIC's LIST ALL ASCII storage format. In addition, variable

declarations must be made at the start of the file. If you don't declare the variables, they will be defined on use, but all integers will be defined as reals with the resultant loss of speed and precision that comes from the use of the real data type.

I was able to convert with no difficulty trivial programs such as the *Byte* benchmarks, and one fairly lengthy musical program came over without much modification. However, large programs such as PC-TALK or RBBS-PC would be easier to rewrite from scratch with BetterBASIC than they would be to convert. This is because of major philosophical differences in the use of procedures and the use of structures for file I/O rather than FIELDs in an I/O buffer. Conceptually, it would probably be easier to translate a Pascal or C program than an existing GW BASIC program.

Refer to Table 3 for a complete summary of the differences between the two BASICs.

The Documentation

If there are two words that describe the BetterBASIC documentation, they are massive and thorough. The manual is an IBM-style 5½ by 8-inch slip-cased binder with over 550 pages! It is broken up into seven sections: General Information, Introduction to BetterBASIC, Quick Reference, Syntax Visuals, Appendices, and two new additions for v1.1—Chain/Overlays and Syscalls.

The syntax visuals are comparable to the IBM BASIC manual with one or more pages in alphabetical order to describe each command and show examples of proper usage. The introductory chapter is very well done with an excellent tutorial. It is geared to beginners, but will provide a refresher for experienced BASIC programmers. It includes in the lesson on recursion a discussion of static versus dynamic storage of variables that will be useful to many microcomputer programmers. The presentation of variable scope and side effects in the procedure section will be helpful to those programmers whose only previous experience is with other BASICs. Appendices include an ASCII code table, keyboard scan codes, interrupts, character sets, module usage, module defined statements, use of Assembler, error codes, and GW BASIC conversion hints.

User Support

Because Summit Software is still a small company, any phone calls in search of support will be handled by one of the program's developers. This kind of direct access can be wonderful, especially when problems involve the more complex aspects of the system. There is no charge for phone support for registered users and Summit can be reached during business hours.

Complaints

As in all new products, there is still room for improvement. My main gripe has to do with errors in the manual. Until I received my v1.1 upgrade, I could find no reference to the PUBLIC keyword that is necessary for preparing to MAKE MODULEs. There also seems to be some confusion when conceptual words from other languages are used interchangeably with BetterBASIC keywords; structure and record are used in this fashion, although record is never defined to mean the same as structure.

Some of the error codes that the compiler can generate are similar to the warnings of other compilers. I'd like to see the compiler add some of the missing parentheses that it can detect, rather than just notifying the programmer. Also, you cannot specify a program on the command line to be executed by BetterBASIC. This precludes the use of B SETCLOCK to read your clock/calendar in an AUTOEXEC.BAT file.

There is no provision made for tracing program execution, as TRON/TROFF are not supported. I hope the people at Summit are listening and provide a trace that, unlike GW BASIC's primitive version, can also show variable contents, not just line numbers.

My final complaint has to do with price—there is a hidden cost that a potential developer must keep in mind. If you plan to market the software you develop with BetterBASIC, you will need to purchase the Runtime System for an additional $250 so your customers will be able to run your program without the necessity of purchasing their own copy of BetterBASIC.

Conclusions

So who will find this package useful?

	BetterBASIC	GW BASIC
BENCH1.BAS—write 64K bytes out to a file	14.6	43.3
BENCH2.BAS—read 64K bytes in from a file	10.4	29.1
BENCH3.BAS—5000 floating point operations	92.9(BCD) 37.2(BNY)	69.4
BENCH4.BAS—calculate all primes from 1 to 7000	35.2	208.1

Table 4
Performance Benchmarks (All times in seconds)

Probably not the C wizards and Forth gurus of the world. They will feel restrained by the line numbering and the lack of mysterious, syntactic constructions. Pascal programmers will feel right at home with procedures that have local variables and statements like READLINE. I think this package will find its place mostly among programmers who are currently developing vertical market (i.e., business) software in BASIC.

A large percentage of software that is created for vertical markets, such as insurance agencies, medical practices, video rental clubs, and accounting firms, is written in some form of the BASIC language. One reason that is often given for this disproportionate popularity is that BASIC is available in some form on most personal computers, thus providing a greater market when the software is ported to a new machine. However, because of the widespread availability of Pascal and C compilers for micros in recent years, I think this reason is less important than the fact that much vertical market software is not developed by trained professional programmers. Often an accountant/car salesman/real estate broker/farmer who has learned a little BASIC on a personal computer realizes a need for some industry-specific piece of software. This combination of industry expertise and programming experience can result in a very successful program.

In the words of Ivar Wold, Summit's president, these people are faced with the "Pascal Dilemma"—they intend to learn Pascal or C but haven't found the time for it yet. With this package, a BASIC programmer will be able to write better, more maintainable code without the learning curve that is associated with learning a new language from scratch. After using BetterBASIC, it will be easy to pick up Pascal with very little effort, as all of Pascal's concepts can be found in BetterBASIC. However, BetterBASIC may provide just the excuse never to have to learn a new language again.

DDJ

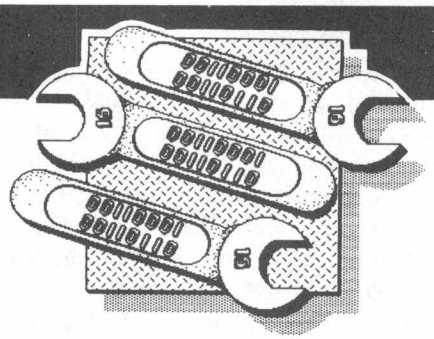
Better Basic Listing

(Text begins on page 108)

```
  Sample assembly language interface
  to a BetterBASIC program

        org     0

;--- define the Symbol Tabl

sym1    db      sym2-$          ;link to next symbol table entry
        db      'EXCHANGE'      ;name of the Procedure
        dw      0               ;fill word ( MUST be here !!)
        dw      header1         ;pointer to Procedure header

sym2    db      sym3-$          ;link to next symbol table entry
        db      'FILL'          ;name of the Procedure
        dw      0               ;fill word ( MUST be here !!)
        dw      header2         ;pointer to Procedure header

sym3    db      0               ;Symbol Table Terminator

;--------------------------

        even                    ;headers must be on
                                ; WORD boundary

header1 dw      154h            ;MUST be 154h
        dw      exchange        ;pointer to actual asm procedure
        dw      2               ;number of Arguments
        dw      1               ;type of argument 0 (Integer)
        dw      1               ;type of argument 1 (Integer)

header2 dw      154h            ;MUST be 154h
        dw      fill            ;pointer to actual asm procedure
        dw      2               ;number of Arguments
        dw      3               ;type of argument 0 (String)
        dw      1               ;type of argument 1 (Integer)

;--------------------------

; These are the procedures proper

exchange        proc fa         ;MUST be a far proc
        .
        .                       ; the EXCHANGE code
        .
exchange        endp

fill            proc f
        .
        .                       ; the FILL code
        .
fill            endp
```

End Listing

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

68000 Square Root Routine

Jim Cathey of Spokane, Washington, writes: "To help [eliminate] the dearth of 68000 programming goodies you were mentioning, here is an integer square root program. It takes only about 10–11 times as long to execute as a divide instruction. I found this algorithm a few years back in *EDN* .... I thought it was a nifty trick, and saved a clipping for when I needed it. Two muls instructions, one add.l, and this subroutine implement a true 16-bit vector length subroutine, in less than 2000 cycles (250 microsec. at 8 MHz)!

"The original program (and my first attempt) had a bug, in that the intermediate variables were half the size of the input data. This works until the last trip through the loop, where you may have overflow errors that could ruin the accuracy. For speed, the main loop could be unrolled using word intermediates for the first 15 passes, and long on the last loop-equivalent. This should save some time if it's a problem. Of course, if speed is really a problem, one of the approximations for vector length you presented a while back would be many times faster." See Listing One (page 122) for the source code for Jim's 68000 square root routine.

MacFeedback

Jim Howell of San Jose, California, a regular correspondent to *DDJ*, writes: "I am writing regarding your recent rather disparaging comments (*DDJ*, December 1984, #98) on the Macintosh. I believe it is much too early for you to be forecasting the demise of the Mac. So 'only' 200,000 Mac's were delivered in its first year. How many IBM PC's were delivered during its first year? And have you forgotten that even a year after the IBM PC came out, people were still complaining about the lack of software for it? As for comparing the Mac's 'lackluster sales' with the Lisa, I would guess that there are already more Macs than Lisas.

"One point I will agree on is the difficulty of developing software for the Mac. If Apple really wants lots of software for it, they need to make the required technical documentation available (and at a reasonable price). Development tools (compilers and assemblers) need to be able to run using a single Mac. There was an ad in *Infoworld* recently for a Mac assembler 'that requires no other hardware or software.' I don't know much about the availability of other language processors for the Mac, though I suspect (and hope) that most of them will be usable with only a single Mac. As for 'weak, bug-ridden' high-level languages, again consider the IBM PC. That's probably a fair description of many of the language processors that were available for the IBM PC during its first year. IBM's Pascal compiler, for example, is atrocious.

"I also agree that a 68000 assembler should be able to run with 128K and one disk drive, but comparing that to the DRI 8080 assembler that ran in 32K is very unfair. 8080 assembly language is much simpler than that of the 68000. The 68000 has more addressing modes, more instruction formats, and its assembly language has a more complex syntax. Also, the DRI 8080 assembler must have been a bare-bones, absolute assembler. A bare-bones 68000 assembler could be written to run on a 64K CP/M system (but probably not on a 32K system). A similar assembler should run, therefore, on a 128K Mac. However, I would assume that Apple does not want to put out just a bare-bones assembler. If, for example, the Apple assembler generates object code that can be linked with separately compiled or assembled modules, this would add considerable complexity to the assembler.

"While a Mac assembler should be able to run in 128K, I don't think it's a big problem if it does not. Anyone doing serious software development on the Mac will have to have 512K (the next step after 128K) and two disk drives, even if some of the tools run in 128K. After all, how many developers of IBM PC software have only 128K in their machines? And even fewer have only one disk drive.

"I will be like the other readers who requested 68000 material and not submit any 68000 code. Like most of these readers (probably), I am interested in the 68000, but do not actually have a 68000 system. Therefore, I also do not have any 68000 code to submit. With the attitudes expressed in your column about the 68000 (and about Intel's 80286), one wonders whether you would want to include any 68000 code or material anyway.

"Finally [in reference to another column on identifying IBM PC environments], location F000:FFFEH on my Corona contains 0FEH, just like the PC/XT. Apparently, they want software to think it's running on an XT, just in case a hard disk happened to be attached (yes, I do own an IBM compatible, though I like my 6809 system better). The AT&T computer that we use at work has 00 (zero) at that location (the date in the ROM is 05/03/84)."

This letter from Jim neatly illustrates the polarization that the Mac has induced within the programming

community. There is a certain class of people who are dazzled by Apple's "insanely great" publicity and will forgive the Macintosh anything. There is *another group*, among whom I guess I must number myself, who find Apple's pretentious advertising and grandiose claims a little ridiculous, considering that almost everything that's good about the Mac's user interface was "borrowed" directly from the Xerox Star that has been around for ten years. Speaking of pretentious, see the recent *Playboy* interview with Steve Jobs for some amusing pronouncements, especially about us programmers over 30 years old.

The fact is that everybody would find the sales of 200,000 Macs in the first year to be very remarkable, if Apple hadn't spouted off all those projections about how it was going to build a Mac every 15 seconds in their new super MacFactory. To argue that the Mac has already sold more machines than the Lisa is not what you'd call a devastating rejoinder, since Apple has already thrown in the towel on the Lisa and reincarnated it as the Macintosh XL.

As for the Macintosh assembler, I feel that I'm on very solid ground with my previous comments, having written several assemblers myself. The fact that the instruction set of the 68000 is powerful and extensive cuts both ways: of course it means that you have to worry about assembling more complex syntax, but it also means that you have a much more powerful language in which to *write* the assembler. Unless, of course, you are so bull-headed as to insist on writing your assembler in a high-level language like Pascal, which is completely unsuited to the job. Then you *do* end up with monstrosities like the Apple Macintosh Assembler or the Microsoft 8086 Macro Assembler. The proof of my contention has been conveniently provided by the product "MacASM" from Mainstay of Agoura Hills, California. This is an integrated full screen editor, macro assembler, and resource compiler for the Macintosh that runs nicely in 128K and costs only $125.00. "MacASM" is not copy protected and supports multiple drives including hard disks. Mainstay can be reached at (818) 991-6540.

MSDOS Device Drivers

Stan Mitchell of San Jose, California, writes: "I thought I would pass along a tool that comes in handy when working with MSDOS device drivers. It depends upon a useful property of the NUL device driver. Unlike other devices, it is embedded in the DOS module (IBMDOS.COM for the PC) and its device header is at the head of the device chain. By using the next device pointer in successive headers, the complete chain can be revealed. It is interesting that if any installed devices are present, the NUL header points to the first of these. When installed devices are exhausted, the link continues with the resident devices (contained in IBMBIO.COM for the PC).

"The only 'trick' in the program is using the OPEN function to return the device header pointer. This is an undocumented feature of MSDOS."

Readers, please note that the format of the reserved area in the file control block is different under MSDOS 2.0 than it is under version 3.0. Stan's C program and the assembly language source for the _peek function are provided as Listing Two (page 122) and Listing Three (page 125).

Microsoft Assembler Bug of the Month

Gregor Owen of Port Jefferson Station, New York, writes: "I don't know if anyone's still counting, but here's my contribution for Microsoft Assembler bug of the month. In the assembly below, the code es:lodsb produces an error; however, the code below it, mov ax,es:[si+1], produces an error too. If you comment out the es:lodsb, *both* errors disappear—so superior to the old-fashioned kind of assembler, where frequently the removal of defective code only removes errors in the vicinity of the defect.

"At this point, I don't really know if es:lodsb represents a legal operation. I don't happen to have an Intel book at this location, and since every 8086/88 instruction has a unique set of addressing modes, it's difficult to get any kind of feel for what's allowed. Needless to say, the IBM/Microsoft assembler manual is not helpful. It's true that if one comments out es:lodsb and assembles lods byte ptr es:[si+0] both errors disappear, but I *don't think* anybody who's used the Microsoft assembler for any short while would take any comfort from that.

"And while we're on the subject, notice the expressive elegance and beauty of the phrase

```
lods byte ptr es:[si + 0]
```

typical of the language Intel has provided for us. The si+0, I concede, is something I've been doing ever since I managed to assemble some kind of complicated expression that had [si] in it—producing code that referenced [si+0FFFFH] and didn't generate an assembler error. Of course I probably don't have to use si+0 in every expression, but who wants to spend the time to find out? That's the beauty of the language: every programmer, in order to produce stuff that works at all, will develop a private dialect containing much myth and legend, thus making the source even more incomprehensible than it is anyway.

"My personal theory about 8086/88 assembly language is that what we have here is a marketing coup on the part of Intel. They had a ridiculous processor, with the traditional inept Intel approach to registers and addressing. Somehow, they sensed it would be awfully embarrassing to expose this thing in all the stark simplicity of their 'lxi h,7'-type assembler [for the 8080 among other processors]. So they commissioned the finest, most convoluted, Pascalized minds in the software industry to come up with an assembler so grand and structured and incredibly intricate that no one would ever notice how awful the thing was for which it was assembling. Judging by the results, they seem to have succeeded.

"And those of us who labor in the vineyards of the information revolution can't be all too high-horsey about this: after all, the Intel/Microsoft language makes it almost impos-

sible for those pesky amateurs to figure anything out, thus preserving the world of 8086/88 assembler for us, the elect: people paid to spend the necessary endless hours figuring out how to use the language and avoid the Microsoft bugs. A sort of 'programmer's employment project' for the new age."

Gregor's program is provided as Listing Four (page 126). My uneducated guess, in this case, is that the Microsoft Assembler interprets the es: portion of the statement es:lodsb as the definition of a label named es. It then gives you the error message "symbol already different type," since the name es is already hardwired into the symbol table as an assembler directive. All the other errors follow from the ensuing type conflicts.

Fun with your PC/AT

Assemble the following instruction sequence into a short assembly language program and execute it on the IBM PC/AT of your choice:

```
mov  bx,0ffffh
mov  ax,[bx]
```

The system dies immediately. The same sequence of instructions will run just fine on an 8086 or 8088-based machine. What is happening here?

It turns out that on the 80286, accessing a word operand at offset 0FFFFH. i.e.. a segment wrap, causes a hardware interrupt 13H. Alas (as Jerry Pournelle would say), this interrupt is already used by the IBM PC ROM BIOS for the disk driver. So when your program accidentally generates a segment wrap fault, it traps to the disk driver with nonsense parameters in the registers and probably writes garbage all over your nice hard disk. This kind of phenomenon, when we move a previously happy program from an IBM PC to a PC/AT and watch it go to heaven, helps us while away those rainy afternoons.

DDJ

16-Bit Toolbox (Text begins on page 118)

Listing One

```
*  Integer Square Root (32 to 16 Bit)
*
*  (Exact method, not approximate)
*
*  Call with:
*         D0.L = Unsigned number
*
*  Returns:
*         D0.L = SQRT (D0.L)
*         D1.L <> 0 if not exact root
*
*  Uses:
*         D1-D4 as temporaries ---
*         D1 = Error term
*         D2 = Running estimate
*         D3 = High bracket
*         D4 = Loop counter
*
*  Notes:
*         Result first in D0.W, but is valid in longword.
*         Takes from 1480 to 1832 cycles (including RTS).
*         (Word version is from 548 to 660 cycles).
*
lsqrt   move.w     #15,d4     ; Loop count (bits-1 of result)
        moveq      #0,d1      ; Result in D1
        moveq      #0,d2
sqrt1   asl.l      #1,d0      ; Get 2 leading bits at a time and
        roxl.l     #1,d1      ; into Error term for extrapolation.
        asl.l      #1,d0      ; (Classical method, easy in binary)
        roxl.l     #1,d1
        asl.l      #1,d2      ; Running estimate * 2
        move.l     d2,d3
        asl.l      #1,d3
        cmp.l      d3,d1
        bls        sqrt2      ; New error term > 2* running est.?
        addq.l     #1,d2      ; Yes, we want 1 bit then.
        addq.l     #1,d3      ; Fix up new error term.
        sub.l      d3,d1
sqrt2   dbra       d4,sqrt1   ; Do all 16 bit-pairs.
        movel      d2,d0      ; Returns answer in D0.W
        rts
```

End Listing One

Listing Two

"C" source code for Stan Mitchell's program to dump the device driver chain.

```
/*************************************************************************
 * dd_dump.c              program for displaying device driver chain     *
 *                        for DOS 2.00,2.10,3.00                         *
 *                                                                       *
 * by stan mitchell       november 21,1984                               *
 *                        Lattice C v. 1.04                              *
 *************************************************************************/

#define  void   int
#define  byte   char
#include "stdio.h"

/* DOS function calls */
#define  OPENFCB    0x0f
#define  CLOSEFCB   0x10
#define  VERSION    0x30

extern   int _peek();

struct   DEVHDR
         {
         unsigned   nxthdr_off;  /* doubleword ptr to next device hdr in chain */
         unsigned   nxthdr_seg;
         unsigned   attr;        /* type of device driver */
         unsigned   strat;       /* device strategy entry point */
         unsigned   intrpt;      /* interrupt entry point */
         char       dname[8];    /* device name */
         };
```

 (Listing Continued, text begins on page 118)

Listing Two

```
29
30      struct  FCB
31              {
32              byte       drive;        /* drive designator */
33              char       fname[11];    /* file or device name */
34              unsigned   curblk;       /* current blk (set to 0 by OPENFCB) */
35              unsigned   recsiz;       /* logical record size (set to 0x80 by OPENFCB) */
36              long       fsize;        /* file size in bytes */
37              unsigned   date;         /* creation or last update date */
38              byte       sys_rsv[10];  /* fields reserved for DOS */
39              byte       bset[5];      /* relative record numbers */
40              );
41
42      struct  RSV2_X
43              {
44              unsigned   time;         /* creation or last update time */
45              byte       attribute;    /* device or file attribute */
46              unsigned   dhdr_off;     /* offset address of device header */
47              unsigned   dhdr_seg;     /* segment address of device header */
48              byte       unk[3];       /* unknown usage */
49              );
50
51      struct  RSV3_X
52              {
53              unsigned   time;         /* creation or last update time */
54              unsigned   attribute;    /* device or file attribute */
55              unsigned   dhdr_off;     /* offset address of device header */
56              unsigned   dhdr_seg;     /* segment address of device header */
57              byte       unk[2];       /* unknown usage */
58              );
59
60      struct  DEVHDR   device;          /* device header */
61      struct  FCB      device_fcb;      /* standard FCB for device */
62      struct  RSV2_X   *dos2x;          /* reserved field definitions DOS 2.x */
63      struct  RSV3_X   *dos3x;          /* reserved field definitions DOS 3.x */
64
65      main()
66      {
67      int       dev_off,dev_seg,i,j;
68
69      initfcb(0,"NUL        ");  /* set up standard FCB for NUL device */
70      if (bdos(OPENFCB,&device_fcb) & 0xff) /* open the device */
71              {
72              printf("Unable to open device\n");
73              exit(1);
74              }
75      if ((bdos(VERSION,0) & 0xff) == 3)
76              {  /* the reserved fields are allocated differently in DOS 3.x */
77              dos3x=(struct RSV3_X *) &device_fcb.sys_rsv[0];
78              dev_off=dos3x->dhdr_off;
79              dev_seg=dos3x->dhdr_seg;
80              }
81      else if ((bdos(VERSION,0) & 0xff) == 2)
82              {  /* ... than they were in DOS 2.x */
83              dos2x=(struct RSV2_X *) &device_fcb.sys_rsv[0];
84              dev_off=dos2x->dhdr_off;
85              dev_seg=dos2x->dhdr_seg;
86              }
87      else
88              {  /* forget it for DOS 1.10 or earlier */
89              printf ("  DOS 2.00 or newer required\n");
90              exit(1);
91              }
92
93      printf("   Device driver chain .... \n");
94      printf("\n");
95
96       /* display the current DOS chain of device drivers */
97      printf("   ptr      type    name       strategy ptr   interrupt ptr  \n");
98
99      for (;;)
100        {
101        printf("%04x:%04x  ",dev_seg,dev_off); /* device header ptr */
102        if (dev_off == 0xffff) break;          /* if last in chain, offset=0xffff */
103        /* move the device header into the data segment structure "device" */
```

```
104         _peek(dev_seg,dev_off,&device,sizeof(device));
105         printf(" %04x  ",device.attr);          /* display the device attribute word */
106         if ((device.attr & 0x8000) == 0)        /* if a block device ... */
107               for (j=0;j<=3;j++) printf("%02x",device.dname[j]); /* there is no device name */
108         else
109               for (j=0;j<=7;j++) printf("%c",device.dname[j]); /* otherwise display device name */
110
111         printf("        %04x:%04x       ",dev_seg,device.strat); /* strategy entry point */
112         printf("%04x:%04x    \n",dev_seg,device.intrpt); /* "interrupt" entry point */
113         dev_seg=device.nxthdr_seg; /* set up ptr to next device header */
114         dev_off=device.nxthdr_off; /*   and loop back to display it */
115         }
116
117      }
118
119      /* initialize standard FCB */
120      void     initfcb(drv,name)
121      byte     drv;
122      char     name[];
123       {
124       int     i;
125
126       device_fcb.drive=drv;  /* drive designation: default drive=0, A=1, etc. */
127       for (i=0;i<=10;i++) device_fcb.fname[i]=name[i]; /* device or file name */
128       for (i=0;i<=4;i++) device_fcb.bset[i]=0; /* fields not zeroed by open call */
129       }
130
```

End Listing Two

Listing Three

Assembly language source for "peek" library function, used in Stan Mitchell's device driver dump program.

```
                                PAGE
                                PAGE    62,132
                                NAME    peek
                          ;*************************************************************************
                          ;
                          ;       void    _peek(segment,offset,buffer,nbytes)
                          ;
                          ;       unsigned  segment;  /* segment portion of memory addr */
                          ;       unsigned  offset;   /* offset portion of memory addr */
                          ;       byte      *buffer;  /* local memory buffer (in data segment) */
                          ;       unsigned  nbytes;   /* number of bytes to transfer */
                          ;
                          ;       Lattice C assembly language interface convention followed
                          ;          (ES=DS on entry)
                          ;*************************************************************************
                          ;
= 0004                    EXTRA   EQU   4
                          ;
                          PGROUP  GROUP   PROG
0000                      PROG    SEGMENT BYTE  PUBLIC  'PROG'
                                  PUBLIC  _PEEK
                                  ASSUME  CS:PGROUP
                          ;
0000                      _PEEK   PROC    NEAR
0000  55                          PUSH    BP              ;
0001  8B EC                       MOV     BP,SP           ;
0003  1E                          PUSH    DS              ;
0004  56                          PUSH    SI              ;
0005  57                          PUSH    DI              ;
0006  51                          PUSH    CX              ;
                                  ;
                                  ;get source segment
0007  8E 5E 04                   MOV     DS,WORD PTR [BP+EXTRA]
                                  ;
                                  ;get source offset
000A  8B 76 06                   MOV     SI,WORD PTR [BP+EXTRA+2]
                                  ;
                                  ;get destination offset in ES
000D  8B 7E 08                   MOV     DI,WORD PTR [BP+EXTRA+4]
                                  ;
                                  ;get byte count for transfer
0010  8B 4E 0A                   MOV     CX,WORD PTR [BP+EXTRA+6]
                                  ;
                                  ;move the bytes into local buffer
0013  F3/ A4                      REP     MOVSB
```

 (Listing Continued, text begins on page 118)

Listing Three

```
                                ;
0015 59                         POP     CX              ;
0016 5F                         POP     DI              ;
0017 5E                         POP     SI              ;
0018 1F                         POP     DS              ;
0019 5D                         POP     BP              ;
001A C3                         RET                     ;
001B                    _PEEK   ENDP
001B                    PROG    ENDS
                                END
```

End Listing Three

Listing Four

```
Gregor Owen's Microsoft Assembler Bug of the Month.
Assemble this brief program for some entertaining error messages,
then comment out the line "es:lodsb" and assemble it again.


microwiff segment

          assume cs:microwiff,ds:microwiff,es:microwiff,ss:microwiff

microwimp:

          es:lodsb
          lods      byte ptr es:[si+0]
          mov       ax,es:[si+1]

microwiff ends

          end       microwimp
```

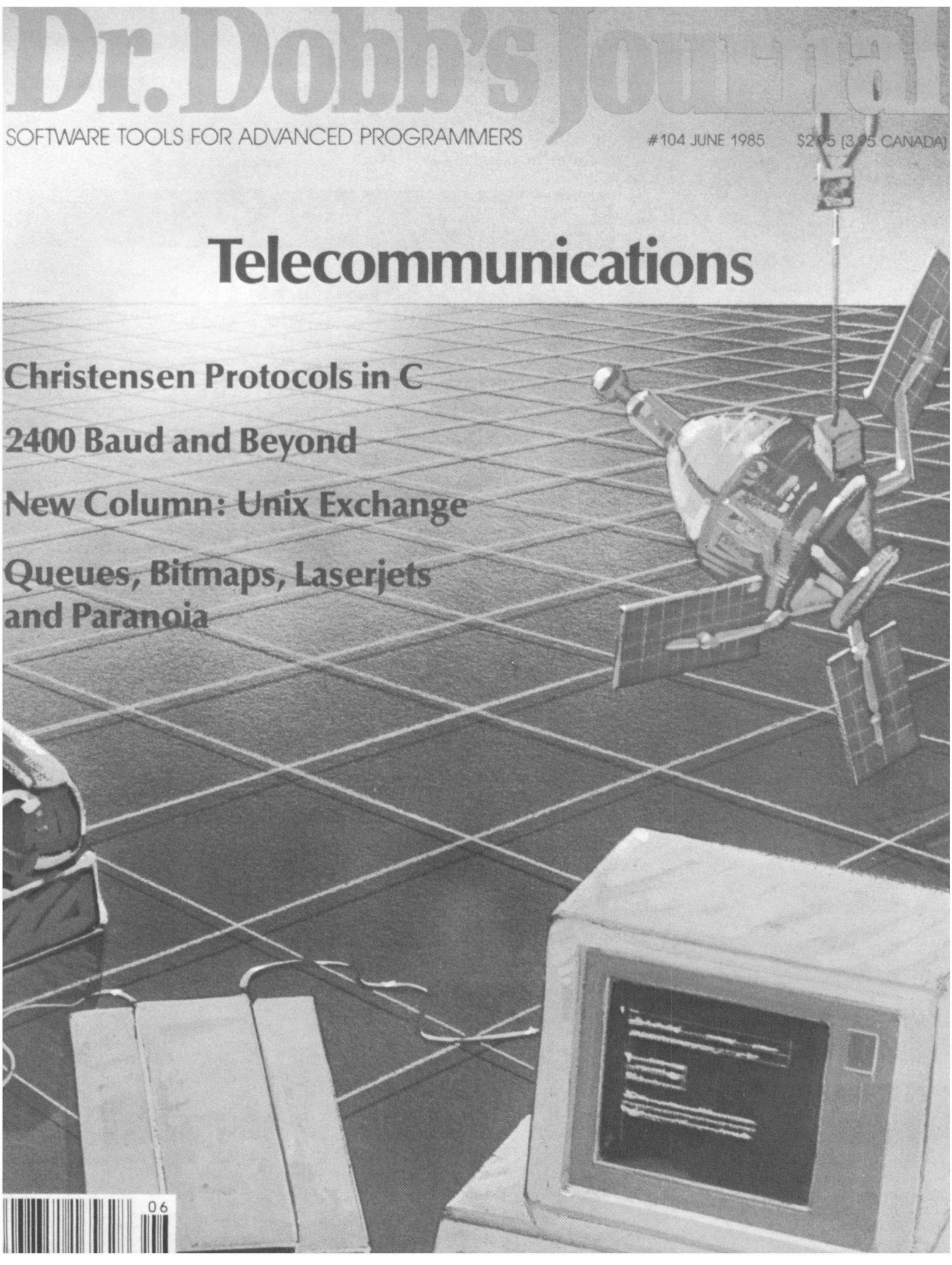
End Listings

Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

#104 JUNE 1985 $2.95 ($3.95 CANADA)

Telecommunications



Christensen Protocols in C

2400 Baud and Beyond

New Column: Unix Exchange

Queues, Bitmaps, Laserjets and Paranoia

06

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Randy Sutherland*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Polly Koch*
Typesetter *Jean Aring*

Production
Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover *William Cone*

Circulation
Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Snee*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Kathleen Boyd*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*

Advertising
Advertising Director
Stephen Friedman (415) 424-0600
Advertising Sales
Walter Andrzejewski (617) 567-8361
Lisa Boudreau (415) 424-0600
Beth Dudas (714) 643-9439
Michele Beaty (317) 875-0557
DDJ Classifieds
Alice Abrams (415) 424-0600
Advertising Coordinators
Jay Horvath (415) 424-0600
Alice Abrams (415) 424-0600

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303. (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank. For subscription problems, call: outside of CA 1-800-321-3333; within CA 1-619-485-6535.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

June 1985
Volume 10, Issue 6

CONTENTS

The View from Mid-Year

These are the articles that generated the most reader response in recent months:

January, ''Fatten Your Mac'' by Thomas LaFleur and Susan Raab, and ''Archiving Files with CP/M 80 and CP/M 86'' by Ian Ashdown

February, Excerpts from *Fire in the Valley* by Paul Freiberger and Michael Swaine, and ''CP/M Exchange,'' in which Bob Blum discusses buffered disk I/O

March, Richard Stallman's ''GNU Manifesto,'' and Dave Cortesi's ''Tour of Prolog''

April, Allen Holub's general purpose Quicksort routine in ''C Chest''

May, ''Solid Shape Drawing on the Commodore-64'' by Richard Rylander

This month, Dean Gengle, our featured author, asks how we are to maintain democracy in an information age. There is more to telecommunications than baud rates and band widths! Special thanks to Ian Ashdown of byHeart Software and Steven Bellovin of AT&T Bell Laboratories for serving as referees.

Next month we deliver three hardware articles: ''Build a Custom PC or Clone,'' ''The Ultimate Parallel Print-Spooler,'' and the feature ''Designing a Real-Time Clock for the S-100 Bus.'' In ''16-Bit Toolbox,'' Ray Duncan presents MSDOS installable device drivers with an overview of the Unix device drivers which inspired them. Allen Holub, in ''C Chest'' offers an MSDOS directory utility which includes a multicolumn print utility. ''Computer Calisthenics'' will announce the winners and answers to Michael Wiesenberg's recent puzzle competition. We will also introduce a new column, ''Mac Toolbox,'' to provide the background needed to write programs that really make use of the Macintosh's capabilities.

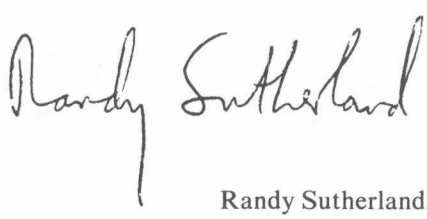
August, we'll C.

In September we'll take a practical look at an academic subject: Algorithms.

October Forth issue: Leo Brodie will report on the implications of the Novix chip for programmers. We also expect to cover Dr. Alan Winfield's Forth machine: an English job, with racing stripes. And we'll review Neon, a Forth-based, Smalltalk-like language for the Mac.

November: Programming Tools for Modula-2 Programmers.

December: Programmer's View of GEM.



Randy Sutherland
Editor

Referees

Ian Ashdown, byHeart Software
Steven Bellovin, AT&T Bell Laboratories

Dr. Dobb's Journal

ARTICLES

Information Age Issues *by Dean Gengle* — **52** Advances in telecommunications carry risks to our civil liberties; the problems and some suggested solutions. **(Reader Ballot No. 193).**

Modems: 2400 Bit/Sec and Beyond *by Dale Walsh* — **56** Two approaches to increasing modem speed: use more of the available bandwidth; construct more densely packed signaling alphabets. **(Reader Ballot No. 197).**

C UART Controller *by Don Gay* — **60** A program for easy modification of the parameters of a programmable USART. **(Reader Ballot No. 195).**

Christensen Protocols in C *by Donald Krantz* — **66** This highly-portable file transfer utility is a basic Christensen protocol engine with a minimal terminal program. **(Reader Service No. 196).**

COLUMNS

Dr. Dobb's Clinic *by D. E. Cortesi* — **16** The end of shadowy printing; Z80 and BASIC code puzzles; pseudo-random numbers; alloc( ) and free( ). **(Reader Ballot No. 190).**

C Chest *by Allen Holub* — **26** Queue management routines; bitmaps for grep.c. **(Reader Ballot No. 191).**

Unix Exchange *by Axel Schreiner* — **42** Tricks with the C preprocessor. **(Reader Ballot No. 192).**

16-Bit Software Toolbox *by Ray Duncan* — **94** WordStar for the HP LaserJet; Microsoft 8086 MacroAssembler 2.0; the TRIM filter. **(Reader Ballot No. 194).**

CP/M Exchange *by Bob Blum* — **106** SID2TTY, an alternate console for SID; CP/M Plus inaccuracy. **(Reader Service No. 198).**

DEPARTMENTS

Editorial **6**

Letters **8**

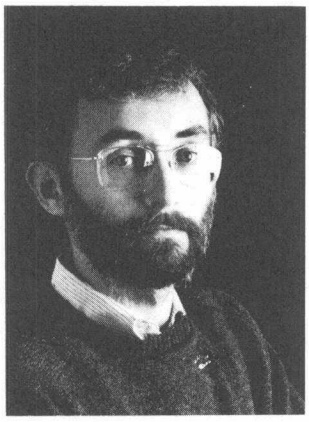
Reviews **92** Artisoft's Envoy

DDJ Classifieds **99**

Of Interest **118**

Advertiser Index **128**

EDITORIAL



You'll find the answers to this month's Industry Awareness quiz upside down at the bottom of the page.

[1] How many computer books have been published in the last five years with the title *Up and Running*?

[2] How many C compilers are available for the IBM PC?

[3] How many IBM 360/75s were there installed in 1979? (Source: International Data Corporation.)

[4] How many DEC PDP-8s? (Source: ibid.)

[5] How many computer magazines folded in 1984? (Source: Marketing Technology.)

[6] What do you call a computer company that drives away people like Steve Wozniak and Andy Hertzfeld, pretends that the division keeping the company alive doesn't exist and thinks that it can make a machine a home computer or a portable by fiat? (Hint: it starts with an *A*.)

[7] Adam Osborne is predicting that within a year virtually all microcomputer software will sell for under $100 and Phillippe Kahn has been heard making predictions of wealth for Borland personnel that sound suspiciously like those we heard from Osborne Computer Company executives on *60 Minutes* a few years ago. Is Osborne emulating Kahn, or is Kahn emulating Osborne?

[8] By what ratio will Bill "The Cracker" Landreth's royalties for his book *Out of the Inner Circle* exceed the fine he paid when convicted of wire fraud? (Hint: his fine was half the dollar amount of the damage that he says the FBI did to his equipment in confiscating it.)

[9] What do you call an Association for DAta-Processing Service Organizations that responds to the vehement rejection by the Microcomputer Managers' Association of its copy-protection proposal (a plan that MMA threatened to respond to by boycotting participating vendors) by saying "Corporate America would find it hard to turn its back on the [software] these companies are offering"? (Hint: it starts with an *A*.)

[10] Back in February, San Francisco police lieutenant Thomas Suttmeier admitted that he had for the preceding two months had access to all computer files in the city's criminal justice system, including the public defender's files of defense strategies and files of the office that handles civilian complaints against the police. In fact, any police officer with access to the computer could have tapped into the files by typing "Tom's Menus." What can be done to prevent such abuses of power?

Michael Swaine

Answers: [1]: At least three. [2]: At least two dozen. We're reviewing them next month. [3] 25. [4] 34,665. [5]: 55. [6]: Arrogant. [7]: Neither. Entrepreneurs do not emulate, imitate, acknowledge predecessors or remember the past. [8]: By conservative estimate, 100:1. [9]: Arrogant. [10]: Be serious. We're talking about the *police* here. They wouldn't do anything wrong.

LETTERS



Prolog

Dear *DDJ*:

I enjoyed reading your March [1985, #101] issue, especially the article entitled "Programming In Logic." I am an avid believer in Prolog, hoping to see it continue to catch on in the U.S.

I'm writing to you in part to thank you for helping promote an understanding of logic programming and Prolog. I am also writing to let you know that while the table of sources of Prolog was a terrific addition to the article, it was not complete. Arity, a spin-off from LOTUS Development Corporation and recently funded by LOTUS, is in the business of building sophisticated Prolog tools and applications.

We take great pride in having built Arity/Prolog, a highly optimized and extended version of Prolog. We have two products available now: the Arity/Prolog Compiler and Interpreter and the Arity/Prolog Interpreter. We are also very excited about our upcoming products.

Sincerely,
Meredith Bartlett
Arity Corporation
336 Baker Avenue
Concord, MA 01742

Dear *DDJ*:

Congratulations on your Prolog issue. The articles, together with Dean Schlobohm's Tax Advisor program, provide an excellent introduction to Prolog.

I would just like to add, for your readers' information, the Quintus name to John Malpas' list of sources of Prolog. Quintus supplies a high-performance Prolog development system for the Sun workstation, and DEC Vax under Unix 4.2 and VMS. Quintus Prolog Release 1.0 features an incremental compiler, integrated Emacs text editor, comprehensive debugger, style checker, C interface, on-line help system, and a high degree of compatibility with DEC-10/20 Prolog and C-Prolog. In addition, Quintus distributes DEC-10/20 Prolog as an unsupported product.

Sincerely,
David Warren
Quintus Computer Systems
2345 Yale Street
Palo Alto, CA 94306

Capitalism vs. Stallmanism?

Dear *DDJ*:

Although I am no fan of Unix, I read Richard Stallman's "Manifesto" with delight. Here is a true adherent to the Hacker ethic! Mr. Stallman's militant advocacy is enhanced by his well-written arguments. Many may find points to dispute in his statement, but all will find it thought-provoking and interesting.

I wish him and his associates luck. The computer industry needs to be turned upside down every so often. Mr. Stallman seems just the fellow to do it this time. Gee—maybe I ought to reconsider my dislike of Unix—I'd hate to be left out of the fun when GNU arrives!

Yours,
James F. Glass
18653 Ventura Boulevard
Suite 351
Tarzana, CA 91356

Dear *DDJ*:

Richard Stallman's "The GNU Manifesto" in your March issue made me furious. I'd love to take 5 pages to dispute all of his 5 pages, but I'll confine myself here to 2 major points.

First, Stallman doesn't believe in property rights. As evidence on a small scale, note the omission of the usual "Unix is a trademark of Bell Labs" in his article, which was present in every advertisement in that issue. Stallman purposely omits this because he doesn't recognize their claim to the name Unix.

However, if it hadn't been for all the time and money invested by Bell Labs in development, then Unix would not be as widespread, popular, or valuable as it is today. If Unix had no value, then neither would GNU. (To see this, reread Stallman's article, substituting "Glorp" for "Unix" and see if he still makes sense.) In effect, Stallman is happy to take full advantage of Bell Labs' investment, with no recompense to them. Would you spend your time and money developing software, only to have Stallman copy it and give it away for free? I wouldn't. (In fact, Stallman wouldn't even give you the choice. He states: "the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.")

Second, Stallman's explicit philosophy is socialist redistributionism. He states that "all sorts of development can be funded with a software tax." Stallman wants to tax *me* to fund *him* to develop software he will give away for *free*. Is this how you want your tax money spent? Not me.

Stallman states: "the fundamental act of friendship among programmers is the sharing of programs"; "good system software [should be] free, just like air"; what deserves a reward is "social contribution." In other words, the Good is what is social, shared, free. On the other hand, he states: "marketing agreements . . . forbid programmers to treat others as friends"; "users [are] at the mercy of one programmer or company that owns the sources"; software vendors

"[extract] money from users of a program by restricting their use of it"; "a person who enforces a copyright is harming society." In other words, the Bad is what is individual, owned, bought, sold.

Stallman obviously prefers the former, which is characteristic of the Marxist ideals of the Soviet Union. Me, I prefer the latter, typified by the good old capitalist U. S. of A.

Stallman says he is looking for contributions. I would be delighted to offer a one-way plane ticket to Moscow.

Sincerely,
Robert Schwartz
P.O. Box 1637
Wakefield, MA 01880

Dear Richard:
Hurrah! Three cheers for GNU! Your proposal and the philosophy behind it are so refreshing, I almost couldn't believe what I was reading in this month's issue of *Dr. Dobb's Journal*. As you can see, I am ecstatic that someone is finally doing something about Unix.

Since about 1978 or so I have been going to Unix users' meetings, and I remember meeting Ron Cain at one of these meetings before he left SRI. I obtained a Unix Version 6 system about that time to run on a PDP-11/70 for a company contract. One of my colleagues spent a few nights writing some very useful programs; one was a combination nroff/troff marriage that output bit-mapped fonts on a Versatec printer. Another rotated the nroff/troff output by 90 degrees so that printed copy came out sidewise, but properly formatted on the Versatec 11×8½ fan-fold paper. With that system our little company was in business, and the number of reports and proposals done on that system was amazing. We had to repair the file system now and then, but that was a minor thing compared to the amount of work that got done.

Rather than waste your time with more of my nostalgia, I want to tell you that I have had in mind the same idea as you have proposed. After some of my friends were involved in the secure Unix kernel work at UCLA, I began to realize that it was in fact possible to write a Unix-like system that would not have the royalty problems that make Unix (even on micros) so ridiculously expensive. It is a crime that the micro community does not have access to the hundreds of programs written in C for Unix systems and that so many (inferior) wheels are being reinvented. I obtained a copy of the XINU book and have accumulated a library of articles on other people's experiences in porting Unix to new machines. I had just about everything I needed except a C compiler for my (don't laugh) Commodore 64. Recently one has been announced by Abacus Software, so I am getting all set.

The C64 is a short-term thing; I am looking forward to getting a 68000-based system soon. My motivation for wanting to do a system on the C64 was to be able to run good programs (e.g., a full-screen editor like Emacs) on that system. Memory limitations will obviously rule out multiple users, but if someone comes along with a hard disk interface, at least limited multi-processing would be nice. Already there are switchable memory banks available in 64K sections up to 512K.

For what it's worth, I would like to offer my help in your enterprise. This would most likely take the form of programming assistance. I suspect you will get a lot of other offers of help as well, now that you have "gone public" with your GNU concept. My resources are somewhat limited; however, it may be possible to get a Southern California "GNU Group" going. Perhaps you will spark enough interest to form a mailing list of volunteers.

Once again, I am glad someone like you has come along, and I hope there is some way I can help out.

Sincerely,
Rollin V. Weeks
7130 Marymount Way
Goleta, CA 93117

Dear *DDJ*:
After reading Richard Stallman's "Manifesto" [March 1985, #101], I felt compelled to respond. To begin, I am certainly not opposed to the idea of a public domain version of Unix, nor to public domain software in general. But my initial reaction to Mr. Stallman's sanctimonious outpouring was to feel rather insulted. I have been a programmer for about ten years now, and I don't believe that programmers are any more or any less greedy than others in this society. In the case of software companies supposedly making large sums of money selling copyrighted products, I rather doubt whether the programmers see very much of the profits.

I believe Mr. Stallman must be suffering from a few other fantasies besides that of a public domain Unix. He perhaps forgets that he is living in a capitalist society. Of course, everybody needs food, even more than free software. So food would be distributed for free, right? I have no doubt that farmers, food processing workers, supermarket clerks, truck drivers and others employed in the food industry will not let personal greed blind them to the logic of this argument. People need clothing, housing, medical care and education, too. All these things are necessities and as such should be guaranteed to everyone .... As for Mr. Stallman, I wonder how he can afford to devote so much of his time to this admirable project. Perhaps because his previous employer paid him such a fat salary that he can afford to forget about the dirty business of making a living for a little while.

I am afraid that I might sound almost as self-righteous as Mr. Stallman. I read *Dr. Dobb's* for its valuable technical information and for enjoyment. If any of your readers are concerned about moral or political issues, I would recommend that they subscribe to the *New York Guardian*. And I would also recommend that Mr. Stallman stick to technical matters, for which I am sure he is highly qualified.

Sincerely,
David Kettle
8 Milepost Place, #308
Toronto, Ont. M4H 1E1

Dear *DDJ*:
I read with excitement Richard Stallman's "The GNU Manifesto" until I reached his justifications for his approach. I feel torn because, on the

one hand, I applaud his objective, but, on the other, I cannot accept his rationale of why this is the proper approach.

My problem is that he says that "social contribution" should be reward for an individual's creativity. Indeed, he claims that a personal reward is "destructive" because it deprives society of the potential benefits of an individual's creativity. I am a programmer and the owner of a business that sells software and from a selfish point of view would reject his arguments. But I am also a free market advocate and I reject his statements on philosophical grounds which I will attempt to enumerate.

First the wealth of society is increased whenever individuals trade freely. If I charge $100.00 for a program and you freely decide to buy the program, we are both richer. I have my $100.00, which I value higher than the copy of the program, and you have the program, which you value higher than the $100.00. If we were not both richer, then one or both of us would not have entered into the trade.

To say that this is "destructive" to society because, by charging, I deny the benefits of my program to society, is to say that whatever one creates does not belong to him, but belongs in some larger sense to society, and, therefore, to deny it to society is being "destructive."

In a narrow sense Mr. Stallman is right. If Lotus Development gave away "Lotus 1-2-3" instead of charging for it, we would all use it and "society" would be richer. The problem is that Lotus Development would never have created "Lotus 1-2-3" in the first place. Further, the hundreds of others who think they can do it better than Lotus would not go to the effort of attempting to make a better "Lotus 1-2-3" without the promise of financial reward. In this sense society would be poorer because of the free software.

I believe that an individual's physical or intellectual product is his own. Unknowingly, he furthers the general good by selfishly trading it for the most he can get for it. Only a lawyer would say that one's original idea is not his own. Only a dictatorship would steal it from the author.

The proof is in the pudding. Today we see hundreds of thousands of programmers working at what they love and doing it to maximize their own rewards. The individual is supreme. The marketplace decides who wins and loses. Society benefits by this enormous effort. If we follow Mr. Stallman's philosophy, we have to elevate "society" to a higher level than the individual. This I cannot do. Indeed, wars have been fought and blood spilled over these very ideas.

Yet, I wish him every success with GNU. I don't know how he keeps food on the table during his efforts, but as he said, his reward is in his love of programming, the pride of seeing his work being used, the recognition of his peers, and just knowing he has contributed something to the world. For some, this is more important than money. Indeed, if given independent means many more would follow the same path.

I hope he succeeds with GNU (I love his recursive name!) because for all the reasons he mentions, I could be a beneficiary. I also wonder whether the success of a "free" operating system will close the door to other operating systems that might be developed if there were some money to be made. He may find that in succeeding in his quest, in the long run he may hurt the "society" that he was hoping to enrich. On second thought, there is nothing to worry about, for if there is a need for a better operating system than GNU in the future, people will be willing to pay for it.

Jim Harlan
Cogitate, Incorporated
24000 Telegraph Rd.
Southfield, MI 48075

BU Patched

Dear *DDJ*:

I read Jim Rosenberg's Letter to the Editor [March 1985, #101] regarding my article "Archiving Files with CP/M-80 and CP/M-86" [January 1985, #99], and would like to respond to his comments regarding BU and CP/M.

Mr. Rosenberg is correct in stating that the BDOS of CP/M v2.2 does *not* reset file attribute bit t3-prime whenever a file is opened, written to and closed. In this matter I admit to being mistaken. I can also confirm that Dave Cortesi's book *Inside CP/M* is also wrong in stating (p. 222) that "the BDOS sets t3' to 0 whenever it updates a directory entry, that is, whenever the data map of an extent is altered."

However, Mr. Rosenberg chose to ignore the second part of my statement: that a directory entry is changed if it is renamed (or created). In this case the BDOS most definitely sets t3-prime to zero, and it is this fact upon which depends the success of both BU and (presumably) QBAX. This can be demonstrated by setting the file attribute t3-prime of a file with a disk utility, using either the built-in "REN" command or BDOS function 23, "Rename File," to change the file's name, and then examining t3-prime again.

To quote Mr. Rosenberg: "Our own product, QBAX, will function without patching for programs such as word processors and compilers that change files by rewriting them from scratch." This is exactly what BU will do. Since "BU is almost useless under 2.2 unless BDOS is patched to install support for the archive bit," cannot the same be said for QBAX under a standard CP/M BDOS?

The answer is of course no. BU is a very useful program, as a number of satisfied users have written to tell me. Its one failure is that it cannot detect files that have been updated in place (i.e., changing records through BDOS random access calls or by appending records via BDOS sequential write calls). To do so would require a modified operating system.

QBAX, according to a review by David Fiedler in the October 1983 issue of *Microsystems*, is a fine commercial software product. The review covered QBAX1, which does not support hard disks. Since that review, Amanuensis has released QBAX2, which offers support for hard disks, automatic split file restoration, version numbering and date stamping. Both products apparently include the BDOS patch, referred to by Mr. Rosenberg, that implements

the archive bit feature for those files updated in place.

I wrote BU as a basic utility program that incorporated and demonstrated a relatively obscure feature of CP/M v2.2. Readers were invited to add their own enhancements, since space limitations prohibited including features such as QBAX1 and QBAX2 offer. If the *Microsystems* review is anything to go by, I would probably recommend the QBAX products to a business user of CP/M, rather than BU. In this matter, I agree with Mr. Rosenberg. I do not believe that businesses should entrust their data to unsupported public domain utilities.

Regarding CP/M's PIP utility, some further investigation revealed that it behaves exactly as you would expect under the explanation given above. When a new file is created on the destination disk, the archive bit is initially zero. The archive bit can be set by BU or another utility, and thereafter is unchanged by any file operation *except* renaming the file.

In summary, I wish to thank Mr. Rosenberg for his critique of my article, although I feel he did *DDJ*'s readers a disservice by failing to mention that the archive bit is reset when a file is first created or renamed, and by strongly implying that BU does not work.

Ian Ashdown
byHeart Software
2 - 2016 West First Avenue
Vancouver, B.C. V6J 1G8
Canada

Dear *DDJ*:

We noted with interest Ian Ashdown's BU program published in the January (1985, #99) *DDJ*. We also noted, as Jim Rosenberg of Amanuensis reported in a letter in the March issue, that the "archive bit" t3' is not automatically cleared by the BDOS on file writes or renames.

After examining the BDOS we came up with a patch to accomplish this under CP/M 2.2. With the patch installed, any time a file is written to, either randomly or sequentially, the "archive bit" will be cleared to zero. We have used this with a number of different programs including word processors, SuperCalc and Microsoft BASIC (including writing a single random record) and have encountered no problems with it.

Since a number of your readers may be interested in having this facility, I am enclosing a printout of the patch (see Listing, below). This patch assumes that the patch code is located outside of the BDOS, probably in the BIOS. The changes to the BDOS can be made by the BIOS on each warm boot, if necessary, though there are other ways to accomplish this.

Before making the patches to the BDOS, be sure that code to be replaced matches that shown in the enclosed patch (the original code is commented out in the patch).

One other approach which we have used to implement the patch is to install it directly into the BDOS source. We have used C. C. Software's excellent CP/M Source Code Generator to create the BDOS source (we have this software available, as well as our own EUREKA! disk cataloger). We then substituted Z80 instructions in enough places to gather the space we needed for the patch and reassembled the BDOS and patched it directly onto the disk.

We hope that this patch will make BU even more useful to your readers.

Bruce Haanstra, President
Mendocino Software Co.,
P.O. Box 1564
Willits, CA 95490

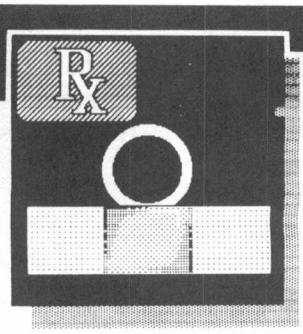
DDJ

Letters Listing (Text begins on page 8)

```
;Desc:Patch to CP/M 2 BDOS to clear update flag on close or rename of a file
;Date:02/02/85
;Time:13:51
;
; The purpose of this patch is to clear the update flag (8th bit of t3) when a
;  file that has been changed is closed.  Part of this patch file will also
;  clear the update flag on a rename.
;
BDOSBASE        EQU     0D800H          ; *** SET THIS TO THE ADDRESS OF THE
                                        ;     BASE OF YOUR BDOS ***

patcharea       equ     00000           ; *** PUT THE ADDRESS WHERE YOUR PATCH
                                        ;     CODE IS LOCATED HERE BEFORE
                                        ;     ASSEMBLY ***

updflg          equ     11              ; offset to update flag byte in fcb
                                        ; 11=t3' (archive bit)
; patch to rename
        org     BDOSBASE+822h
        jmp     patch1
;
; was   lxi     d,16
;       dad     d
;       mov     m,a


; patch to clear update flag on a file that has been changed
        org     BDOSBASE+91Ch
        jmp     patch2
; was   JMP     BDOSBASE+810h   (update1)

;
        org     patcharea
;
; ***** UPDATE FLAG PATCH *****
;  01/06/85 - by BWH
;  this patch forces the directory entry being written to have its update
;   flag bit reset to 0 so FD and MDIR can handle ALL updated files.
;

patch1:                         ; RENAME patch
        lxi     d,16                    ; move over to desired name
        dad     d
        mov     m,a                     ; keep the same user number
        call    clruflg                 ; clear the update flag
        jmp     BDOSBASE+827H           ; (chgnam1) go back and continue

patch2:                         ; UPDATE flag patch
        call    BDOSBASE+55Eh   ; (fcb2hl) get address of fcb in dir buffer
        call    clruflg         ; offset to FD update flag byte (0-11) of fcb

        jmp     BDOSBASE+810h   ; (update1) do original stuff
;
; 01/27/85 - broke out clearing of update flag so rename can do it too
;       IN    -> HL=address of base of FCB with flag to clear
;       USES -> A, HL, DE
;
clruflg:                        ; clear update flag routine
        lxi     d,updflg        ; offset to FD update flag byte (0-11) of fcb
        dad     d
        mov     a,m             ; get the current byte
        ani     7FH             ; clear 8th bit
        mov     m,a             ; and put it back
        ret
;
;
```

End Listing

DR. DOBB'S CLINIC



by D. E. Cortesi

Printer Out of Shadow

In the April column, we asked for help finding competent maintenance for our Diablo printer, but we wrote that column back in February, and the problem wouldn't wait. We posted the same query on several local RBBSs (and very glad we were that the Oxgate, PicoNet, and Simms boards exist). One Eugene Jones came through with a recommendation of The Printer Works (1961 Alpine Way, Hayward, CA 94545; (415) 887-6116). And they did the job: made an informed diagnosis; discussed it with us, explaining exactly what they would do and what it would cost; then did the work. They were thoroughly professional, helpful, and prompt. May they get every broken Diablo in Silicon Valley.

But our problems getting the Diablo fixed have made us wonder: What else is hard to get fixed? Who else is doing a good job fixing things? If you've had maintenance adventures—with happy endings or otherwise—why don't you describe them to us? Especially tell us about repair people who know their stuff. Let us in on the scams, too (with enough horror stories, we can revive "Dobb's Ex Machina"), but it would be super if we could point to good repair services in every major city.

Watt Duzzit Dew

We asked in January David Tilton asked us: What does this short Z80 assembly routine do?

```
BTS:
   cp    (hl)
   ret   z
   rl    L
   djnz  BTS
   ld    L,0
   ret
```

Only Thomas Cage of Crestview, FL, attempted to give us an answer (yea, Tom!), and while he got the essence of it, he was wrong in detail (boo, Tom!). Here is how Tilton described it.

"[The routine] actually outperforms the CPIR instruction of the Z80 in searching a list of bytes for a match by using a binary rather than a linear search, but the restrictions on the sequence of the list and its placement in memory make it less than attractive.

"On entry, A contains the byte to search for, B contains the number of levels in the binary tree, and HL points to the root of the binary tree. One of the unfortunate restrictions is that the value of L for the root must be 01h. Since the value of H remains constant, it is the value of L on return that indicates the result.

"It is helpful to think of the list as a byte array indexed by L. We start with a comparison. If there is a match, the second instruction returns with L indicating where the match occurred.

"If there was no match, the value in the tree was either too large or too small. The carry flag will be set if it was too large. The next instruction multiplies the contents of L by two and, if the carry flag was set, adds one. The DJNZ instruction checks for another level in the tree and, if there is one, loops to check the already-selected node in it. If there is not another level, L is set to zero to indicate that the byte sought was not in the tree.

"There are many disadvantages to this arrangement. It is inflexible. For one thing, a full tree must be maintained; that is, the size of the tree must always be some power of two less one. It is significantly faster than CPIR only for the larger trees. Furthermore, if you can spare a full page of 256 bytes [to hold the tree] then there is an even faster way of doing the same thing:

```
BTL:
   ld    L,A
   ld    L,(HL)
   ret
```

This is, of course, simply a byte look-up table. I thought I was being really clever with my BTS routine, but it turns out to be virtually useless."

Oh, now, cheer up. You got deep into binary trees, right? It's very profound that the successive carry-bits shifted into L are a record of the path through a tree to the desired node, where one means "go left" and zero means "go right." The binary number that defines the path to a node is a unique label for that node; it's basically a "Dewey Decimal Number" for the node.

There's a discussion of this in Knuth's Volume 1, Section 2.3 (especially see the answer to Exercise 15, p. 315). There also is a connection to data compression: if the compared values were wider than a byte, their binary node labels would be a compressed representation of them.

Watt Dozen It Dew

OK, faithful readers, you didn't do too well on Tilton's puzzle; perhaps you can do better with one from David Ross, of Iowa City, IA. He sends the BASIC code you can read in Listing One (page 18). "In theory," he says, "it does something quite remarkable. In practice, it doesn't do much of anything: it is extraordinarily sensitive to roundoff errors. Question: what ought it to do, and to what extent can it be made to work?"

Now, Tilton sent a full explanation

of his puzzle. Ross did not. That means (since the Intern hasn't a clue) that if you don't figure it out and explain it, nobody will ever know the answer.

Random Probing

In February we showed a simple generator of pseudo-random numbers (a PRNG, pronounced "prang"). We're getting mixed feedback on that; at least one reader has written to say it's no good. It would be nice if someone would subject it to some rigorous testing and tell us.

In fact, the letters we are seeing indicate a lot of ignorance about what's good in a PRNG and how to test one. Add to that a paper that Mike Swaine ran across: "Random Number Generation in Microcomputers" by Modianos, Scott, and Cornwell in *Interfaces*, Vol. 14, No. 4 (July-August 1984). We aren't familiar with *Interfaces*; it appears to be a publication of The Institute of Management Sciences.

Anyway, Modianos, et al., studied the characteristics of the PRNGs in various micro BASICs. The net of their findings is "that the random number generators which are intrinsic to the Apple II+ and IIe (using either Applesoft BASIC, Integer BASIC, or CP/M BASIC), the Osborne Executive, and the IBM PC are so flawed that we cannot recommend their use for simulation studies." The only acceptable PRNGs they found were in the HP-86, Apple III, and (oddly) the TRS-80 model III. Plus one for the Apple published by Hare, Faulkner, and Sparks in *Call APPLE*, Vol. 6, No. 1.

These authors apparently expended a lot of time and CPU cycles testing PRNG; unfortunately, they don't discuss their methods in any detail. Sure, that subject has been covered many times before, but not for some years in this magazine. Volunteers?

Object Module

By way of filling up this column, we present a module of C code that has helped us make some quick-and-dirty programs. Sometimes it's easier to do things if you have the ability to allocate a lot of small objects and free them again in any order. In *The C Programming Language*, Kernighan and Ritchie described a pair of functions, alloc( ) and free( ), to do just that. Probably these are in most C libraries.

The compiler we use implements an alloc( ), but contains no free( ). Alloc( ) alone gives the essential ability to allocate chunks of storage for buffers and what-not. But we wanted to make dozens or hundreds of little objects and then discard them again. So we did what you'd do: we rolled our own.

The module for dispensing objects appears in Listing Two (page 18). It duplicates the functions of alloc( ) and free( ), including quite a bit of validation to guard against bugs in the using code. It adds the useful functions of objmax( ) and objtrim( ), which allow you to size an object to match the data it will hold.

The method it uses is not that of Kernighan and Ritchie, but rather an old scheme that you may not have seen before: chaining adjacent blocks by fencing them in with length-words.

The more we use C's #define, the more impressed we are with the flexibility it gives. In the Object module, we used it to define an adjective, WORDAT, to stand for the awkward cast required to say, "the unsigned integer in storage where this character pointer points."

The Object module is definitely written for 8-bit machines; it relies on 16-bit addresses stored in 16-bit integers. The first step in making it portable would be to convert all the hex constants to #defined names. Also, it would be more efficient if it started the search for a free block at the point where it last found or freed one. You might enjoy making those improvements.

The performance of this design won't differ a great deal from that of Kernighan and Ritchie. You might enjoy analyzing the two and working out which is faster, under the assumption of a high turnover of many, small objects.

DDJ

Dr. Dobb's Clinic (Text begins on page 16)

Listing One

```
100 REM BASIC PUZZLE BY D. ROSS
105 REM MODIFIED FOR MBASIC BY DEC
110 DEF FNL(X)=LOG(X)/T
120 INPUT "K=",K : DIM A$(K+1)
130 FOR I=1 TO K+1 : A$(I) = "0" : NEXT I
140 T = LOG(10)
150 Z = ASC("0")
160 A0 = K*FNL(2)
170 B = 1
180 WHILE B > 0
190 A = A0+FNL(B)
200 N = INT(A)
210 IF N<0 THEN GOTO 280
220 L = A-N
230 X = 10^L
240 D = INT(X)
250 A$(K-N+1) = CHR$(D + Z)
260 B = B-10^(N-A0+FNL(D))
270 WEND
280 FOR I=1 TO K+1 : PRINT A$(I); : NEXT I
```

End Listing One

Listing Two

```
/*
        OBJECT.C
        general purpose object-allocation module
        (8-bit systems only, relies on 16-bit addresses)

        Copyright (C) 1985 David E. Cortesi
        430 Sherman Ave #212
        Palo Alto, CA 94306
        All rights reserved
        Permission granted to copy for nonprofit purposes.

An "object" is a block of storage of at least two bytes.
Functions in this module manage a pool of storage to
allocate and reclaim objects.

unsigned objinit(u) unsigned u
        Initializes a storage pool of size u, returning the
        size of the largest object that may be allocated.
        If called a second time, does nothing and returns the
        size of the largest object that can now be allocated.

unsigned objmax()
        Returns the size of the largest object that can be
        allocated from the storage pool as it is now.

char *objget(u) unsigned u
        Allocates an object of size u and returns its address,
        if possible.  If it can't, it aborts the program.

void objfree(p) char *p
        Frees the object addressed by p, which must have been
        allocated by objget().  If it wasn't, or if it is now
        free, or if it's been damaged, the program is aborted.

unsigned objsize(p) char *p
        Returns the size of the object addressed by p.  Aborts
        if *p is not a valid object.
```

```
void objtrim(p,u) char *p; unsigned u
        Trims the object addressed by p back to have size u,
        releasing excess space.  Use in a sequence like,
                p = objget(t=objmax());
               ...process data *p to size u, u <= t...
                objtrim(p,u);
        Aborts if *p isn't a valid object or if its size is
        less than u.

void objprt()
        Compiled only if DEBUG is defined, this function
        prints a map of object-space on stderr.

Allocation is first-fit.  Garbage is melded as it is created.
An object is allocated as an even number of bytes L, fenced in
by two words WWWW:

                   A     B               C     D
 ...object (WWWW)(WWWW)<== L bytes ==>(WWWW)(WWWW) object...

Words WWWW are essentially 2+L, where L is the length of the
object they bound.  The low-order bit is the in-use flag.
If (W & 0x0001), the object is active; if not, it is free.
The value K == (WWWW & 0xfffe) == 2+L is used to chain from
block to block.  For addresses A,B,C, and D as shown, A+K->C,
B+K->D, B-2->A, D-2->C, D-K->B, etc etc.  Address B is the
object's address, returned by objget() and given to objfree.

The whole storage pool is bounded by two sentinel words, 0003.
These appear to be active blocks with K=2, hence L=0, and stop
a scan over the pool.

*/
#include "a:stdio.h" /* for error messages */

/*efine DEBUG 1*/ /* or don't -- to exclude objprt() */


 #define WORDAT *(unsigned *)    /* using "char *" ptrs for words */
 #define VOID /* not supported in Aztec C */
 static char *objbase = 0,       /* base of storage pool, uninitialized /
           *objtop;             /* upper limit, for integrity checks */
static unsigned maxobj,         /* largest possible object */
                j,k;            /* scratch words */
static char *p, *q,             /* scratch pointers */
        minit[] = "Objinit",    /* abort-message parts */
        mget[] = "Objget",
        mtrim[] = "Objtrim",
        mvali[] = "Object validation",
        mpool[] = "Object pool damage";

VOID    objinit(u)
        unsigned u;
{
        if (objbase) return(objmax());

/* ensure an even number of bytes in the entire pool */
        u = (u==0xffff) ? 0xfffe : (u + 1) & 0xfffe;

/* set out 4 bytes for sentinels, 4 for first 2 WWWW words */
        maxobj = u-8;

/* ensure a reasonable size of pool, allocate it */
        if (248 > maxobj)
                objerr(minit,1);
        if (NULL==(objbase = alloc(u))
                objerr(minit,2);
```

Listing Two

```
/* set up sentinel words 0003h at ends of pool */
        objtop = objbase + u - 2;
        WORDAT objbase = 0x0003;
        WORDAT objtop = 0x0003;

/* fill pool with a single free object */
        objbase += 2;
        k = maxobj+2;
        WORDAT objbase = k;
        q = objbase + k;
        WORDAT q = k;

        return (maxobj);
}

char *objget(u)
        unsigned u;
{
/* eliminate cases of zero and too-big sizes */
        if (u==0) objerr(mget,1);
        if (u>maxobj) objerr(mget,2);

/* ensure even length, establish enventual K == L+2 */

        k = 2 + (u = (u + 1) & 0xfffe);

/* Scan to the top sentinel, looking for the first free
   object longer or equal to the requested size.  The pool
   is vulnerable to damage --  be very suspicious and test
   every link in the chain before using it. */
        p = objbase;
        while (0x0003 != (j = WORDAT p))
        {
                if (0==(j & 0x0001)) /* free */
                        if (j >= k) break; /* good size */
                q = p + 2 + (j & 0xfffe);
                if ((q <= p)||(q > objtop))
                        objerr(mpool,1);
                p = q;
        }
/* if we hit the sentinel, the requested size is not available
   (could return NULL here instead of aborting) */
        if (j==0x0003) objerr(mget,3);
 /* set p->low WWWW, q->high WWWW, validate block */
         q = p + j;
         if ((q <= p)||(q > objtop))
                 objerr(mpool,2);

 /* if the wasted space in this block would accomodate
    another object of 4 bytes or more, split off the
    high end as a new, smaller free object */
         if (j > (k+8))
         {
                 j = j-k-2; /* j=WWWW of excess object */
                 WORDAT q = j;
                 q -= j;
                 WORDAT q = j;
                 q -= 2;
                 WORDAT q = k;
                 WORDAT p = k;
         }

 /* activate chosen block, return address of data */
         WORDAT p += 1;
```

```
        WORDAT q += 1;
        return (p+2);
}

unsigned objsize(ob)
        char *ob;
{ /* This is mostly an internal subroutine.  It validates
     a user-supplied object-pointer, leaving static globals
     p, q, and k set up so that p->(WWWW) ... q->(WWWW)
     and k== WWWW & 0xfffe.

        p = ob-2;

/* address must fall within in pool, */
        if (p < objbase) objerr(mvali,1);
        if (p > objtop) objerr(mvali,2);

/* ..must have active lower WWWW word, */
        k = WORDAT p;
        if (0==(k & 0x0001)) objerr(mvali,3);
        k--; /* strip active bit */

/* ..showing size of L=2 or more, but not more than max, */
        if (k<4) objerr(mvali,4);
        if (maxobj < k) objerr(mvali,5);

/* ..which must yield a valid upper-WWWW address, */
        q = p + k;
        if ((q <= p)||(q > objtop)) objerr(mvali,6);

/* ..and the upper and lower WWWWs must match */
        if (WORDAT p != WORDAT q) objerr(mvali,7);

/* object is valid, return its data size */
        return (k-2);
}

VOID    objfree(ob)
        char *ob;
{
/* use objsize() to set up p, q, k and validate block */
        objsize(ob);

/* object is valid, make it free */
        WORDAT p = k;
        WORDAT q = k;

/* if object next above is free, meld with it.  Afterward
   and regardless, ensure that p->KKKK... q->KKKK */
        q += 2;
        j = WORDAT q;
        if (0==(j & 0x0001))
        {
                k = k + j + 2;
                WORDAT p = k;
                q += j;
                WORDAT q = k;
        }
        else q -=2;

/* if block next below is free, meld with it, too */
        p -= 2;
        j = WORDAT p;
        if (0==(j & 0x0001))
        {
                p -= j;
                k = k + j + 2;
                WORDAT p = k;
                WORDAT q = k;
```

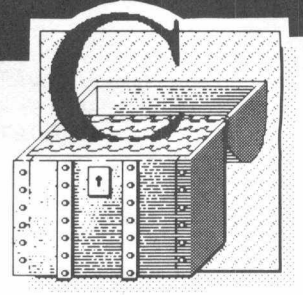
Listing Two

```
        }
}

VOID    objtrim(ob,sz)
        char *ob;
        unsigned sz;
{
/* validate object, ensure call is to reduce it not enlarge it */
        j = objsize(ob);
        if (sz > j) objerr(mtrim,1);

/* ensure new size will be even */
        sz = (sz + 1)&0xfffe;

/* we will create a new, free object in the high end of
           the present one --  its k is k-sz-4. */
        j = k - sz - 4;

/* if that block doesn't amount to 8 bytes of data, forget it */
        if (j <= 10) return;

/* the k of the existing block will become sz+2 */
        k = sz + 2;

/* create the trimmed block, including active bit */
        WORDAT p = k+1;
        p += k;
        WORDAT p = k+1;

/* create the freed block (no active bit) */
        p += 2;
        WORDAT p = j;
        WORDAT q = j;

/* if its upper neighbor is free, meld them */
        q += 2;
```

```
        k = WORDAT q;
        if (0==(k & 0x0001))
        {
                j = j + k + 2;
                WORDAT p = j;
                q += k;
                WORDAT q = j;
        }
}
unsigned objmax()
{
        p = objbase;
        k = 0;

/* scan the whole pool to find the largest free object. */
/* be suspicious, validate all chain links as we go. */
        while (0x0003 != (j = WORDAT p) )
        {
                if (j & 0x0001)
                        j &= 0xfffe;
                else
                        if (k < j) k = j;
                q = p + j;
                if ((q <= p)||(q > objtop))
                        objerr(mpool,3);
                p = q+2;
        }
        return (k-2);
}

VOID    objerr(m,n)
        char *m;
        int n;
{
        fprintf(stderr,"\n\n%s error #%d\n",m,n);
        exit();
}

#ifdef DEBUG

VOID    objprt()
{
        fputs("\n\nmap of allocation space...\n",stderr);

/* display what should be lower sentinel of 0003, and its address */
        p = objbase-2;
        fprintf(stderr,"%04x->%04x\n",p,WORDAT p);
        p +=2;

/* display bounds and WWWW words of each block to upper sentinel */
        while(0x0003 != (j = WORDAT p))
        {
                q = p + (j & 0xfffe);
                k = WORDAT q;
                fprintf(stderr,
                "%04x->%04x ... %04x->%04x\n",
                p, j, q, k);
                if ((q<=p)||(q > objtop)) break;
                p = q+2;
        }

/* display upper sentinel unless chain was broken */
        if (j == 0x0003)
                fprintf(stderr,"%04x->0003\n",p);
        else
                fputs("...stopped for invalid chain\n",stderr);
}
#endif
```

End Listings

C CHEST



Queues and Bit Maps

by Allen Holub

Queues

I've never been happy with a store-bought word processor. I cut my teeth on vi and nroff on a Unix system, and I've yet to find an acceptable commercial version of either that will run under MSDOS.

Because I had an editor that I could live with, not having nroff was my greatest problem. None of the versions of nroff from users groups were acceptable; they invariably were subsets of the real thing, often loaded with bugs, and at least one of them (roff4) was formatted so poorly that it was unmaintainable. The writers of these programs also took considerable liberties with nroff's instruction set.

So, being a masochist, I wrote my own nroff. In the course of this project, I added several useful functions to my library and these functions will turn up in this column from time to time. This month we'll look at the queue manager .

A queue is a FIFO (first in, first out) data structure. The first object to be enqueued is also the first object to be dequeued (unlike a stack where the first object pushed is the last object popped).

The rate at which objects are enqueued can be higher than the rate at which they are dequeued, as long as you never allow the queue to get full. Therefore, you can use queues to implement type-ahead in character I/O routines and the like. The interrupt service routine that talks to the keyboard UART puts characters into the queue, and getc or its system equivalent takes the characters out of the queue. As long as the queue never fills up, getc doesn't have to keep up with the data rate at the UART.

Queues have other uses. In multitasking operating systems, they hold pending messages while a task is busy. In a word processing application (like nroff), they're used for line filling. Input text to nroff is unformatted: there are a random number of words on a line. Nroff collects words until it has enough to fill an entire line, then it prints the collected line. My version of nroff enqueues characters until it has collected one too many words. It then dequeues all but the last word to produce an output line.

Listing One (page 31) contains a package of queue management routines; you need to declare a pointer to a QUEUE somewhere in your program to use them. Like a FILE, a QUEUE is a structure maintained by the queue manager. However, also like a FILE, your program need know nothing about the details of queue management or the contents of this structure. I usually typedef a QUEUE as a character pointer when I use it. Now let's turn to the various routines.

```
typedef char *QUEUE;

QUEUE *makequeue( qsize, objsize )
int qsize, objsize;
```

Makequeue( ) is the equivalent to fopen( ). "Qsize" is the size of the queue in objects (not bytes), and "objsize" is the size of a single object in bytes. You can make a queue of anything; even a queue of structures is permissible. Like fopen( ), makequeue( ) returns a pointer that must be passed to all the other queue management routines. A zero is returned if there isn't enough memory to make the queue.

```
del_queue( qp )
QUEUE *qp;
int empty;
```

Del_queue( ) deletes a queue created with a previous makequeue( ) call. Qp is the pointer returned by that makequeue( ) call. This routine *only* deletes empty queues. If you need to delete a queue that isn't empty, use free(qp); a one is returned if the queue is deleted, and a zero otherwise.

```
enqueue ( obj, qp )
char *obj;
QUEUE *qp;
```

enqueue( ) puts an object into the queue. "Obj" points at the object to be enqueued. Alhough it's declared as a character pointer, obj actually can point at anything. "Qp" is a pointer to the queue itself, as returned from a previous makequeue( ) call.

```
dequeue ( obj, qp )
char *obj;
QUEUE *qp;
```

Dequeue( ) is the opposite of enqueue( ). It takes an object out of a queue pointed to by "qp" and puts that object at the memory location pointed to by "obj."

```
car *show_next( qp )
QUEUE *qp;

int *sp_used ( qp )
QUEUE *qp;

int sp_avail ( qp )
QUEUE *qp;
```

You use these routines for getting information about a queue. Show_next( ) returns a pointer to the object to be dequeued by the next call to dequeue( ), without actually removing the object from the queue. Sp_used( ) returns the number of objects

now in the queue. Sp_avail( ) returns the amount of space left in the queue. The sum of the return values of sp_used( ) and sp_avail( ) is the queue size.

The QUEUE data structure is a block of memory that includes a header describing the queue and the queue itself. The mechanism is similar to that used by alloc( ) and free( ). This structure is shown in Figure 1 (page 27).

"Start" points at the beginning of the queue proper. "Head" and "tail" are the offsets, in objects, from the start of the queue to the first and last valid objects in the queue. "Size" is the maximum number of objects that the queue can hold. "Nobj" is the number of objects currently in the queue. "Objsize" is the size of an object in bytes.

There are actually more things in the header than are, strictly speaking, necessary. For example, you can determine the start of the queue's data area by adding the header size to the queue pointer. Similarly, you can compute the number of objects in the queue from head and tail. However, these extra fields do reduce the amount of computation you need to do. If RAM is tight and ROM isn't, take them out.

Makequeue( ) (Listing One, line 27) allocates space for the queue and initializes the header. The real queue size is the number of bytes required for the queue itself (qsize × objsize) plus the size of the header.

Del_queue( ) (line 43) just calls free to return the memory used by the queue to the free list (if the queue is empty).

Objects are put into the queue with enqueue( ) (line 57). They are enqueued at the tail and dequeued from the head. The address where the object is to go is computed on line 71. Because qp –>start is declared as a character pointer, pointer arithmetic is defeated (if you add 1 to a character pointer, you actually modify it by 1, not so with an integer pointer). The object is transferred into the queue, one byte at a time, on line 75, and the tail pointer is incremented on lines 77-78. If the pointer advances past

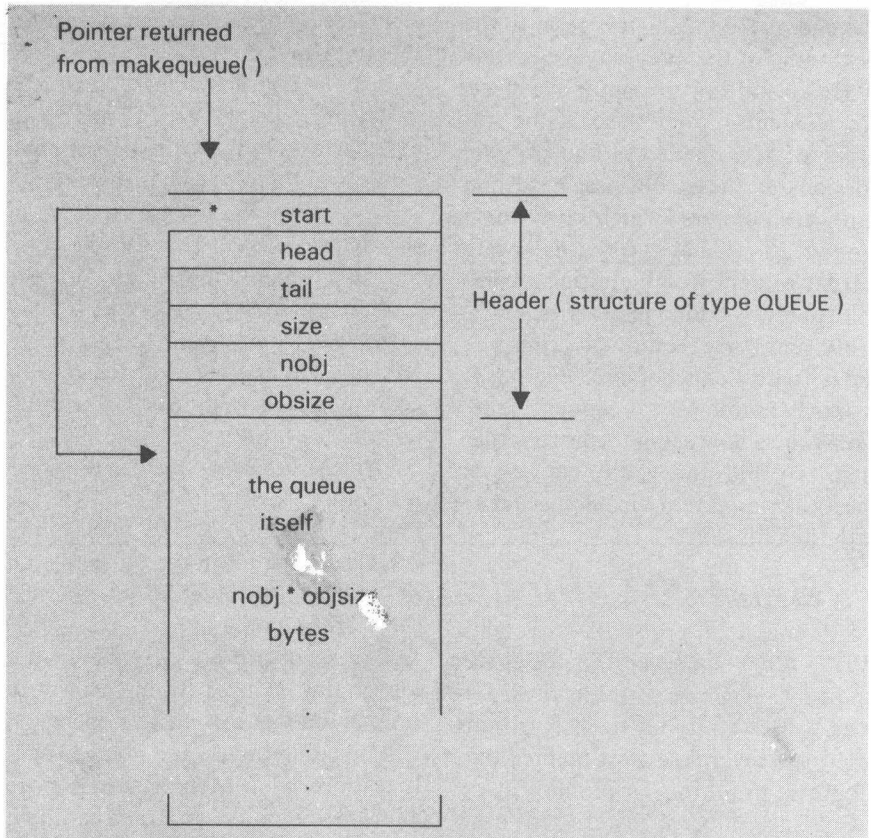


Figure 1
The queue data structure

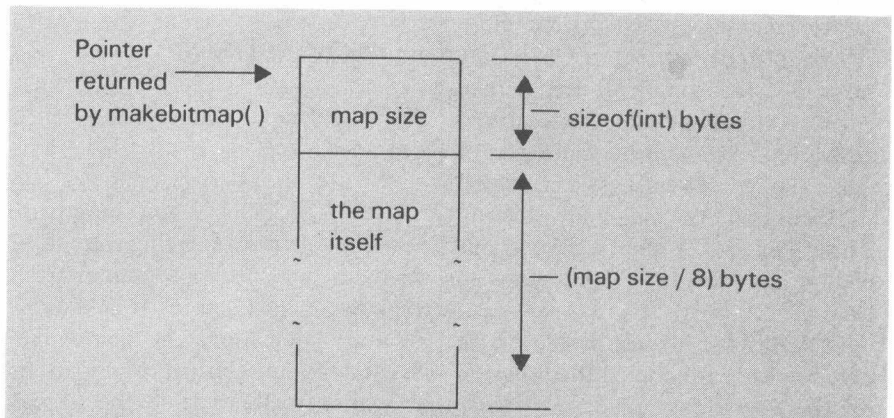


Figure 2
The bit map data structure.

the end of the queue, it is reset to the beginning of the queue.

Head and tail are actually offsets (in elements) from the starting address of the queue. In the original versions of these routines, head and tail were pointers (which saved some computation), but the routines were harder to debug and additional computation was added in other places.

Objects are removed from the queue with dequeue( ) on line 82. It works the same way as enqueue( ) except you're now using the head rather than the tail, and you're copying in the other direction (from the queue into the object).

Bit Maps

My version of grep (*DDJ*, October 1984, #96) had several inefficiencies built into it. Some of these inefficiencies were basic to the algorithms used. As several readers pointed out, a state machine implementation would have been faster. However, changing something this fundamental would have meant throwing away most of the program.

Because the thing did work after all, I wasn't tempted to rewrite it completely. Other of the suggested improvements, however, were more reasonable. In particular, Dave Cortesi suggested using a bit map to take care of character classes; his code, which I have transformed into a set of general purpose routines, is presented in Listing Two (page 36). Listing Three (page 37) shows the various changes needed to add bit maps to grep.

Probably the biggest problem with grep.c is that I modeled the program on the version given in *Software Tools in Pascal*. I've never been able to figure out why Kernighan and Plauger did some things the way they did. Pascal does support a pointer type; why don't they ever use it?

Anyway, the problem of character classes is really a problem of set recognition. You want to define a set of characters that are legally in the character class and then to test for membership in that set. The *Software Tools* version creates a string of characters that are legal in a character class, and then searches that string sequentially when testing for membership.

Because character classes are often pretty big (i.e., [A–Za–z] has 52 elements), you both waste memory and slow down the program if the character you're looking for is at the end of the list. [A–Za–z] requires 31 comparisons to find a lower-case "e," the most common letter in English.

A better solution to the set problem is a bit map in which each bit corresponds to a single ASCII character; that is, bit 0 is a NULL, bit 1 is a control-A, bit 2 is a control-B, bit 65 is an upper-case A, and so forth. So, to put a letter into the set, you set the corresponding bit in the bit map, and to see if a letter is in the set, you test the corresponding bit.

The set operation will take a little longer than it used to, but you need set a bit only once. The test operation will be much faster, and you generally test bits a lot. Memory use is more efficient too; a bit map that can hold all 128 ASCII characters needs only 16 bytes. An equivalent string would require 129. On the other hand, 16 bytes are always required, even in a small character class.

Listing Two contains three routines:

```
typedef char BITMAP;

BITMAP *makebitmap( size )
unsigned size;
```

This will make a bit map with "size" elements, requiring (size/8 + 2) bytes, and return a pointer to it. Because calloc( ) is used to do the memory allocation, the pointer returned by makebitmap( ) can be passed to free( ) to delete the map. A zero is returned if makebitmap( ) can't get enough memory.

```
setbit ( c, map, val )
unsigned c, val;
BITMAP *map;
```

This will set bit "c" of the bit map pointed to by "map" if "val" is not a zero; it will clear the bit if "val" is zero. C must be in the range 0 to size − 1, where "size" is the argument originally given to makebitmap( ). Map is a pointer returned by a previous makebitmap( ) call. A one is returned on success, and a zero if c is out of range.

```
testbit ( c, map )
unsigned c;
BITMAP *map;
```

This will test bit "c" of the bit map pointed to by "map." It returns a one if the bit is set, and a zero if it isn't or if c is out of range.

Listing Two also contains a main( ) routine for testing the three utility routines (and demonstrating how they work). A bit map containing 32 bits is made on line 66. The contents of the map are printed to the screen with the for loop on lines 74 and 75. The bit is actually set on line 80.

The subroutines are relatively straightforward. Makebitmap( ), like makequeue( ), creates a header with the bit map itself appended to it (see Figure 2, page 28). The pointer returned by makebitmap( ) points at this header, which is a single unsigned sized object (containing the number of bits in the map).

Makebitmap( ) figures the number of bytes needed on line 24 with:

```
numbytes = (size >> 3) + ((size
& 0x07) ? 1 : 0);
```

The right shift does a divide by 8, and the conditional adds 1 to the result of the divide if "size" isn't an even multiple of 8. Calloc( ) is called (on line 28) to get the memory so the bit map will be initialized to all zeroes (all bits cleared). The additional sizeof(unsigned) is for the header. A pointer to the map is returned on line 30.

Setbit( ) starts on line 32 of Listing Two. It tests for "c" out of bounds on line 39 ("map" must be cast into a pointer-to-unsigned to get the map size). We *then add sizeof* (unsigned) to map to skip past the map size to the map itself. The bit is set on line 43 or cleared on line 45. Testbit( ) (line 49) works just like setbit( ), the only difference being that it returns the bit's value instead of modifying it.

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 191**.

Listing One

```
#define DEBUG 1

#ifdef DEBUG
#include <stdio.h>
#endif

/*-----------------------------------------------------------------
 *      QUEUE.C:        General purpose queue management routines:
 *
 *     Copyright (c) 1985, Allen I. Holub.  All rights reserved
 *  This program may be copied for personal, non-profit, use only.
 *-----------------------------------------------------------------
 *
 *  The QUEUE data structure. No external routine needs to know anything
 *  about how this structure is put together. These routines need only
 *  remember a pointer to the queue (in a manner similar to the FILE
 *  pointer used by the i/o routines).
 */

typedef struct
{
        char    *start;         /* Pointer to beginning of queue        */
        int     head;           /* Index of current head                */
        int     tail;           /* Index of current tail                */
        int     size;           /* Max num of objects queue can hold    */
        int     nobj;           /* Number of objects now in the queue   */
        int     objsize;        /* Size of one element                  */
} QUEUE;

/*-----------------------------------------------------------------------*/

QUEUE   *makequeue( qsize, objsize )
{
        /* Make a queue of the specified size containing objects of the
         * specified size. Return a pointer to the queue or 0 if there is
         * not enough memory to make the queue. Queues are created using
         * calloc(). They require sizeof(QUEUE) + (qsize * objsize) bytes.
         */

        register QUEUE  *qp;

        if( !(qp = (QUEUE *) malloc(sizeof(QUEUE) + (qsize * objsize)) ))
                return 0;

        qp->start       = (char *)(qp + 1);
        qp->size        = qsize   ;
        qp->objsize     = objsize ;

        qp->head = qp->tail = qp->nobj = 0;

        return( qp );
}
del_queue( qp )
QUEUE   *qp;
{
        /*   Delete a queue and free the memory. The queue will NOT
         *   be deleted unless it is empty. Return 1 if the queue
         *   was deleted, 0 otherwise. If you don't care if the queue
         *   is actually empty, use free(qp).
         */

        if( qp->nobj )
                return 0;

        free( qp );
        return 1;
}
/*-----------------------------------------------------------------------*/
```

(Continued on next page)

Listing One

```
 57: enqueue( obj, qp )
 58: char    *obj ;
 59: QUEUE   *qp  ;
 60: {
 61:         /*      Put an object into the queue. Obj is a pointer to the
 62:          *      object qp is a pointer to a QUEUE. Return 1 on success,
 63:          *      0 if there's no more room in the queue.
 64:          */

 65:         int     i;                              /* Counter              */
 66:         char    *bp;                            /* points into queue    */

 67:         if( qp->nobj  >=  qp->size )            /* If the queue is full */
 68:                 return 0;                       /* return failure.      */

 69:         qp->nobj++;                             /* One more object in   */
 70:                                                 /* the queue            */

 71:         bp= qp->start + (qp->objsize * qp->tail); /* Get target address */
 72:                                                 /* within the queue;    */
 73:                                                 /* then move object     */
 74:                                                 /* into it:             */

 75:         for( i = qp->objsize;  --i >= 0;  *bp++ = *obj++ )
 76:                 ;

 77:         if( ++qp->tail >= qp->size)             /* Wrap around if we've */
 78:                 qp->tail = 0;                   /* gone off the end of  */
 79:                                                 /* the queue.           */
 80:         return 1;
 81: }
 82: dequeue( obj, qp )
 83: char    *obj ;
 84: QUEUE   *qp  ;
 85: {
 86:         /*      Get an object from the queue. Qp is a pointer to a QUEUE,
 87:          *      The dequeued object is copied into the place pointed to by
 88:          *      obj. Return 0 if the queue is empty and no object was
 89:          *      dequeued, 1 otherwise.
 90:          */

 91:         register int    i;
 92:         register char   *bp;
 93:
 94:         if( qp->nobj <= 0 )
 95:                 return 0;                               /* queue empty */

 96:         qp->nobj-- ;

 97:         bp = qp->start + (qp->objsize * qp->head) ;

 98:         for( i = qp->objsize; --i >= 0; *obj++ = *bp++ )
 99:                 ;

100:         if( ++qp->head >= qp->size )
101:                 qp->head = 0;

102:         return 1;
103: }

104: /*-------------------------------------------------------------------------*/
105: /*  Little access routines:                                                */
106: /*      Show_next returns a pointer to the object at the head of the       */
107: /*      queue; sp_used returns the number of objects in the queue          */
108: /*      sp_avail returns the number of slots available in the queue.       */

109: char    *show_next (qp)
110: QUEUE   *qp;
111: {
```

(Continued on page 34)

Listing One

```
112:            return( qp->start + (qp->head * qp->objsize)  );
113: }

114: int        sp_used     (qp)
115: QUEUE      *qp;
116: {
117:            return( qp->nobj );
118: }

119: int        sp_avail    (qp)
120: QUEUE      *qp;
121: {
122:            return( qp->size - qp->nobj );
123: }
124: #ifdef DEBUG

125: main()
126: {
127:            int     num, c, *ip;
128:            QUEUE   *qp;

129:            qp = makequeue( 4, sizeof(int) );

130:            while( 1 )
131:            {
132:                    num = c = -1;

133:                    ip = (int *)qp->start ;

134:                    printf("\n\nqueue: %d %d %d %d\n",ip[0],ip[1],ip[2],ip[3]);
135:
136:                    printf("start     =0x%x\n",     qp->start       );
137:                    printf("head      =%d\n",       qp->head        );
138:                    printf("tail      =%d\n",       qp->tail        );
139:                    printf("size      =%d\n",       qp->size        );
140:                    printf("objsize   =%d\n",       qp->objsize     );
141:                    printf("nobj      =%d\n",       qp->nobj        );

142:                    printf("there are %d slots left in the queue\n\n",
143:                                                        sp_avail(qp) );

144:                    printf("(d/e/q) ->");
145:                    while( c != 'e' && c != 'd' && c != 'q' )
146:                            c = getchar();

147:                    if( c == 'e' )
148:                    {
149:                            printf("enter decimal number ->");
150:                            scanf("%d", &num );
151:                            printf("enqueue(%d) returned %d\n",
152:                                                    num, enqueue(&num,qp) );
153:                    }
154:                    else if( c == 'd' )
155:                    {
156:                            printf( "dequeue returned %d, loaded %d\n",
157:                                                    dequeue( &num, qp ), num );
158:                    }
159:                    else
160:                            break;
161:            }

162:            printf(" deleting queue, queue was %sempty\n",
163:                                            del_queue(qp) ? "" : "not " );
164: }

165: #endif
```

End Listing One

(Listing Two begins on page 36)

 (Listing continued, text begins on page 26)

Listing Two

```
 1: /*      BITMAP.C        makebitmap, setbit, testbit: bit map manipulation
 2:  *                      routines.
 3:  *
 4:  *      Copyright (c) Allen I. Holub, all rights reserved. This program
 5:  *      be copied for personal, non-profit use only.
 6:  */

 7: #ifdef DEBUG
 8: #include <stdio.h>
 9: #endif

10: typedef char BITMAP;

11: /*--------------------------------------------------------------------*/

12: BITMAP    *makebitmap( size )
13: unsigned  size;
14: {
15:         /*      Make a bit map with "size" bits. The first entry in
16:          *      the map is an  unsigned int representing the maximum
17:          *      bit. The map itself is concatenated to this integer.
18:          *      Return a pointer to the map on success, 0 if there's
19:          *      not enough memory.
20:          */
21:
22:         unsigned *map, numbytes;
23:
24:         numbytes = (size >> 3) + ((size & 0x07) ? 1 : 0 ) ;

25: #ifdef DEBUG
26:         printf("Making a %d bit map (%d bytes required)\n", size, numbytes);
27: #endif

28:         if( map = (unsigned *) calloc( numbytes + sizeof(unsigned) ,1 ) )
29:                 *map = size;

30:         return (BITMAP *) map;
31: }
32: setbit( c, map, val )
33: unsigned  c, val;
34: char      *map;
35: {
36:         /*   Set bit c in the map to val.
37:          *   If c > map size, 0 is returned, else 1 is returned.
38:          */

39:         if(  c >= *(unsigned *)map )            /* if c >= map size */
40:                 return 0;

41:         map += sizeof(unsigned);                /* Skip past size   */

42:         if( val )
43:                 map[c >> 3]  |=    1 << (c & 0x07) ;
44:         else
45:                 map[c >> 3]  &=  ~(1 << (c & 0x07)) ;

46:         return( 1 );
47: }

48: /* ------------------------------------------------------------------- */


49: testbit( c, map )
50: unsigned c;
51: char     *map;
52: {
53:         /*   Return 1 if the bit corresponding to c in map is set.
54:          *   0 if it is not.
```

```
55:          */

56:         if( c >= *(unsigned *)map )
57:                 return 0;

58:         map += sizeof(unsigned);

59:         return(  map[ c >> 3 ]  &  (1 << (c & 0x07)) );
60: }
61: #ifdef DEBUG

62: main()
63: {
64:         int     bitnum, set, i, *map;

65:         printf("Making a 32 bit wide bit map\n");

66:         if( !(map = makebitmap( 32 )) )
67:                 printf("Can't make map\n");

68:         while( 1 )
69:         {
70:                 /* Print the bit map. Try to print past the end of the
71:                  * map to make sure overflow detection works (bit 32 should
72:                  * come back as a 0).
73:                  */

74:                 for( i = 0; i <= 32 ; i++ )
75:                         putchar( testbit( i, map ) ? 'X' : '.' );

76:                 printf("\n\nBit number :");
77:                 scanf("%d", &bitnum );
78:                 printf("\nl to set, 0 to clear: ");
79:                 scanf("%d", &set );

80:                 if( ! setbit(bitnum, map, set) )
81:                         printf("Bit out of range\n");
82:         }
83: }

84: #endif
```

End Listing Two

Listing Three

```
 1: /*-------------------------------------------------------------------+
 2:  * GREP:  Changes needed to add bit maps to character classes:       |
 3:  *-------------------------------------------------------------------+
 4:  */

 5: typedef struct token                              /* In tools.h */
 6: {
 7:         char            tok;
 8:         char            lchar;
 9:         char            *bitmap;
10:         struct token    *next;
11: }TOKEN;

12: /*-------------------------------------------------------------------------*/

13: TOKEN   *makepat(arg, delim)                      /* In tools.c   */
14: char    *arg;
15: int     delim;
16: {
17:                                 *
18:                                 *
19:                                 *
20:                 case CCL:

21:                         if (*(arg+1) == NEGATE)
22:                         {
23:                                 ntok->tok = NCCL;
24:                                 arg += 2;
25:                         }
```

(Continued on next page)

Listing Three

```
26:                             else
27:                             {
28:                                     ntok->tok = CCL;
29:                                     arg++;
30:                             }

31:                             if( ntok->bitmap = makebitmap(128) )
32:                                     arg = dodash(CCLEND, arg, ntok->bitmap );
33:                             else
34:                             {
35:                                     fprintf(stderr,"Not enough memory for pat\n");
36:                                     error = 1;
37:                             }
38:                             break;

39:                                     *
40:                                     *
41:                                     *
42: }
43: char    *dodash( delim, src, map )              /* In tools.c   */
44: int     delim;
45: char    *src, *map;
46: {
47:         register int    first, last;
48:         char            *start;

49:         start = src;

50:         while( *src  &&  *src != delim )
51:         {
52:                 if( *src != '-')
53:                         setbit( esc( &src ), map, 1 );

54:                 else if( src == start  ||  *(src+1) == delim )
55:                         setbit( '-', map, 1 );
56:                 else
57:                 {
58:                         src++;

59:                         if( *src < *(src - 2) )
60:                         {
61:                                 first = *src;
62:                                 last  = *(src-2);
63:                         }
64:                         else
65:                         {
66:                                 first = *(src - 2);
67:                                 last  = *src;
68:                         }

69:                         while( ++first <= last )
70:                                 setbit( first, map, 1);

71:                         src++;
72:                 }
73:         }

74:         return( src );
75: }
76: int     omatch (linp, pat, boln)                /* In tools.c   */
77: char    **linp, *boln;
78: TOKEN   *pat;
79: {
80:                                 *
81:                                 *
82:                                 *

83:                 case CCL:
84:                         if( testbit( **linp, pat->bitmap ) )
```

(Continued on page 40)

Listing Three

```
 85:                                    advance = 1;
 86:                            break;

 87:                    case NCCL:
 88:                            if( !testbit( **linp, pat->bitmap) )
 89:                                    advance = 1;
 90:                            break;

 91:                                    *
 92:                                    *
 93:                                    *
 94: }

 95: /*--------------------------------------------------------------------------*/

 96: pr_tok( head )                                        /* In tools.c   */
 97: TOKEN   *head;
 98: {                                  *
 99:                                    *
100:                                    *

101:                    if (head->tok == CCL || head->tok == NCCL)
102:                    {
103:                            printf("string (at 0x%x) =<", head->bitmap );

104:                            for( i = 0; i < 0x7f ; i++)
105:                                    if( testbit(i, head->bitmap) )
106:                                            putchar(i);

107:                            printf(">, ");
108:                    }

109:                                    *
110:                                    *
111:                                    *
112: }

113: unmakepat(head)                                        /* In tools.c   */
114: TOKEN   *head;
115: {
116:            register TOKEN  *old_head;

117:            while (head)
118:            {
119:                    switch (head->tok)
120:                    {
121:                    case CCL:
122:                    case NCCL:
123:                            free(head->bitmap);
124:                            /* no break, fall through to default */

125:                    default:
126:                            old_head = head;
127:                            head = head->next;
128:                            free(old_head);
129:                            break;
130:                    }
131:            }
132: }
```

End Listings

by Axel Schreiner

This column is reprinted from the German quarterly unix/mail *(Hanser Verlag, Munich, Germany).* ©1984 *by Axel T. Schreiner, Ulm, West Germany. It may be reproduced as long as the copyright notice is included and reference is made to the original publication.*

/lib/cpp

This column deals with uses and abuses of the C preprocessor. It demonstrates some techniques that can save a lot of work and even more errors. The discussion applies to programming in C in general and makes only very elementary assumptions:

> C programs are run through a preprocessor *before* they are handed to the actual compiler. The preprocessor performs (parameterized) text substitution (#define), inserts header files (#include), and can exclude parts of the source from compilation (#if).

Because the preprocessor is independent of the actual compiler—and does not know C at all—one can use it in particular to extend the C language. Only taste limits one's imagination here . . . .

Excluding Text

Every programmer writes occasional comments. Sometimes we comment to exclude program parts from a compilation. Because in Standard C comments may not be nested, there is considerable temptation not to comment such excluded program parts anymore.

The following technique for text exclusion is much more appropriate:

```
#ifdef not_defined
   crash_the_system(NOW);
   /* this definitely goes wrong */
#endif not_defined
```

Of course, the name not_defined should really not be defined.

Vector Dimensions

In principle, one can determine the size of a vector by using the sizeof operator. However, sizeof yields the size in bytes, not in elements. The following macro determines the number of elements in an arbitrary vector:

```
#define DIM(x) (sizeof (x) / sizeof ((x)[0]))
```

sizeof does not really need parentheses, if it is used to determine the size of an object and not of a data type. One should, however, enclose macro parameters in parentheses. Then things work out for a vector with more than one dimension, too:

```
main( )
{
      struct { int a; char b; } v[10] [20] [30];
      printf("%d %d %d\n", DIM(v), DIM(v[1]),
            DIM(v[1] [2]));
}
```

The program produces the values 10, 20, and 30.

Parentheses should not be necessary in this use of sizeof because a vector subscript should have precedence over sizeof. However, my copy of the Mark Williams CP/M-86 C compiler does not seem to know this.

We can carry these ideas somewhat further. The last element of a vector is

```
#define LAST(x) ((x)[DIM(x)-1])
```

and the customary for loop is, for example:

```
#define END(x) ((x) + DIM(x)-1)

int vector[10], * vp;
   . . .
   for (vp = vector; vp <= END(vector); ++ vp)
      . . .
```

The compiler evaluates sizeof during its evaluation of constant expressions. This circumstance can be used to determine the length of constant strings in an efficient and flexible fashion:

```
#define STRLEN(s) (sizeof (s) - 1)

char buf[STRLEN("model") + 1];
   . . .
   strcpy(buf, "model");
```

There is the danger, however, that STRLEN is used for other objects (i.e., non-strings) by mistaking it for strlen.

Trace

It is well known that a *macro call* is not recognized in a

constant string. Less well known, but more useful perhaps, is that a *macro parameter* is recognized and replaced within the replacement text of a macro definition. Rather than

```
printf("variable = %d\n", variable);
printf("formula = %f\n", formula);
```

we write

```
#define SHOW(val,fmt) fprintf(stderr,\
        "SHOW: val = fmt\n", val)

        SHOW(variable, %d);
        SHOW(formula, %f);
```

The latter is easier to use and conveys more information because val is replaced in the format by the entire macro argument.

A bit of caution is required: if the % operator is used within val, there will be problems with the format. This can be corrected as follows:

```
#define SHOW(val, fmt)  fprintf(stderr, "%s\
   = fmt\n", "val", val)
```

A macro can be defined without a replacement text. Uses of SHOW thus can be eliminated easily from the compiled program altogether. Alternatively, we can specify a condition:

```
#ifdef   DEBUG
         char debugflag;
#        define SHOW(val,fmt) (debugflag &&\
               fprintf( . . . ))
#else    ! DEBUG
#        define SHOW(val,fmt) /* null */
#endif   DEBUG
```

In this example, SHOW is always used as a statement and not as an expression. Using && rather than if has two advantages: we do not *have* to use SHOW as a statement, and use of SHOW does not invite an unintentional else. debugflag, by the way, should be used as a bit vector, e.g.:

```
#define SHOW(level, val, fmt) (debugflag &\
   1<<level && fprintf( . . . ))
```

Now we can maintain different sets of trace information at levels 0 through 7.

Global Variables

Do you like modular programs with lots of sources, makefile, a central header file, and the (feeble) hope that all global declarations really match? Do you like to lint, too? The following technique simplifies maintaining global variables. The central header file contains about the following:

```
#ifndef   GLOBAL
#         define GLOBAL extern
#endif    GLOBAL

GLOBAL  int   global_variable;
```

If GLOBAL is not defined, a variable declared GLOBAL is declared extern.

Within exactly *one* of the source files that include the header file, we have to take care that the variables that were declared extern elsewhere are really defined. In the main source file, we therefore write:

```
#define GLOBAL /* to define global variables */
#include "definitions.h"
```

One can even initialize global variables in this context *without* resorting to the −m flag instructing the loader ld to accept multiple definitions:

```
#ifdef    GLOBAL
#         define     INIT(x) = x
#else     ! GLOBAL
#         define     GLOBAL extern
#         define     INIT(x)
#endif    GLOBAL

GLOBAL  int variable INIT(10);
```

This technique is not very practical for aggregates. The following variant is easier to use:

```
#ifdef    GLOBAL
#         define     INIT(x) =x
#         define     GINIT
#else     ! GLOBAL
#         define     GLOBAL extern
#         define     INIT(x)   ;
#         undef      GINIT
#endif    GLOBAL

GLOBAL  struct { int a; char b; } variable INIT( )
#ifdef    GINIT
                     { 10, 'b' };
#endif    GINIT
```

This method requires that the C preprocessor permit a macro call with an empty argument list and that the C compiler not complain about superfluous semicolons between global declarations. This method is admittedly no longer very elegant, but it has the significant advantage that the text of central definitions exists only once in all cases.

/bin/lex

Now you see it . . .

lex programs have lots in common with fashions: the effect is not always what the pattern promises. If a function generated by lex is used as a front end for a parser generated by yacc, it is sometimes very hard to decide where to place the blame for a bug: is there a bug in the grammar presented to yacc, or are the patterns that were processed by lex at fault?

The following technique[1] permits the construction of a source file for lex that is conditionalized so that a debugging version can be compiled at any time without any changes to the source. To test the results of lex, all inputs that the parser is to receive later are first presented to the debugging version. This version of the front end then prints a mnemonic version of the values that the parser would receive:

```
%{
#ifdef  TRACE

#       include "assert.h"

        main( )
        {
                char *cp;
                assert(sizeof(int) >=
                   sizeof(char *));
                while (cp = (char *) yylex( ))
                   printf("%-.10s is \"%s\"\n",
                      cp, yytext);
        }

#       define token(x)   (int) "x"

#else   ! TRACE

#       include "y.tab.h"
#       define token(x)   x

#endif  TRACE
%}
```

Normally, TRACE is undefined, and the tokens (i.e., the values that are to be returned to the parser) are defined in the file y.tab.h generated by yacc as:

```
#define NAME     257
        . . .
```

These defined names are used directly in the source presented to lex and are returned as a result of the function yylex( ).

If TRACE is defined, y.tab.h need not yet exist. In this case (i.e., in the debugging version), we want to return a string as a result of yylex( ) that is then printed by the main( ) program included here.[2] Analyzing the debugging output is most easily accomplished if the output uses exactly those words that later will appear in y.tab.h—i.e., that are a result of %token statements in the source presented to yacc.

We are using the fact that macro parameters are replaced within strings in the replacement text of a macro. token(x) returns either x itself (to be passed on to yacc) or

a string "x" for the purposes of TRACE.

The remainder of the lex program is now quite obvious:

```
%%

[0-9]+                  return token(NUMBER);
[a-z_A-Z][a-z_A-Z0-9]*  return word( );
[ \t\n]+                ;
                        return token(yytext[0]);

%%

struct reserved { char * text; int yylex; } reserved[ ] = {
    { "begin", token(BEGIN) },
    { "end", token(END) },
    (char *) 0 };
int word( )
{

    struct reserved *rp;
    for (rp = reserved; rp->text; ++ rp)
        if (strcmp(yytext, rp->text) == 0)
            return rp->yylex;
    return token (NAME);
}
```

Yes, there should have been a binary chopped search, but we are dealing only with the principles.

/usr/src/main.c

Argument Standards

Command arguments are always good for surprises. Sometimes several options may be combined into one argument; sometimes each option must be a separate argument; sometimes a parameter value follows as part of the argument; sometimes it does not; sometimes all of the above; sometimes some of the above . . . .

If one consults the sources of certain Unix utilities, one learns to appreciate the flexibility of C (or the infinite patience of the C compiler). Everybody does his or her own thing, and most do it differently in every program! However, it would be so simple to develop a standard as in Listing One, page 48).

At show( ), argc contains the number of arguments that have not yet been processed, and *argv is the first one of these. This argument can be a single – character; in some ancient (cat) and almost new (tar) utilities, this indicates that standard input or output is to be used in place of a file argument.

Flags can be combined at will. If an option requires a value, it can follow immediately or it can be an argument of its own.

Following a standard proposed in the "USENIX login," an option – – serves to terminate processing of the option list. Apart from that, options must start with – and must precede other arguments. These rules, however, still do not cover all possibilities of pr.

The skeleton above is useful but anatomically somewhat terrifying. The following incarnation is perhaps more attractive:

```
#include <stdio.h>
#include "main.h"

#define show(x) printf("x = %d\n", x)
#define USAGE fputs("cmd [-f] [-v  #]\n",\
   stderr ), exit(1)

MAIN
{
      int f = 0, v = 0;

      OPT
            ARG 'f':
                  ++ f;
            ARG 'v':
                  PARM
                  v = atoi(*argv);
                  NEXTOPT
            OTHER
                  USAGE;
      ENDOPT
show(f), show(v), show(argc);
if (argc)
      puts(*argv);
}
```

The trick, of course, is concealed in the header file main.h: here the macros OPT, ARG, PARM, NEXTOPT, OTHER, and ENDOPT must be defined using exactly those texts that were given explicitly in the previous example, as in Listing Two, (page 50).

The definitions are not exactly beautiful, especially if they need to be compacted so that the C preprocessor accepts the lengthy replacement texts, but they need to be developed only once to make the argument standard available for all applications. An application then is almost self-documenting:

MAIN
is the function header of the main program.

OPT
starts the loop during which the options are processed.

ENDOPT
completes this loop.

ARG
within the loop starts the processing of one option; the name of the option (a single character) enclosed in single quotes and a colon must follow.

PARM
follows the option specification if the option has a value parameter; the parameter itself is then available as *argv.

NEXTOPT
is used once such a parameter has been processed to advance to the next command argument.

OTHER
must follow all options; following this, one specifies what should be done if an option could not be recognized. NEXTOPT may be specified in this case, too. The unknown option itself is **argv.

After the OPT ENDOPT loop, argc contains the number of command arguments that have not yet been processed, and *argv is the first such argument. Arbitrarily many (different) options ARG can be specified. pr would be implemented approximately as follows:

```
MAIN
{
    do
    {
        OPT
            ARG 'h':
                PARM
                header = *argv;
                NEXTOPT
            ARG 'w':
                PARM
                width = atoi(*argv);
                NEXTOPT
            ARG 'l':
                PARM
                length = atoi(*argv);
                NEXTOPT
            ARG 't':
                tflag = 1;
            ARG 's':
                PARM
                delimiter = **argv;
                ++*argv;
                NEXTOPT
            ARG 'm':
                mflag = 1;
            OTHER
                if (isdigit(**argv))
                    columns = atoi(*argv),
                        NEXTOPT
                else
                    USAGE, exit(1);
        ENDOPT

        if (argc)
        {
            if (**argv == '+')
            {
                PARM
                first_page = atoi(*argv);
                continue;
            }

            dopr(*argv);
        }
        else
            dopr("-");
    } while (argc > 1);
}
```

There is a blemish: – columns must be secified as a single argument (because – alone refers to standard input).

Notes

[1] This technique was developed for the book *Introduction to Compiler Construction* by A. T. Schreiner and H. G. Friedman, Jr., Prentice-Hall, 1985.

[2] The technique requires that a pointer to a character string be returned in place of an int value. This is not possible across all implementations of C; e.g., it is probably not allowed on the 7300 systems. We guard against a portability problem by using assert( ). **DDJ**

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 192**.

```
#include <stdio.h>

#define show(x) printf("x = %d\n", x)
#define USAGE fputs("cmd [-f] [-v  #]\n", stderr), exit(1)

main(argc, argv)
int argc;
char **argv;
{
    int f = 0, v = 0;

    while (--argc > 0 && **++argv == '-')
    {
        switch (*++*argv)
        {
            case 0:                                    /* - */
```

Listing One

(Continued on page 50)

(Continued from page 48)

```
                    --*argv;
                    break;
                case '-':
                    if (! (*argv) [1])                    /* -- */
                    {
                        ++ argv, -- argc;
                        break;
                    }
                default:
                    do
                    {
                        switch (**argv)
                        {
                            case 'f':                     /* -f */
                                ++ f;
                                continue;
                            case 'v':
                                if (*++*argv)
                                    ;                     /* -v# */
                                else if (--argc > 0)
                                    ++argv;               /* -v  # */
                                else
                                    break;
                                v = atoi(*argv);
                                *argv += strlen(*argv) -1;
                                continue;
                        }
                        USAGE;
                    } while (*++*argv);
                    continue;
            }
            break;
        }
        show(f), show(v), show(argc);
        if (argc)
            puts(*argv);
    }
```

Listing One

```
#define MAIN    main(argc, argv)                    \
                int argc;                           \
                char ** argv;
#define OPT     while (--argc > 0 && \ **++argv     \
                    == '-')                         \
                {                                   \
                    switch (*++*argv)               \
                    {                               \
                        case 0:                     \
                            --*argv;                \
                            break;                  \
                        case '-':                   \
                            if (! (*argv) [1] )     \
                            {                       \
                                ++ argv, -- argc;\
                                break;              \
                            }                       \
                        default:                    \
                            do                      \
                            {                       \
                                switch (**argv)     \
                                {
#define ARG     continue;                           \
        case
#define OTHER   continue;                           \
            }
#define ENDOPT  } while(*++*argv);                  \
                continue;                           \
            }                                       \
            break;                                  \
        }

#define PARM    if (*++*argv)                       \
                    ;                               \
                else if (--argc > 0 )               \
                    ++argv;                         \
                else                                \
                    break;

#define NEXTOPT *argv += strlen(*argv)-1;
```

Listing Two

Information Age Issues

by Dean Gengle

The real issues in telecommunications dwarf questions about protocol and baud rate. This is an essay on responsibility.

My first experiences with a modem, a microcomputer, a television set, and a telephone linked together—with me in the loop—were much like my first discoveries regarding the true significance of portions of my own anatomy. These experiences combined the joys of intellectual insight, emotional pleasure, physical involvement, and all the wonder we associate with the notion of "surprise."

On-Line Life

I conducted my early experiments in on-line living with no misgivings at all. I was able to consider only the positive possibilities contained in this new technical capability. Modemocracy! Distributed networks! Grass roots anticipatory guidance systems! It all seemed within reach if only everyone would use these tools. The sheer weight of the new experiences themselves bolstered my initial faith that folks would come to use these tools in the "right" way.

I remember distinctly the moment when the inevitable mental disappointment occurred and reality set in—about six weeks after my first modem fix (ca 1980). I was logged on a local bulletin board system (BBS). I had just finished entering a private reply to a particular message; it was meant to go directly to another user of the system. On my screen flashed "Hi." Weird response. Commands issued to the remote computer no longer worked. Instead, the system operator (sysop) had taken over and was typing messages at me directly.

"I've been watching your session," he wrote, in the most casual and matter-of-fact manner. "Thought I'd say hello." In that instant, the scales fell from my eyes, so to speak. My illusion of privacy was shattered, an illusion maintained without my awareness by a number of supporting factors. I was working out of my own apartment, a bedroom with a spare corner space, in fact. What could be more personal or private? Also, I've never had any reason to think that my phone was tapped, so I've always regarded phone conversations as "private" My modem is hooked to the phone, ergo . . . . Besides, the remote computer system itself reinforced my illusion by allowing "private" messages between individuals and by requiring a password to read "mail."

All along, that sysop might have been looking over my shoulder, reading what I typed, aware of when and how I used the system. The implications of that made me decidedly uncomfortable. Since then, I have become aware of other sysops who've turned their computers into dictatorchips. They not only eavesdrop at will but run their systems with the firm style of a Khomeni. (Confess or I pull out your passwords!) Large systems and small are run this way. The truly astounding thing is that some people seem to *like* that sort of relationship. Still, I hope, there numbers are not great. Not here. Not in the U.S. of A.

Dean Gengle, 1150 Bryant St., San Francisco, CA 94103 (415) 861-8733.

Info Liberties

The net result of this is that I've been led, willy-nilly, into what has become an ongoing interest in civil liberties and technology. I have found that specific design decisions of an apparently "purely technical" nature can and often do have an effect on what we consider to be our personal liberties and freedoms. Conversely, social values can affect what and how new technology is designed.

The controversies that arise from this interplay taken together can be called *information age issues*. They all focus on the handling of information: who generates it; who can store it; who can ship it around; who can have it; who can control it; who can copy it and who cannot; what is and is not legitimate information in a given instance—hence, "information age."

It is becoming increasingly difficult to draw a firm line between the idea of telecommunications and the idea of computing. Some commentators have coined terms such as *telecomputing* and *compunications* to make this point. (I will spare the reader any further use of these, however.) Distinctions between media are blurring fast. All kinds of information now can take digital form to be zapped across various channels, whether phone lines, fiber optic cables, cellular networks, or satellite transponders.

For the time being, I find it useful to think of *telecommunications* as any form of electronic transmission requiring temporary or permanent physical connections.

The examples that follow are representative of the dilemmas we already confront. I can only hint at telecommunications-related conundrums (or nightmares) we may have to face in the months and years directly ahead.

Freedom from Eavesdropping

The kind of surveillance conducted by the sysop I described earlier is not the only kind that new technologies have made possible. Signals of all kinds are now being intercepted on a widespread scale. On the governmental level, spy agencies around the world engage in SIGINT (signal intelligence) work:

- Radio and television broadcasts, voice and data phone calls, satellite channels, international telex and TWX and Western Union transmissions, and amateur radio broadcasts are all monitored. The superpowers conduct the most sophisticated monitoring, as one would expect, but the technology of signal interception is getting cheaper every day.
- Telephone company officials, FBI agents, and credit card companies go on "fishing expeditions" in local microcomputer-based message systems. Ostensibly, they are looking for exchanges of "illegal" data, such as computer access numbers and passwords, phone credit card numbers, and so on. Who knows what other subversive material these official intruders might turn up?
- Although laws surrounding the tapping of voice telephone calls are stringent when they deal with what may and may not be used as trial evidence or when warrants are required and when they are not, no such laws cover eavesdropping on private data transmissions. With large computers doing the listening and sifting, it is a simple matter to monitor thousands of data phone calls.

Meanwhile, in the commercial sector, cellular mobile telephone systems are being marketed heavily in major metropolitan areas across the country. However, cellular channels can be monitored with easily built "scanners" similar to those used to monitor police and CB channels.

Thus, as the cellular service suppliers themselves admit, those top level business calls we make from the middle of traffic jams may not be private. The service suppliers downplay this aspect of cellular mobile telecommunications. After all, there are 666 assigned channels, and the channel assignment for a particular call *does* shift as the user moves from cell to cell. But even if it isn't technically feasible to build a scanner capable of following a particular call, you probably shouldn't conduct sensitive business by cellular phone anyway, according to the suppliers. So much for productivity in traffic jams.

A few companies already are planning to connect modems and micros to cellular telephones. Get your electronic mail from anywhere, anytime. There is no reason to think this evolution will stop until someone makes a device that is a cellular phone, voice and data terminal, and microcomputer, all in one portable package. The temptation to use such a device will grow right along with the temptation to listen in.

"Freedom from eavesdropping" is more apt as a rallying cry than "right to privacy" because certain forms of eavesdropping do not intrude directly on our privacy. For example, others can gather and analyze information about your phone calls even if they do not monitor directly the content of the calls themselves. Within many corporate settings, this already is being done.

Information about whom you call, when you call, the duration of the call, and the frequency of calls to particular numbers all can be collected without your knowledge. To some extent, the same laws covering voice phone tapping hedge the ability of police and government to use such information. However, no such laws cover intraorganizational phone monitoring. Monitoring systems are marketed under the rubric of "phone management" and "automatic call distribution and reporting systems."

One company advertisement for this kind of system is a touch Orwellian in its copy: "To check out your staff's performance, you can either walk around the office. Or get one of [our] automatic call distribution systems. They actually print out detailed reports on every single person's hourly performance: Who needs a raise. And who needs improvement."

Freedom to Play vs. the Drive for Security

Given the complexity of our society, we are far more likely to lose our valued freedoms bit by bit than all at once. Freedoms also are more likely to erode faster as a result of nongovernmental institutional decisions.

As technicians, software designers, hardware designers, and computer users, we have as much to lose through the misapplication of technology as anyone else. We are

also, perhaps, in a better position to prevent misapplications than anyone else.

There is much attention paid these days to the ubiquitous "hacker," whoever that may be. The hacker is linked in the public mind with computer crime. This was not always so, although the reasons for this association are many. Yet, whether one thinks of oneself as a hacker or not, the solutions to the problems of computer crime don't seem to lie in the direction of hiring more consultants. Nor does it make much sense to try to rehabilitate the image of the hacker by making him or her into some sort of cultural hero(ine).

Along with the hacker image, the public is being sold a bill of goods concerning computer crime. Such crime, by and large, is portrayed as something an individual does to an organization, using a microcomputer. The public sees computer crime as "breaking and entering" large systems, whether to take money gains, information gains, or both. But other kinds of computer abuse are not labelled as "crimes" or "criminal" and, for the most part, are not even mentioned in the mass media that shape public consciousness on such matters. The daily papers do not discuss the idea that data eavesdropping by government, corporate, or even private individuals on private citizens may also be criminal. Nor does the idea that the IRS or NSA may be tapping the communications and computers of private individuals receive much consideration. The laws that are discussed and passed are laws aimed at individual abuse, not large-scale institutionalized abuse.

Often, the quest for security tramples our rights to privacy and individuality. In the drive for absolute security in an absolutely efficient society, we may lose the one thing we need in order to survive and prosper in the global information economy: the freedom to play and its attendant creativity.

No matter what the end product, creative (and therefore innovative) ideas are nurtured in an atmosphere of play. The balance between control and freedom must be struck in actual design decisions we make from day to day. The impact of one implementation vs. another on our overall freedoms must be part of design decisions.

High on the list of needed design developments in an era of mass telecommunications is some sort of public key encryption and signature system that is mass-producible and inexpensive and secure from any kind of eavesdropping. The jury still seems to be out on whether or not the DES (Data Encryption Standard) is really secure from megacomputer cracking power. This is an area where some effective designs could improve all our on-line lives, but it is only one example.

The Future of Paranoia

Future problems in telecommunications may make today's problems seem relatively trivial by comparison. As expert systems take up residence in our telecommunications networks, designers will have to become even more concerned with individual freedoms—constitutional and civil liberties—than they are now.

For example, it is feasible to expect that various information suppliers, data banks, and mass information utilities will be able to learn about users in sophisticated ways: what your interests are, what items to call to your attention based on your past use of the system, what you read and what you ignore, what kind of equipment you have, your age, marital status, and economic position, and so forth.

It would be nice to interact with a system that appears custom tailored to your needs and use. However, it is not a big leap to go from information about how you've interacted with, say, a "smart Source" or "intelligent CompuServe" to a psychological and/or political profile. This sort of information analysis is good not only for marketers who want to sell you things but for politicians, power junkies, and competitors who want to know all they can about you for reasons of their own.

Approaches to Action

People directly involved in making technical decisions for design or implementation are, as I said earlier, in a good position to act as *ad hoc* advisers to the general public, the media, and lawmakers at local, state, and federal levels. Technical people, along with artists, can act as "antennae for the race" in matters of sociotechnical importance. If you are involved in the early design stages of an expert software system to correlate IRS data with SSI information, to take one hypothetical example, you'll know before the rest of us do how that system can be abused and put to uses never intended in the first place.

We've all heard that the price of liberty is eternal vigilance. But just how one remains vigilant in an age of rapid technological change is not so obvious. Responses to upcoming issues can take a variety of forms, whether individual, institutional, or governmental. It has become an article of faith for me, having studied and worked in this arena for several years, that individual responses are preferable to institutional ones and institutional ones preferable to governmental intervention. However, we will need all these approaches.

Individual Responses

Individual technicians and designers play a large role in the creation of technical standards and on standards bodies. In telecommunications, for instance, global standards for videotext, viewdata, and teletext systems have yet to emerge. Standards can affect the kinds of competition possible and hence the marketplace. It is not unheard of for standards to be adopted, not on their merit, but for overtly political reasons.

More individual participation in the creation of new standards, therefore, is one way in which technicians and designers can respond.

Educating our friends and neighbors who are not technically inclined is another way to respond to the changes raining down. Explicit information on the connections between technical possibilities and quality of life is needed on a wider scale than ever before.

Organizational Responses

When biotechnology began to hint that doctors and biological technicians could have a profound influence on the

direction and quality of human life, medical schools and other establishments of higher learning began to teach formal courses in medical and biological ethics.

Today, when computer technology has had—is having—an equally profound effect on our quality of life, almost none of the establishments responsible for technical education are providing courses equivalent to those taken by medical students. A few formal studies of "technical ethics" and "information age ethics" are being conducted, but these are one-shot projects, not long-term educational efforts. Therefore, one possible institutional response to information age issues is to provide courses that combine the study of ethics and morals with the study of information science and the future of telecommunications.

Legal Responses

It's important that, in the discussion of what is and is not "computer crime," we not let high-priced "security consultants" or politicians with their own axes to grind make our laws for us.

It is especially important that we not let the rare occurence of a "hacker break-in" distract us from the larger and potentially more serious issues at hand. The Privacy Act and the Freedom of Information Act, for example, have suffered consistent erosion over the last 10 years by administrative decree and bureaucratic red tape.

As more government information is put on line and made accessible by computer, no one is talking about our rights to access information that is *not* private and that was collected legally at taxpayers' expense. Census data and Library of Congress information come to mind. (You can get census data now, certainly, if you *have certain* kinds of computers and are willing to pay a lot for it.) The government is probably the largest single source of useful information in this country. Making that information available to the micro-using public is an issue worthy of Congressional attention.

Last Words

I don't have any. I am willing to bet that some of the people who read this piece are engaged, even now, in debates on design that will come to wider attention only after a particular piece of outrageous hardware or software has been manufactured and distributed. Even now, you probably can think of a dozen or so issues I've not been able to cover here because of my own ignorance or lack of space.

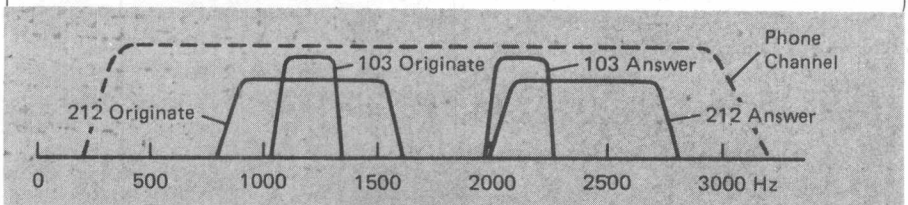
In this business of telecommunications, there are no last words. Therefore, for all our sakes, I urge you to add yours to the heap.

Dean Gengle is a software documentation specialist and consultant at CommuniTree Group in San Francisco. He is the author of The Netweaver's Sourcebook. **DDJ**

Modems: 2400 Bit/Sec and Beyond

by Dale Walsh

Electronic data communication, in its simplest form, employs binary signaling common in logic and microprocessor circuits. A "high" signal is one, and a "low" signal is zero. We have seen dramatic increases in how fast electronic logic devices can switch binary logic levels. Each new computer, it seems, is several times faster than its predecessor. This is achieved primarily by reducing large-scale integration (LSI) devices in size to increase device speeds. We in the communications field would say "they kept binary signaling but increased signal bandwidth."



SIGNAL BANDWIDTHS
Figure

We usually associate bandwidth with frequency limits: our hearing bandwidth is from 20 Hz to 12,000 Hz, for example. But bandwidth also limits how fast things can move. Run in water and you appreciate that water is a narrow bandwidth medium, at least for physical movement.

In their efforts to keep pace with the demand for higher speeds, modem designers immediately collided with a limited resource: by and large, signal bandwidth, set by the telephone channel, is fixed. Two approaches will increase speed: (1) use more of the available bandwidth, and (2) construct densely packed signaling alphabets so that each signaling interval uniquely specifies more than one data bit.

Dale M. Walsh, U.S. Robotics Inc., 477 East Butterfield Rd., Lombard, IL 60148.

Available Bandwidth

The available telephone channel bandwidth is approximately 200–3200 Hz. Lower speed modems carve out narrow slices near the center where channel distortions are at a minimum. But as you move out from the middle, things get sticky. Band-edge characteristics vary widely from call to call and, near the limits, are extremely hostile to data signals. Modem evolution, as a result, favored development of complex signal alphabets before the final assault on unused bandwidth.

The Bell 103 and other dial-up, full duplex modems belong to a class called Frequency Division Multiplexing (FDM) modems. They send and receive simultaneously using two non-overlapping frequency bands—much like two radio stations tuned at different frequencies. By protocol, the calling modem uses the lower frequency band; hence the terms "Originate" and "Answer" bands. The figure (page 57) shows these bands for the Bell 103 and Bell 212 modems.

Signal States

"One if by land, two if by sea," and Paul Revere was on his way. The flash of light was a data symbol, and one bit per data symbol did the job.

The 300 bit/sec 103 modem uses a similar scheme: one tone for a one and a different tone for zero. It's a fairly simple scheme and easily implemented with today's technology. The Originate and Answer bands each use about 300 Hz of the roughly 3,000 Hz bandwidth that is available.

To quadruple data rates to 1200 bit/sec, modem designers made three changes. First, they changed the basic signaling scheme from frequency modulation (as in FM radio) to phase modulation; the latter more efficiently uses available bandwidth. Second, they increased the signaling alphabet from two states (0 and 1) to four states (0, 1, 2, 3); hence, each data symbol uniquely signals two bits. Finally, they doubled the baud rate (the rate at which data symbols are sent from transmitter to receiver) from 300 to 600; this boosted the Originate and Answer bandwidths to approximately 800 Hz each.

These changes significantly increased modem complexity. But LSI technology rode to the rescue. Faster microprocessors handled the modulation and demodulation tasks. More importantly, the complicated 212 analog filters that separate high and low (Answer and Originate) bands were reduced to single LSI chips.

2400

The move from 1200 to 2400 bit/sec distinctly ups the ante.[1] Designers decided not to increase signal bandwidth further, primarily so that they could use 212 filter chips in 2400 bit/sec modems and so that 2400 modems could conveniently fall back to a 212 mode.

Instead, they increased speed by changing the signaling scheme from phase modulation to quadrature amplitude modulation (QAM). In QAM, you can increase the signal alphabet by varying both signal phase and amplitude. A 2400 bit/sec modem sends 16 phase-amplitude combinations. Each combination corresponds to four data bits. Therefore, the modem baud rate between transmitter and receiver is the same for both 212 and 2400, but each 212 data symbol carries two data bits while each 2400 modem data symbol carries four data bits.

Performance

What does this cost? First, design complexity quadruples. The design is more exacting and permits fewer simplifying short cuts.

Second, an inherent performance loss is associated with using 16 different signals rather than four. Imagine a signal system that flashes one of four colors, each color representing two bits. To convert to 16 colors would degrade the system's ability to distinguish colors, especially when they are surrounded by other interfering color flashes.

However, the performance difference between 212 and 2400 modems is not as great as one might expect. The 2400 modems use adaptive equalizers that "learn" telephone channel characteristics, compensating for call-to-call variations. An adaptive equalizer is a little like an auto-focusing camera device that also balances the color spectrum and corrects lens imperfections. Because a sharply focused receiver signal is more noise tolerant, the 2400 receiver can achieve 80–90 percent of its potential noise immunity on every call. Most 212 designs use a compromise equalizer ("one size fits all") approach that sacrifices performance for design simplicity. This difference significantly narrows the performance gap between the two modems.

But, more to the point, how well the 2400 modem works depends on its performance potential compared to the performance limits imposed by the U.S. telephone network. After reading a recent survey of signal levels, noise, and distortions on the U.S. network, one can conclude that well-designed 2400 modems will perform successfully on a very high percentage of such connections. Indeed, this is borne out by units already in the field: a limited test using a U.S. Robotics Courier 2400 modem on long distance connections yielded 95 percent successful calls.

Higher Speeds

Doubling speed from 2400 to 4800 is not as straightforward as going from 1200 to 2400 because 2400 bit/sec is the maximum practical speed for FDM dial-up modems (like the Courier 2400). There are two barriers. First, network changes will make extending the signal alphabet beyond 16 states highly questionable. Second, the remaining option (to use more of the available signal bandwidth) chews up more bandwidth than is available! At 4800 and above, the Answer and Originate bands overlap, and design complexity escalates dramatically.

It *is* possible to mix Transmit and Receive signals in the same frequency band. Each end must ignore its own transmitter and pick up the remote transmitter. This is called echo cancellation. Echo cancellation and adaptive equalization (mentioned above) are similar in that they both "learn" line conditions and adaptively compensate. But echo cancellation requires more computations with about twice as many bits of arithmetic. Several European modems successfully use echo cancellation, but they are fairly expensive compared to 2400 bit/sec modems.

CCITT recommendation V.32 defines a new dial-up 9600 bit/sec modem that includes a 4800 bit/sec mode. The Originate and Answer bands completely overlap, and each occupies 90 percent of the available bandwidth. Originally, the modem was conceived for local access to digital data networks. However, added features now allow long haul operation on international circuits and in large countries like the U.S.

Recommendation V.32 is very new and subject to change as the first models are introduced. Measured by computations per second and bits of resolution, these modems are roughly 64 times more complex than 2400 bit/sec modems. For this reason, it will be quite some time before V.32 modems can compete in the high-volume modem market.

Notes

[1] The discussion here concerns dial-up, full duplex modems in the 103 and 212 class. Leased-line modems typically operate 4,800–14,400 bit/sec.

DDJ

C UART Controller

by Don Gay

This article describes a program that allows easy modification of the parameters of a programmable USART. The program, written in C, is readily adapted to a number of different UART chips. This particular program is for the Signetics/National Semiconductor 2651 USART.

Recently, I constructed a serial port board to let me use a modem on my microcomputer. After looking at some of the designs in *Interfacing to S-100/IEEE 696 Microcomputers*[1] and checking the local availability of UART chips, I decided to implement the design on page 183 of the book, which makes use of the 2651 USART.

This USART is fully programmable with respect to baud rate, stop bits, parity, word length, and so on, and changes to my BIOS would program the USART at boot time. However, I needed more flexibility. When transferring files to an adjacent machine, I would have appreciated a simple way to change the USART parameters, namely, the baud rate, resetting it for higher rates.

A tasty little morsel from Wahroonga.

SETP.C

The Listing on page 62 shows the results of my attempt at solving this problem. The program is called with a number of command line switches, which can be used to modify the port parameters. I have designed the program to run entirely from the command line to make it easily adaptable to systems that have an autoexecute on boot: anyone can boot up in whatever mode is required without modifying the BIOS.

An examination of the listing reveals four command line switches. These allow the user to change the baud rate (–b), number of stop bits (–s), parity (–p), and word length (–l). All invalid switches are ignored, as are values out of range for any of the parameters. After all command line switches have been effected, the program shows the values of all the port parameters. Trying to execute the program with no switches on the command line produces a "Usage" description, followed by the current port parameters. Here is an example of how the program typically is called:

```
A> setp –b1200 –s1 –pn –l8
```

This would configure the USART to transmit/receive at 1200 baud, 1 stop bit, no parity, and a character length of 8 bits.

USART Programming Details

A 2651 USART uses four consecutive I/O ports:

(1) Transmit/Receive holding register
(2) Status register
(3) Mode register *1* and *2*
(4) Command register

The USART parameters are controlled by writing to the mode registers. By reading the values in the mode registers, you can determine the current parameters of the USART.

There are actually two mode regis-

Don Gay, 136 Fox Valley Road, Wahroonga, 2076, N.S.W., Australia.

ters at the same I/O address, each accessed by a consecutive read or write. Read from the mode register once and you will get the contents of one of the mode registers; read from it again and you will get the contents of the other mode register. A similar situation exists for writing. After a reset, the first read/write accesses the first mode register. As long as reading and writing are done in pairs, you will always know with which register you are dealing.

The command register controls the level of *RTS and *DTR lines, as well as enabling and disabling transmit/receive. Because the values written to the command register need not change once the port has been initialized, the program does not allow alteration of the command register from the command line switches. These switches alter only the contents of the mode registers.

As is usually the case with programmable peripheral interfaces, each bit has a special significance in the controlling mode registers. Figures 1 and 2 (at right) show the importance of each bit in the mode registers of the 2651.

The USART is configured in this program in asynchronous mode with internal receive and transmit clocks. This is probably the most common mode used with ordinary devices, such as modems and terminals, so these characteristics are "hard coded" in the program. Refer to the 2651 documentation[2] to find out other ways to operate the port.

References

[1] Sol Libes and Mark Garetz. *Interfacing to S-100/IEEE 696 Microcomputers.* California: Osborne/McGraw-Hill, 1981.

[2] National Semiconductor Data Sheet. "INS2651 Programmable Communications Interface." October 1980.

DDJ

(Listing begins on next page)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**

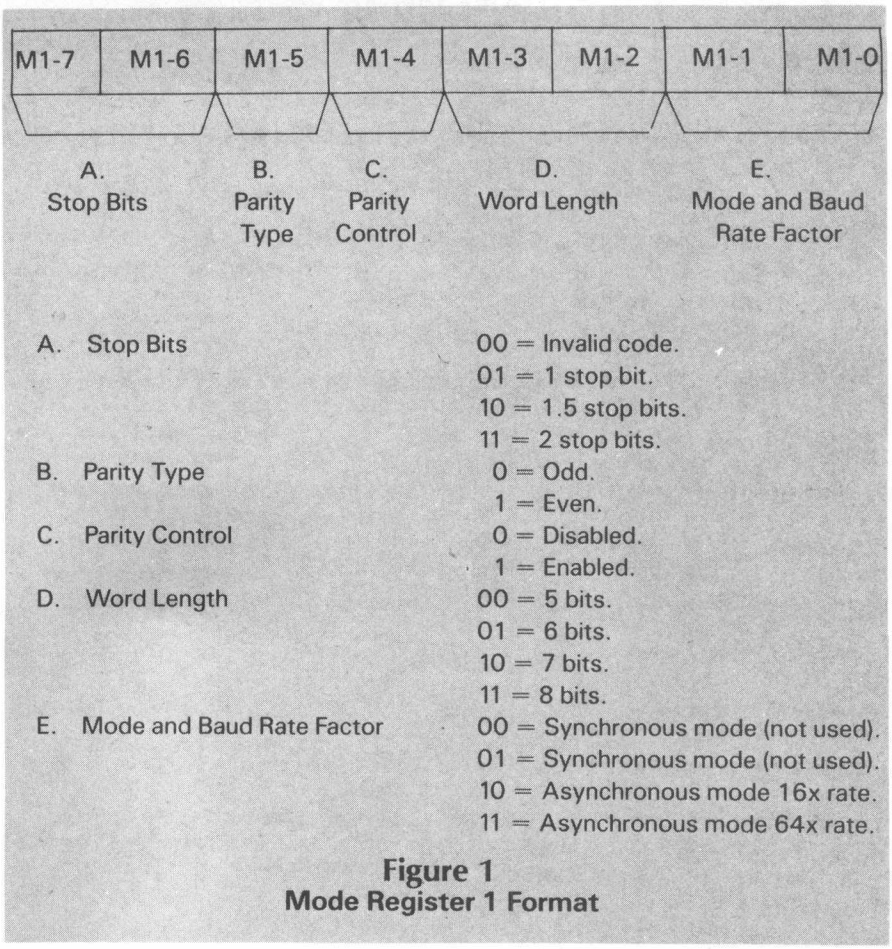


Figure 1
Mode Register 1 Format

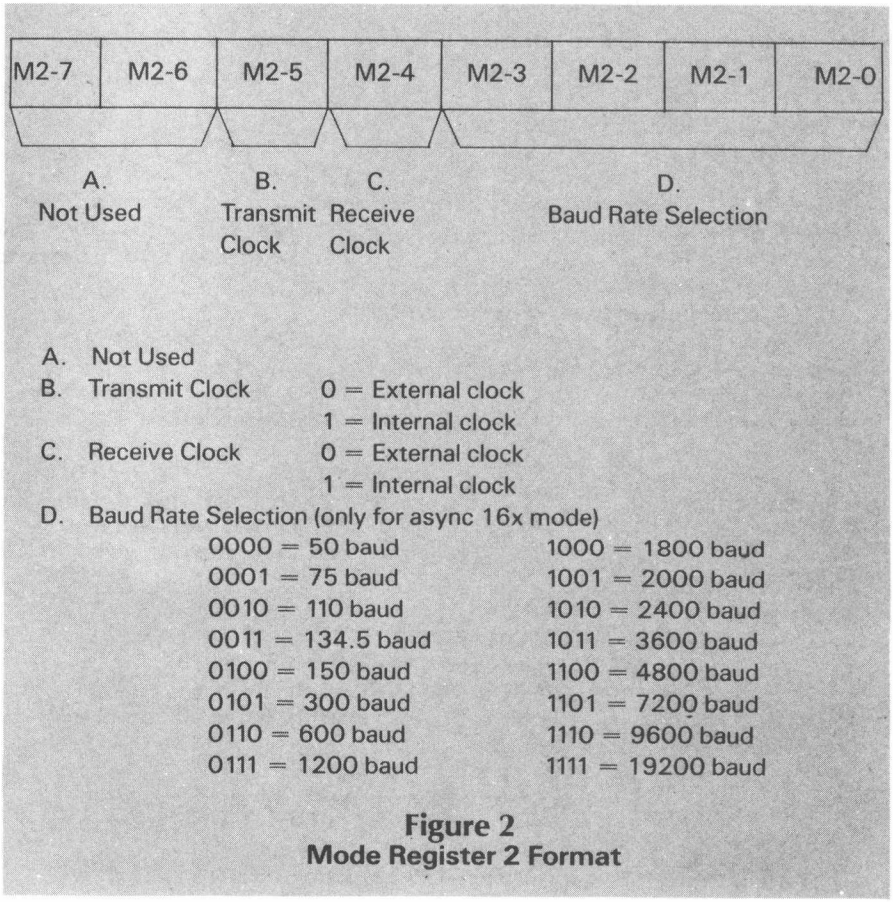


Figure 2
Mode Register 2 Format

C UART Controller Listing (Text begins on page 60)

```
/*************************************************************************/
/*                                                                       */
/*           Serial Port Configuration Program  (2651 PCI)               */
/*                                                                       */
/*   Source filename  : SETP.C                                           */
/*   Program Language : C/80 3.0 (Software Toolworks)                    */
/*   Program Author   : Don Gay                                          */
/*   Last Revision    : June 11, 1984                                    */
/*                                                                       */
/*************************************************************************/

#include printf.h                    /* Header for formatted output */

#define REV_DATE        "11/06/84"

#define PORT_BASE       0               /* 2651 PCI base address    */
#define MODE_REG        PORT_BASE+2     /* Mode register            */
#define CMND_REG        PORT_BASE+3     /* Command register         */

#define DISABLE         0x32            /* Disable UART recv/xmit   */
                                        /* Force *DTR and *RTS low  */
#define ENABLE          0x37            /* Enable UART recv/xmit    */

/* Store legal values for all paramters */

char *baudr[] = {"50","75","110","134.5","150","300","600",
                 "1200","1800","2000","2400","3600","4800",
                 "7200","9600","19200"},
     *sbits[] = {"0","1","1.5","2"},
     *parit[] = {"N","O","N","E"},
     *wordl[] = {"5","6","7","8"};

int baud_rate, stop_bits, parity, word_length;

main(argc,argv)
int argc;
char *argv[];
{
   printf("\nSerial port configuration program as of %s\n",REV_DATE);
   get_prm();                       /* get current port parameters */
   if (argc>1)                      /* process the command line */
      while (argc-- > 1)
         process(argv[argc]);
   else                             /* Give some help */
      show_usage();
   show_prm();                      /* Display parameters */
   set_prm();                       /* Re-program PCI     */
}

process(arg)                        /* Try to match the command line */
char *arg;                          /* argument with a valid switch  */
{                                   /* If valid, set up new value    */
   int n;
   if (*arg == '-') {
      switch(toupper(*++arg)) {
         case 'B': n=match(++arg,baudr,16);
                   if (n>=0) baud_rate=n;
                   else error("baud rate value",arg);
                   break;
         case 'S': n=match(++arg,sbits,4);
                   if (n>=0) stop_bits=n;
                   else error("stop bits value",arg);
                   break;
         case 'P': n=match(++arg,parit,4);
```

```
                    if (n>=Ø) parity=n;
                    else error("parity value",arg);
                    break;
          case 'L': n=match(++arg,wordl,4);
                    if (n>=Ø) word_length=n;
                    else error("word length value",arg);
                    break;
          default : error("switch",arg); break;
       }
    }
    else
       error("switch",arg);
}

match(value,table,size)                 /* Check if switch value is valid */
char *value,                            /* Command line value */
     *table[];                          /* Table of valid arguments */
int size;                               /* No. of entries in table */
{
   int ndx;

   ndx=Ø;
   while ((ndx<size) && (strcmp(value,table[ndx]) != Ø))
      ndx++;
   return( ndx<size ? ndx : -1 );/* Return -1 if not found */
}

get_prm()                               /* Read mode ports to determine */
{                                       /* current port parameters      */
        int mode1,mode2;

        mode1 = input(MODE_REG);/* get mode register 1 */
        mode2 = input(MODE_REG);/* get mode register 2 */

        baud_rate   = getbits(mode2,3,4); /* isolate bit fields */
        stop_bits   = getbits(mode1,7,2);
        word_length = getbits(mode1,3,2);
        parity      = getbits(mode1,5,2);
}

getbits(x,p,n)                          /* get n bits from position p */
unsigned x, p, n;                       /* Ref: K&R pg 45 */
{
        return((x >> (p+1-n)) & ~(~Ø << n));

}

set_prm()                   /* Re-program PCI with new parmameters */
{
        int mode1, mode2;

        mode1 = Øx02 |                      /* set asynch. 16x mode */
                ( stop_bits   << 6) |  /* merge parameters     */
                ( (parity     << 4) & Øx3Ø ) |
                ( (word_length << 2) & ØxØc );

        mode2 = Øx3Ø |                      /* set internal clock   */
                baud_rate;                  /* merge baud rate      */

        output(CMND_REG,DISABLE);           /* disable recv/xmit    */
        output(MODE_REG,mode1);             /* send new parameters  */
        output(MODE_REG,mode2);
        output(CMND_REG,ENABLE);            /* now enable PCI       */
}
```

```
show_parm()                          /* Display current parameters */
{
        printf("\nBaudrate   : %s\n",baudr[baud_rate]);
        printf(  "Stopbits   : %s\n",sbits[stop_bits]);
        printf(  "Parity     : ");
        switch(parity) {
           case 0: printf("None\n"); break;
           case 1: printf("Odd\n"); break;
           case 2: printf("None\n"); break;
           case 3: printf("Even\n"); break;
        }
        printf(  "Wordlength : %s\n",word1[word_length]);
}

show_usage()
{
   printf("\nUsage: SETPORT [-Bb][-Ss][-Pp][-Ll]\n\n");
   printf("        where -Bb sets Baudrate,    b = 50..19200\n");
   printf("              -Ss sets Stopbits,    s = 0, 1, 1.5, 2\n");
   printf("              -Pp sets Parity,      p = O, E, N\n");
   printf("              -Ll sets wordLength, l = 5, 6, 7, 8\n");
}

error(type,value)                    /* Warn command line argument error */
char *type, *value;
{
   printf("Illegal %s ignored : %s\n",type,value);
}

input(port)                          /* Return input from I/O port */
int port;
{
#asm
        pop     b               ;Return address
        pop     h               ;Port number
        push    h               ;Restore stack
        push    b
        mov     a,l             ;Get port into A
        sta     inadr           ;Modify next instruction------+
        in      0               ;Dummy input instruction      ¦
inadr   equ     $-1             ;Actual port gets poked here <+
        mov     l,a
        mvi     h,0
#endasm
}

output(port,byte)                    /* Output to I/O port */
int port, byte;
{
#asm
        pop     b               ;Return address
        pop     d               ;Byte to output
        pop     h               ;Port number
        push    h               ;Restore stack
        push    d
        push    b
        mov     a,l             ;Get port into A
        sta     outadr          ;Modify next instruction--------+
        mov     a,e             ;Get byte to output             ¦
        out     0               ;Dummy output instruction       ¦
outadr  equ     $-1             ;Actual port gets poked here <--+
        mov     l,a
        mvi     h,0
#endasm
}
```

End Listing

Christensen Protocols in C

by Donald Krantz

The most-often-ported of telecommunications programs meets the most-often-called-portable of languages.

About two years ago, when I acquired my first 8-inch machine, I needed a file transfer program to load data between the 8-inch machine and my 5¼-inch machines. I started looking around for a version of MODEM7 on 8-inch disk.

Before I found one, I ordered a couple of 8088 co-processors for my systems. As this development promised to complicate my file transfer problems considerably, I decided to write a file transfer utility in C so I would have a single, portable program for all my systems.

I started writing in SuperSoft C, because that was what I had back then, and several weeks later had a version working on my Zorba CP/M machine. I thought I'd be in Fat City when the co-processors came.

Well, the co-processors arrived, I wired them in, and I started planning the big port. I was unpleasantly surprised to find that the co-processors had no assembler or documentation and wouldn't read my 5¼-inch IBM disks. As far as I could tell, I had no means of getting any new programs or files into the system short of writing a program with DEBUG. This was unacceptable, especially since I had promised myself that I would never learn nor use 8088 assembly language and had given myself a waiver only to port the source for the file transfer program.

I put the 8088s aside and continued with my life. As time passed, I used the file transfer program, which by now had the name XFR, and gradually started adding bells and whistles.

Some time later, I had occasion to write XMODEM in C for my portable BBS series. I thought I'd just take the old XFR program and do a little surgery; I figured the whole thing ought to take no longer than a couple of hours.

By this time, I had switched all of my new program development to Eco C, which is closer to the Unix Version 7 standard than was SuperSoft. As I started work on XMODEM, I found more and more places where the basic structure of XFR made porting the thing less and less attractive. Still, I took out the digital chain saw and started to work on XFR.

Some of the items I took out include an assembly language CRC generator, interrupt-driven receive FIFO, XDIR-style directory, wildcard filename expansion for the batch mode, file eraser, disk drive logger, half duplex and host terminal mode, and a data line monitor for watching control characters. What's left is a basic Christensen protocol engine with a minimal terminal program. The advantage is that the file transfer primitives are now pretty much independent of the operating system and C compiler—to the extent allowed by the protocol—and can be dropped

Donald Krantz, 2845 42nd Ave. South, Minneapolis, MN 55406.

into other environments easily.

The basic features supported by XFR are dumb terminal emulation with host system terminal attributes and checksum or CRC file transfer in single or batch file mode. I'll run through the Listing (page 71) and discuss features and portability considerations on a function-by-function basis. First, let me mention up front that my goal was operation at 19.2K-baud, so I have violated some software development standards in the interest of speed, specifically by using C's interfunction GOTO to reduce conditionals in loops. Also, Z80 implementations of C tend to access globals faster than automatics, so globals perhaps are overused.

Function main( )

This is basically a menu processor. Most of the functions do returns rather than using the longjmp( ) to resume the menu loop. It's a trivial job to add features to the menu if you plan on actually using this as a terminal program and need more functions.

Function term( )

This is a simple dumb terminal. For you MSDOS or Unix folks who haven't memorized the CP/M BDOS function calls, the function _bdos(6, 0xFF) is a specialized version of getchar( ) that immediately returns a NULL if no character is waiting. I've seen a similar function, sometimes called inkey( ) or keyscan( ), in other C libraries. The presence or absence of term( ) doesn't affect the file transfers at all, but because you usually should be able to communicate with the remote system to set up the file transfer, it's nice to have some sort of dumb terminal.

Function transmit( )

This function has the only cheezy user interface in the program. It selects batch file transfer, as opposed to single file transfer, by looking for a comma at the end of the input line when asking for the transmit filename; should it find a comma, it will select batch mode and ask for the next name. Most versions of MODEM7 either have a command option switch for this purpose or recognize that you want batch mode if an ambiguous filename is specified. The original XFR used the latter method. Unfortunately, there is no portable way to expand filenames and get a list of matching files.

If you have a system that will expand wildcards on the command line and pass matching names through the argc/argv method, you could easily convert this program from interactive to command line mode. Otherwise, on CP/M systems, if you have access to the BDOS calls, you can use the function unparse( ) to convert CP/M fcb names to a normal format and pass them one at a time to trans( ). The one advantage the current method has over wildcard expansion is that it allows sending files with dissimilar names. However, if you do use the CP/M "search for file" and "search for next" calls, be sure to collect all the names before you allow the program to proceed.

Function trans( )

This is the file transmission primitive executive (if that's not a contradiction in terms). A filename is passed in, in a format compatible with the fopen( ) function. If batch mode is selected, the name is sent to the remote system, followed by the file itself. This function, and the supporting functions it uses, are all pretty much system independent within the confines of the protocol.

Function txrec( )

This function sends a single record to the remote system and does any retransmission necessary due to errors. The record size is fixed at 128 bytes by the macro RECSIZE. If you don't need compatibility with the rest of the world, you could try optimizing performance by increasing the size of the record, if you have a good communication link, or decreasing the size of the record if you use noisy lines. The macro RECSIZE fixes all the parameters necessary for change of record size.

Function txname( )

This function sends a filename to the remote system when batch mode is selected. The protocol calls for a fixed 11-character field in CP/M fcb format. This part of the protocol has a slightly different flavor to it than the rest, so I suspect that somebody other than Ward Christensen did it. If you're not interested in compatibility with MODEM7, the size of the name is fixed by the macro NAMESIZE. You're safe up to about 30 characters or so before you have to start hunting for other items affected by the name size that NAMESIZE doesn't control.

Function receive( )

This is the file reception primitive executive. All it really does is figure out if you want batch mode or single file mode. It's not possible to tell what the other system has selected just by listening to the line. If an asterisk is entered in the receive filename specification, batch mode is selected while preserving any drivespec. Those with non-CP/M systems could remove the drivespec material, but this probably won't be necessary. It's triggered by a colon in the second character position of the filespec.

Because the Christensen protocol is receiver driven, you have to have made a choice whether to use checksum or CRC error detection before you initiate the receive mode. *Some* versions of MODEM7 will hunt back and forth between CRC and checksum mode to match what the other system uses, but this program doesn't.

Function rxbatch( )

This takes over as receive executive if batch mode is selected. It receives a filename from the remote system and converts it from the fcb format used for transmission back to a format suitable for use with C functions. The name is then passed to the file receive function rxfile( ) to pick up the body of the file.

Function rxname( )

This is the companion to txname( ). Enough said.

Function rxfile( )

This handles reception of an entire file. This function is the time-critical part of the program, which is why I didn't break it into smaller modules.

It doesn't matter if the program is slow when transmitting, but when receiving you have to catch all of the characters sent from the other end as they come in. I put off the computation of checksum or CRC to the end of the record for this reason, instead of building it into the receive character function where it probably belongs.

It's also important not to output messages to the display when a packet transfer is in progress. One of the versions of MODEM7.11 that is floating around can't run above 1200 baud in receive mode because it makes a message output call immediately after receiving the packet header and so misses a couple of characters.

Function make( )

This opens a file to receive to. The

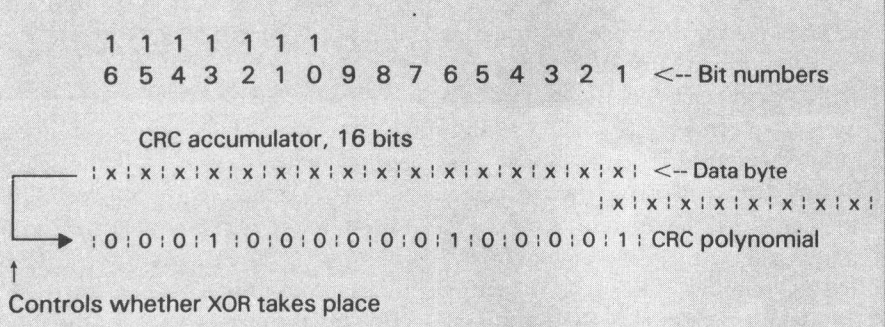


Figure 1.
A 1 shifted out of the left end of the CRC accumulator causes the CRC polynomial to be exclusive-ORed with the post-shift CRC accumulator. The CRC polynomial appears to have no relation to the polynomial equation in the text until you realize that it has been shifted left along with the accumulator.

Byte Number	Definition
1	SOH (Start of Header)*
2	Record number mod 256*
3	[One's complement of record number*
4	Data byte 1
.	
.	
131	Data byte 128
132	CRC high byte†
133	CRC low byte†

* Not included in CRC or checksum calculation
† Included as zeros in CRC calculation

Figure 2.
Christensen packet makeup is shown for a CRC error detection packet. In checksum, byte 133 is not sent, and byte 132 is the checksum.

Programmers writing MODEM7-compatible programs traditionally learn about the protocol by getting as many copies of the source to MODEM7 as possible and tracing program flow. In violation of that tradition, I have written down my interpretation in English. Glock's Fourth Law states that this must have been done before, but the corollary to that law states that you can't find out where, even if it has.

Ward Christensen (according to legend) developed the protocol, which was extended by several other people who ought to receive credit but whose names I don't have. Ward has been around so long that I was surprised, and perhaps a little indignant, to find that he is still alive. He is responsible for some of the more clever public domain programs, so I'm willing to bet that he didn't do some of the peculiar things in the Christensen protocol—such as the filename transfer.

There are two levels to the Christensen (also called XMODEM) protocol: file-at-a-time and batch transfer. The batch transfer mode provides for the unattended transmission of multiple files, including the filename.

The protocol uses two different error detection schemes: checksum and CRC (Cyclic Redundancy Code). Older versions of MODEM7 use the checksum method, while the newer versions use the more accurate CRC method. Most programs that can do CRC will also do checksum.

Only the data bytes in a packet are included in the calculation of the error detection code. The protocol handles errors in headers and handshakes by ignoring garbage characters or assuming the worst.

Checksums are calculated by adding all data bytes mod 256, with any carry ignored.

The CRC polynomial used is the

original version of XFR made a backup file if a file would be overwritten, but file renaming isn't portable in C (the current version will happily destroy an old file, even if reception is terminated prematurely). I left the code in a false conditional assembly block because I thought it rather clever; it's a good feature if your C library supports renaming. The function link takes the arguments (newname, oldname).

Function parse( )

This takes a filespec useable by C functions and converts it to the CP/M fcb format needed by the Christensen protocol. On systems with pathnames, you'll have to hack this function into shape to deal with filename preambles. It will operate on names that don't follow CP/M conventions in most circumstances, although the use of periods will give some unusual blanks expanding the dot should the filename be less than 11 characters. It is important in CP/M systems to make sure that any high bits are stripped from the filename. See Figure 3 (page 69).

CCITT $X^{16} + X^{12} + X^5 + 1$. I'm told that there are people who know what that means (I'm taking the word of an old XMODEM listing). In simple language, the procedure for generating the CRC is this: A byte to be added to the CRC is fed into the generator, one bit at a time, high bit first. The bit is shifted into a 16-bit CRC accumulator low end. If the high bit shifted out of the CRC accumulator is a 1, the CRC accumulator is exclusive-ORed with 0x1021. The process repeats for all eight bits of the input character. See Figure 1 (page 68).

In the case of the Christensen protocol, after all the data bytes in a given packet are sent through the CRC generator, two zero bytes are sent through at the end. I'm guessing that these bytes take the place of the CRC bytes in the packet, although I don't know why this is needed. I do know that it doesn't come out right unless you do this, and for an empirical kind of guy like myself that's plenty good enough.

At the single file level of the protocol, both systems are told to begin, and the transmitter waits for the receiver to send a sync byte. The initial sync byte tells the transmitting system whether CRC or checksum will be used. If checksum, the sync byte is a NAK. If CRC, the sync byte is a 'C.' If the receiver is thinking CRC and the transmitter isn't capable of it, both systems hang until they time out. Some programs send five or six 'C' sync bytes and, if not answered, then send NAKs in case the transmitter is an old version.

In CRC mode, once the systems are in sync, the 'C' is dropped in favor of ACK/NAK handshaking for subsequent records.

Once in sync, the transmitter sends a packet consisting of an SOH character, the record number mod 256, the one's complement of the record number, a 128-byte data block, and the error detection byte(s). See Figure 2 (page 68).

The receiver must accept the entire packet at once. Most versions of MODEM7 wait for the SOH, so a missed SOH is an error from which it is difficult to recover. After reception, conditions that will cause record rejection are: the record number doesn't match the one's complement of the record number; the error detection number doesn't match; SOH is not intact; and the record number either is not the record number expected or is one less than the record number expected (indicating a previous ACK was trashed). If the record is rejected, the receiver sends a NAK to the transmitter, which retransmits the record. If the record is accepted, the receiver sends ACKs at about one-second intervals until the next SOH.

This process repeats until all records are transmitted. At the end of file, the transmitter sends EOT at one-second intervals until the receiver ACKs the EOT.

Record numbers are "natural" numbers; that is, the first record is numbered 1, as opposed to the computer natural 0.

Batch level of the protocol includes a filename header in front of the file. Filenames always use checksum error detection, even if CRC mode is selected for the rest of the file. The filename is sent as it would appear in a CP/M file control block (fcb), a fixed block, 11 characters long, with

When preparing to receive a name, the receiver sends NAKs at about one-second intervals until the transmitter responds with an ACK followed by the first character of the filename. The receiver then handshakes with ACKs until the transmitter sends EOF. This may mean that more than 11 characters are received (because of line noise). When the receiver gets the EOF, it responds with the checksum of the name, including the EOF byte. If the checksum matches, the transmitter sends an ACK, and the single file process takes over. If the checksum doesn't match, the transmitter sends a NAK, and the process repeats.

If the transmitter sends an EOT in place of any name character, it causes batch transfer mode to terminate. This is the normal end to a batch transfer.

Both the transmitter and the receiver should be able to recognize CAN (Cancel) in place of ACK, NAK, EOT, SOH, or EOF. In practice, timeouts will replace some tests for CAN.

All transmit and receive loops should have timeouts and retries built into them to avoid hanging if the two systems get out of sync.

Standard UART setting for the Christensen protocol is eight bits, no parity, one stop bit.

TEST.DOC becomes:	¦T¦E¦S¦T¦ ¦ ¦ ¦ ¦D¦O¦C¦
VERY.LONG.NAM becomes:	¦V¦E¦R¦Y¦L¦O¦N¦G¦N¦A¦M¦
NAME becomes:	¦N¦A¦M¦E¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦

Figure 3.
CP/M filename expansion

results. If the NAMESIZE macro is resized, parse( ) will still operate in a reasonable manner.

Function fillbuf( )
This loads the transmit file buffer with a record. I modularized this section because soon I'll add the ability to extract members from a library file. On systems capable of file sizes that aren't multiples of system records, this function pads the last record with zeros. Note that this code will never be executed on a CP/M system.

Function unparse( )
This undoes what the parse( ) function does. Notice that a period always is inserted into a filename unless the filename is less than (NAMESIZE–3) characters long.

Function updcrc( )
This adds a character to the CRC or checksum accumulator. This uses the CCITT CRC polynomial, but characters are fed into the generator high bit first. I'm not certain, but I think that makes it incompatible with communications chips that do a hardware CRC, in case you have thoughts in that direction. At any rate, look into it before you spend a lot of time on it.

Function abort( )
This scans the console for an operator-requested abort. As far as I know, this function is called inside any loop where the system can hang (but I never guarantee anything about a program). The BDOS keyboard scan is the same as in term( ).

Function error( )
This is called when fatal transmission or reception errors occur. It ties up loose ends then jumps back to the menu.

Functions wait( ), swait( ), and waitcan( )
These do different delays while waiting for characters. You could combine them all into one function if you're willing to pay the overhead for more conditionals. At high baud rates, file reception timing is pretty critical, but for 1200 baud or so there won't be any problems.

Remaining Functions
In CP/M systems, all of the above functions should work as is. The rest of the functions are all machine dependent—or potentially so.

The init( ) and quit( ) functions are called at the beginning and end of the program. I had code to set up and remove an interrupt-driven FIFO in these functions; UART initialization or I/O channel acquisition would probably go here as well.

The I/O primitives are set up for an I/O-mapped Intel 8351A UART. These functions are the ones most likely to require customization to make the program work.

How does the whole thing perform? On a Z80 running at 3.58 MHz, the program will easily do transfers in checksum mode at 19.2K baud. I'm reasonably sure it will do the same in CRC mode, but I haven't the resources to test CRC at that speed. If anyone does do a benchmark, I'm interested in the result.

As far as error handling, the program is not ready for an end user, but a reasonable person would have to work at it to blow the thing up. Consider this a starting point, and if anyone gets it up on 8-inch MSDOS for the Bigboard and Co-Power 88, let me know.

Source is available on TCOG BBS (612) 724-7779 300/1200 baud.

DDJ

Protocols Listing (Text begins on page 66)

```
/*-----------------------------------------------------------------
                              XFR
                    By Donald G. Krantz 3/84
              (C) Copyright 1984 Donald G. Krantz
                      All Rights Reserved

XFR is a packet protocol file transferer similar to modem7. It is
designed for 19.2K baud transfers "nose-to-nose" between computers
and works very well in modem use at low speeds. XFR is compatible
with most versions of Modem7.

-----------------------------------------------------------------*/

#include "stdio.h"                /* ECO C */

/*-----------------------------------------------------------------
ASCII and XMODEM control characters
-----------------------------------------------------------------*/

#define SOH 1                     /* start of header              */
#define EOT 4                     /* end of transmission          */
#define ACK 6                     /* true acknowledge             */
#define NAK 0x15                  /* false acknowledge            */
#define CRC 'C'                   /* request CRC mode             */
#define CAN 0x18                  /* cancel transmission          */
#define EoF 26                    /* end of file (used for name)  */
#define BADNAME 0x75              /* received bad name checksum   */
#define BELL 7                    /* console bell                 */
#define NEWLINE 10                /* linefeed char                */
#define CR 13                     /* carriage return char         */
#define BS 8                      /* backspace character          */
#define ESCTERM 'E'-'@'           /* escape from terminal mode    */

/*-----------------------------------------------------------------
User accessable system equates
-----------------------------------------------------------------*/

#define RETRY 5                   /* no. of retrys before abort   */
#define RECSIZE 128               /* transfer record size         */
#define NAMESIZE 11               /* filename fixed length        */
#define TITLE printf( "\n\t\t\t  XFR - File Transfer Utility")
#define VERSION printf("\n\t\t\t ZORBA Version 2.00 as of 3/85")
#define CLS printf( "\033E")      /* clear screen                 */
```

Protocols Listing (Listing continued, text begins on page 66)

```
#define ERROR -1
#define MAGIC_NUMBER 10000      /* time constant - machine dep. */
#define VOID int                /* function type                */
/* #define LINK                 / * file rename supported       */

/*-------------------------------------------------------------
Machine dependant port addresses - ZORBA Version (Intel 8251A)
-------------------------------------------------------------*/

#define RXDATA 0x20             /* rec'd data UART port         */
#define TXDATA 0x20             /* tx'ed data UART port         */
#define UARTCMD 0x21            /* Command/Status port          */


/*-------------------------------------------------------------
Global variables
-------------------------------------------------------------*/

char *line;                     /* scratch input line           */
char *buffer;                   /* record buffer                */
FILE *fd;                       /* file POINTER!                */
unsigned rec;                   /* record number                */
char checksum;                  /* checksum accumulator         */
unsigned crcaccum;              /* global crc bytes             */
char crc;                       /* CRC on/off flag              */
char batch;                     /* batch on/off flag            */
jmp_env to_menu;                /* long jump envelope           */
struct {                        /* batch list array             */
            char name_part[ 32 ];
        } list[ 32 ];

/*-------------------------------------------------------------
Function type declarations
-------------------------------------------------------------*/

VOID main(), term(), transmit(), trans(), txname(), txrec();
VOID receive(), rxname(  rxfile(), make(), error(), clrcrc();
VOID updcrc(), abort(), sleep(), quit(), init(), tx(), rxbatch();
char *index(), *parse(), *unpa se();
int fillbuf(), wait(), swait(), waitcan(), rxstat(), rx();
int txstat();

/*-------------------------------------------------------------
Executable code follows.

main() is (of course) the master program loop. Uses setjmp()
to un-nest function calls on error exits.

References globals: crc, tty;
Modifies globals: crc, tty;
-------------------------------------------------------------*/

VOID main()
{
   char menu;

        menu = FALSE;           /* initially expert mode        */
        init();                 /* initialize modem and UART    */
        TITLE;
        VERSION;
        printf( "\nType 'M' for MENU display....");
        setjmp( &to_menu );     /* menu loop                    */
        while( TRUE )
        {
                printf( "\n\nMode: %s", crc ? "CRC" : "Checksum" );
                if( menu )
                {
                  printf( "\n\n\tS Send file    R Receive file\n" );


                  printf( "\tT FDX term     V CRC/Checksum\n" );
                  printf( "\tM toggle menu  X Exit\n" );
                }
                printf( "\n\nCommand: " );
                switch( toupper( getchar() ) )
                {
                        case 'S':               /* send file(s) */
                                transmit();
                                break;
                        case 'R':               /* RX file(s)   */
                                receive();
                                break;
                        case 'T':               /* dumb terminal*/
                                term();
```

(Continued on page 74)

```
                        break;
                case 'V':                       /* toggle crc   */
                        crc = !crc;
                        break;
                case 'M':                       /* toggle menu  */
                        menu = !menu;
                        break;
                case 'X':                       /* quit         */
                        quit();
                default:                        /* option bad   */
                        putchar( BELL );
            }
        }
}


/*---------------------------------------------------------------
term() is the dumb terminal section. Escape by typing ESCTERM
character (see ASCII #defines).
Some machine dependent stuff.
---------------------------------------------------------------*/

VOID term()
{
   char ch;                             /* scratch char         */

        CLS;
        printf( "\n\n" );
        while( TRUE )                   /* the loop.            */
        {
                if( rxstat() )          /* if char coming in    */
                        putchar( rx() );/* display local        */
                if( (ch = _bdos( 6, 0xFF )) != NULL )
                        if( ch == ESCTERM )     /* check exit   */
                                return;
                        else
                                tx( ch );       /* disp remote  */
        }
}

/*---------------------------------------------------------------
transmit() is the transmission executive. It requires no input
parameters. This function takes filename(s) as input from the
user, one line at a time. A comma at the end of a filename
signals the intention to enter anther filename on the next
line. More than one filename entered sets "batch" mode. 32
names maximum, for no especially good reason.

Modifies globals: buffer, batch;
---------------------------------------------------------------*/

VOID transmit()
{
   int ct = 0;
   int i;

        printf( "\n" );
        while( TRUE )
        {
                printf( "Transmit File specification: " );
                gets( list[ ct ].name_part );
                if( index( list[ ct ].name_part, ',' ) == NULL )
                        break;
                *index( list[ ct ].name_part, ',' ) = '\0';
                ct++;
        }
        batch = (ct ? TRUE : FALSE);
        for( i=0 ; i <= ct ; i++ )
        {
                while( rxstat() )       /* gobble garbage       */
                        rx();
                trans( list[ i ].name_part );   /* send file    */
        }
        if( batch )
        {
                while( swait() != NAK ) /* await name request   */
                        abort();        /* operator abort scan  */
                tx( ACK );              /* handshake NAK        */
                sleep();                /* decent interval      */
                tx( EOT );              /* end of transmission  */
        }
}
```

(Continued on page 76)

```
/*-----------------------------------------------------------------
trans() does transmission of a file.

References globals: buffer, fd, batch;
Modifies globals: fd, rec, crc;
-----------------------------------------------------------------*/

VOID trans( in_name )
   char *in_name;
{
   char name[ NAMESIZE+1];      /* expanded name                */

   char ch;                     /* scratch character            */
   register char *i;            /* buffer pointer (index)       */
   int y;                       /* scratch counter              */

        rec = 1;                /* uses "natural" numbering     */
        if( (fd = fopen( in_name, "rb" )) == NULL)
        {
                printf( "Unable to open: %s\n", in_name );
                return;
        }
        if( batch )             /* need to send name?           */
                txname( parse( in_name, name ) );
        printf( "\nTransmitting: %s", in_name );
        while( rxstat() )
                rx();
        while( TRUE )           /* handshake w/ receiver        */
        {
                printf("\nSyncronizing: " );
                if( (ch = waitcan( 10 )) == NAK )
                {
                        printf( "Received Checksum Request\n" );
                        crc = FALSE;
                        break;
                }
                if( ch == CRC )
                {
                        printf( "Received CRC Request\n" );
                        crc = TRUE;
                        break;
                }
        }
        while( fillbuf() )              /* file not empty?      */
                txrec( buffer );        /* send record          */
        tx( EOT );
        while( (ch = waitcan( 1 )) != ACK )
        {
                tx( EOT );              /* show end             */
        }
        fclose( fd );                   /* dump file            */
        printf( "\n%s transferred.", in_name );
}


/*-----------------------------------------------------------------
txname() transmits a file name from the CP/M fcb format
parameter 'name'.

References globals: buffer, checksum, crc;
Modifies globals: crc;
-----------------------------------------------------------------*/

VOID txname( name )
   char *name;
{
   register int i;              /* scratch counter              */

   char ch;                     /* scratch char                 */
   char crcsav;                 /* holds state of global 'crc'  */

        crcsav = crc;           /* we always use checksum for   */
        crc = FALSE;            /* name error detection         */
        while( TRUE )           /* main loop for retries        */
        {
                while( rxstat() )
                        rx();
                i = 0;          /* retry count                  */
                while( TRUE )
                {
                        printf( "\nWaiting for filename request");
                        if( waitcan( RETRY ) == NAK )
                                break;
                        if( i++ ) RETRY )
```

```
                        error( "Can't send filename" );
                printf( " - Received garbage" );
        }
        tx( ACK );          /* handshake name request       */
        sleep();            /* decent interval              */
        printf( "\nSending name: " );
        clrcrc();           /* clear checksum accumulator   */
        for( i = 0 ; i < NAMESIZE ; i++ )/* name loop       */
        {
                tx( name[ i ] );
                putchar( name[ i ] );   /* local echo       */
                wait( 10 );     /* wait for ACK             */
        }
        tx( EoF );              /* name terminated w/EoF*/
        if( wait( 10 ) == checksum )    /* handshake        */
        {
                printf( " - Name sent OK" );
                tx( ACK );      /* handshake OK checksum*/
                crc = crcsav;   /* replacce 'crc'           */
                return;         /* normal exit              */
        }
        tx( BADNAME );          /* handshake bad chksm      */
        printf( " - Bad name transfer" );
    }
}


/*-----------------------------------------------------------------
txrec() transmits a single record, with retransmit on error from
receiver. Input is a pointer to the I/O buffer.

References globals: rec, crcaccum, checsum, crc;
Modifies globals: rec;
-----------------------------------------------------------------*/

VOID txrec( buf )
   char *buf;
{
   register int i;
   unsigned cr;

    while( TRUE )                   /* do it until right    */
    {
        printf( "\rTransmitting record %d   ", rec );
        tx( SOH );                  /* start of header      */
        tx( rec );                  /* rec #                */
        tx( ~rec );                 /* 1's comp             */
        clrcrc();                   /* clear CRC accum      */
        for( i = 0 ; i < RECSIZE ; i++ )
            tx( buf[ i ] );         /* send record          */
        updcrc( 0 );                /* finish up CRC        */
        updcrc( 0 );                /* again                */
        cr = crcaccum;              /* save crc lobyte      */
        if( crc )                   /* send hi byte first   */
        {
            tx( crcaccum >> 8 );
            tx( cr );
        }
        else
            tx( checksum );
        if( waitcan(10)==ACK )  /* quit if correct          */
            break;
        printf( "Error in transmission\n" );
    }
    rec++;                          /* bump record count    */
}


/*-----------------------------------------------------------------
receive() is the receive executive. It requires no input
parameters except 'crc' should be set. Does not "hunt" from
CRC to checksum.

References globals: line;
Modifies globals: line;
-----------------------------------------------------------------*/

VOID receive()
{
    printf( "\n\nReceive File specification: " );
    gets( line );
    if( rxstat() )
        rx();                       /* gobble trash         */
    if( index( line, '*' ) != NULL )/* 'batch' set by a     */
        rxbatch( line );            /* asterisk             */
    else                            /* in input file name   */
        rxfile( line );             /* otherwise nobatch    */
}
```

(Continued on next page)

```
/*-----------------------------------------------------------------
rxbatch() is the batch mode reception executive.

References Globals: line;
Modifies Globals: line;
-----------------------------------------------------------------*/

VOID rxbatch()
{
   char fcb[ NAMESIZE + 1 ];     /* used by unparse()            */
   char name[ NAMESIZE + 5];     /* unparsed name                */

        while( TRUE )
        {
                while( rxstat() )       /* gobble trash         */
                        rx();
                printf( "\nWaiting for file name:  " );
                rxname( fcb );  /* receive file name into fcb   */
                unparse( name, fcb );   /* make "standard"      */
                if( line[ 1 ] == ':' )
                        line[ 2 ] = '\0';
                else
                        line[ 0 ] = '\0';
                strcat( line, name );
                rxfile( line );         /* receive file         */
        }
}


/*-----------------------------------------------------------------
rxname() loads a CP/M style fcb with a filename from remote
sender. On receipt of an EOT instead of a name character,
aborts to master menu via longjmp().
-----------------------------------------------------------------*/

VOID rxname( fcb )
   char *fcb;                   /* points to CP/M style fcb     */
{
   char *fcbptr;                /* index to fcb                 */
   register int i;              /* scratch counter              */
   int ch;                      /* scratch char (must hold ERR) */
   char chksum;                 /* checksum accumulator         */

        while( TRUE )
        {
                fcbptr = fcb;           /* align index          */
                i = RETRY * 5;          /* retry for name h.s.  */
                while( TRUE )           /* Handshake NAK        */
                {
                        abort();        /* check operator abort */
                        tx( NAK );
                        if( swait() == ACK )
                                break;
                        if( !( i-- ) )
                                error( "Timed out waiting for name" );
                }
                chksum = EoF;           /* init checksum        */
                for( i = 0 ; i < 34 ; i++ )     /* accept noise */
                {
                        if( (ch = swait()) == EOT )
                        {
                                tx( ACK );
                                printf(
                                "\nAll transfers completed" );
                                longjmp( &to_menu, 0 );
                        }
                        if( ch == EoF ) /* End of name chars    */
                                break;
                        if( ch != ERROR )       /* not timeout  */
                        {
                                *(fcbptr++) = ch & 0x7F;
                                putchar( ch );
                                chksum += ch & 0x7F;
                        }
                        abort();        /* operator abort       */
                        tx( ACK );      /* handshake name char  */
                }
                fcb[ NAMESIZE ] = 0;    /* terminate name field */
                do {
                        abort();        /* operator abort       */
                        tx( chksum );   /* handshake checksum   */
                } while( (ch = swait()) == ERROR );
```

(Continued on page 80)

```
                if( ch == ACK )          /* ACK is good name      */
                        return;
                printf(" - Checksum error\nRetrying file name: " );
        }
}


/*-----------------------------------------------------------------
rxfile() receives a file. Input parameter is the name, including
drivspec if any, to save incoming data under. To maintain speed
the function is rather long to reduce function calls where
possible. Have 'crc' set before entry.

References globals: buffer, fd, rec, crcaccum, checksum, crc;
Modifies globals: buffer, rec;
-----------------------------------------------------------------*/

VOID rxfile( name )
   char *name;
{
   char ch;                      /* scratch handshake var        */
   char response;                /* ACK/NAK/CRC handshake        */
   char crcrlo;                  /* rec'd CRC low byte           */
   char crcrhi;                  /* rec'd CRC hi byte            */
   char r1;                      /* current record number        */
   char r2;                      /* 1's comp record number       */
   unsigned int j;               /* wait loop timer              */
   int i;                        /* scratch counter              */
   register char *bptr;          /* buffer index                 */

        while( rxstat() )
                rx();

        make( name );            /* open/make backup             */
        printf( "\n" );
        rec = 1;                 /* uses natural numbering       */
        if( crc )                /* set initial handshake to     */
                response = CRC; /* CRC or checksum               */
        else
                response = NAK;
        while( TRUE )            /* record receive loop          */
        {
                bptr = buffer;           /* align index          */
                printf( "\rWaiting for record %d       ", rec );
                for( i=1 ; i <= RETRY * 5 ; i++ )
                {
                        abort();
                        tx( response ); /* send handshake        */
                        if( (ch = swait()) == SOH )
                                break;  /* SOH indicatees rec    */
                        if( ch == EOT ) /* EOT indicates done    */
                        {
                                printf( "\n%s received OK", name );
                                fclose( fd );
                                tx( ACK );       /* handshake    */
                                return;          /* normal exit  */
                        }
                        if( ch == CAN ) /* Xmit request abort    */
                        {
                                fclose( fd );
                                error("\rReceived cancel request");
                        }
                        if( i == RETRY * 5 )     /* timeout exit */
                                error( "Can't sync to sender" );
                }
                r1 = swait();            /* record number        */
                r2 = swait();            /* 1's comp record #    */
                while( bptr - buffer < RECSIZE ) /* test count   */
                {
                        *(bptr++) = swait();     /* accept char  */
                }
                if( crc )                /* get hibyte CRC       */
                        crcrhi = swait();
                crcrlo = swait();        /* lobyte CRC or chksm  */
                response = NAK;          /* init response        */
                if( (~r1 & 0xFF) != (r2 & 0xFF) )
                {
                        printf( "- Record numbers don't match\n" );
                        continue;
                }
                clrcrc();                /* calc checksum/CRC    */
                for( j = 0 ; j < RECSIZE ; j++ )
                        updcrc( buffer[ j ] );
```

(Continued on page 82)

```
                updcrc( 0 );            /* required to finish   */
                updcrc( 0 );            /* off CRC - why?       */
                if( crc )               /* CRC test             */
                        if( (crcrlo + (crcrhi << 8)) != crcaccum )
                        {
                                printf( "- Bad CRC received\n" );
                                continue;
                        }
                if( !crc )              /* checksum test        */
                        if( crcrlo != checksum )
                        {
                                printf("- Bad checksum received\n");
                                continue;
                        }
                if( (rl == (rec - 1) & 0xFF ) ) /* duplicate?   */
                {
                        printf( " - Received duplicate record\n" );
                        response = ACK; /* dup is OK - ACK was  */
                        continue;       /* trashed - ignore it  */
                }
                if( rl != (rec & 0xFF) )/* fatal sequence error */
                        error( "File record numbering error" );
                rec++;                  /* bump record count    */
                for( j = 0 ; j < RECSIZE ; j++ ) /* write data  */
                        putc( buffer[ j ], fd );
                response = ACK;         /* normal loop end      */
        }
}


/*-----------------------------------------------------------------
make() opens a file for output. It renames any existing file
with the same name to ".BAK", and erases any previous backup.
Input parameter is filespec in "normal" (compressed) format.
Note: file rename by the link() function isn't supported
by several CP/M C Compilers, and others call this function
rename(), or reverse the arguments. At any rate, you can determine
how to handle backups in your own system and take any
appropriate action.

References globals: fd;
Modifies globals: fd;
-----------------------------------------------------------------*/

VOID make( name )
   char *name;
{
#ifdef LINK
   char bak[ NAMESIZE + 5 ];    /* holds .BAK filespec          */

                                /* does this name exist?        */
        if( (fd = fopen( name, "r" )) != NULL )
        {
                fclose( fd );   /* we were just checking        */
                strcpy( bak, name );    /* save old name        */
                if( index( name, '.' ) )/* extension spec'ed?   */
                        strcpy( index( bak, '.' ), ".BAK" );
                else
                        strcat( bak, ".BAK" );
                unlink( bak );  /* dump any old .BAK file       */

                link( bak, name ); /* rename new .BAK file      */
        }
#endif
        if( (fd = fopen( name, "wb" )) == NULL )
                error( "Can't create file - Aborting" );
}


/*-----------------------------------------------------------------
parse() expands  non - ambiguous filespecs to the
standard CP/M fcb format, excluding drive byte.
Sets the globals
'batch' if ambigous, and 'driv' for drivespecs.
Inputs are "normal" filespec (inspec), and exanded fcb.
-----------------------------------------------------------------*/

char *parse( inspec, fcb )
   char *inspec, *fcb;
{
   register int i;                      /* fcb index           */
   int inptr;                           /* input spec index    */
```

(Continued on page 84)

```
        for( i = 0 ; i < NAMESIZE ; i++ )/* blank fill name      */
                fcb[ i ] = ' ';
        if( inspec[ 1 ] == ':' )        /* check for drivspec    */
                inptr = 2;              /* point past drivespec */
        else
                inptr = 0;              /* index to start        */
        i = 0;                          /* pointer into fcb      */
        while( TRUE )
                switch( inspec[ inptr++ ] )
                {
                        case '\0':      /* end of input spec     */
                                fcb[ NAMESIZE ] = 0;
                                return( fcb );
                        case '.':       /* extension spec'ed     */
                                i = NAMESIZE - 3; /* extension  */
                                break;
                        default:
                                if( i < NAMESIZE )
                                        fcb[ i++ ] = toupper(
                                        inspec[ inptr - 1 ] );
                }
}


/*-----------------------------------------------------------------
fillbuf() loads the I/O buffer with a record from the input file.
Returns TRUE if data was available, FALSE if no data left.

References globals: buffer, fd;
Modifies globals: buffer;
-----------------------------------------------------------------*/

int fillbuf()

{
   register int i;              /* scratch counter               */
   int errorchk;                /* holds EOF in Eco C            */

        for( i = 0 ; i < RECSIZE ; i++ )
        {
                if( (errorchk = getc( fd)) == EOF )
                        break;
                buffer[ i ] = errorchk;
        }
        if( i == 0 )                        /* no data read      */
                return( FALSE );
        for( ; i < RECSIZE ; i++ )
                buffer[ i ] = 0;            /* zero fill at EOF  */
        return( TRUE );
}


/*-----------------------------------------------------------------
unparse() reassembles a filename from a CP/M fcb into "normal" or
cmpressed form, so that our C functions can deal with them.
Inputs are a pointer to a string to receive the name (name), and
a pointer to a CP/M style fcb entry (buf).
-----------------------------------------------------------------*/

char *unparse( name, buf )
   char *name, *buf;
{
   register int i;              /* 'name' index                  */
   int j;                       /* 'buf' index                   */

        i = 0;                  /* 'driveless' name     */
        for( j = 0 ; j < NAMESIZE ; j++ )/* transfer chars       */
        {
                if( buf[ j ] != ' ' )   /* (skip spaces)         */
                        name[ i++ ] = buf[ j ] & 0x7F;
                if( j == NAMESIZE-3 )   /* don't forget dot      */
                        name[ i++ ] = '.';
        }
        name[ i ] = '\0';               /* terminate string      */
                                        /* eat terminal dot      */
        if( *(index( name, '.' ) + 1) == '\0' )
                *(index( name, '.' )) = '\0';
        return( name );                 /* return pointer        */
}


/*-----------------------------------------------------------------
clrcrc() clears the crc accumulator. Not much to it, actually.
```

(Continued on page 86)

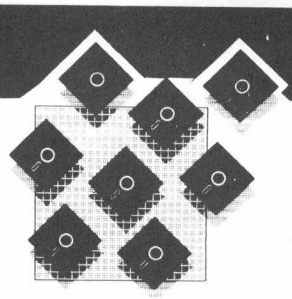
```
References globals:
Modifies globals: crcaccum, checksum;
-----------------------------------------------------------------*/

VOID clrcrc()

{
        crcaccum =
        checksum = 0;
}

/*-----------------------------------------------------------------
updcrc() updates the crc accumulator, if 'crc' is TRUE, else
updates the checksum.
'x' is the byte to be added to CRC or checksum.
CCITT polynomial.

References globals: crc;
Modifies globals: crcaccum, checksum;
-----------------------------------------------------------------*/

VOID updcrc( x )
   char x;
{
   unsigned shifter, i, flag;

        if( crc )
        {
                for( shifter = 0x80 ; shifter ; shifter >>= 1 )
                {
                        flag = (crcaccum & 0x8000);
                        crcaccum <<= 1;
                        crcaccum |= ((shifter & x) ? 1 : 0);
                        if( flag )
                                crcaccum ^= 0x1021;
                }
        }
        else
                checksum += x;
}


/*-----------------------------------------------------------------
abort() scans console for ^X abort code. Exit via error().
System Dependent. If not supportable, convert to null function
or delete and use a preprocessor macro to make it go away.
-----------------------------------------------------------------*/

VOID abort()
{
        if( _bdos( 6, 0xFF ) == 'X'-'@' )
                error( "\rOperator requested abort" );
}


/*-----------------------------------------------------------------
error() does error abort cleanup

References globals: fd;
-----------------------------------------------------------------*/

VOID error( str )
   char *str;
{
        fclose( fd );                   /* dump any open files  */
        tx( CAN );                      /* alert remote         */
        putchar( BELL );                /* alert operator       */
        printf("\n\nError - %s\nPress any key to continue", str );
        getchar();                      /* pause                */
        longjmp( &to_menu, 0 );         /* rejoin menu          */
}


/*-----------------------------------------------------------------
sleep() does a short delay to account for transmission line
latency, etc.
-----------------------------------------------------------------*/

VOID sleep()
{
   register unsigned int i;

        for( i=0 ; i < MAGIC_NUMBER ; i++ )
                ;
}
```

(Continued on page 88)

```
/*-----------------------------------------------------------------
wait() waits for a character - timeout built in. Timeout
condition causes error return to menu.
Input parameter 'time' controls duration of wait.
-----------------------------------------------------------------*/

int wait( time )
   int time;
{
   register unsigned int i;             /* loop timer           */
   int j;                               /* timeout count        */

        j =                             /* timeout count        */
        i = 0;                          /* inner timeout count  */
        while( !rxstat() )
                if( i++ ) MAGIC_NUMBER * 3 )/* about 1.5 secs   */
                        if( !(time--) ) /* check retry count    */
                                error( "Receiver timed out" );
                        else            /* more tries available */
                        {
                                abort(); /* scan for ^X         */
                                if( !j ) /* \n first timeout    */
                                        printf( "\n" );
                                printf( "\rtimeout %d   ", ++j );
                                i = 0;
                        }
        return( rx() );                 /* send back char       */
}

/*-----------------------------------------------------------------
swait() does a short wait for a char from rx, returns ERROR on
a timeout, or char received.
-----------------------------------------------------------------*/

int swait()
{
   register unsigned int i;             /* loop timer           */

        i = 0;
        while( i++ < MAGIC_NUMBER )     /* loop timing test     */
                if( rxstat() )          /* anything yet?        */
                        return( rx() ); /* yes, return char     */
        return( ERROR );                /* timeout exit         */
}


/*-----------------------------------------------------------------
waitcan() waits for a character - If recv's CAN aborts to main
menu. Uses wait() for timing loop, so timeout also returns
to main menu. Input argument 'time' controls duration of wait.
-----------------------------------------------------------------*/

int waitcan( time )
   int time;
{
   char ch;

        if( (ch = wait( time )) == CAN )
                error( "Received cancellation request" );
        return( ch );
}


/*-----------------------------------------------------------------
=================================================================
The following functions are machine dependant and should be
modified to run on your machine.
=================================================================
-----------------------------------------------------------------*/


/*-----------------------------------------------------------------
init() initializes globals. In addition, UART initialization
goes here as well, if required. No input parameters. May have
some machine dependant stuff here.

References globals:
Modifies globals: crc, batch, buffer, line;
-----------------------------------------------------------------*/

VOID init()
{
        buffer = alloc( RECSIZE );      /* transfer buffer      */
```

```
        line = alloc( 81 );             /* scatch input line    */
        crc =                           /* CRC mode             */
        batch = TRUE;                   /* not really required  */
}


/*-------------------------------------------------------------
rx() receives a char from uart. No timeout checking is performed.
Also, no status check is done. Bad news if you call this without
checking status first... Done that way to speed up things.

                        ZORBA Version
-------------------------------------------------------------*/

int rx()
{
        return( inp( RXDATA ) );
}


/*-------------------------------------------------------------
rxstat() returns TRUE if char waiting for input.

              ZORBA Version (interrupt fifo)
-------------------------------------------------------------*/

int rxstat()
{
        return( inp( UARTCMD ) & 2 );
}


/*-------------------------------------------------------------
tx( ch ) transmits the char 'ch', and calls the CRC/checksum
update. A timeout is built in for transmission faults.

                        ZORBA Version
-------------------------------------------------------------*/

VOID tx( ch )
   char ch;
{

   register unsigned int i;             /* timing loop counter  */

        i = 0;
        while( !txstat() )              /* ready?               */
                if( i++ > MAGIC_NUMBER )/* loop about 3 secs    */
                        error( "UART not ready??" );
        outp( TXDATA, ch );             /* to UART              */
        updcrc( ch );                   /* stir CRC a little    */
}


/*-------------------------------------------------------------

txstat() returns TRUE if tx register is empty

                        ZORBA Version
-------------------------------------------------------------*/

int txstat()
{
        return( inp( UARTCMD ) & 4 );
}


/*-------------------------------------------------------------
quit() does exit processing.
-------------------------------------------------------------*/

VOID quit()
{
        exit( 0 );
}
```

End Listing



Envoy, Version 1.1

Company: Artisoft Inc., 2450 E. Speedway, Suite 4, Tucson, AZ 85719 (602) 327-4305
Computer: IBM PC family, 16-bit compatibles, 8-bit CP/M
Price: $49.95
Circle Reader Service No. 144
Reviewed by David W. Carroll

The trend in microcomputer software is toward high-volume, high-quality, low-cost software (under $100). Borland International pioneered this "volks-software" concept with Turbo Pascal and Sidekick, and other companies are now following this path.

Commercial telecommunications software typically costs $200 and more, but Artisoft of Tucson, is now offering its Envoy communications package with versions that run on almost any microcomputer system for only $49.95.

"With over a dozen major commercial programs available, who needs another?" you might well ask. First, the price of Envoy represents a breakthrough. Second, the program is small, fast, and easy to use. Third, Envoy offers some unique features. Finally, compatible versions are available for almost any type of microcomputer and operating system.

Operation

Envoy is a menu-driven communications program. This means that it has two modes: command and terminal. In command mode, the current menu is shown on the screen. In terminal mode, you are on line with your serial port or modem, and Envoy is transparent to all keyboard input except for your terminal mode "exit" character, usually escape <Esc>. Envoy requires you to set your terminal mode exit character when you start each session. The program's main menu appears below:

*** Main Menu ***

<T> — Smart Terminal
<E> — ENVOY file transfers
<X> — XMODEM file transfers
<P> — Port Parameters
<U> — Utilities
<Ret> — Previous menu
<Esc> — Exits ENVOY

Enter your selection –>

The program offers dumb terminal operation, ASCII file send and buffer capture, and protocol file transfers. Envoy supports both the Ward Christensen XMODEM protocol and the little known ANSI standard X3.28 file transfer protocol.

The ANSI protocol is designed so that either computer in a data link can act as master while the other is a slave. This allows multiple file transfers and remote system utility operation (you can use the remote system's utility menu). The transfer protocol is faster than XMODEM and provides greater error checking.

This menu appears when you select Smart Terminal:

*** Smart Terminal ***

Capture Buffer is Off

<T> — Terminal Mode
<S> — Send Text File
<M> — Modem Functions
<O> — Terminal Mode Options
<C> — Capture Buffer On
<B> — Buffer Utilization
<W> — Write Buffer to Disk
<D> — Display Buffer
<U> — Utilities
<Ret> — Previous menu
<Esc> — Exits ENVOY

Enter your selection –> B

Capture Buffer 57362 bytes
Remaining Space 57362 bytes

[Enter any key to continue]

Envoy uses an autodial menu and even supports simple "script" files for autologon. It will support almost any modem because of its free-form modem command/phone number directory. The directory consists of a file with entries created using a word processor or editor (like WordStar or EDLIN). A few typical directory entries for a Hayes Smartmodem follow:

Redial [A/|]
Disconnect [_+++_ATH0]
Information [ATDT1-555-1212]

Here is a sample directory listing:

*** Modem Functions ***

<A> — Redial
<B> — Disconnect
<C> — Information
<D> — ARTISOFT, Inc.
<E> — CompuServe
<Ret> — Previous menu
<Esc> — Exits ENVOY

Enter your selection –>

Envoy scripts can include keyboard control characters, delays, and character-matching strings. The special characters ^, _, and | provide these features when inserted in script entries. Question marks indicate strings to match.

You can set the terminal mode

Copyright by David W. Carroll

communications parameters, which include stripping line feeds, local echo mode, and delay of up to 9 sec between each line sent for ASCII text file transfers:

*** Terminal Mode Options ***

<L> — Send LFs On
<D> — Delay Lines (0)
<E> — Local Echo On
<X> — Change Exit Key
<Ret> — Previous menu
<Esc> — Exits ENVOY

Enter your selection –>

You may also set port parameters (speed, parity, etc.) from within Envoy.

Full operating system utilities are available from within Envoy, and PC/MSDOS users have full access to DOS 2.0 subdirectory commands and paths. The program even supports remote system operation (unattended). This menu is for the IBM PC version of Envoy PC:

*** Utilities ***

<D> — Directory
<L> — Log to another disk
<E> — Erase files
<T> — Type files
<C> — Copy a file
<R> — Rename a file
<P> — Print files
<N> — Change directory path
<S> — Create a directory path
<X> — Remove a directory path
<Ret> — Previous menu
<Esc> — Exits ENVOY

Enter your selection –>

Envoy is written in assembly code; that means it is fast and compact. The program takes a total of 7–9K disk space, depending on the version. That is small enough to put on every floppy—not much more space than the old standby MODEM7 program requires—and Envoy offers many more features.

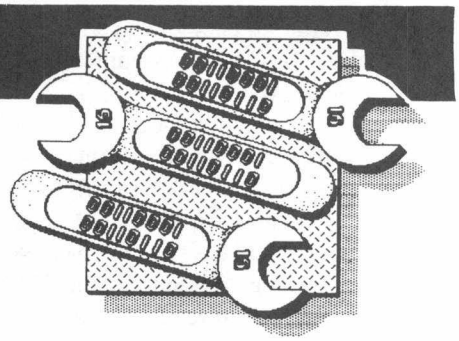
Evaluation

Envoy PC operates reliably in all modes. My only complaints were regarding the effort required to make new directory entries. I found it clumsy to exit the program, start up a word processor, and enter numbers in the directory file. The lack of a real-time directory entry mode, similar to what Crosstalk and Smartcom II offer, was irritating. I also didn't like to reenter the required modem commands in each directory entry, though this is a minor point.

Overall, the Envoy communications programs are well designed, easy to use, and versatile. For a reasonable price, they provide virtually every communications feature you could require.

DDJ

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

IBM PC WordStar and the HP LaserJet

We at Laboratory Microsystems recently purchased an HP LaserJet printer to produce our various software manuals. After buying the LaserJet, we were dismayed to find that it is essentially impossible to configure IBM PC WordStar to properly use the printer's boldface, italics, and various fonts. WordStar allows the user to patch in only four-character escape sequences, while nearly every important LaserJet control sequence is five bytes or longer. When we contacted the HP dealer about this problem, his response was, "Well, you'll just have to switch to Spellbinder; it supports all the LaserJet capabilities."

Because we have around 10 Mb of documentation files that we would have to convert if we switched word processors, this option didn't sound too attractive. My first attempt at getting around the problem was to write a character filter that read a WordStar document file, translated the various control codes (such as ^B) into the proper LaserJet escape sequences, and sent the result to the printer. This worked acceptably as far as print quality, but it required an exit from WordStar whenever you wanted to print something, which made WordStar's capability of background printing while editing useless.

It began to look as if we would have to face the facts and do some heavy-duty disassembly and patching of WordStar. On a long shot, I dropped a note to MicroPro Technical Support, asking them if they knew of anyone who had already worked out the necessary patches. Much to my amazement, I got an immediate phone call from a nice lady at MicroPro who said that a LaserJet version of WordStar was already available to licensed users for $25.00. I duly sent off my check and in a few days received a diskette containing a new WS.OVL file and a batch file that performs some patches to the WS.COM file.

After following the installation steps in the accompanying documentation, I ended up with a WordStar that supports italics, boldface, double-strike, underline, over-strike, micro-justification, superscripting, subscripting, variable character pitch, variable line height, and multiple copies on the HP LaserJet. It's a little slow (only 2–3 pages/min throughput) and has a few bugs (a ^Y at the start of a line makes it go crazy), but it works pretty well. You can get this enhancement disk by sending a polite letter, including your WordStar version 3.3 serial number and $25.00, to LaserJet Technical Support, MicroPro International Corp., 33 San Pablo Ave., San Rafael, CA 94903.

Microsoft 8086 MacroAssembler 2.0

Many moons after Microsoft announced version 2.0 of the 8086 MacroAssembler, it has appeared in the IBM Product Centers in the last few weeks without fanfare. Along with the new, higher version number comes a new, higher price ($175.00). Silly me. I thought that because I had bought version 1 with its zillions of bugs I might be entitled to a reduced cost update, but the Product Center salesman nearly killed himself laughing when I brought *that* up. Finally, gritting my teeth and hoping that the new version would be a little more robust than the old, I hauled out my well-worn MasterCard and took home a copy.

If you choose to go the same route, what will you get for your money? Well, the first thing you get is a MacroAssembler that has an even more severe case of bloat than the first version (the EXE file is 76,544 bytes compared to 67,584 bytes for version 1). The 8087 mnemonics are now fully supported, along with most of the additional 80286 mnemonics that would be used in that CPU's real mode. The performance of the MacroAssembler has improved considerably: a 106K source file (approximately 3200 lines), which required 13 min 57 sec to assemble with version 1, is assembled in 4 min 13 sec with version 2.

Version 2 provides a new version of the Linker (which has also suffered more bloat to the tune of about 6K), a documented Library manager, and a new program called SALUT (Structured Assembly Language Utility), which converts an assembly language source file containing high-level control structures into a source file that can be fed to the assembler. Although IBM pushes SALUT pretty hard in the MacroAssembler manual, I see no evidence in the BIOS listings or distributed PCDOS assembler source files (such as VDISK.ASM) that IBM programmers ever use it. Perhaps IBM is adopting some of Apple's policies for "the rest of us."

The manual for the MacroAssembler has been drastically overhauled. The previous "encyclopedia" format, which devoted a page to each of the assembler directives and 8086 mnemonics with examples, has been abandoned. The new manual is heavily oriented to *operation* of the assembler and associated utilities (everything is called "sessions" now) and provides almost no programming guidance whatsoever. A copy of Rector and Alexy's *The 8086 Book* or the Intel *iAPX 86 User's Manual* will be

indispensable to people who start out with the new MacroAssembler and no previous experience.

Some of the known bugs in version 1 appear to have been fixed; these include the operand order in the SHL and SHR operators, the XOR that used to work like OR, the packed decimal data generated by the DT pseudo-op, the failure of DUP to work properly inside the invocation of a STRUC, and the lack of error messages and generation of "funny" opcodes for some nonexistent instructions (such as CMP ES,0). Error detection (and the quality of the error messages) seems generally improved.

Some bugs, however, remain. I've provided a few examples in Listing Two (page 104). A major problem seems to be inconsistent handling of signed and unsigned values by the logical operators. This is "explained" to some extent by a comment in the manual on page 2-28: "When relational operators are used, results are produced by comparing 17-bit signed numbers, where the 17th bit can contain only a sign bit." This accounts for the behavior of the LT and GT operators, but if it were strictly true, the expression "–1 EQ 0FFFFH" would return a false flag (which it doesn't, as you can see by the listing). Anyway, it's a mystery to me why Microsoft thought it should support within expressions a 17-bit data type that has no physical equivalent in the 8086 processor family. Wouldn't it make more sense to have the assembler evaluate expressions to the same results that you would get with in-line machine code?

The TRIM Filter

In the last few months, we have printed three new "filters" for MSDOS: CLEAN, TEE, and TK. In this month's column, we print another filter, named TRIM, contributed by A. K. Head of Melbourne, Australia (Listing One, page 96). I have taken the liberty of retyping and reformatting the program from his original listing, and I have added some error messages; hence, credit for the concept and implementation should go to Mr. Head, while blame for any bugs or other deficiencies specially introduced by publication in *DDJ* is mine.

In spite of its brevity, TRIM is a fairly sophisticated filter that can perform one of several operations:

- Exclude all the characters *outside* a given range of columns from each line of a file
- Exclude all the characters *inside* a given range of columns from each line of a file
- Delete trailing blanks from each line of a file
- Delete blank lines from a file

Documentation for the use of TRIM appears in the beginning of the source listing itself. You can use TRIM in combination with SORT and various other commands by means of the Unix-like pipes of MSDOS 2.x to perform complex operations. For example, the following command line will isolate the filename and extension from each line of a directory listing, sort the list, and leave the result in a file named SORTED.DIR:

```
A>DIR ¦TRIM 1,12¦SORT
    >SORTED.DIR
```

Incidentally, you can download the source for all the programs printed in this column, as well as many other useful MSDOS utilities, from the Laboratory Microsystems Bulletin Board System at (213) 306-3530 (300 or 1200 baud). This BBS is up between 6 PM and 9 AM, Pacific Time, on weekdays and for 24 hours on weekends and holidays. **DDJ**

Listing One

```
        name    trim
        page    55,132
        title   'TRIM - excerpt lines of a file'
;
; TRIM --- excerpts selected columns from each line
;          of a file and writes them to the selected
;          output device or file.
;
; A "filter" for MS-DOS or PC-DOS version 2 or higher,
; after the fashion of Unix.   Reads from the standard input
; (redirectable) and writes to the standard output (redirectable).
; Error messages are directed to the standard error device.
;
; TRIM can be (and usually would be) used in a pipe, e.g.
;
;    | TRIM 7,45 |
;
; transmits only the characters in columns 7 to 45 (inclusive)
; of each line.  A minus sign reverses the action, e.g.
;
;    | TRIM -7,45 |
;
; transmits all characters except those in columns 7 to 45.
;
; Special actions:
;
;    | TRIM 0  |        deletes trailing spaces from lines, and
;    | TRIM -0 |        also discards empty lines
;
; By A. K. Head, 6 Duffryn Place, Melbourne, Australia 3142
; reformatted and error handling added by Ray Duncan
;

command equ     80h             ; buffer for command tail
fcb1    equ     5ch             ; default file control block #1
fcb2    equ     6ch             ; default file control block #2

buflen  equ     16384           ; buffer length, alter to taste

cr      equ     0dh             ; ASCII carriage return
lf      equ     0ah             ; ASCII line feed
ff      equ     0ch             ; ASCII form feed
eof     equ     01ah            ; End-of-file marker
tab     equ     09h             ; ASCII tab code
blank   equ     20h             ; ASCII blank

                                ; DOS 2.x pre-defined handles
stdin   equ     0000            ; standard input file
stdout  equ     0001            ; standard output file
stderr  equ     0002            ; standard error file
stdaux  equ     0003            ; standard auxilliary file
stdprn  equ     0004            ; standard printer file


cseg    segment para public 'CODE'
        assume  cs:cseg,ds:cseg

        org     100H            ; start .COM at 100H

start:  jmp     near ptr trim

param1  dw      0               ; command parameter #1
param2  dw      0               ; command parameter #2
sign    dw      0               ; nonzero if "-" in command
count   dw      0               ; column count, current line
topin   dw      0               ; chars in input buffer - 1
char    db      0               ; current character
```

 (Listing continued, text begins on page 94)

Listing One

```
trim    proc    far

        xor     si,si           ; initialize buffer pointers
        xor     di,di
        mov     bx,fcb1+1       ; addr of parsed parameter 1
        call    getprm          ; convert it
        cmp     ax,0
        je      trunc           ; zero parameter, go truncate
                                ; trailing blanks etc.
        mov     param1,ax       ; save first parameter
        mov     bx,fcb2+1       ; addr of parsed parameter 2
        cmp     byte ptr [bx],blank ; is it present at all?
        jne     trim0           ; yes, proceed
        jmp     err3            ; no, exit
trim0:
        call    getprm          ; convert it
        mov     param2,ax       ; save 2nd parameter
        cmp     ax,param1       ; is end column < start column?
        jnb     trim1
        jmp     err3            ; yes, exit with error message

trim1:  mov     count,0         ; starting a new line,
                                ; initialize column counter

trim2:  inc     count           ; count characters
        call    cin             ; read a character
        mov     al,char         ; is it carriage return?
        and     al,07fh         ; (ignore high bit in case
        cmp     al,cr           ;  this is Wordstar file)
        je      trim5           ; yes, found end of line
        mov     ax,count        ; fetch current char count
        cmp     sign,0          ; is this include or exclude call?
        jne     trim4           ; jump, -, exclude range
                                ; proceed, +, include range
        cmp     ax,param1       ; is column counter within
                                ; desired range?
        jb      trim2           ; no, discard this char.
        cmp     ax,param2
        ja      trim2           ; no, discard this char.
trim3:  call    cout            ; yes, use this character
        jmp     trim2           ; get next char.

trim4:  cmp     ax,param1       ; is column counter outside
                                ; of excluded range?
        jb      trim3           ; yes, use this character
        cmp     ax,param2
        ja      trim3           ; yes, use this character
        jmp     trim2           ; no, discard this character

trim5:                          ; found end of line
        call    cout            ; write carriage return
        call    cin             ; read presumed line feed
        call    cout            ; write line feed
        jmp     trim1
trunc:                          ; come here if zero parameter
                                ; to delete trailing blanks from
                                ; all lines.  If - sign was in
                                ; command parameter, also delete
                                ; empty lines completely.

trunc1: mov     count,0         ; initialize column counter
        xor     bp,bp           ; init line pointer

trunc2: call    cin             ; read a character
        mov     al,char         ; is it carriage return
        and     al,07fh         ; (ignore high bit in case
        cmp     al,cr           ;  this is Wordstar file)
        je      trunc3          ; yes, go process end of line
        mov     al,char         ; transfer char. to forming line
        mov     byte ptr ds:[line+bp],al
```

```
        inc     bp
        cmp     char,blank      ; is character a space code?
        je      trunc2          ; yes, get next char
        mov     count,bp        ; no, update column count
        jmp     trunc2          ; get next char.

trunc3: xor     bp,bp           ; text string now in LINE
        call    cin             ; discard line feed
        cmp     count,0         ; was line empty?

        jne     trunc4          ; no, go output it
        cmp     sign,0          ; deleting empty lines?
        jne     trunc1          ; yes, discard this one
        jmp     trunc5          ; no, send cr-lf sequence

trunc4:                         ; now transfer LINE to BUFOUT
        mov     al,byte ptr ds:[bp+line]
        mov     char,al         ; get next char and
        call    cout            ; send it to output
        inc     bp
        cmp     bp,count        ; entire line sent yet?
        jb      trunc4          ; no, send another char

trunc5: mov     char,cr         ; send carriage return
        call    cout
        mov     char,lf         ; and line feed
        call    cout
        jmp     trunc1

exit:   cmp     di,0            ; output buffer empty?
        je      exit1           ; yes
        call    outbuf          ; no, flush it
exit1:  mov     ax,4c00h        ; exit with return code=0
        int     21h             ; if no errors were encountered

err:                            ; print error message and exit.
                                ; DS:DX = addr of message
                                ; CX    = length of message
                                ; AL    = return code
        push    ax              ; save return code
        mov     ah,40h          ; function 40 = write
        mov     bx,stderr       ; handle for error output
        int     21h
        pop     ax              ; retrieve return code
        mov     ah,4ch          ; function 4C = exit
        int     21h

err1:   mov     dx,offset err1msg  ; print "output device error"
        mov     cx,err1len
        mov     al,1            ; return code = 1
        jmp     err

err2:   mov     dx,offset err2msg  ; print "disk is full".
        mov     cx,err2len
        mov     al,2            ; return code = 2
        jmp     err

err3:   mov     dx,offset err3msg  ; print "bad parameter"
        mov     cx,err3len
        mov     al,3            ; return code = 3
        jmp     err

err4:   mov     dx,offset err4msg  ; print "input device error"
        mov     cx,err4len
        mov     al,4            ; return code = 4
        jmp     err

trim    endp

cout    proc    near            ; output contents of "char"
                                ; with autobuffering
        mov     al,char
        mov     byte ptr [di+bufout],al
        inc     di
```

 (Listing continued, text begins on page 94)

Listing One

```
        cmp     di,buflen       ; buffer full yet?
        jb      coutl
        call    outbuf          ; write buffer
coutl:  ret                     ; back to caller
cout    endp
outbuf  proc    near            ; write buffer to std output
        mov     ah,40h          ; function 40 = write
        mov     bx,stdout       ; predefined handle
        mov     cx,di           ; number of characters
        lea     dx,bufout       ; DS:DX = buffer addr
        int     21h             ; request DOS service
        jc      err1            ; jump, device write error
        cmp     ax,di
        jne     err2            ; jump, disk is full
        xor     di,di           ; initialize output buff pointer
        ret                     ; back to caller
outbuf  endp

cin     proc    near            ; input next char with buffering
        inc     si              ; bump input buffer pointer
        cmp     si,topin        ; buffer exhausted?
        jbe     cin2            ; no, jump
        mov     ah,3fh          ; yes, read some more data
        mov     bx,stdin        ; predefined handle
        mov     cx,buflen       ; max length to read
        lea     dx,bufin        ; DS:DX = input buffer addr
        int     21h             ; request DOS service
        jc      err4            ; jump, input device error
        cmp     ax,0            ; end of file?
        jne     cin1
        jmp     exit            ; yes, goto success exit point
cin1:   dec     ax              ; save offset of top of data
        mov     topin,ax
        xor     si,si           ; zero input buffer pointer
cin2:   mov     al,byte ptr [si+bufin]
        mov     char,al         ; get next char
        ret
cin     endp

getprm  proc    near            ; convert numeric parameter to
                                ; binary and return it in AX
        xor     ax,ax           ; initialize forming answer
        mov     cl,[bx]         ; get first char
        cmp     cl,'-'          ; is it minus sign?
        jne     getp2           ; no, jump
        inc     sign            ; yes, set flag and
getp1:  inc     bx              ; bump command string pointer
                                ; past the "-" sign
        mov     cl,[bx]         ; get next char
getp2:  cmp     cl,'0'          ; at least 1 legal digit?
        jb      getp6           ; no, exit
        cmp     cl,'9'
        ja      getp6           ; no, exit
        jmp     getp4
getp3:  inc     bx              ; advance through string
        mov     cl,[bx]
        cmp     cl,'0'          ; make sure legal digit 0-9
        jb      getp5           ; not digit, jump
        cmp     cl,'9'
        ja      getp5           ; not digit, jump
        mov     dl,10           ; previous answer * 10
        mul     dl
getp4:  sub     cl,'0'          ; add in the new digit
        xor     ch,ch
        add     ax,cx
        cmp     ah,0            ; new answer > 255?
        je      getp3           ; no, keep converting
        jmp     err3            ; yes, illegal parameter, exit
getp5:  cmp     byte ptr[bx],blank ; if not digit, must be blank
```

```
        jne     getp6           ; exit, bad parameter
        ret                     ; back to caller
getp6:  jmp     err3            ; ... since too far to reach
getprm  endp                    ; direct with conditional branch

err1msg db      cr,lf
        db      'trim: output device error'
        db      cr,lf
err1len equ     (this byte)-(offset err1msg)

err2msg db      cr,lf
        db      'trim: disk is full.'
        db      cr,lf
err2len equ     (this byte)-(offset err2msg)

err3msg db      cr,lf
        db      'trim: bad parameter'
        db      cr,lf
err3len equ     (this byte)-(offset err3msg)

err4msg db      cr,lf
        db      'trim: input device error'
        db      cr,lf
err4len equ     (this byte)-(offset err4msg)

bufin   equ     this byte       ; data is read here
                                ; from the standard input

bufout  equ     bufin+buflen    ; data to be written to
                                ; standard output is built here

line    equ     bufout+buflen   ; temporary line buffer

cseg    ends

        end     start
```

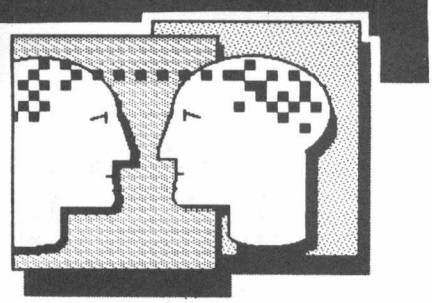
End Listing One

Listing Two

```
1                                       name    optest
2                                       page    55,132
3                                       title   'OPTEST -- test MASM bugs'
4                                   ; Demonstrate some anomalies in the Macro Assembler v. 2.
5                                   ; This was run on MASM.EXE file dated 7-18-84
6
7       0000                        cseg    segment para public 'CODE'
8                                       assume  cs:cseg,ds:cseg,es:cseg
9       0100                            org     100h
10                                  ;
11                                  ; SHR and SHL don't distinguish between signed and unsigned data
12      0100  0001                       dw      -2 shr 1
13      0102  0001                       dw      2 shr 1
14      0104  0002                       dw      -1 shl 1
15      0106  0002                       dw      1 shl 1
16                                  ;
17                                  ; EQ and NEQ seem to recognize the decimal and hex equivalents...
18      0108  FFFF                       dw      -1 eq 0ffffh
19      010A  0000                       dw      -1 ne 0ffffh
20                                  ;
21                                  ; but the comparison operators don't ...
22      010C  FFFF                       dw      1 gt -1
23      010E  0000                       dw      1 gt 0ffffh
24      0110  0000                       dw      1 lt -1
25      0112  FFFF                       dw      1 lt 0ffffh
26                                  ;
27                                  ; on MOV instructions, if source operand is missing, no
28                                  ; error message is produced and machine code is generated
29                                  ; as though the source operand were an immediate "0".
30      0114  3D 0000                    cmp     ax,
31
32                                  ; generates the same machine code as
33      0117  3D 0000                    cmp     ax,0
34                                  ;
35                                  ; Assembler generates erroneous machine code when last digit of a
36                                  ; literal is "B" or "D" and the default radix is hexadecimal
37      0010                             .radix  16
38      011A  83 EB 0A                   sub     bx,0a
39      011D  83 EB 00                   sub     bx,0b   ; generates same code as SUB BX,0
40      0120  83 EB 0C                   sub     bx,0c
41      0123  83 EB 00                   sub     bx,0d   ; generates same code as SUB BX,0
42      0126  83 EB 0E                   sub     bx,0e
43
44      0129                        cseg    ends
45                                      end
```

End Listings

CP/M EXCHANGE



by Robert Blum

Alternate Console for SID

Necessity was the mother of invention that spurred Bridger Mitchell of Plu*Perfect Systems to develop this useful modification for ZSID, SID, or DDT. Apparently during the development and debugging of a CRT screen-handling routine, Mitchell realized that he had to find an alternate path for the messages coming from the debugging program. Each time SID displayed a message, it altered the context of the original screen, making it practically impossible to decide whether the screen-handling routine was functioning properly. Even worse, performing an instruction trace completely destroyed the screen contents: as each instruction executed, a new line of information from SID was displayed, scrolling yet another data line off the top of the CRT screen, until the screen was filled with nothing but messages from the debugger. This was no help at all in determining how the routine under test was interacting with the CRT.

Mitchell developed SID2TTY to provide a standard way of interacting with the BDOS to direct application program messages to one display device while providing an additional path for any messages coming from the debugging program. This program dynamically patches the SID debugger to redirect all of its console input and output to the logical TTY: device while allowing the program under test to continue using the CRT: device undisturbed. If you are using either ZSID or DDT, you should use ZSID2TTY and DDT2TTY, respectively, to patch the debugger.

By using a second terminal as the TTY: device to "command" SID, you can debug programs without interfering with standard console input and output. Assuming the usual CON: =CRT: device assignment, a program's standard output will appear on the video monitor exactly as if the debugger were not in use; program input is entered on the standard keyboard. All debugger input comes from the TTY: console, and debugger output appears on the TTY: video monitor. The patch in the Listing (page 107) is only 23 bytes long, the remainder of the code providing the necessary relocation and memory protection services.

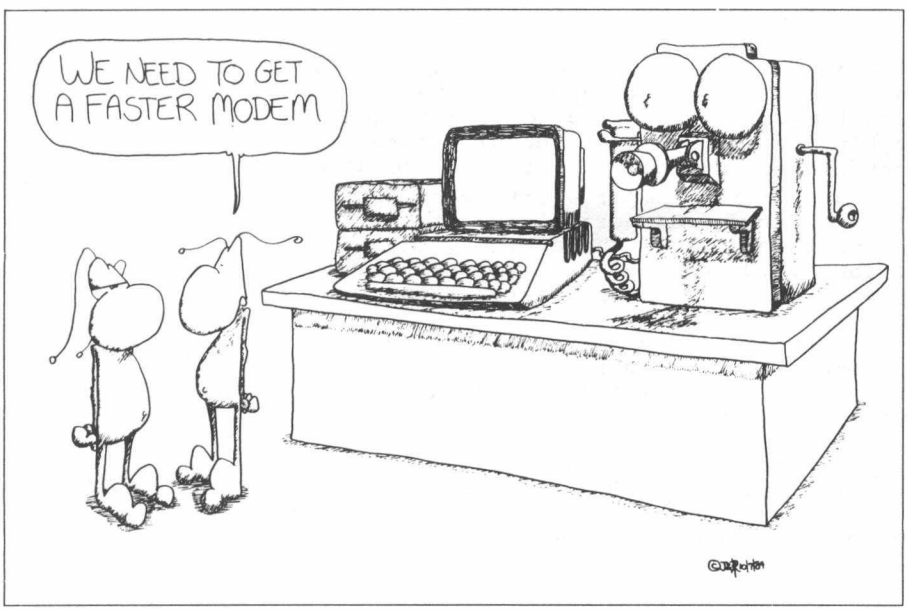
In operation, each time SID calls for BDOS console I/O service, SID2TTY changes the IOBYTE to reflect a console assignment of TTY: and restores it to the original value afterwards. To protect high memory, the patch is relocated below SID and also below any .SYM and .UTL files, which should be loaded first. Operating procedures are well documented within the program source listing.

CP/M Plus Inaccuracy

The CP/M Plus documentation states that the fourth field of a RSX header, labeled PREVIOUS, contains a 16-bit pointer to the previous RSX in the chain or to memory location 5 if no other RSXs are active. This suggests that the PREVIOUS pointer, in fact, always points to the NEXT jump instruction of the previous RSX in the chain. Thus, compatibility is maintained throughout the system, and each RSX is chained in both forward and backward directions.

In actuality, though, this pointer is to location 7, the last byte of the jump NEXT field in the previous RSX, or to memory location 7 if no other RSXs are active. This inaccuracy has no effect on CP/M Plus's operation or on any RSX currently supplied by DRI. But beware if you are planning to write any custom RSXs that might use reverse chaining.

DDJ



Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 198.**

CP/M Exchange Listing

```
;       SID2TTY.ASM v 1.1  01/10/84     B. Mitchell (Plu*Perfect Systems)
;
;
;           A Debugging Console for SID, ZSID and DDT
;
;SID2TTY is a small, public-domain program that
;dynamically patches the SID debugger (Data Resources, Inc.) to
;redirect all of SID's console input and output to the logical TTY:
;device.  (ZSID2TTY and DDT2TTY are the ZSID and DDT versions.)
;
;By using a second terminal as the TTY: device to "command" SID,
;one can debug programs without interfering with standard console
;input and output.  Assuming the usual CON:=CRT: device assignment,
; a program's standard output will appear on the video
;monitor exactly as if the debugger were not in use, and program
;input will be entered on the standard keyboard.  All debugger
;input comes from the TTY: console, and debugger output appears
;on its video monitor.
;
;The patch itself is just 23 bytes -- it changes the IOBYTE each time
;SID calls for BDOS console i/o service and restores it afterwards.
;The patch is relocated to protected high memory, below SID (and
;below any .SYM and .UTL files, which should be loaded first).
;
;Running SID2TTY will display a help message.
;
;Source file:   SID2TTY.ASM for assembly with ASM or MAC.
;               The ZSID and DDT versions are obtained by
;               changing three equates.
;
;Requirements:  CP/M 2.2 - 3.0 with IOBYTE implemented in bios.
;               8080, 8085 or Z80 CPU.
;               SID v. 1.4 or ZSID v. 1.4 or DDT v2.2.
;               External terminal connected to TTY:.
;
;Installation:  Set the equates for SID, ZSID or DDT.
;               If your TTY: device needs initialization or
;               de-initialization, insert the code in place
;               of dummy routines.
;
;Bugs:          Exiting from SID with ^C leaves the console
;               redirected to TTY: (unless the BIOS improperly
;               restores the IOBYTE on warmboots).  The
;               recommended fix is simply to exit with a "G0".
;               Should you happen to hit ^C, reload SID and reset
;               the IOBYTE to its original value, then exit with
;               ^C from the CRT terminal.
;
;Remarks:       1. Although it would be possible to trap the
;               warmboot and restore the IOBYTE, it's difficult to
;               do correctly.  Trapping 0001h violates the basic CP/M
;               addressing convention, making it impossible to
;               run other resident modules that need to locate the BIOS
;               and BDOS.  Trapping the warmboot at the BIOS jump
;               vector does preserve addressing, but it still requires
;               special efforts to maintain compatibility with any higher
;               resident modules, and the trap must remove itself
;               without disturbing them.  Since the application here
;               is manual, interactive debugging, it seems best to
;               favor compatibility and do a manual restoration.
;
;               2. The assembler portion of SID/ZSID can be removed
;               with the "-A" command to gain 680h bytes, but
;               this must be done BEFORE installing the patch
;               with the "g103" command.
;
;Version:       1.0     --      10 October 1984 B. Mitchell
;               1.1     --      10 January 1985 B. Mitchell
;
;Revisions:     Please forward any revisions and improvements
;               to the author.
```

(Continued on next page)

```
;
;Author:         Bridger Mitchell
;               Plu*Perfect Systems
;               Box 1494, Idyllwild CA 92349
;               (714) 659-4432
;
;Note:           Not tested on CP/M 3.0.
;
;
VERS    equ     1$1
;
FALSE   equ     0
TRUE    equ     NOT FALSE
ADDRESS equ     0FFFFH
;
; exactly ONE of the next 3 equates must be TRUE --
;
SID     equ     TRUE
ZSID    equ     FALSE
DDT     equ     FALSE
;
CR      equ     0dh
LF      equ     0ah
BELL    equ     07h
;
IOBYTE  equ     0003h
BDOS    equ     0005h
;
        org     100h
START:  jmp     HELP
        jmp     INSTALL
        jmp     DEINITTTY
;
signature:db    'NEXT  PC'      ;common SID/ZSID/DDT code
siglen  equ     $-signature

        IF SID
;
modsiz  equ     1800h   ;size of debugger module
sigad   equ     0E42h   ;offset to signature
callbd  equ     06A4H   ;offset to call_bdos routine
fn2ad   equ     1004h   ;offset to addr of function 2 bdos call
fn10ad  equ     0FF2H   ; ditto fn 10
fn11ad  equ     1065h   ; ditto fn 11

;
usage:  db      CR,LF,'SID2TTY v. '
        db      vers/10+'0','.',(vers mod 10)+'0'
        db      '  --  Redirect SID console i/o to TTY:',CR,LF
        db      '          -- Usage --',CR,LF
        db      'Load SID, then .SYM and .UTL files',CR,LF
        db      '  #I* filename.SYM',CR,LF
        db      '  #R',CR,LF
        db      '  #ISID2TTY.COM',CR,LF
        db      '  #R',CR,LF
        db      '  #G103       ==>> SID''s console is now TTY:',CR,LF
        db      '  #Ifilename.COM  (or .HEX) to debug',CR,LF
        db      '  #R',CR,LF
        db      '  #D3,3       ==>> display IOBYTE for reference',CR,LF,LF
        db      'To exit from SID, deinitialize the TTY: (if needed) by',CR,LF
        db      '  #ISID2TTY.COM',CR,LF
        db      '  #R0',CR,LF
        db      '  #G106',CR,LF
        db      'Then use "G0" (not "^C") to exit from SID.',CR,LF
        db      '(This message can be displayed under SID by the command:'
        db      CR,LF,'   G100  or   C100 before loading ''filename'')'
        db      CR,LF,LF,'$'
;
errmsg: db      CR,LF,BELL,'Can''t find SID v 1.4!$'
        ENDIF   ;IF SID

        IF ZSID
```

(Continued on page 110)

CP/M Exchange Listing (Listing continued, text begins on page 106)

```
;
modsiz  equ     2200h
sigad   equ     1682h
callbd  equ     0EAFH
fn2ad   equ     1880h
fn10ad  equ     186Eh
fn11ad  equ     18E1h
;
usage:  db      CR,LF,LF,'ZSID2TTY v. '
        db      vers/10+'0','.',(vers mod 10)+'0'
        db      '  --  Redirect ZSID console i/o to TTY:',CR,LF,LF
        db      '          -- Usage --',CR,LF
        db      'Load ZSID, then .SYM and .UTL files',CR,LF
        db      '  #I* filename.SYM',CR,LF
        db      '  #R',CR,LF
        db      '  #IZSID2TTY.COM',CR,LF
        db      '  #R',CR,LF
        db      '  #G103       ==>> ZSID''s console is now TTY:',CR,LF
        db      '  #Ifilename.COM  (or .HEX) to debug',CR,LF
        db      '  #R',CR,LF
        db      '  #D3,3       ==>> display IOBYTE for reference',CR,LF,LF
        db      'To exit from ZSID, deinitialize the TTY: (if needed) by',CR,LF
        db      '  #IZSID2TTY.COM',CR,LF
        db      '  #R0',CR,LF
        db      '  #G106',CR,LF
        db      '(This message can be displayed under ZSID by the command:'
        db      CR,LF,'   G100  or   C100 before loading ''filename'')'
        db      CR,LF,LF,'$'
;
errmsg: db      CR,LF,BELL,'Can''t find ZSID v 1.4!$'
        ENDIF   ;IF ZSID

        IF DDT
;
modsiz  equ     1000h
sigad   equ     0A71h
callbd  equ     06A2h
fn2ad   equ     0BCEh
fn10ad  equ     0C25h
fn11ad  equ     0BBCh
;
usage:  db      CR,LF,LF,'DDT2TTY v. '
        db      vers/10+'0','.',(vers mod 10)+'0'
        db      '  --  Redirect DDT console i/o to TTY:',CR,LF,LF
        db      '          -- Usage --',CR,LF
        db      'A>DDT DDT2TTY.COM',CR,LF
        db      '  -G103       ==>> DDT''s console is now TTY:',CR,LF
        db      '  -Ifilename.COM  (or .HEX) to debug',CR,LF
        db      '  -R',CR,LF
        db      '  -D3,3       ==>> display IOBYTE for reference',CR,LF,LF
        db      'To exit from DDT, deinitialize the TTY: (if needed) by',CR,LF
        db      '  -IDDT2TTY.COM',CR,LF
        db      '  -R0',CR,LF
        db      '  -G106',CR,LF
        db      '(This message can be displayed under DDT by the command:'
        db      CR,LF,'   G100 before loading ''filename'')'
        db      CR,LF,LF,'$'
;
errmsg:db       CR,LF,BELL,'Can''t find DDT v 2.2!$'
        ENDIF   ;IF DDT
;
;
; Display usage message & quit.
;   This is executed if SID2TTY is run as a .COM file
;   It can also be C'd or G'd from the debugger.
;
HELP:   lxi     d,usage
print:  mvi     c,9
        call    bdos
```

(Continued on page 112)

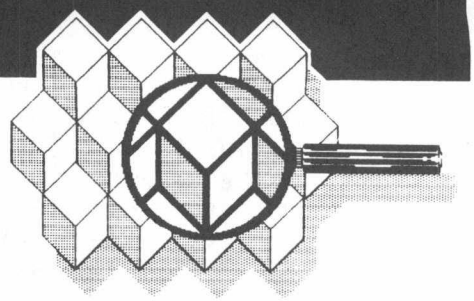
```
        lxi     h,0             ;is debugger in use on default stack?
        dad     sp
        mov     a,h
        cpi     2
        rnc                     ;return to caller (ccp)
        rst     7               ;return to debugger
;
ERR:    lxi     d,errmsg
        jmp     print
;
;
; Search for debugger signature in high memory.
;   Search downward to find first image of the debugger,
;   in case other copies are lying around.  This also
;   allows other resident modules to exist above the
;   debugger.  Debugger is always located on a page boundary.
;
INSTALL:
        lhld    1
        lxi     d,-(modsiz+3)
        dad     d               ;start at max possible addr
search: push    h
        lxi     d,sigad
        dad     d
        call    compar
        pop     h
        jz      found
        mov     a,h
        cpi     10h             ;quit looking at 1000h
        jc      err
        dcr     h               ;try 1 page lower
        jmp     search

compar: lxi     d,signature
        mvi     b,siglen
complp: ldax    d
        cmp     m
        rnz
        inx     h
        inx     d
        dcr     b
        jnz     complp
        ret
;
; Debugger located.  Calculate run-time addresses & patch
;   into code and debugger.
;
found:  push    h               ; save base addr of debugger
        lxi     d,callbd        ;put 3 run-time addresses into code
        dad     d
        xchg
        lxi     h,patch1        ;1: the call_bdos routine address
        mov     m,e
        inx     h
        mov     m,d
        lhld    6               ;2: the protect address/base of high code
        shld    patch2
        lxi     d,-codelen      ;protect the code
        dad     d               ;(don't trace from here to lpend)
        shld    6
        push    h               ;save run-time base of high code
        lxi     d,saviob-code
        dad     d
        shld    patch3          ;3: the save-iobyte address

        pop     d               ;get destination
        push    d
        lxi     h,code          ;move the code into place
        mvi     b,codelen
```

(Continued on page 114)

```
lp:     mov     a,m
        stax    d
        inx     h
        inx     d
        dcr     b
        jnz     lp
;
lpend:  pop     b               ;patch debugger's console io calls
        inx     b
        inx     b
        inx     b               ; ..to point at code entry ('divert')
        pop     d               ;de = base of debugger
        lxi     h,fn10ad        ;index into debugger module
        dad     d
        mov     m,c             ;plug 3 diversion addresses into calls
        inx     h
        mov     m,b
        lxi     h,fn2ad
        dad     d
        mov     m,c
        inx     h
        mov     m,b
        lxi     h,fn11ad
        dad     d
        mov     m,c
        inx     h
        mov     m,b
        jmp     INITTTY         ;do any user-supplied initialization
;
;
; The REDIRECTION CODE, which is moved up just below the debugger
;
patch2  equ     $+1
CODE:   jmp     ADDRESS         ;jmp to debugger entry
divert: lxi     h,iobyte
        mov     a,m
patch3  equ     $+1
        sta     ADDRESS         ;save the iobyte
        ani     0fch            ; assign CON:  to TTY: for debugger
        mov     m,a
patch1  equ     $+1
        call    ADDRESS         ;debugger's call_bdos routine
saviob  equ     $+1
        mvi     a,00            ;restore user's iobyte
        sta     iobyte
        mov     a,l             ;get bdos return param back to A
        ret
codelen equ $-code

;
; User-supplied TTY: initialization routine.
;   Use this to set baud rate, channel parameters, etc.
;
INITTTY:
        rst 7                   ;return to debugger on TTY: device
;
; User-supplied TTY: de-initialization routine.
;   Use this to restore TTY: device to regular settings.
;
DEINITTTY:
        rst 7                   ;return to debugger on TTY: device


        END
```

End Listing

OF INTEREST



by Alex Ragen

If you've ever traveled outside the United States, you've probably noticed that the household electricity varies from country to country. The line voltage may be 110 or 120 or 220 or 240 volts, and the frequency may be either 50 or 60 Hz. Even if your electric shaver or hair dryer has a little switch somewhere to flip back and forth among all the possible combinations, you still may find that you can't use it because the plug doesn't fit the wall receptacle. It turns out that almost every country in the world has its own unique plug shape, which is not only different but also superior to every other country's plug shape. Does this remind you of anything?

Software and hardware expend a tremendous amount of computing energy in converting between languages, protocols, disk formats, operating systems, and so on. You can run CP/M 80 or MSDOS on your Macintosh or use your IBM PC (and compatibles, of course!) to write Apple II disks or run a 68000 board. Whatever designers intended users to do on one configuration, they have managed to implement on another as well. This Babel of standards and not-so-standards is chaotic, confusing, pointless, and counterproductive—and we've all had lots of fun and worked hard getting around it.

Perhaps one day we'll live in a perfect world. Everything will interface with everything else, Macintosh add-ons won't be called Mac-this or Mac-that, and a word processor will emerge that is so good that nobody will want to write another one.

C Language

Lattice Inc. is offering an upgrade kit to users of Microsoft's MSDOS C compiler. For $150 and the original Microsoft diskettes, Lattice will provide version 2.20 of its C compiler, the C-SPRITE debugger, and a typeset manual. The regular retail price of the package is $675, and the offer is good until July 31, 1985. Contact Lattice Inc. at P.O. Box 3148, Glen Ellyn, IL 60138 (312) 858-7950. **Reader Service No. 101.**

QTools, a programmer's toolbox of 19 tools adapted from Unix to PCDOS, is available for $49.95 postpaid from QCAD Systems, 1164 Hyde Ave., San Jose, CA 95129 (800) 538-9787 or (408) 255-5574 in California. **Reader Service No. 103.**

A new C language **scientific subroutine library** that provides IBM PC users with 112 pretested and precompiled mathematical and statistical subroutines is available from Wiley Professional Software. The subroutines include the most commonly used operations such as differentiation, polynomials, probability, numerical integration, regression, differential equations and others. Over 400 pages of documentation provide a description of methodology, notes on special considerations, source code, test programs, and results. The library requires the Lattice C compiler, version 2.12 or later, and a two-drive IBM PC. Contact Leslie Bixel, Wiley Professional Software, 605 Third Ave., New York, NY 10158 (212) 850-6788. **Reader Service No. 105.**

TGL Inc. has released version II of **The Converter**, an IBM PC program that automatically converts UCSD, MT+, and MS-Pascal source programs into equivalent C programs. Ted Lewis of TGL points out that The Converter correctly handles nested procedures, separately compiled units or modules, and intrinsic functions; it also enables developers to move quickly from UCSD or MSDOS to Unix. Contact Mr. Lewis at TGL Inc., 4400 Sulphur Springs Rd., Corvallis, OR 97330 (503) 745-7476. **Reader Service No. 107.**

Thunder Software has announced version 2.0 of its **Thunder C**, the only C compiler available for the Apple Pascal and Apple ProDOS environments. Thunder C generates native 6502 assembly language routines, and its developers rate the new version as 300 percent faster than the old one. It runs under Apple Pascal 1.1 or 1.2, although the ProDOS version requires an external ProDOS macro assembler. Price is $49 plus $3 shipping. The company also offers LINKIT, a linking loader and library generator; XREF, a Pascal cross-reference utility; and ASSYST, a 6502 assembler. Contact Thunder Software at P.O. Box 31501, Houston, TX 77231 (713) 728-5501. **Reader Service No. 109.**

Hippo-C is the only (so far) C compiler for the Macintosh with a source-level debugger—or so its developers claim. Level 1 provides access to over 380 Toolbox and Quickdraw routines and features a compiler, editor, standard C library, on-line tutorial, debugger, linker, shell command processor, many sample programs, and over 200 pages of documentation. It's priced at $149.95. Level 2 includes all Level 1 features plus an optimizing compiler, assembler, and full floating-point support. It sells for $399.95. Contact Hippopotamus Software at 1250 Oakmead Pkwy., Sunnyvale, CA 94086 (408) 738-1200. **Reader Service No. 111.**

Modula 2

Release 1.10 of **Modula-2/86 for PCDOS** is now available from Logitech. A library of programming tools is also available. Contact Chris Cale at Logitech, 805 Veterans Blvd.,

Redwood City, CA 94063 (415) 365-9852. **Reader Service No. 113.**

Modula-2 for Z80-based CP/M systems is available from Hochstrasser Computing AG, Leonhard-*shalde 21*, CH-8001 Zurich, Switzerland, phone 01/47 55 48. The cost is sFr 400 or about $150. **Reader Service No. 115.**

MSDOS

Guess who is throwing its hat into the cutthroat world of the IBM PC add-on market? None other than Intel, which has established a **Personal Computer Enhancement Operation**. The first two products are its well-known 8087 and 80287 math co-processors (what's next, the 8086?). Contact Intel at 520N.E. Elam Young Pkwy., Hillsboro, OR 97124-6497 (503) 629-7369. **Reader Service No. 117.**

PC-68K is an upgrade package—plug-in board and software—that adds development support for Motorola's 68000 family to IBM's PC XT and PC AT running PCDOS. PC-68K provides a symbolic debugger, linker/locator, Motorola compatible macro assembler, and IEEE floating-point package. Pascal and C compilers and communications utilities are extra-cost options. The board features an 8 MHz 68000 cpu and 256 – 1024K of RAM. Prices start at $2995 for the 256K version. Contact Carrie Ann Moran at Language Resources, 4885 Riverbend Rd., Boulder, CO 80301 (303) 449-8087. **Reader Service No. 119.**

The **Buddy System** enables a support person located at a central facility to "take over" a remotely located IBM PC in order to resolve a customer's problem. The developer has been using it internally for three years to support its products and now offers it to others at $199. You load the Buddy System when you turn on your PC; thereafter, it's dormant until you activate the "copilot" mode and contact the support center via modem. The support person then observes you, the user, as the application is executed and can even take control of the application to show you what to do. A number of other features, such as help, screen snapshot, screen copy to file, and record/replay support session, are designed to increase the productivity of the support operation. Contact Edward Murphy at Solvation Inc., 302 Turnpike Rd., Southboro, MA 01722 (617) 481-9390. **Reader Service No.121.**

Phoenix Software Associates has introduced an IBM PC/AT-compatible ROM BIOS that was developed under strict controls specifically to ensure originality and to avoid copyright infringement suits. The company had already developed versions for the PC and PC/XT. The PC/AT version is offered to OEMs for unlimited use licensing at a price of $440,000. The PC and PC/XT versions are $290,000. Contact Richard Levandov at 1420 Providence Hwy., Suite 101, Norwood, MA 02062 (617) 769-7020. **Reader Service No. 123.**

Ryan McFarland's **RM/Cobol** is now available for the IBM PC/IX user. It was already available for the IBM PC under PCDOS, as well as for the IBM Series 1 under Unix and the IBM 370 computers under VM/CMS. The new RM/Cobol is priced at $230 for the runtime version and $750 for the full version. Contact Ryan McFarland Corp., 609 Deep Valley Dr., Rolling Hill Estates, CA 90274 (213) 541-4828. **Reader Service No. 125.**

A full ANSI **Fortran 77 compiler** for the IBM PC has been announced by Lahey Computer Systems. The complete package, including a 250 page manual, customer telephone support, and newsletters, costs $477 and requires 256K RAM and an 8087 co-processor. The compiler claims Lattice C compatibility. Contact the vendor at 31244 Palos Verdes Dr. West, Suite 243, Rancho Palos Verdes, CA 90274 (213) 541-1200. **Reader Service No. 127.**

Scroll and Recall lets the IBM PC user scroll back through 27 pages of previously displayed screens while saving you DOS commands, which you can then recall, edit, and reuse without having to type them in again. The cost is $69. Contact Dennis P. Olenick, Opt-Tech Data Processing, P.O. Box 2167, Humble, TX 77347 (713) 454-7428. **Reader Service No. 129.**

The **GTP Program Development System** is a productivity tool for Turbo Pascal programmers. Basically, it generates the Turbo Pascal statements for handling screen data entry. It requires PCDOS or MSDOS version 2.0 or higher, and lists for $99.95. Contact AEF Software, P.O. Box 928, Katy, TX 77449 (713) 391-8570. **Reader Service No. 131.**

FREECOPY is functionally equivalent to DISKCOPY, which is distributed as part of PCDOS. Although it is basically DISKCOPY reverse engineered, the author claims not to have violated the proprietary rights of either IBM or Microsoft. Commented source code is included. The author is placing the program in the public domain but does request a $25 contribution. Contact Donald Buresh, Squire Buresh Associates, 18 Dorothy Rd., Millbury, MA 01527 (617) 865-3435. **Reader Service No. 133.**

Modular Bridge is a product that allows you to transfer files between PCDOS and the PICK PC-XT operating system. Contact Modular Software, P.O. Box 204, Union City, GA 30291 (404) 964-7171. **Reader Service No. 135.**

HFORMAT and **HTEST** are two programs that address the problems of formatting hard disks on the *PC* family. The programs test and perform the equivalent of factory formatting on the drives. Contact Marcus Kolod at Kolod Research, P.O. Box 68, Glenview, IL 60025 (312) 291-1586. **Reader Service No. 137.**

A **Universal PROM programmer** for the IBM PC is offered for $250 by Advanced Microcomputer Systems of 6802 N.W. 20th Ave., Ft. Lauderdale, FL 33309 (305) 975-9515. **Reader Service No. 139.**

HelpDOS is a menu-driven on-screen reference for PCDOS and includes a technical dictionary. It requires PCDOS or MSDOS, version 2.0 or later, and is available for $49.95 from Help Technologies, P.O. Box 50834, Palo Alto, CA 94303 (415) 856-3431. **Reader Service No. 141.**

Superkey, a resident keyboard enhancer for data encryption and macro processing, has been announced by Borland International, the Turbo Pascal people. The program is basically a shortcut to routine entry of often-used commands, like those re-

quired for logging onto RBBS services or formatting letters and spreadsheets. The user can bring up Superkey, move around in the program, and exit with single-keystroke operations, as well as pull its full-screen macro editor down on top of the main program. It also provides on-line help, automatic recall of the last 20 entered commands, cut and paste capability, data encryption/decryption, and screen burn-in protection. Contact Borland International, 4113 Scotts Valley Dr., Scotts Valley, CA 95066 (408) 438-8400. **Reader Service No. 143.**

SEE is an editor for "professional programmers" from Prologic Corp., 31324 Via Colinas, Suite 111, Westlake Village, CA 91362 (818) 991-5062. **Reader Service No.145.**

A new version of **PC/Forth** for the PC/AT, running under 80286 XENIX 3.0, has been announced by Laboratory Microsystems, 3007 Washington Blvd., Suite 230, P.O. Box 10430, Marina del Rey, CA 90295 (213) 306-7412. **Reader Service No. 147.**

The Echo PC2 is a speech synthesizer card for the IBM PC. The manufacturer, having previously provided similar products for the Apple, claims to be the first to produce one for the IBM PC. It features two speech modes: a limited vocabulary (about 700 words) in a natural-sounding female voice and an unlimited vocabulary in a robotic-sounding voice. Some 400 English language and pronunciation rules have been utilized to produce the computer-synthesized speech. Software can control pitch and volume. The board fits into the PC's short slots and comes with a speaker, software, and an instruction manual. The price is $149.95. Contact Street Electronics Inc., 1140 Mark Ave., Carpinteria, CA 93013 (805) 684-4593. **Reader Service No. 149.**

BIS7705 allows an IBM PC to emulate all the functions of a Honeywell 7700/7705 terminal. The price is $695, and it's available from IE Systems Inc., 112 Main St., Newmarket, NH 03857 (603) 659-5891. **Reader Service No. 151.**

CP/M 80

DSD80 is a fully DDT-compatible full-screen symbolic debugger for 8080, 8085, and Z80-based CP/M 80 systems. DSD80 includes port I/O, string searching, and symbol definition. The Z80 instruction set is fully supported using either extended Intel or Zilog mnemonics. There is an on-line help facility, and the program comes with a 50-page user's manual. The price is $125. Contact Soft Advances, P.O. Box 49473, Austin, TX 78765 (512) 478-4763. **Reader Service No. 153.**

CP/M users needn't feel left out in the cold by the latest fad of pop-up programs. Poor Person Software has announced **Write-Hand-Man**, which includes a notepad, phone book, desk calendar, file and directory viewing programs, and a communication program. Users familiar with CP/M programming can add new functions. Write-Hand-Man requires CP/M 2.2 and will run on all CP/M machines. The price is $49.95. The program is available from Poor Person Software, 3721 Starr King Circle, Palo Alto, CA 94306 (415) 493-3735. **Reader Service No. 155.**

MB+ Tools is a set of programmer productivity tools for Pascal MT+. It's currently available for CP/M 80 and CP/M 86, and it should be available for MSDOS late this year. The price is $175. Contact Michael Nunamaker, Minnow Bear Computers, P.O. Box 2233, Champaign, IL 61820-8233 (217) 398-6883. **Reader Service No. 157.**

Macintosh

MacCharlie is not a trampburger but a $985 box that attaches to the Macintosh and allows it to run programs written for the IBM PC. MacCharlie also allows Macintosh users to connect to IBM PC serial networks and to use IBM PC-compatible printers. Contact Robert Barrett at Dayna Communications, 50 South Main St., Suite 530, Salt Lake City, Utah 84144 (801) 531-0600. **Reader Service No. 153.**

Mighty Mac is a personal information manager priced at $99 from Advanced Logic Systems at 1195 E. Arques Ave., Sunnyvale, CA 94086 (408) 730-0307. **Reader Service No. 155.**

FileMaker (the third product in the MACWARE line) is a single-file data base system that allows a user to create a variety of custom-designed reports and forms that include text and graphics. FileMaker takes full advantage of the "visually active" Macintosh environment and functions within the Macintosh Office recently announced by Apple. Contact Forethought Inc., 1973 Landings Dr., Mountain View, CA 94043 (800) MACWARE. **Reader Service No. 129.**

DDJ

Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS #105 JULY 1985 $2.95 ($3.95 CANADA)

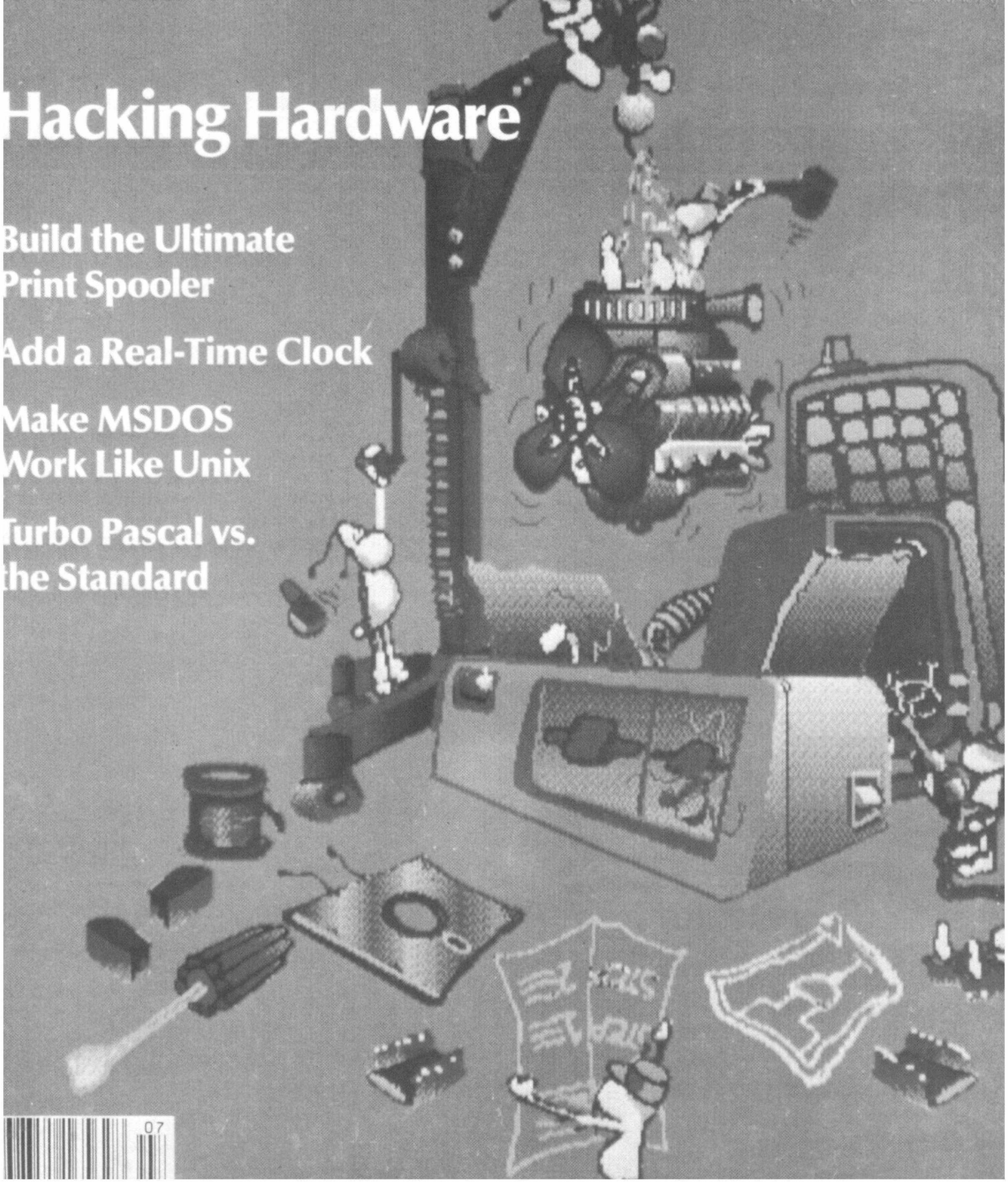
Hacking Hardware

Build the Ultimate Print Spooler

Add a Real-Time Clock

Make MSDOS Work Like Unix

Turbo Pascal vs. the Standard



07

Dr. Dobb's Journal

Editorial

Editor-in-Chief *Michael Swaine*
Editor *Randy Sutherland*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Polly Koch*
Typesetter *Jean Aring*

Production

Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover Cartoon *Rand Renfroe*

Circulation

Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Snee*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Kathleen Boyd*

Administration

Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda López-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*

Advertising

Advertising Director
Stephen Friedman (415) 424-0600
Marketing Manager
Shawn Horst (415) 424-0600
Advertising Sales
Walter Andrzejewski (617) 587-8361
Lisa Boudreau (415) 424-0600
Beth Dudas (714) 643-9439
Michele Beaty (317) 875-0557
DDJ Classifieds
Tim Ortiz (415) 424-0600
Advertising Coordinators
Jay Horvath (415) 424-0600
Alice Abrams (415) 424-0600

M&T Publishing, Inc.

Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $46 for airmail to Canada, $62 for airmail to other countries. Foreign subscriptions must be pre-paid in U.S. Dollars, drawn on a U.S. Bank. For subscription problems, call: outside of CA 800-321-3333; within CA 619-485-9623 or 566-6947.

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 686-1520 TELEX: 620430 (WUI)

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

July 1985
Volume 10, Issue 7

Referees

Michael Enright
Robert Tripp

Cover

The cover cartoon by Rand Renfroe was colored on a Via Video Graphics System by Doyle Puppo and Associates.

Editor's Note

In C chest this month Allen Holub offers an MSDOS version of the Unix directory utility ls. As the magazine went into production, we discovered that the program did not operate exactly as promised. The command ls /, which was supposed to list the contents of the root directory from another directory, did not function properly under DOS 2.x. Also, when run on a Compaq, ls did not underline directories, but printed them in half intensity. This is probably due to the particular ANSI.SYS file used.—Ed.

Dr. Dobb's Journal

ARTICLES

Build a Custom PC or Clone 42
by Jim Kronman
From individual components you can assemble a PC XT work-alike that is totally IBM compatible and costs substantially less. **(Reader Ballot No. 192).**

The Ultimate Parallel Print Spooler 46
by Don Rindsberg
Build a parallel print spooler with a 60K buffer, software refresh of RAM and a peek-a-boo ROM. **(Reader Ballot No. 193).**

Designing a Real-Time Clock for the S-100 Bus 56
by Alan Wilcox
Construct a clock board for your S-100 system that prints the time, date and filename as a header on your programs. **(Reader Ballot No. 194).**

COLUMNS

Dr. Dobb's Clinic 12
by D. E. Cortesi
A design error in the 8-bit version of Turbo Pascal; Turbo compared to Standard Pascal. **(Reader Ballot No. 190).**

C Chest 20
by Allen Holub
Ls, a Unix-like MSDOS directory utility; multi-column printing; a new MSDOS access function for Lattice C. **(Reader Ballot No. 191).**

Realizable Fantasies 92
by Michael Swaine and Bob Albrecht
Progress reports on fantasies presented to date. **(Reader Ballot No. 195).**

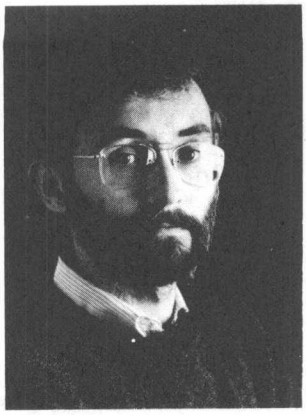
16-Bit Software Toolbox 94
by Ray Duncan
MSDOS installable device drivers. **(Reader Ballot No. 196).**

Computer Calisthenics 110
by Michael Wiesenberg
Answers to puzzles presented in previous columns. **(Reader Service No. 197).**

DEPARTMENTS

Editorial 6
Letters 8
DDJ Classified 115
Of Interest 120
Advertiser Index 128

EDITORIAL



We do this for you.

"We:" that's us, the editorial staff; "this" is the magazine. It's that "you" that makes things tricky.

"You" are a statistical construct. You are 40,000 readers with at least 40,000 opinions on any random subject, 40,000 ideas about what *Dr. Dobb's Journal* should be.

You are a systems programmer for IBM. A DoD Ada specialist. You are the DP Manager for a large corporation. You are the only programmer for a small firm. You are an independent software developer. A consultant. You're a structural engineer who happens to write software. A researcher. A teacher. A student.

You program in a high-level language because you're a high-level thinker, and you use assembly language only to tighten the bolts. You have concluded that all serious computer languages are fundamentally equivalent, so why doesn't everyone just use Pascal? You think that computer languages differ widely in power, efficiency and naturalness, so why doesn't everyone use Forth (since, as surely everyone knows, Forth code is invariably faster and tighter and more readable than anything else)? You've forgotten in the past year that there is any high-level language but C.

You refuse to program in a high-level language, pouring all your creative efforts into high-torque 68K assembly language code. 8086/8088 code. Z80/8080 code. You know the 6502 inside out. You know, even if you are in the vanishing minority, that the 6809 is all the processor anyone really needs. You know things about the 80286 that they haven't discovered at Intel yet.

You know what you like about *DDJ*. The *DDJ* articles for 1985 that you were most interested in (so far) included pieces dealing with the Unix, CP/M and MSDOS operating systems; the C and Prolog languages; the IBM PC, the PC/AT, the Mac, the Commodore 64, Z80 machines and "machines in the 68000/16000 class with virtual memory."

You're intelligent, knowledgeable, creative and opinionated, but let's face it: you are not particularly consistent. Neither, consequently, are we.

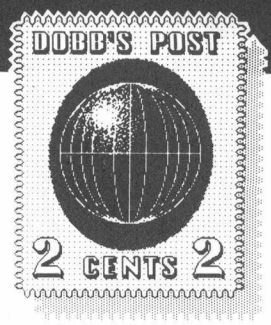
This is the hardware issue of *Dr. Dobb's Journal* of "software tools for advanced programmers." Why are we publishing hardware construction articles in a software magazine?

We are doing it, of course, for you. The "Fatten Your Mac" article in January was one of our most popular articles to date, and other hardware-related pieces have been well-received.

We have a broad constituency, trying as we do to address all advanced programmers, and magazine space is heartbreakingly limited. So we try to be as broad as possible without sacrificing depth. We publish generalizable code whenever we can, emphasizing good algorithms.

But the canvas tears when stretched to include hardware articles. Although there is much non-hardware material in this issue, we have this month committed a sizeable portion of the magazine to hardware-related material. We offer this collection of hardware articles with a question: how would you like to see us cover hardware in the future? With an occasional article when it seems appropriate? With an annual hardware issue like this? Or a hardware issue of a different sort? In separate monographs/project papers, leaving the magazine pages for software only? Or not at all?

Michael Swaine



Small C

Dear *DDJ*:

I don't know if anybody's still keeping track of these things, but I've found a Small C bug. The source

```
int lptr; /* global integer */
...
routine( ){
    int r;
    char bytes[2];
    ...
    lptr += r - bytes;
    ...
```

compiled to (Z80 mnemonics)

```
... (usual stuff)
LD HL, (LPTR)
PUSH HL;;;   (save left side of
                += statement)
LD HL,4
ADD LH,SP
CALL CCGINT
EX DE,HL;;
LD HL,2
ADD HL,SP
CALL CCSUB
POP DE       (retrieve left side of
                += statement)
ADD HL,DE    (add it)
LD A,L       (forget what's going
                on ....)
LD (DE),A    (should be LD
                (LPTR),HL)
...
```

which is entirely the wrong idea and has very nasty effects (the bug also occurs with local destinations for the assignment, but with somewhat less dangerous results). The problem disappeared when I *removed* the last code line from plung2( ) (aka plnge2 in "The Small-C Handbook" by James E. Hendrix) in cc31.c:

```
if ((oper==ffsub)|
    (oper==ffadd))
  result(lval, lval2);
```

In my copy of the compiler, this is the only place result( ) is used, so I removed it from cc33.c. The compiler compiled itself in this shape, and successfully compiled a reasonably large program, so maybe this is the correct fix, but I'm not deeply convinced.

Fixing the compiler is a real adventure for me, because I don't know how it works. I do wonder what solemn oath bound the original authors, so that no trace of comment is allowed to stain the pure pages of code ....

Cordially,
Gregor Owen
μ Software/Hardware
35 Admiral St.
Port Jefferson Station,
New York, NY 11776

Other fixes and a status update for Small C will appear in the August issue—Ed.

Iconoclasm

Dear *DDJ*:

By your response to Frank Gaude's criticism of icons, it appears as though you don't fully appreciate his point. In fact, your apparent praise of 'active, growing icons' only strengthens his argument. What will we have if software developers everywhere start creating systems with user-definable icons? We will have the second stage in the evolution of language, as icons were the first. What is more, we will have multitudinous languages, with each programmer or user designing the icon interface to [his] own personal taste. Nobody will be able to sit down at somebody else's system and run it. We will have the Tower of Babel all over again. Confusion. Insanity. Let's not do it all over again.

For those who insist upon performing all operations with a single keystroke, there are menus.

Another thing: metaphors. This isn't really related to icons, except that it's tangled up in your discussion of them. I wouldn't quibble with anyone who said that the purpose of software was to entertain, educate, or to simply get the job done ... or to obfuscate, mislead, and lie. But 'The purpose of software is to realize metaphors'? Could you say that again, please? Could you illustrate with an example?

Well, gee, now that I've gotten going it's just too hard to stop. One more thing. I greatly doubt that all of your readers know that Richard Conn wrote ZCPR3. As a consequence, I would have thought it to be a prudent editorial decision to add a note to that effect to his review [*DDJ* #103, May 1985] of the Ampro computers, which comes bundled with ZCPR3. What do you think?

(By the by, I mean no criticism of ZCPR3. It's the best damn thing to come down the pike since the Z80. Conn—and Echelon—are to be applauded roundly.)

Sincerely,
Dreas Nielsen
234 NW 30th St.
Corvallis, OR 97330

Indeed we should have identified Richard Conn's affiliation.—Ed.

CP/M

Dear *DDJ*:

I have been using CP/M 2.2 for several years, and acquired CP/M Plus about 9 months ago. I am pleased with it, but I have run up against one difficulty, namely that the SAVE program does not correctly save .REL files (produced by Microsoft's BASCOM) loaded by MICROSOFT's L80 relocating loader. I wrote a short BA-

SIC program to list ASCII files, including line numbers; after finishing a listing it prompts the operator for another file name. L80 loads the .REL file, and shows a starting address of 013CH. (The starting address of CP/M programs is 0100H.) If 013C is used as the starting address when the program is saved (using CP/M Plus's SAVE program) it does not run properly. It returns to the operating system after a few seconds, having done nothing visible. (It does not even ask for a file to list.) If the file is saved with a starting address of 0100H, the resulting .COM program does execute. It lists the file, and returns to ask for another file when the first file is completed. However, it does not operate properly: with each line it prints a set of characters that do not belong to the file, the same set of characters for each line. If loaded and saved under CP/M 2.2 it operates properly, even when moved to a CP/M Plus diskette. However, if the .REL file is loaded by L80 with the /G option the program operates properly, even under CP/M Plus.

If the .REL file is loaded by Digital Research's LINK, the resulting .COM program executes and lists the file properly, but hangs without returning to request more input when it has finished listing the file. Recovery requires rebooting the system.

I have written to Digital Research and Microsoft, as well as the vendor from whom I bought the computer, a MAX80, and have not received any assistance. Have you heard about this problem? Or might some of your readers have some information that might be helpful.

Sincerely yours,
Maynard B. Neher
16637 Diaz Drive
San Diego, CA 92128

Tiny BASIC

Dear *DDJ*:

I have enjoyed your journal very much over the years. I especially liked your articles on Tiny BASIC and Small-C. I implemented Tiny-BASIC on the 6800 after your first articles. I wrote a Small-C for the 6809 when you started the Small-C series and did one for the 68000 later. When you published the article on Tiny BASIC for the 68000 in the February issue [*DDJ* #100 February 1985] I couldn't resist downloading the code and trying it.

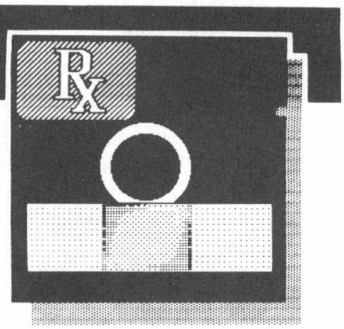
I have a suggestion for improvement which I would like to pass along. The RND function does not generate very good random numbers. I wrote the enclosed test program which generates pairs of 1 digit numbers. These should be evenly distributed over the 100 possible combinations. If you try the program you will see that there are several pairs of digits that don't appear at all.

I have enclosed an improved RND function which eliminates this problem. It is only a first try and there is room for improvement in two areas. First it is restricted to a limit of 65535 since I didn't bother to implement a 32-bit multiply. Second it is based on the multiplicative algorithm and I didn't spend any effort picking the best multiplier. It should have a sequence length of 4,294,967,296 however.

John P. Byrns
1953 Governors Ln.
Hoffman Estates, IL 60195

DDJ

DR. DOBB'S CLINIC



by D. E. Cortesi

Random Moves

We are still collating the letters we've received on that pseudo-random number generator (PRNG) we showed in February. Don't use it. We promise emended and improved versions in Forth and assembly language next month.

Turbo and CP/M Plus

The estimable Turbo Pascal product has a design error that makes its 8-bit version unusable under CP/M Plus (CP/M 3.0) and dubious under CP/M 2.2. When you compile a program to form a .com file, Turbo includes several kilobytes of runtime support in the output, along with the code generated from your Pascal source. That's not the problem; runtime support code is necessary.

Part of the job of that runtime library is to initialize the program's environment when it is loaded and begins execution. Anyone who has written assembly code for CP/M knows what must be done: the stack pointer must be set up and the size of available storage discovered. The usual way of sizing storage is to load the BDOS entry address from location 6.

It's a universal CP/M convention that system programs can reserve storage for themselves at the top of the program area by reducing that address. DDT and SID do it; XSUB does it; spoolers and key-macro programs do it; and CP/M Plus has elevated it from an informal convention to an elaborate system of Resident System Extensions (RSXs). As a result, the top of storage is a dynamic value in CP/M. It can change from day to day if the BIOS expands and from moment to moment as system programs are loaded.

The Turbo compiler, however, treats the size of storage as a constant. At compile time, it places in the runtime code a constant number reflecting the size of the program as the compiler found it; when the .com is executed, it expects storage to be exactly the same size. If programs compiled under Turbo V.3 find less storage than the compiler had, they abort with the message "Not enough memory." This has nothing to do with how many bytes the program needs. Indeed, our test program didn't need any; it simply wrote "hello" 10 times. If we compile it when no RSXs are present but run it when one is, it aborts. Run under SID, it aborts. It's a strange oversight in an otherwise solid product.

Turbo and Standard Pascal

Understand, please, that we *like* Borland and its marketing policies. Its products give good value for the money, and Borland backs them with support that is much better than average. Between Borland and the "shareware" pioneers, a whole new level of price/performance is being established in the software industry. So spare us the letters about how Borland has done so much for software, etc., etc. We appreciate all that.

Turbo Not Standard

But we are getting just a tad tired of the claim that Borland's Turbo Pascal is "standard." It is definitely *not* standard Pascal, which, in a compiler that seems destined to dominate the micro world, is a damn shame. Its price and good user interface commend it for school use, but we certainly couldn't recommend it for that purpose, and we doubt any college computer science department would accept it despite its friendliness.

Let's talk about the Pascal standard, how Turbo violates it, and what that costs its users.

The Ignored Standard

There *is* a Pascal standard. In the U.S., it is sanctioned by the IEEE Computer Society and the American National Standards Institute (ANSI); abroad, by the International Standards Organization (ISO). Although ANSI finalized it in December 1982, it was available at least two years earlier in drafts that differed from each other in only the minutest points and that nobody expected would change in any significant way prior to final approval.

The standard language omits features that many people want. Although it defines a minimum Pascal, it explicitly opens the door to extensions. It defines an extension as "a modification to . . . this standard that does not invalidate any program complying with this standard . . . except by prohibiting the use of one or more particular spellings of identifiers." In other words, you may add any features you like to Pascal as long as you compile standard programs correctly. The last clause even lets you add new reserved words to the language, words that could be user-defined names under the standard.

The standard lays two burdens on the developer. First, you must document variations from the standard in certain ways; that's usually no problem. The other is that a translator must "be able to determine whether or not a program violates . . . this standard . . . and report the result of this determination to the user." In other words, you must warn the user when a program is not standard Pascal. That's essential to portability; otherwise what you intend to be a standard (hence, portable) program can slip

into an unstandard state that is undetectable until you attempt to port it.

Standard Pascal is a small language but still bigger than, say, Fortran IV; more flexible than, say, COBOL; and easier to teach, to use, and to implement than, say, Ada. If there were general compliance with the standard, we could teach people a single language they could use anywhere, at least for small and medium problems. If you moved to a new implementation, you would have to learn the peculiar extensions it supported, but you could still do useful programming while you learned. And programs you wrote using the standard language would be portable anywhere.

It's ironic that many newer and less forgiving standards (e.g., for graphics, for terminal escape sequences, for various buses) garner wide respect and conformance, while the modest and unconfining Pascal standard, draws only scorn from developers.

Turbo Versus the Standard

The Turbo manual (*Turbo Pascal Version 3.0 Reference Manual*, Borland International, Inc., 1985) doesn't contain the kinds of documentation called for by the standard, but it does have Appendix D, "Turbo vs. Standard Pascal" (p. 319). This shows at once that Borland is unaware of the ANSI/ISO standard. "The Turbo Pascal language," it says, "follows the Standard Pascal defined by Jensen & Wirth in their *User Manual and Report* . . . ."

We hate to disturb such blissful slumber, but the language has changed in the 10 years since the *User Manual and Report* was published. The ANSI/IEEE committee convened in 1979 and published its first draft standard in 1980, two years before Turbo was born.

Turbo follows its outdated standard "with only minor differences introduced for the sheer purpose of efficiency." Efficiency for whom or what? For users who want to port programs or to port their hard-won knowledge of the language? Surely not! What then? Efficiency in the execution of the generated object programs? We will examine the case for that as we go; it doesn't hold up.

Could the changes be to improve the efficiency of the compiler itself? Very possibly; we'll see that the elimination of some hard cases may have let the compiler be a little simpler.

Or could the manual be referring to the efficiency with which the developers can bring a product to market? Did Turbo drop the hard parts so the developers could finish their compiler more quickly?

New and Dispose

Turbo v.1.0 skipped the standard way of managing dynamic storage, the procedures New and Dispose. V.2 added them but in incomplete form. Turbo supports the basic features: New(*ptr*) creates a new one of whatever *ptr* is declared to address, and Dispose(*ptr*) reclaims the space used by whatever *ptr* points to.

The standard allows a special case for allocating variant records (records that may have different lengths depending on a tag field). Suppose that *ptr* bases a variant record type and that *tag* is one of the constants that select among the variants. New(*ptr*) will allocate space to hold the longest variant, but New(*ptr*,*tag*) should allocate just enough space to hold the *tag* variant. Dispose(*ptr*,*tag*) is then required to release such a record.

Turbo omits these special forms. Why? Not to enhance the user's efficiency; it will seriously complicate your life if you try to import to Turbo a program that uses this feature. Nor for the program's efficiency; the shortest variant of a record is often the most common one, and lacking a way to allocate short records, you can waste a lot of dynamic storage—sometimes enough to make a program infeasible.

Turbo didn't have to ban these cases. It could have accepted them while ignoring the *tag* parameters. Then standard programs would at least compile. But why not support them in full? The standard says *tag* must be a constant; therefore, Turbo could still tell at compile time how many bytes New allocates or Dispose releases. The extra compiler code couldn't be large, and the cases have no effect on the runtime library.

But the manual adds insult to injury. "The restriction," it chirps, "is easily circumvented by using the standard procedure GetMem." Wrong on every count! GetMem is not "standard"; that name is unique to Turbo. Nor is GetMem type-secure, as New is. Plus it requires a count of bytes to allocate, so to use it you must know in detail how Turbo allocates space in variant records. Then heaven help you if you change the record type but forget to change the GetMem calls! (Turbo has a nonstandard SizeOf function, but it won't give you the size of a particular record variant.)

The Page Procedure

The standard specifies a list of procedures and functions that a compiler must predefine. Turbo has all the trigonometric functions, all but two of the I/O procedures (discussed below), and dozens of nonstandard routines for "turtle" graphics, special I/O operations, DOS calls, and so on.

But it lacks the simple procedure Page. Page(f), where f is a text (ASCII) file, is supposed to append a newline if the file isn't currently at the head of a line then "cause an implementation-defined effect on the textfile f, such that subsequent text . . . will be on a new page if the textfile is printed on a suitable device." Like the other Pascal I/O routines, Page with no parameter works on the file output by default.

Page may not seem like much of a loss, but its absence puts the user in an infuriating bind. Like the other Pascal I/O procedures, Page cannot be duplicated by user code. User procedures can't have default parameters, and there is no way to perform the essential test of a file variable to see if it is at the head of a line. (A Page that always writes a newline will sometimes force an extra blank sheet where the standard Page will not.)

What do you do to import a standard program that relies on Page? You find every use of Page and change it to call one of two procedures that you must write yourself—procedures that can't duplicate the standard actions perfectly. If the output isn't satisfactory, you must dig into the program's logic. Changing

the logic of an imported program, one written perhaps by total strangers, because of a stupid omission from a compiler is not a good use of time. If you write a program that you will later export, you must include your homebrew PageNamedFile and PageDefault procedures, surrounded with big comment blocks explaining how to alter them to use Page if it should be available.

What would Page cost: 64 bytes of code in the runtime library and maybe 32 bytes of table space in the compiler? Its overhead would be undetectable in a compile and zero at runtime. Why did Turbo omit it? We'll never know. Appendix D says it's because "the CP/M operating system does not define a form-feed character," but that is pure waffle. If there is one thing common to all CP/M systems, it's ASCII, whose form feed is respected by every printer we know.

Get and Put

Wirth gave a mathematician's definition of I/O. He began with a rigorous definition of the file data type. Then he chose a minimal set of operators on that type, operators whose actions are regular enough to be useful in formal verifications of program correctness. Finally he defined all file operations in terms of these minimal operators. The standard follows the *User Manual and Report* in giving a rigorous logician's definition of file I/O, all based on the operators Get and Put.

What do these fundamental operations do? First, understand that to Wirth a file variable is a species of pointer variable. He says that a file is a sequence of objects, while a file variable is a pointer to only one of the objects in the sequence. The declaration

```
fr : file of real;
```

asserts two things: somewhere there is a sequence of real numbers (possibly empty), and under the right circumstances *fr* will address one of them. The statement

```
Reset(fr);
```

sets up the circumstances; *fr* now addresses the first real number in the file. For comparison, recall that

```
pr : ^real;
```

sets up a variable *pr* that points to a single real number, as in

```
x := 3.5 * pr^;
```

Therefore, Reset(*fr*) sets things up so that *fr* addresses a real number:

```
x := 3.5 * fr^;
```

Thus did Wirth unify the treatments of I/O and dynamic storage.

Standard Get and Put

The Get procedure advances a file one unit in its sequence. After Get(*fr*), *fr* addresses the next real number in the sequence.

Rewrite opens a file for output; the statements

```
nr : file of real;
Rewrite(nr);
```

create an empty sequence of real numbers and make *nr* point to a real of undefined value. You may use *nr* exactly like any other pointer to reals:

```
nr^ := sqr(pr^) + sqr(fr^);
pr^ := sqrt(nr^);
```

The statement

```
Put(nr);
```

appends the current referent of *nr* to its file and makes *nr* point to undefined space again.

Do you find these ideas confusing, difficult, baffling? Do you think they make for an inflexible system? Or do you agree that they are simple, logical, consistent, and highly useful?

The Borland programmers did not. Turbo does not support Get and Put, nor does it permit a file variable to be used as a pointer. It offers only the Read and Write procedures (which to Wirth are composite operations defined in terms of Get and Put).

Simplicity?

Appendix D offers four excuses for the omission. One is that Read and Write are "easier to understand." A computer science teacher would never say that. Good teachers explain the basis of things. You cannot explain the basis of Read and Write without referring to Get and Put; you can state their actions as arbitrary rules to learn by rote, but you can't explain them.

Of course, if you are already an experienced programmer—one used to Fortran or BASIC, perhaps—you might find the Read and Write procedures more *familiar* than Get and Put, and you might confuse familiarity with simplicity. That's a common error, as anyone will attest who has tried to explain the arbitrary, but totally familiar, rules of English to a foreigner.

Versatility?

One of the three other excuses is that Read and Write are "far more versatile" than Get and Put. That's rubbish, but our hypothetical ex-Fortran user wouldn't realize it without close study of Pascal.

With Put, you can assign and reassign an output value as often as you like; the value isn't committed to the file until Put is issued. That concept isn't in the thinking of a PL/I user; in most languages, once you associate data with a file, it's gone. Once you comprehend the idea, however, you can find lots of uses for *writing* data without *putting* it.

Get is even more useful. If you've read Kernighan and Plauger's *Software Tools*, you may recall how often they resort to the ungetc( ) procedure to replace the character last read from a file. With Pascal standard input, you can look at the next value coming from a file without committing yourself to removing it from the file. You simply test the value addressed by the file variable. If you want it, you assign it somewhere else and issue Get. You may also leave it for another part of the program to use, or discard it with Get. Here's a procedure, free of side effects, to strip blanks from an input file:

```
procedure fstrip(var f: text);
begin
   while (not eof(f))
```

```
    and (' '=f^) do
      Get(f)
  end;
```

Afterward, either the next element of the file is a nonblank or the file is finished. Try writing a comparable function in C or BASIC—or in Turbo! It takes either ungetc( ) or your own input function with a static variable.

Because Get and Put are standard and highly useful, you may assume that Pascal programmers make use of them. A lot of Pascal programs are floating around on minis and mainframes and in textbooks. The odds are good that if you try to import one to Turbo, you'll fail because the program uses Get or Put. You may expect to find as well that the assumptions behind Get and Put are woven deeply into the program's logic; most programs will need major changes to run without them.

Less Overhead?

Appendix D offers the excuse that, without Get and Put, "variable space overhead is reduced, as file buffer variables are not required." True, the compiler would have to allocate a buffer the size of one file data item in addition to the sector buffer it needs for DOS I/O. Alternatively, it could round the size of a data item up to a whole number of DOS sectors, add one sector, and allocate a single buffer of that size (this ensures a complete data item will always fit in the buffer).

Banning standard use of the file variable lets the compiler get by with a single sector buffer. Does that reduce "variable space overhead"? Of course not! Now *you* have to define the variables! You need extra variables to hold data items before they are written to the file, instead of merely assigning and reassigning until a Put is done. You need variables to hold items after reading them from the file and before using them in expressions. And if it turns out the items shouldn't have been read, Turbo offers no ungetc( ), so *you* get to define one—again adding code overhead.

Speed?

Appendix D's premier excuse is that "Read and Write give much faster I/O." Do they? True, a naive implementation of Get might often copy data from a sector buffer to an item buffer. But Read *requires* copying data to a user-defined variable, whereas a smart implementation of Get could avoid moving data at all. Many times when the file data type is smaller than half a sector (the typical case), Get need do no more than increment a pointer, check for end of sector, and return. That's quick!

If a new sector must be read, a Get implementation might have to move part of the last sector from the end of the buffer to its head before reading more data. But that move always involves *less* than a single data item! Under MSDOS, when the file data type is composite (record, set, or array), a clever compiler would push the whole burden onto the DOS. It would allocate no sector buffer at all, just an item buffer, and use "handle" I/O to read or write only as many bytes as needed.

A good implementation of Get and Put might entail 15 percent more code than the present Turbo library, but by moving less data, it might well execute in 25 percent less time.

Procedure Parameters

One of the least-regarded features of standard Pascal is the ability to pass a procedure or function as a parameter. Let's say "routine" instead of "procedure or function." Then the feature sounds simpler: the ability to pass one routine the name of another. Its purpose is to let a routine's behavior be parameterized along with its data. Unfortunately, neither Jensen nor Wirth nor the standards committee could find a good example of its use except "finding the minimum of a function by bisection." Now really, what can an honest programmer say to that but "Gah, Wha?"—meaning, surely you can omit such an abstruse feature from a language without anyone's noticing!

Let's try to suggest some examples to show why it should stay. The first might be a generalized sort that takes, as a parameter, the name of a function that compares two data items and returns True if one is less than or equal to the other. Now define two comparison routines:

```
function Up(a,b:real):Boolean;
begin Up := (a <= b) end;

function Down(a,b:real):Boolean;
begin Down := ( b<=a ) end;
```

The identical generalized routine will sort ascending if we pass it the name of Up and descending if we pass it Down.

Notice that if we make Up and Down take, not the data to be compared, but ordinal indices to the data, and if we also pass a procedure Swap that interchanges data items given their indices, we will have removed all dependencies on data type from the general sort routine. We need only pass it the Up or Down and the Swap that are appropriate to the data type we want to sort.

As another example, consider a situation where a service routine can detect an error, for which the correct recovery action depends on the conditions under which the routine was

called. There are three ways to handle this: (1) You can write a version of the routine for each recovery method; (2) you can pass the routine an error-recovery action number and put a big error-handling case statement in it; or (3) you can pass the routine the name of an error-handling procedure. Each client may then contain a local procedure to do error recovery in its own style. The service routine remains general, while the recovery actions are localized in those parts of the program that establish the need for them.

Turbo Pascal is not alone in banning procedure parameters; it's the most ignored feature of the language. Why? It seems like a fairly simple thing to implement—at first. At runtime, it amounts to just pushing the address of a routine on the stack. At compile time, the compiler must check that the parameter list of the routine that is passed matches the parameter list that the receiving routine declared, which is easier to do than to describe.

The real problem is the implementation of Pascal's rules for the scope of variable names. When procedure parameters are forbidden, the dynamic scope of a name matches its lexical scope; the compiler can tell from the program text exactly what names are accessible at any point in the program. When procedure parameters are allowed, cases occur where the referent of a name can be resolved only by extra stack linkages. That complicates the runtime code for stack management and for procedure call and exit.

Although Borland may seem to have made a good tradeoff in dropping procedure parameters, the usual arguments can be made for the feature: it's useful and likely to appear in published programs, especially in the more sophisticated ones, those that are at once most valuable and hardest to modify. We might add that, inasmuch as Turbo threatens to be the single most common implementation of Pascal ever, Borland has an obligation to support the whole language.

The stack management is not that complicated anyway. Full Pascal stack linkage no doubt would slow down an 8080 or Z80 implementation (for once there's a legitimate reason for restricting the CP/M version!), but that is not true of the 8086.

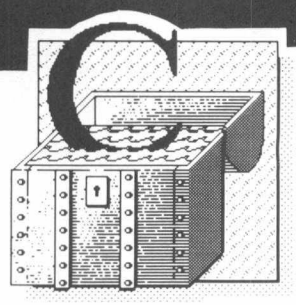
What would be the penalty for full Pascal in the MSDOS version of Turbo? The compiler would be slightly more complicated, as noted. And the runtime code? Studies have shown that the overwhelming majority of references are to local and global names and to parameters. The troublesome references, to names defined in intermediate scopes, are very rare. (It's tempting to modify the scope rules to forbid just these references, but that would create incredibly subtle portability bugs.) If the generated code optimizes common usages and penalizes the rare ones, the cost should be small.

Summary

The Pascal standard defines a nice little language, but despite the language's popularity and the standard's age, the features of the language and the liberality of the standard are still misunderstood or unappreciated. The justifications in the Turbo manual's Appendix D just don't hold water; likely the real reason Turbo is nonstandard is that the Borland programmers were in a hurry to get to market so they dropped some features that were obscure (to them) or of little use (to people who don't understand Pascal well). Yet Turbo Pascal has many admirable features. If it added to them full compliance with the Pascal standard, it would be without question the finest implementation of Pascal for personal computers and among the finest on any computer.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**



by Allen Holub

As part of my ongoing struggle to turn MSDOS into Unix, I write a lot of Unix-like utilities. This month we'll look at a better MSDOS directory utility called "ls" (after the Unix model). Ls uses several subroutines useful in their own right. A multicolumn print utility and an MSDOS interfacing utility (which corrects various deficiencies in the one supplied by Lattice) are two of these.

How To Use Ls

Ls prints a sorted directory in a multicolumn format. The columns are read like a magazine; that is, an entire column is read from top to bottom, then the next column is read. Ls prints volume labels, which are in boldface, first. Next it prints directories; they are underlined. Then it prints the files. Finally, it prints the amount of space used by the directory (instead of the space left on the disk), along with a file and directory count. The figure (page 21) shows various sample outputs from ls.

Usage is: ls [options] <file list>. The legal command line options are:

-a

List all files, including hidden files. Like Unix, all files whose names begin with a period (.) are considered invisible (even if the hidden attribute bit isn't set); -a causes these files to be printed too.

-c<num>

Print the output in <num> columns instead of the default five. Up to six columns can fit on a screen. The command

```
ls -c1 *.c >foo.bat
```

is useful for making batch files. No spaces are allowed between -c and the number.

-d

List only directories (no files).

-e

Sort by extension and then by filename. This groups all the .c files together and all the .obj files, and so on. The filenames are sorted within their extension categories (i.e., bar.c precedes foo.c, and both precede bar.obj).

-f

List only files (no directories).

-l

List in long format. -l lists the file size, the create time and date, and the file's attributes. Possible attributes are:

```
D – (D)irectory
H – (H)idden
L – Volume (L)abel
M – File has been (M)odified
        since last backup
R – (R)ead only
S – (S)ystem
```

-s

Suppress directory name-underlining and label-boldfacing so that you can direct the output to a printer without the extra escape sequences. Note that directories sort to the top of the list because the graphics require an ESC (which has a smaller value than any letter) at the beginning of the name string. Because the ESC is no longer present, the directory names will be mixed in with the filenames when you use -s.

-u

Print directory unsorted.

You may combine options (ls -ac3s *.c) or list them separately (ls -a -c3 -s *.c). You can place them anywhere on the command line (ls *.c -l is the same as ls -l *.c).

The <file list> is a list of the files you're looking for. You can list files explicitly (ls foo.c bar.c) or with ambiguous references (ls *.c). A directory may precede the filename (ls \src\tools\ls.c). You can use / as well as \ to separate directory names (ls /src/tools/ls.c). If you request a single directory, then ls lists the contents of that directory. If no <file list> is given, ls prints all files in the current directory.

To make the underlining and boldfacing work, you must find the line "device=ANSI.SYS" in your config.sys file when the system boots. If you don't do this, your screen will look funny unless you use the -s option.

Getting a Directory

You can access MSDOS directories in two ways. A set of low-level DOS function calls "Search for First Entry" (0x11) and "Search for Next Entry" (0x12) work much like the equivalent CP/M functions. (An example of CP/M directory accessing appeared in the March 1985 C Chest.) Because functions 11 and 12 force you to construct an FCB, and they search only the current directory, they're pretty hard to use.

Luckily, DOS versions 2 and higher provide two easier-to-use functions: "Find First" (0x4e) and "Find Next" (0x4f). When you pass Find First a pointer to a string holding a filename, it loads information about the requested file into the current disk transfer area (DTA). If you give an ambiguous file reference, subsequent files that match the reference are retrieved with a series of Find Next calls. Find Next must use the same DTA as *Find First used.*

If an error is encountered (such as not finding the requested file), the carry bit is set when the DOS interrupt returns, and an error code is put into the AX register. DOS v.2 sets the AX register to 0 on success, but DOS v.3 doesn't seem to do this; you actually have to look at the carry bit.

These two routines have several nice features. The name can contain ambiguous file references and directory specifiers, so you can search for a file anywhere in the file system. You can even use constructs like ". ." or ". .\. ." if you want. Perhaps nicer, you can use / as well as \ to separate directory names in a complete file reference. (This is actually true for all the low-level MSDOS file functions. The insistence on that idiotic backslash is an anomaly of the MSDOS-supplied shell, command-.com. Why couldn't they use / to separate directories and - for command line switches?) Another nice feature is that you can specify a disk id as part of the filename (c:/foo).

All is not wonderful, though. The two directory-searching routines do have problems. Unix directories are just files. The only difference be*tween a* Unix directory and any other file is the value of the directory attribute bit. MSDOS uses a similar mechanism, except that directories are special: they can't be accessed like other files, they have a length of zero, and so forth. However, they will show up in a directory search as if they were files (with an attribute bit that says you're looking at a directory).

So, if you request a directory from Find First, you'll get a single listing with the matching directory name. You won't get a listing of the files in that directory; to do this, you have to specify /*.* after the directory name. A search for ". ." will return a structure that actually contains the name of the parent directory.

A second problem is the root directory. MSDOS doesn't think that the root directory exists. If you request / or \ from Find First, it comes back with a file-not-found error. Similarly, requesting b: produces the same error. You need to ask for a file (or *.*).

Another problem is the volume label. A filename returned by DOS has all the blank padding removed and a period inserted between the name and extension. Unfortunately, DOS also puts the period into a volume label. If a disk is labeled LONGLABEL, the DOS directory search will yield LONGLABE.L as the volume label. Ls doesn't do anything to correct this problem.

The final problem is actually a deficiency in the Lattice C I/O library. (The people at Lattice claim that the next revision of the compiler fixes this, but that doesn't help us now.) To use Find First/Next, you need to mess with the DTA (the place where the directory will end up after the function call). Unfortunately, your disk I/O system uses the DTA. If you're not doing any file I/O, you can put the DTA where you want it (with a DOS function 0x1a call) and then forget about it. *But*, if you're working with files too, you'll have to put the DTA back where it came from to do file accesses.

```
C:\SRC\LS ls/

HARDFILE    GAMES       SRC         AUTOEXEC.BAT
BIN         INCLUDE     TEXT        COMMAND.COM
DOS         LIB         UTIL        CONFIG.SYS

3 files (22172 bytes, 21 K), 8 directories
C:\SRC\LS ls -la/

HARDFILE              0     12-20-84 17:22:18 LM
BIN                   0     12-24-84 15:13:04 D
DOS                   0     12-20-84 17:23:04 D
GAMES                 0     12-24-84 15:20:32 D
INCLUDE               0     12-24-84 15:17:56 D
LIB                   0     12-24-84 15:22:26 D
SRC                   0     12-24-84 15:13:54 D
TEXT                  0     12-24-84 15:20:58 D
UTIL                  0     12-24-84 15:17:58 D
AUTOEXEC.BAT         95      4-09-85 17:42:50 M
COMMAND.COM       22042      8-14-84  8:00:00
CONFIG.SYS           35      1-22-85 22:19:06 M
IBMBIO.COM         8964      7-05-84 15:00:00 RHS
IBMDOS.COM        27920      7-05-84 15:00:00 RHS

3 files (59056 bytes, 58 K), 8 directories
C:\SRC\LS ls -a/text/drdobbs

.       FOO         LS.NR           SAVE.BAT
..      CROOT.NR    Q_BITMAP.NR     SORT.NR

5 files (56197 bytes, 54 K), 3 directories
C:\SRC\LS ls -c1fe/text/drdobbs

SAVE.BAT
CROOT.NR
LS.NR
Q_BITMAP.NR
SORT.NR

5 files (56197 bytes, 54 K)
C:\SRC\LS ls -d/text/drdobbs

FOO

1 directory
```

Figure 1.
Sample Outputs from Ls

DOS function 0x2f returns the current DTA in ES:BX. Unfortunately, the Lattice DOS interface functions (bdos, int86, etc.) won't let you get at the ES returned by the get DTA function. They push all the registers, do the DOS call, then restore the registers, overwriting the ES returned by DOS. Nor will a subsequent segread( ) call return the value you need for ES.

My solution to this problem was to write a more sensible DOS interface, which we'll talk about in a moment. Although ls doesn't need to put the DTA back, an example of how to restore the DTA is given in Listing Five (page 41).

Program Description

Ls itself appears in Listing One (page 25). MAXDIR (line 12) is the maximum number of files that will be listed; change this if you need more than 132 (132 names are 6 columns by 22 lines).

Ls uses two subroutines that have appeared in previous columns: qsort (*DDJ* #102, April 1985) to sort the directory and getargs (*DDJ* #103, May 1985) to parse command line arguments. The argument table for getargs is on lines 27–37 of Listing One (getargs.h on line 2 is used by getargs, as we discussed back in May).

The global variables on lines 19–26 are set by the various command line switches. The variables on lines 39–44 are more general purpose.

Directories are stored in an argv-like array of pointers to character strings: Dirs (line 39). Dirv and Dirc (40–41) operate like argv and argc. Total (line 42) is the total size, in bytes, of all listed files. Numfiles and Numdirs (43–44) are the number of files and directories found, respectively.

Find_first( ) and find_next( ) (lines 47–77) do the DOS calls. They use the DOS interface function given in Listing Three (page 34).

The REGS structure is *not* the structure of the same name defined in the dos.h supplied by Lattice. This version of REGS is defined in mydos.h (Listing Two, page 33, lines 14–23). REGS is actually a functional superset of the Lattice REGS structure (that is, you can use the REGS defined in mydos.h to talk to the Lattice interface functions, but you can't use the Lattice REGS to talk to the dos( ) function in Listing Three).

REGS has fields for all the registers you need to access MSDOS, including the segment registers. It is a union that defines the normal registers so that you can easily access the high, low, or both bytes. The two register images are superimposed (i.e., definitions for byte offsets are superimposed over definitions for word offsets). For example, given a pointer p to a REGS structure, p→x.ax accesses the AX register, p→h.ah gets the high byte (the AH register), and p→h.al gets the low byte (the AL register). You can retrieve only the AX, BX, CX, and DX registers in this way. The others (SI, DI, ES, CS, SS, DS) are all accessed through the x part of the union (p→x.es).

The interface function, dos( ) (Listing Three), is called with dos(regp); where regp is a pointer to a REGS type structure. It returns the status register, as returned by DOS. You can use dos( ) only with the small and one of the medium (the P) memory models because it assumes a 16-bit pointer.

Dos( ) copies all fields of *regp except x.ss and x.cs into their equivalent registers (lines 59–66 of Listing Three). DOS is invoked with the INT 21 on line 68, then *all* the registers are copied back into *regp. The routine returns with registers restored to their original state (i.e., before the dos( ) call). The function gregs( ) (lines 93–124) gets the register contents but doesn't do a DOS call; it's used for initializing a REGS structure.

Find_first( ) and find_next( ) both return 0 on success or the error code passed back by DOS on failure. When they succeed, the current DTA will be loaded with a shuffled-around version of the directory entry for the requested file. You use the FILE_INFO structure, typedefed in mydos.h (Listing Two, lines 30–39), to access this information. Mydos.h also has a bunch of macros for extracting stuff from packed fields (lines 59–79). With these, you can get at the various fields of the creation time and date, as well as test for specific attributes.

Dirtoa( ) (Listing One, lines 86–135) converts a FILE_INFO structure into an ASCII string (see the figure). The ANSI escape sequences used for underlining and boldfacing are added to the string in dirtoa( ).

The final directory-related routine is fixup_name( ) (Listing One, lines 174–223). This routine takes care of all the problems with accessing the root directory and getting the files inside a specified directory that we discussed earlier. It appends *.* or /*.* onto directory names where necessary.

Add_entry( ) (Listing One, lines 225–253) puts the strings returned from dirtoa( ) into the Dirv array. It also processes the -a and -f flags. If an attribute bit is set when you call find_first( ) or find_next( ), then only those files that have the indicated attributes will be found; the -d option is processed on line 281 using this mechanism. Unfortunately, because there's no way to request files (only directories), -f must be processed by hand in add_entry( ) (lines 241–246).

The only remaining nonobvious function in ls.c is ecmp( ) (lines 147–162), which is used by qsort to do a sort by extension. It skips over to the extension parts of the two strings, compares the extensions, and if they match, goes back and compares the actual filename. Refer to the April C Chest for more information on how qsort works.

Multicolumn Printing

The multicolumn printing is done by ptext( ) (Listing Four, page 37), which prints out an argv-like array of pointers to strings in multicolumn format. Ptext( ) originally was written for a version of the Unix print utility pr and works quite well in that application.

Ptext( )'s calling syntax is:

```
ptext(linec, linev, outfile, numcols,
   colwidth, numrows);
char **linev;
FILE *outfile;
```

Linev is the array of string pointers, linec is the number of pointers in the

array, outfile is the stream to which output will be sent, numcols and numrows are the number of rows and columns, respectively, and colwidth is the width of a single column.

Ptext( ) is called in Listing One on line 170. Here num_cols is the number of columns. The column width is 80/Num_cols. The number of rows is computed with (dirc/Num_cols) + (dirc % Num_cols != 0); dirc is the total number of names to be printed. Dirc % Num_cols != 0 evaluates to 1 if Num_cols doesn't divide evenly into dirc and to 0 if it does divide evenly.

Several practical problems having to do with multicolumn printing will vary from application to application. The first problem is a lone carriage return. Some text formatters underline text by printing the text itself, then a carriage return, then a line containing all the underscores. When you have several columns, you can't print the carriage return because that takes you to the left edge of the page rather than the left edge of the current column. The problem is solved in ptext( ) by backspacing to the correct place. If your printer doesn't support a backspace, you'll have to modify this part of the code (lines 93–103).

The next problem is ESC sequences (which take up no space in the output). Different sequences are composed of different numbers of characters. Rather than try to recognize various escape sequences, ptext( ) just assumes that all sequences use four characters (one for the ESC and three normal characters that follow). This assumption lets us handle the various Set Graphics Rendition (SGR) sequences correctly (we need these for the underlining and boldfacing), but it won't handle the escape sequences needed to change fonts on printers and the like. So, you may need to make this part of ptext( ) more sophisticated (the relevant code is on lines 117–129).

The third problem is with the ANSI.SYS driver provided by IBM (at least with DOS v.3 running on a PC/AT—I'd appreciate someone testing other versions of DOS on other machines and sending me the results). If you try to underline the left-most character on a line, the entire line ends up underlined. If you explicitly turn off underlining, the nonunderlined characters overwrite the underscores, but underscores still extend from the last character you wrote to the physical end of line on the screen (i.e., from the place where you wrote a carriage return to the right edge of the screen). To make matters worse, the underline attribute sticks on, so subsequent lines end up underlined too. I didn't want to rewrite ANSI.SYS, so I kludged a solution to this problem in ptext( ). If "IBM" is #defined at the top of the module, a space will be inserted as the left-most character of every line. I realize this isn't a real solution, but it works everywhere I've needed to use ptext( ).

While we're on the subject of ANSI.SYS, I found another bug a few days ago. The "Erase in Display" and "Erase in Line" leave the cursor in the wrong place. Erase in Display should home the cursor; instead, it leaves the cursor at the left edge of row 2. Similarly, Erase in Line puts the cursor at the left edge of the next line; it shouldn't move the cursor at all. Obviously, the driver is just printing blanks to erase the line, but it should put the cursor back where it belongs when it's done.

The Erase in Line problem is particularly bad when you try to erase the bottom line: the screen scrolls up a line when the cursor is sent past the bottom, so, even if you put the cursor back where it belongs, you lose the top line of the screen. If anyone knows any other bugs in ANSI.SYS (or has fixes for them, maybe a new version of ANSI.SYS), please send them in so I can put them in the column.

This month we talked about directories; next month we'll continue the discussion with a version of the Unix utility "make," which automates the recompilation of modular programs.

C Chest (Text begins on page 20)

Listing One

```
 1: #include <stdio.h>
 2: #include <getargs.h>
 3: #include <mydos.h>

 4: /*-------------------------------------------------------------------------+
 5:  *                  LS.C: An MSDOS directory utility                       |
 6:  *                                                                         |
 7:  *       (c) Copyright 1985, Allen I. Holub. All rights reserved.          |
 8:  *   This program may be reproduced for personal, non-profit use only.     |
 9:  *-------------------------------------------------------------------------+
10:  */

11: #define CARRY    0x01     /* Carry flag mask for an 8086                  */
12: #define MAXDIR   132      /* Largest directory which will be printed      */

13: #define BOLDFACE  "\033[1m"  /* Ansi esc sequence to turn bold face on    */
14: #define UNDERLINE "\033[4m"  /*                    "         underline on  */
15: #define ALL_OFF   "\033[0m"  /*                    "        attributes off */
16:
17: #define EOS(s)    while(*s) s++   /* Position s at the end of the string */
```

 (Listing Continued, text begins on page 20)

Listing One

```
/*------------------------------------------------------------------------*/

int     Longfmt     = 0;        /* Global variables set by command line */
int     Unsorted    = 0;        /* switches.                            */
int     Files_only  = 0;
int     No_graphics = 0;
int     Dirs_only   = 0;
int     List_all    = 0;
int     Num_cols    = 5;
int     Ext_sort    = 0;

ARG Argtab[] =
{
        { 'a', BOOLEAN, &List_all,   "List all files (including hidden)" },
        { 'c', INTEGER, &Num_cols,   "Print output in <num> columns"     },
        { 'd', BOOLEAN, &Dirs_only,  "List only directories"             },
        { 'e', BOOLEAN, &Ext_sort,   "Sort by extension"                 },
        { 'f', BOOLEAN, &Files_only, "List only files"                   },
        { 'l', BOOLEAN, &Longfmt,    "Print directory in long format"    },
        { 's', BOOLEAN, &No_graphics,"Suppress ANSI graphics chars"      },
        { 'u', BOOLEAN, &Unsorted,   "Print directory unsorted"          },
};
#define TSIZE (sizeof(Argtab)/sizeof(ARG))

static char     *Dirs[MAXDIR]   ;       /* Place to put directory       */
static char     **Dirv    = Dirs ;      /* Like argv for directory      */
static int      Dirc      = 0    ;      /* Like argc for directory      */
static long     Total     = 0L   ;      /* Sum of sizes of all files    */
static int      Numfiles  = 0    ;      /* Number of  files   found     */
static int      Numdirs   = 0    ;      /*  "     " directories  "      */
extern char  *malloc();

/*------------------------------------------------------------------------*/

find_first( filespec, attributes, regp )
char    *filespec ;
short   attributes;
REGS    *regp;
{
        /*      Get directory information for the indicated file.
         *      Ambiguous file references are ok but you have to use
         *      find_next to get the rest of the file references.
         *      In this case, The regs structure used by find_first
         *      must be passed to find_next. 0 is returned on success,
         *      otherwise the DOS error code is returned.
         */

        regp->h.ah = (char)  FINDFIRST  ;
        regp->x.dx = (short) filespec   ;
        regp->x.cx = attributes         ;

        return (int)( (dos(regp) & CARRY)  ?  regp->x.ax  :  0 );
}

/*------------------------------------------------------------------------*/

find_next ( regp )
REGS    *regp;
{
        /*      Get the next file in an ambiguous file reference. A
         *      call to this function must be preceded by a
         *      find_first call. The regp argument must be the
         *      same register image used by the find_first call.
         *      0 is returned on success, otherwise the error code
         *      generated by DOS is returned.
         */
        regp->h.ah = FINDNEXT ;
        return (int)(  (dos(regp) & CARRY)  ?  regp->x.ax  :  0 );
}

/*------------------------------------------------------------------------*/

doscall( id, regp )                     /* Do the DOS system call     */
REGS    *regp;                          /* specified by "id", *regp   */
{                                       /* is modified to reflect the */
        regp->h.ah = id ;               /* results of that call       */
        dos( regp );
}

/*------------------------------------------------------------------------*/

dirtoa( s, p, longfmt )
char            *s;
FILE_INFO       *p;
{
```

```
90:         /* Convert a FILE_INFO structure to an ascii string. In longfmt
91:          * all information about the file (name, size, date & time,
92:          * mode) is used. Possible modes are:
93:          *
94:          *      R - Read only           H - Hidden
95:          *      S - System              L - Volume label
96:          *      D - Directory
97:          *      M - archive bit is set (file has been modified)
98:          *
99:          * If longfmt isn't true, only the name is printed. Maximum
100:         * string length in long format is 52, in short format is 21
101:         * Volume label names are printed in bold face (since this will
102:         * put an ESC in the name the volume label will sort to the front
103:         * of the list). Directories are printed underlined (these will
104:         * immediately follow the directories). The number of characters
105:         * in the string (not including the terminating null) is returned.
106:         */

107:        char *startstr = s;
108:        int  i;

109:        if( !No_graphics && (IS_LABEL(p) || IS_SUBDIR(p)) )
110:                sprintf(s, "%s%s%s", IS_LABEL(p) ? BOLDFACE : UNDERLINE,
111:                                     p->fi_name, ALL_OFF );
112:        else
113:                sprintf(s, "%s", p->fi_name );
114:        EOS(s);

115:        if( longfmt )
116:        {
117:                for( i = strlen(p->fi_name); i++ < 12 ; *s++ = ' ' )
118:                        ; /* Pad out the name field */
119:                sprintf( s, " %6ld ", p->fi_fsize );
120:                EOS(s) ;
121:                sprintf(s, "%2d-%02d-%02d %2d:%02d:%02d",
122:                                C_MONTH(p), C_DAY(p), C_YEAR(p)-1900,
123:                                C_HR(p),    C_MIN(p), C_SEC(p)    );
124:                s   += 17 ;
125:                *s++ = ' ';
126:                if( IS_READONLY(p) ) *s++ = 'R' ;
127:                if( IS_HIDDEN(p)   ) *s++ = 'H' ;
128:                if( IS_SYSTEM(p)   ) *s++ = 'S' ;
129:                if( IS_LABEL(p)    ) *s++ = 'L' ;
130:                if( IS_SUBDIR(p)   ) *s++ = 'D' ;
131:                if( IS_DIRTY(p)    ) *s++ = 'M' ;
132:                *s = 0;
133:        }
134:        return( s - startstr );
135: }
136: /*---------------------------------------------------------------------------*/

137: haswild(s)
138: char    *s;
139: {
140:        /*      Return true if s has a '*' or '?' in it  */

141:        for( ; *s ; s++)
142:                if( *s == '*' || *s == '?' )
143:                        return 1;
144:        return 0;
145: }

146: /*---------------------------------------------------------------------------*/

147: ecmp( s1p, s2p )
148: char    **s1p, **s2p;
149: {
150:        /*  Comparison routine for sorting a directory by extension  */

151:        char    *s1, *s2;
152:        int     rval;

153:        for( s1 = *s1p; *s1 && *s1 != '.' && *s1 != ' '; s1++ )
154:                ;
155:        for( s2 = *s2p; *s2 && *s2 != '.' && *s2 != ' '; s2++ )
156:                ;

157:        if( rval = strcmp(s1, s2) )     /* If the extensions don't match */
158:                return rval ;           /* return the strcmp value       */

159:        for(s1=*s1p, s2=*s2p ; *s1==*s2 && *s1 && *s1 != '.'; s1++, s2++)
160:                ;

161:        return(  (*s1 == '.' ? 0 : *s1)  -  (*s2 == '.' ? 0 : *s2) );
162: }
```

 (Listing Continued, text begins on page 20)

Listing One

```
163: /*------------------------------------------------------------------------*/

164: printdir(dirc, dirv)
165: int     dirc   ;
166: char    **dirv ;
167: {
168:         if( !Unsorted )
169:                 qsort( dirv, dirc, sizeof(*dirv), Ext_sort ? &ecmp : 0);
170:         ptext( dirc, dirv, stdout, Num_cols, 80/Num_cols,
171:                                 (dirc/Num_cols) + (dirc % Num_cols != 0) );
172: }

173: /*------------------------------------------------------------------------*/

174: char    *fixup_name( name, regs, info )
175: char            *name;
176: REGS            *regs;
177: FILE_INFO       *info;
178: {
179:         /*  If the name specifies an implicit file (ie. it asks for
180:          *  the directory rather than the files in the directory),
181:          *  modify it to ask for files (eg. ".." becomes "..\*.*").
182:          */

183:         static char         buf[80], *p ;

184:         if( !find_first( name, ALL, regs) )
185:         {
186:                 /* If the requested name was found, and that name is
187:                  * a sub directory, append  \*.* to the requested name
188:                  */

189:                 if( !IS_SUBDIR(info) )
190:                         return( name );

191:                 strncpy( buf, name,   80 );
192:                 strncat( buf, "\\*.*",80 );
193:                 return ( buf );
194:         }
195:         else
196:         {
197:                 /* If you didn't find anything, and a non-explicit
198:                  * directory is requested (ie. "..\.." or "b:" ) then
199:                  * set up the name to get the root directory on the
200:                  * indicated disk.
201:                  */

202:                 if( name[1] == ':' )            /* Take care of the      */
203:                 {                               /* disk designator first */
204:                         buf[0] = name[0];
205:                         buf[1] = ':' ;

206:                         if( !*(name += 2) )
207:                         {
208:                                 buf[2] = '*';   /* If the current dir    */
209:                                 buf[3] = '.';   /* on another disk is    */
210:                                 buf[4] = '*';   /* requested...          */
211:                                 buf[5] =  0 ;
212:                                 return( buf );
213:                         }
214:                         else
215:                                 buf[2] =  0 ;
216:                 }

217:                 for( p = name ; *p ; p++ )
218:                         if( !(*p == '.' || *p == '\\' || *p == '/'))
219:                                 return( name );

220:                 strncat( buf, "\\*.*", 80 );
221:                 return( buf );
222:         }
223: }

224: /*------------------------------------------------------------------------*/

225: add_entry( p )
226: FILE_INFO       *p;
227: {
```

```
228:         /*      Add an entry to the directory array (dirv) and update
229:          *      various associated globals. Hidden files are not
230:          *      printed unless -a was given on the command line.
231:          *      (which will cause List_all to be set). As with
232:          *      Unix, file names starting with the character '.' are
233:          *      also not printed unless -a is specified. If -d was
234:          *      given on the command line (Dirs_only will be true)
235:          *      and files won't be printed.
236:          */

237:         char            buf[64];
238:         register int    strlen;

239:         if( !List_all && (IS_HIDDEN(p) || *p->fi_name == '.')  )
240:                 return 1;

241:         if( IS_SUBDIR(p) )              /* Entry is a directory         */
242:                 Numdirs++;              /* Adjust directory count       */

243:         else if( Dirs_only )            /* Entry is a file              */
244:                 return 1;               /* Don't print files            */

245:         else if( !IS_LABEL(p) )         /* Entry is a file              */
246:                 Numfiles++;             /* Adjust file count            */


247:         strlen = dirtoa( buf, p, Longfmt );
248:         if( ++Dirc > MAXDIR || !(*Dirv = malloc( strlen + 1 )) )
249:                 return 0;

250:         strcpy( *Dirv++, buf );
251:         Total += p->fi_fsize;
252:         return 1;
253: }
254: /*---------------------------------------------------------------------*/

255: main(argc, argv)
256: int     argc;
257: char    **argv;
258: {
259:         /* Get a directory. Given a list of files on the command line
260:          * (file names may be ambiguous) will print a sorted directory
261:          * of those files along with a total file and byte count.
262:          */

263:         static REGS         regs          ;     /* Needed for DOs calls */
264:         static FILE_INFO    info          ;     /* DOs puts dirs here   */
265:         static char         *name  = "*.*";     /* File name            */

266:         gregs  ( &regs  );              /* Initialize the regs structures */
267:         regs.x.dx = (word) &info;       /* Change the Disk Transfer Addr  */
268:         doscall( SETDTA, &regs );       /* to point at info structure     */


269:         argc = getargs( argc, argv, Argtab, TSIZE );

270:         if( Longfmt )                   /* -l is always printed in 1 column */
271:                 Num_cols = 1;           /* even if -cN was given on the cmd */
272:                                         /* line                             */

273:         if( argc >= 2 )                 /* Put the requested file name into */
274:         {                               /* "name." The default is *.*       */
275:                 name = *(++argv);
276:                 if( argc == 2  &&  !haswild(name)  )
277:                         name = fixup_name( name, &regs, &info );
278:         }

279:         do
280:         {
281:                 if( !find_first(name, Files_only ? ALL_FILEs : ALL, &regs))
282:                 {
283:                         if( !add_entry(&info) )
284:                                 break;

285:                         if( haswild(name) )
286:                                 while( !find_next( &regs ) )
287:                                         if( !add_entry(&info) )
288:                                                 goto abort;
289:                 }
290:                 name = *(++argv);

291:         } while( --argc > 1 );

292: abort:  printf   ("\n");
```

 (Listing Continued, text begins on page 20)

Listing One

```
293:          printdir ( Dirc, Dirs );
294:          printf   ("\n");
295:          if( Numfiles )
296:          {
297:                  printf("%d file%s (%ld bytes, %d K)",
298:                                          Numfiles, Numfiles == 1 ? "" : "s",
299:                                          Total, Total/1024 );
300:          }

301:          if( Numdirs  )
302:          {
303:                  if( Numfiles )
304:                          printf(", ");

305:                  printf("%d director%s",Numdirs, Numdirs ==1 ? "y" :"ies");
306:          }
307: }
```

End Listing One

Listing Two

```
  1: /*-------------------------------------------------------------------+
  2:  *                                                                   |
  3:  *       MYDOS.H         Various defines for talking to dos          |
  4:  *                                                                   |
  5:  *-------------------------------------------------------------------+
  6:  *
  7:  *    Typdefs for using dos(). Note that this structure can be used
  8:  *    by the various routines supplied by Lattice (intdos, bdos etc.)
  9:  *    but dos() can't use the structure defined in Lattice's dos.h.
 10:  *
 11:  *    Since "REGS" and "byte" are both also defined in dos.h, you
 12:  *    shouldn't #include both of them in the same place.
 13:  */

 14: typedef short   word;
 15: typedef char    byte;

 16: struct LREG { word  ax,    bx,    cx,    dx,     si, di, es, cs, ss, ds;};
 17: struct SREG { byte  al,ah, bl,bh, cl,ch, dl,dh;                          };

 18: typedef union
 19: {
 20:         struct LREG     x ;
 21:         struct SREG     h ;
 22: }
 23: REGS;

 24: /*--------------------------------------------------------------------
 25:  *      Stuff needed to get a directory from MSDOS
 26:  *
 27:  *  The FILE_INFO type structure is filled by a find first or find
 28:  *  next command (dos system calls 0x4e and 0x4f).
 29:  */

 30: typedef struct
 31: {
 32:         char    fi_resv[21];    /* Bytes 0-20   Reserved by DOS       */
 33:         char    fi_attrib;      /* Byte  21     File attribute        */
 34:         short   fi_time;        /* Bytes 22-23  Create/update time    */
 35:         short   fi_date;        /* Bytes 24-25  Create/update date    */
 36:         long    fi_fsize;       /* Bytes 26-27  File size in bytes    */
 37:         char    fi_name[13];    /* Bytes 28-40  File name & extension */
 38: }
 39: FILE_INFO;


 40: /*
 41:  *   Macros to extract information from a FILE_INFO. In all these macros
 42:  *   the argument "p" is a poshorter to a FILE_INFO structure. Note that
 43:  *   the C_YEAR and C_SEC macros compensate for MSDOS wierdnessess.
 44:  *
 45:  * IS_READONLY(p)        File is read only.
 46:  * IS_HIDDEN(p)          File is invisible in normal directory searches
 47:  * IS_SYSTEM(p)          File is a system file
 48:  * IS_LABEL(p)           Info is a volume label, not a file.
 49:  * IS_SUBDIR(p)          File is a directory
 50:  * IS_DIRTY(p)           True when file is written to and closed, set to
 51:  *                       False by the program backup.
 52:  * C_HR(p)               Hour of last update or create   (0-23)
 53:  * C_MIN(p)              Minute of last update or create (0-59)
 54:  * C_SEC(p)              Second of last update or create (0-59)
```

```
 * C_YEAR(p)            Year of last update or create   (1980-2099)
 * C_MONTH(p)           Month of last update or create  (1-12)
 * C_DAY(p)             Day of last update or create    (1-31)
 */

#define READONLY        0x01            /* Attribute bits       */
#define HIDDEN          0x02
#define SYSTEM          0x04
#define LABEL           0x08
#define SUBDIR          0x10
#define DIRTY           0x20

#define ALL           (READONLY | DIRTY | SYSTEM | HIDDEN | SUBDIR | LABEL)
#define ALL_FILES     (READONLY | DIRTY | SYSTEM | HIDDEN )
#define NORM_FILES    (READONLY | DIRTY )

#define IS_READONLY(p)  ((p)->fi_attrib & READONLY      )
#define IS_HIDDEN(p)    ((p)->fi_attrib & HIDDEN        )
#define IS_SYSTEM(p)    ((p)->fi_attrib & SYSTEM        )
#define IS_LABEL(p)     ((p)->fi_attrib & LABEL         )
#define IS_SUBDIR(p)    ((p)->fi_attrib & SUBDIR        )
#define IS_DIRTY(p)     ((p)->fi_attrib & DIRTY         )

#define C_HR(p)         ( ((p)->fi_time >> 11) & 0x1f )
#define C_MIN(p)        ( ((p)->fi_time >>  5) & 0x3f )
#define C_SEC(p)        ( ((p)->fi_time <<  1) & 0x3e )
#define C_YEAR(p)       ((((p)->fi_date >>  9) & 0x7f ) + 1980)
#define C_MONTH(p)      ( ((p)->fi_date >>  5) & 0x0f )
#define C_DAY(p)        ( ((p)->fi_date      ) & 0x1f )

/*-------------------------------------------------------------------------
 *      Directory related BDOS function numbers
 */

#define FINDFIRST       0x4e
#define FINDNEXT        0x4f
#define SETDTA          0x1a
#define GETDTA          0x2f
```

End Listing Two

Listing Three

```
        TITLE   DOS INTERFACE FUNCTION
        SUBTTL  Copyright 1985 by Allen I. Holub
        NAME    DOS
        INCLUDE DOS.MAC

;-------------------------------------------------------------------------
;
; Data types: The REGS structure is defined as type allregs in mydos.h
;

REGS    STRUC           ; Structure used to transfer register
AX_REG  DW      0       ; contents. Note that this structure is
BX_REG  DW      0       ; a supperset of that defined in the
CX_REG  DW      0       ; dos.h supplied by Lattice. This particular
DX_REG  DW      0       ; structure is defined in \include\mydos.h
SI_REG  DW      0
DI_REG  DW      0
ES_REG  DW      0
CS_REG  DW      0
SS_REG  DW      0
DS_REG  DW      0
REGS    ENDS

OFF     EQU     4       ; Offset to arguments


;-------------------------------------------------------------------------
;
; name:         dos  --  Do a dos function call
;
; synopsis:     #include <mydos.h>
;
;               status = dos( regsp );
;               int     status  ;       /* returned status register   */
;               REGS    *regp   ;       /* pointer to register struct */
;
; description:  This function is a more useful version of the bdos()
;               function provided with the lattice C compiler. It
;               saves the existing machine state, replaces the
;               contents of all registers except CS & SS from the
;               structure pointed to by regp, does an int 21, loads
;               the registers back into the structure, restores the
;               machine state and returns.
```

Listing Three

```
39: ;
40: ; notes:          This function will only work with the SMALL model, since
41: ;                 it assumes a 16 bit pointer. The stack frame is
42: ;                 non-standard too (ie the BP holds the structure pointer
43: ;                 arg rather than the frame pointer).
44: ;
45:           PSEG

46:           PUBLIC  DOS
47: DOS       PROC    NEAR

48:           PUSH    BP                ; Save the old stack frame.
49:           MOV     BP,SP             ;
50:           MOV     BP,[BP+OFF]       ; BP = pointer to register structure.
51:
52:           PUSH    BX                ; Save the current machine state. No
53:           PUSH    CX                ; point in saving AX because it's
54:           PUSH    DX                ; going to be used for the return
55:           PUSH    SI                ; value.
56:           PUSH    DI
57:           PUSH    ES
58:           PUSH    DS
59:           MOV     AX,SS:[BP].AX_REG       ; Set up a new machine state
60:           MOV     BX,SS:[BP].BX_REG       ; using the contents of the
61:           MOV     CX,SS:[BP].CX_REG       ; REGS structure. Don't modify
62:           MOV     DX,SS:[BP].DX_REG       ; the SS or CS registers.
63:           MOV     SI,SS:[BP].SI_REG
64:           MOV     DI,SS:[BP].DI_REG
65:           MOV     ES,SS:[BP].ES_REG
66:           MOV     DS,SS:[BP].DS_REG
67:           PUSH    BP                      ; Make the DOS call. Save the BP
68:           INT     21H                     ; out of irrational paranoia.
69:           POP     BP
70:           MOV     SS:[BP].AX_REG,AX       ; Now update the original structure
71:           MOV     SS:[BP].BX_REG,BX       ; to reflect the return values from
72:           MOV     SS:[BP].CX_REG,CX       ; the dos call.
73:           MOV     SS:[BP].DX_REG,DX
74:           MOV     SS:[BP].SI_REG,SI
75:           MOV     SS:[BP].DI_REG,DI
76:           MOV     SS:[BP].ES_REG,ES
77:           MOV     SS:[BP].CS_REG,CS
78:           MOV     SS:[BP].DS_REG,DS
79:           MOV     SS:[BP].SS_REG,SS
80:           POP     DS
81:           POP     ES
82:           POP     DI              ; Restore the previous machine state
83:           POP     SI
84:           POP     DX
85:           POP     CX
86:           POP     BX
87:           LAHF                    ; Return the flags:
88:           MOV     AL,AH           ;       Move flags to LSB
89:           XOR     AH,AH           ;       Clear high bytes

90:           POP     BP              ; Restore the previous stack frame's
91:           RET                     ; frame pointer and return.
92: DOS       ENDP
93: ;---------------------------------------------------------------------
94: ;
95: ; name:           gregs  --  Initialize a REGS structure to the current
96: ;                           register contents.
97: ;
98: ; synopsis:       #include <mydos.h>
99: ;
100: ;                gregs( regp )
101: ;                REGS    *regp   ;       /* pointer to register struct */
102: ;
103: ; description:   This function is used before before calling dos().
104: ;
105: ; notes:         This function will only work with the SMALL model.
106: ;
107:           PUBLIC  GREGS
108: GREGS     PROC    NEAR

109:          PUSH    BP                ; Save the old stack frame.
110:          MOV     BP,SP             ;
111:          MOV     BP,[BP+OFF]       ; BP = pointer to register structure.

112:          MOV     SS:[BP].AX_REG,AX ; Initialize the structure pointed
113:          MOV     SS:[BP].BX_REG,BX ; to by bp.
```

```
114:            MOV     SS:[BP].CX_REG,CX
115:            MOV     SS:[BP].DX_REG,DX
116:            MOV     SS:[BP].SI_REG,SI
117:            MOV     SS:[BP].DI_REG,DI
118:            MOV     SS:[BP].ES_REG,ES
119:            MOV     SS:[BP].CS_REG,CS
120:            MOV     SS:[BP].DS_REG,DS
121:            MOV     SS:[BP].SS_REG,SS

122:            POP     BP                      ; Restore the previous stack
123:            RET                             ; frame and return.

124: GREGS      ENDP
125:            ENDPS
126:            END
```

End Listing Three

Listing Four

```
  1: #include <stdio.h>

  2: /*
  3:  *                  PTEXT.C: Multi-column print utility
  4:  *
  5:  *        Copyright (c) 1985 Allen I. Holub,  all rights reserved.
  6:  *    This program may be reproduced for personal, non-profit use only
  7:  *
  8:  *      Bugs and Features:
  9:  *
 10:  *      1) Pr_line assumes that the output device supports backspace
 11:  *         ('\b' == ^H ).   This is only a problem when  the  source
 12:  *         has underlined text implemented as:
 13:  *                    texttexttext\r
 14:  *         We can't just ouput a '\r' in this case because we may
 15:  *         not be in the leftmost column. A \r should get us to the
 16:  *         left edge of the current column.
 17:  *
 18:  *      2) When printing multi-column stuff. If a line is truncated it
 19:  *         will run into the column to its right. That is, there is no
 20:  *         seperator between columns other than the  whitespace needed
 21:  *         to pad the column out to a particular width. If no  padding
 22:  *         is necessary (ie. the lines have been truncated), then  the
 23:  *         columns will run together.
 24:  *
 25:  *      3) When an ESC is found, the escape and the next three characters
 26:  *         take up no space in the output. This lets us print out the
 27:  *         various SGR commands without messing up the column width.
 28:  *
 29:  *      4) Strange things happen to an IBM screen when the lettmost
 30:  *         character on a line is printed with underline. To compensate
 31:  *         for this, a single blank is printed as the left-most
 32:  *         character on every line if IBM is #defined.
 33:  */
 34: #define ESC     0x1b
 35: #define IBM     1
 36: /*------------------------------------------------------------------------*/

 37: ptext(linec, linev, outfile, numcols, colwidth, numrows)
 38: int     linec, numcols, colwidth;
 39: char    **linev;
 40: FILE    *outfile;
 41: int     numrows;
 42: {
 43:         /*      Print out the array of strings "linev" which consists of
 44:          *      linec entrys. Output is sent to "outfile" formated as
 45:          *      follows:
 46:          *
 47:          *      "numcols"  : number of columns
 48:          *      "colwidth" : width of a column in characters. Any text
 49:          *                   longer than colwidth is truncated off.
 50:          *      "numrows"  : columns are numrows long. The left-most
 51:          *                   column is printed in its entirety
 52:          *                   first then the next column in its
 53:          *                   entirety, and so on.
 54:          */
 55:         register int    j ;
 56:         register char   **lineend, **line, **nextline ;

 57:         lineend = &linev[linec-1];      /* Last linev entry to print    */

 58:         for( j = numrows ; --j >= 0 ; )
 59:         {
 60: #ifdef IBM
 61:                 putc( ' ', outfile );
 62: #endif
```

 (Listing Continued, text begins on page 20)

Listing Four

```
63:                 for( line = linev++;  line <= lineend; line = nextline)
64:                 {
65:                         nextline = line + numrows;
66:                         /*   Print the line. Don't pad the rightmost    */
67:                         /*   column.                                    */
68:
69:                         pr_line( *line, outfile, colwidth,
70:                                                 nextline <= lineend);
71:                 }

72:                 fputs("\n" , outfile);
73:         }
74: }
75: /*-----------------------------------------------------------------------*/

76: static pr_line( str, stream, width, padded )
77: register char   *str;
78: FILE            *stream;
79: int             width, padded;
80: {
81:         /*      Print out "str" into "stream" padding it to "width"
82:          * columns wide. Non-printing characters and 3 printing characters
83:          * immediatly following an ESC are not counted as having printed.
84:          * '\n' characters are treated as line teminators but are not
85:          * printed. If "padded" is 0 then no padding is done, though the
86:          * line will still be truncated if it's too long.
87:          */
88:         int     col = 0 ;
89:         while( col < width  &&  *str )
90:         {
91:                 if ( *str == '\n' )
92:                         break;
93:                 else if( *str == '\r' )
94:                 {
95:                         /* Back up to the left edge of the current column
96:                          */

97:                         while( col >  0 )
98:                         {
99:                                 --col;
100:                                putc('\b' , stream);
101:                        }

102:                        str++;
103:                }
104:                else if( *str == '\t' )         /* expand tabs */
105:                {
106:                        str++ ;
107:                        col++ ;
108:                        putc( ' ' , stream );

109:                        while( (col % 8)  &&  col < width)
110:                        {
111:                                putc( ' ' , stream );
112:                                col++;
113:                        }
114:                }
115:                else
116:                {
117:                        if( *str == ESC )
118:                        {
119:                                putc( *str++ , stream );
120:                                if( !*str )
121:                                        break;

122:                                putc( *str++ , stream );
123:                                if( !*str )
124:                                        break;

125:                                putc( *str++ , stream );
126:                                if( !*str )
127:                                        break;
128:                        }
129:
130:                        else if (*str == '\b')
131:                                --col ;
```

```
132:                        else if( *str >= ' ' )
133:                                ++col ;

134:                        putc( *str++ , stream );
135:                }
136:        }

137:        if (padded)
138:                while( col++  <  width )
139:                        putc( ' '  , stream );
140: }
```

End Listing Four

Listing Five

```
 1: /* Example of how to get and restore the MSDOS Disk Transfer address
 2:  * using the dos() and gregs() functions defined in listing 3
 3:  */

 4: #include <mydos.h>

 5: #define SETDTA  0x1a
 6: #define GETDTA  0x2f

 7: main()
 8: {
 9:         static FILE_INFO info;
10:         static REGS      oregs, nregs;

11:         gregs( &oregs );                /* Get the old DTA. It will be put */
12:         oregs.h.ah = GETDTA;            /* into ES:BX of oregs.            */
13:         dos( &oregs );

14:         gregs( &nregs );
15:         nregs.x.dx = (word) &info;      /* Change the Disk Transfer Addr  */
16:         nregs.h.ah = SETDTA ;           /* to point at info structure     */
17:         dos( &nregs );

18:         /*   . . .   */

19:         oregs.x.ds = oregs.x.es;        /* Now put the DTA back. You have  */
20:         oregs.x.dx = oregs.x.bx;        /* to copy ES:BX into DS:DX first. */
21:         nregs.h.ah = SETDTA ;
22:         dos( &oregs );
23: }
```

End Listings

Build a Custom PC or Clone

by Jim Kronman

I have owned an 8-bit S-100 microcomputer since 1978. Old habits are hard to kick, so I set about assembling a PC-compatible computer the same way I built my S-100 system. I individually selected each component of the system to suit my specific needs. I now have a PC XT work-alike that is totally "IBM compatible," costs substantially less than the closest off-the-shelf equivalent, and fits my requirements exactly.

My approach may not be for everyone. I am pleased with my completed project, but you should consider the pros and cons before taking the plunge yourself. Before going into the advantages of rolling your own PC, I'll point out some of the drawbacks you should consider.

Careful shopping, gentle persuasion with vise grips and a lot of patience yield an XT work-alike at a PC price.

You will need patience. If you must have a computer tomorrow, go buy something off the shelf of your local computer store. I ordered pieces from five separate mailorder suppliers (including IBM), and it was over five weeks from the time I placed the orders by telephone until I first booted the system. At that point, I was still missing a vital element, but more on that subject later.

Do not attempt to assemble your own PC if you are not experienced with microcomputer hardware or are timid when handling equipment costing, in the aggregate, up to several thousand dollars. When you put Manufacturer A's board and Manufacturer B's board into your computer, you alone are responsible for determining that the boards are compatible both with each other and with the PC itself. The vendors and manufacturers usually will try to help if you have a problem, but they may never have encountered exactly the configuration you have assembled and therefore may be unable to determine the cause of your problem.

Now that you have received fair warning of some of the potential pitfalls, I'll shamelessly promote the advantages of building your own PC. You can save a bundle of money when you compare your homegrown computer to one with the same functionality in a store: for the same amount of money, you can get a lot more capability. With my project now complete, I realize, too, that there are other benefits I had not anticipated when I began.

An obvious reason to undertake this adventure is to assemble a system exactly suited to your needs. One of my prime requirements was to have a Selectric style keyboard, which IBM does not offer with the PC. I did not want to buy the standard IBM PC because I would end up throwing away a perfectly good (if poorly laid out) key-

Jim Kronman, 6085 Venice Blvd., No. 16, Los Angeles, CA 90034 (213) 558-3281

board and buying a replacement. (As far as I know, IBM will not sell the PC System Unit without a keyboard.)

I purchased the Keytronic KB5151 deluxe keyboard with a Selectric style layout and separate numeric and arrow keypads; it also has the function keys across the top of the keyboard so they will align with the function key legends in row 25 of the display in many application programs. My one criticism of the Keytronic design was that the "touch" was too soft; sometimes the slightest tap would cause a character to appear on the screen. This problem was easy to remedy. I carefully took each keytop off of the keyboard, stretched the spring under the keytop to about 0.8 inches, and then replaced the keytop. Do one key at a time and you should have no problems with this slight modification; it's well worth the time!

I had decided that I wanted the capability of a hard disk and that I wanted it internal to the computer. I also did not like the crowded arrangement in the eight-slot PC XT or the XT clones. These considerations and others led me to opt for a custom assembly with a five-slot PC motherboard. I'm sure you can think of plenty of your own reasons to customize, too.

While my project was in progress, I had occasion to help a friend with an IBM PC that had a failed power supply. What I have come to call the "$290 Fuse Syndrome" is a strong argument to buy as few components as possible from IBM. Big Blue's warranty policy (which I have seen reviewers praise in several magazine articles) is to replace defective components on the spot and without question. That's great while you are within the warranty period and the tab is on Blue, but unfortunately the policy continues and is *mandatory* after the warranty expires and you are paying the bill. If you try to obtain a schematic or other service information for an IBM PC component that is designated a FRU (Field Replaceable Unit), you will encounter as firm a stonewall from IBM as you will ever want to see. Even if you figure out who made the component for IBM, you can't get any help because IBM has its vendors contractually bound to stony silence as well.

The $290 Fuse Syndrome is so named because inside of each IBM power supply is a standard slow-blow radio-type fuse—in a fuse holder! If this fuse blows (or even simply fails), IBM's policy is that the supply is terminally (if you'll pardon the expression) ill and must be replaced. A PC XT (130 watts) power supply costs $290 when ordered from IBM; as I write this article, the going rate for a clone replacement seems to be around $170. By purchasing your power supply, plug-in boards, disk drives, and so on from other sources, you can obtain servicing information in most cases. If the availability of servicing information is vital to you, check with each manufacturer before you buy.

My original plan was to purchase a PC motherboard, the case, and a 130 watt power supply from IBM. This would have given the machine the appearance of being "true Blue." IBM was one step ahead of me: I was unable to obtain the entire case from IBM, even though all the pieces are listed in the IBM PC Service Manual's illustrated parts breakdown and the Greencastle Parts Depot accepted my order. I did buy the power supply, which has given me no problems to date (although the first supply sent to me was dead on arrival), but I regret not saving the $100 by buying a supply from a third party.

My inability to obtain the case from IBM was frustrating; after the IBM Parts Center accepted my order for all of the pieces, it simply failed to deliver all of them. I never received notification that the entire order would not be filled. As the weeks dragged by, I assembled all of the components of the by now functional computer on a 24 × 27-inch piece of particle board. When I finally found out what was happening with my order from IBM, the Parts Depot at least let me return for credit the pieces of the case I had obtained and could not use.

I found a clone chassis made in Taiwan for about half the price of the IBM parts, and after a small amount of persuasion with vise grip pliers to straighten out a bracket, everything fit fine. The case is identical in appearance to the IBM from the front, except for the absence of the famous logo, and the back panel has a variety of cutouts adaptable to various hardware configurations.

One piece of the system that had to be genuine IBM was the motherboard, which contains the IBM ROM BIOS, thus assuring total PC compatibility. Although some of the clone boards might be close to 100% compatible, the difference in price was not worth the risk. If you have reliable information about a specific non-IBM motherboard that meets your needs, then go for it!

One other specific piece of hardware made the choice of the PC (rather than XT) motherboard attractive: Maynard Electronics' "Sandstar Series" WS-2 disk controller board supports up to four floppy disk drives and two 10 Mb hard disks and takes up only a single slot. An added advantage of this configuration is that the Maynard hard disk software driver is in a PROM that is installed on the main circuit board, occupying address space in the F0000h segment, while the IBM XT controller is a separate board and has its driver in the middle of the C0000h segment. Thus, by not using the XT controller or any other board with ROM code in the C0000h segment, you make available 64K of memory space along with the D0000h and E0000h segments, which allows you to install a 192K RAM disk without using any of the 640K available to DOS. Use caution in dealing with the C0000h through E0000h segments; according to the IBM Technical Reference Manual (page 2-11), plug-in boards can use the area from C2000h to F4000h for software drivers that the ROM BIOS will recognize at power-on.

When you do not buy the System Unit from IBM, you do not receive the IBM PC Guide to Operations manual, the BASIC manual, or the IBM Diagnostic Disk. You may purchase these items from IBM if you need them. If you purchase the IBM Technical Reference Manual, you will not need the Guide to Operations. (If you need this Guide to Operations, you probably shouldn't be

assembling your own computer!) I highly recommend that you purchase the Technical Reference Manual; it is the definitive source of information on the ROM BIOS function calls, containing the source code for the entire ROM BIOS.

After I completed my project (and after my friend purchased a replacement power supply), I discovered that Howard W. Sams publishes a "Computerfact" package for the IBM PC. It costs $39.95 and includes schematics, circuit board photos with component identifications and trouble-shooting instructions. The package covers the PC motherboard, power supply, keyboard, display adapters, floppy disk controller, and a few other optional cards.

You must also purchase an operating system separately. The various choices are PCDOS 2.1 or 3.0 from IBM and Digital Research's CP/M-86, Concurrent CP/M v3.1, or Concurrent DOS v3.2. By far the most software is available for PCDOS. Because v3.0 seems to offer the single user little advantage over v2.1, I selected PCDOS 2.1. The choice is yours. You can use more than one operating system (but not simultaneously).

That is about all there is to tell about my experiences. Not only would I do it again this way if I had to (less the mistakes, of course), I would not consider doing it any other way! What follows is a summary of my advice and some useful tabulations of information, components, and sources.

If you intend to assemble your own PC:

(1) Be confident of your skills and knowledge in microcomputers.
(2) Be prepared to wait; this isn't a project for the impatient.
(3) Select your components carefully; make sure each item is compatible with the others.
(4) Understand each vendor's return policy before you order, just in case.
(5) Select your software with as much or more care as your hardware.
(6) If you have a specific software application in mind, make sure the hardware you select is compatible.

The components you will need (as a minimum) are:

(1) Motherboard, with CPU, ROM BIOS, memory, keyboard interface, and support circuits
(2) Power supply, 62 to 130 watts (130 watts recommended)
(3) Display adapter card
(4) Monitor compatible with display adapter
(5) Disk controller card (possibly a second card for hard disk)
(6) Case
(7) Keyboard
(8) Floppy disk drive, 5¼-inch DSDD, 48 tpi, PC-compatible

Optional components include (but are not limited to):

(1) Multifunction card (additional memory, battery run clock, serial and parallel ports, game port, etc.)
(2) MODEM card
(3) Math coprocessor chip (8087)
(4) Coprocessor board (Z80 CP/M-80, Apple, 10286, etc.)

The following items are available from IBM:

(1) PC motherboard with 64K memory installed, P/N 8654213, $640.00
(2) PC XT motherboard with 128K memory installed, P/N 8529254, $750.00
(3) Speaker with mounting bracket, P/N 8529143, $8.15
(4) Guide To Operations manual
(5) Technical Reference Manual, P/N 1502234
(6) BASIC reference manual, P/N 6025010
(7) PCDOS 2.1, P/N 6024120
(8) PC Diagnostic Disk, P/N 1502212

All of the above except for the motherboards and speaker should be available through the IBM Product Centers or other retail outlets (Sears Business Centers are usually a good place to check). The replacement parts (motherboard, etc.) are available from the IBM Parts Depot in Greencastle, IN. It is best to deal by telephone: (317) 658-2022. The address is P.O. Box 505, Greencastle, IN 46135. The hardware prices may change; they are included here for reference only.

For the other pieces that you will need for your system, look through the ads in this magazine, *The Computer Shopper*, and *Byte*. Many firms sell PC clone cases, power supplies, keyboards, and so on. I have even found bargains at surplus companies; a firm in the Los Angeles area recently had surplus Keytronics keyboards manufactured with special legends on the keys for just $35!

Happy hunting and happy cloning. I hope this article encourages fellow computer hobbyists to explore the PC bus in the same spirit as the S-100 bus. I have found the experience to be well worth the effort.

DDJ

The Ultimate Parallel Print Spooler

by Don Rindsberg

Don Rindsberg has come up with a printer spooler design incorporating low-cost 6665 (64K) dynamic RAM chips. The result—a 60K print spooler that you can build for about a hundred dollars.

If you are tired of waiting while your parallel printer does its thing, here is a project that will allow you to print and compute at the same time. It is a print spooler that provides a whopping 60K of buffering. The spooler, designed to stand alone, will interface almost any parallel printer port.

The project incorporates "software refresh" of the dynamic RAM memory—a concept little publicized, to my knowledge, except in Motorola's application literature. By using software refresh of RAM, you can eliminate a number of integrated circuits from the design of the spooler. The refresh cycle ties up the microprocessor for less than 10 percent of its time, a penalty easily tolerated when you are controlling a printer, which operates at a snail-like speed from the microprocessor's point of view. You implement software refresh by carefully writing the software so that the 128 consecutive row addresses are read at least every 2 ms: 128 rows in a dynamic RAM must be refreshed, and the maximum allowable time between refreshes is 2 ms.

Don Rindsberg, 5958 S. Shenandoah Rd., Mobile, AL 36608.

I built this spooler to interface my Apple II computer to an Epson parallel printer. However, it should be easily adaptable to virtually any computer with a parallel printer port and any parallel printer. The 60K of buffering holds at least 12 pages of solid, single-spaced text. In the graphics mode, an entire screen is dumped to the spooler in a couple of seconds.

This design uses software refresh of dynamic ROM to create a 60K spooler for about $100.

The Hardware

The 6809 microprocessor, which is now available for about $12, is ideal for our software-refreshed memory system: its internal clock can provide the time-critical signals to the dynamic memory without the use of additional clocks or delay lines. The 6809 (not the 6809E) has its own on-chip clock generator and requires only a crystal in the 3.5 to 4.0 MHz range; a low-cost TV color-burst crystal is a simple and inexpensive choice.

The 6809 produces two clock signals, E (enable) and Q (quadrature), which are both 50 percent duty cycle square waves at a frequency exactly one-fourth that of the crystal. As shown on the timing diagram (Figure 1, page 47), the Q signal leads the E signal by one-fourth of a cycle—the E signal, by the way, is the same as the phase two signal on the 6800 and 6502 microprocessors. We use the rising edge of Q to trigger RAS (row address strobe) on the dynamic RAM. The rising edge of E causes the switch from row addresses to column

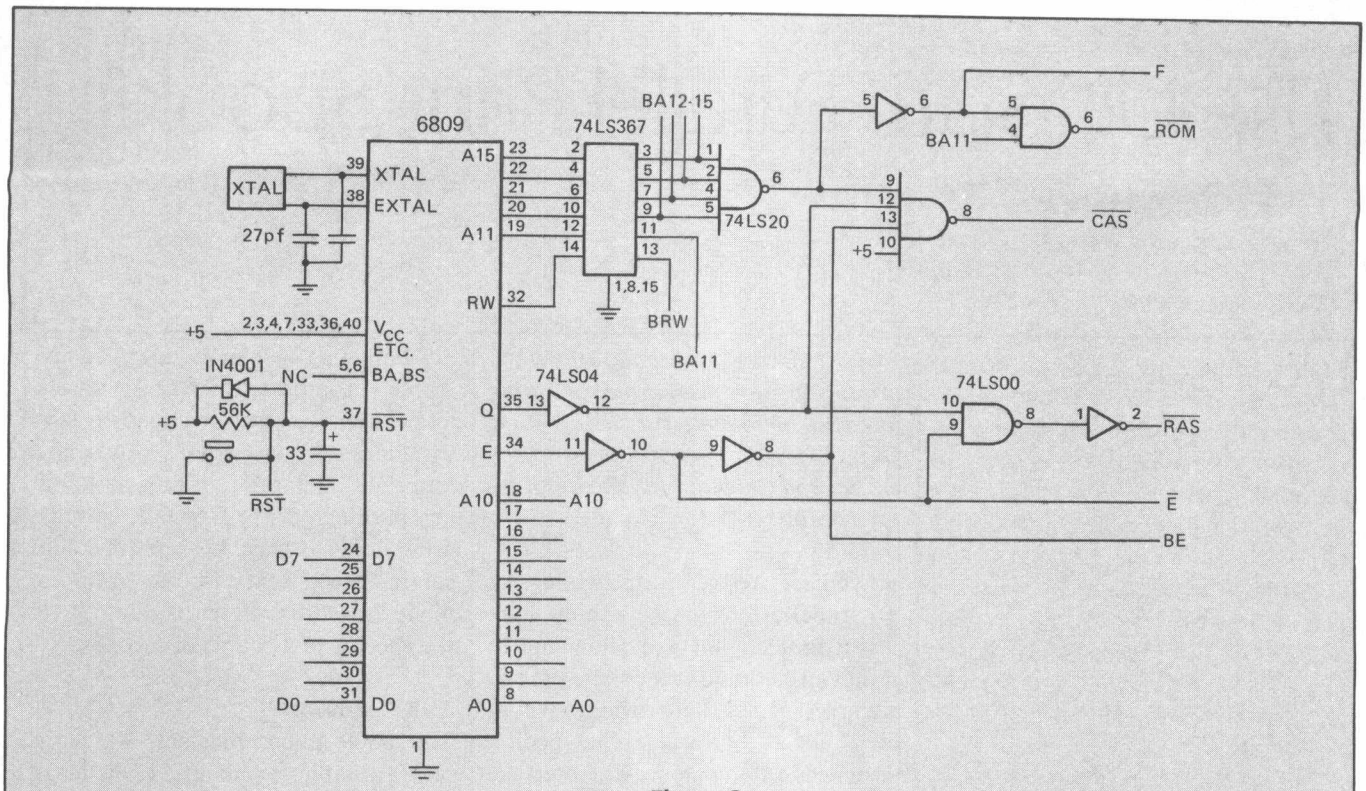


Figure 2.
The wiring diagram shows the 6809 and the address decoding. A 3–4 MHz crystal is required (frequency is not critical).

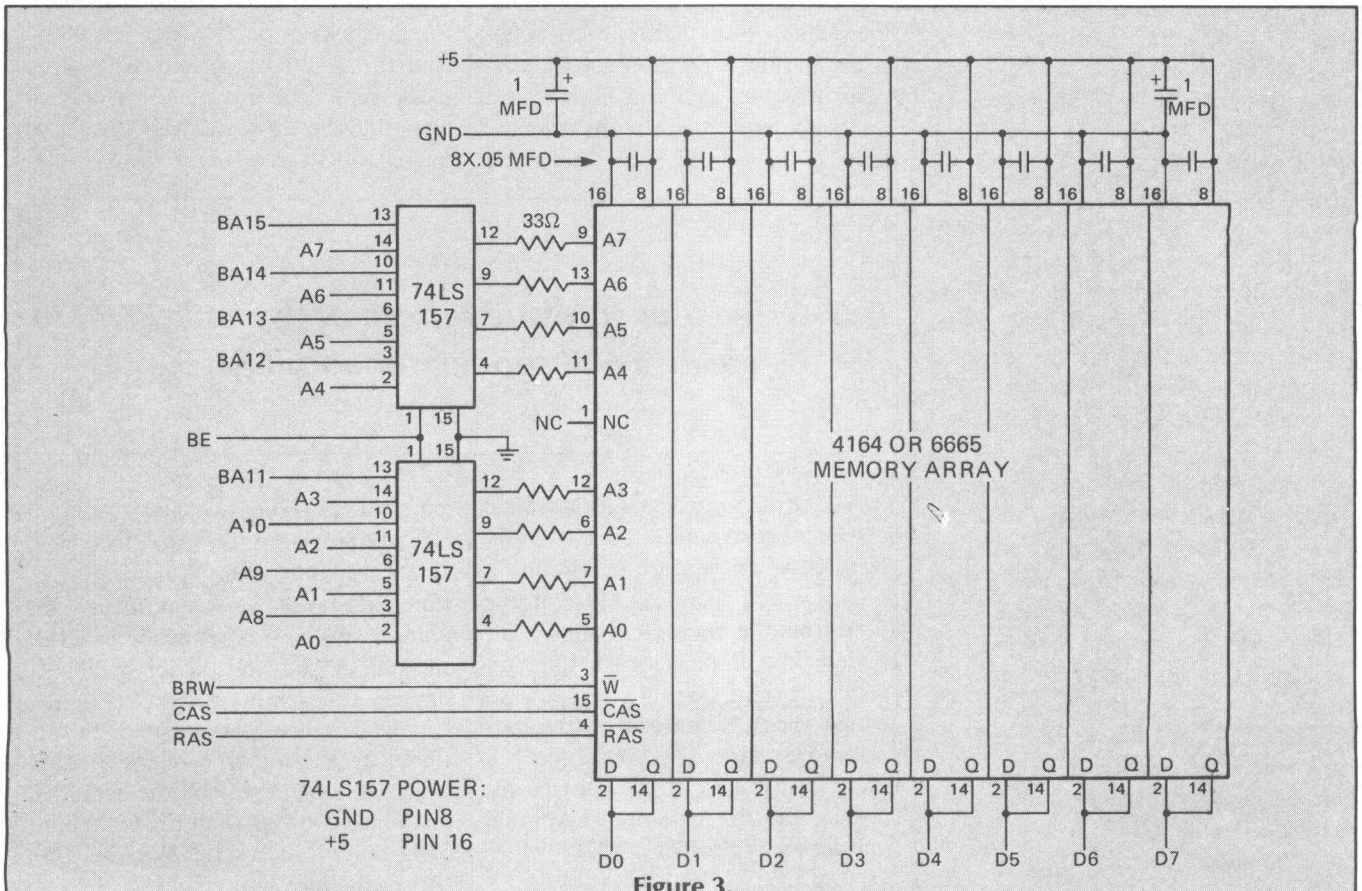


Figure 3.
The eight dynamic RAM chips are wired in parallel. Make identical connections to each chip with the exception of the data lines; a different data line connects to each chip as shown. Be sure to install a decoupling capacitor near each 6665.

addresses, while the falling edge of Q triggers CAS (column address strobe) to the RAM chips. RAS is exerted every cycle to force a refresh, while CAS is exerted only if the address is not an "F" address ($F000–FFFF).

Figure 2 (page 48) shows that we decode address F in the 74LS20, which produces a low when A12–A15 are all high. The 2716 ROM is at $F800–FFFF, and the PIA (parallel interface adapter) is at $F400–F403; address 11 provides the distinction between the ROM and the PIA. We buffered A11–A15 with a 74LS367 because the 6809 has limited drive capabilities. Figure 3 (page 48) shows how the switching between the row addresses A0–A7 and column addresses A8–A15 was done in a pair of 74LS157 chips. The 33 ohm resistors prevent ringing on the RAM's address lines.

The 6821 PIA in Figure 4 (page 50) was designed for applications such as this, having exactly the number of functions we need. It automatically provides the handshaking signals used by parallel printers. When the A side is used as a parallel input, STR from the host computer sets a flag. When the spooler reads the data, the PIA automatically provides a one-cycle negative-going signal, which is used for ACK. On the B side of the PIA, a similar STR signal automatically is generated on a write to port B, and the printer's ACK signal sets a flag to signal that the printer is ready for another character.

Figure 5 (page 50) shows the completed spooler. A 74LS74, used in conjunction with the pause switch, allows manual shutdown of the printout to change paper and so on. This chip changes state at the falling edge of E when you change the switch position. At this time, the opposite half of the ROM is bank-switched into memory, which contains a holding routine. The two halves of the ROM are selected by its A10 line. The power supply (Figure 6, page 51) provides regulated 5 volts at somewhat less than 500 ma.

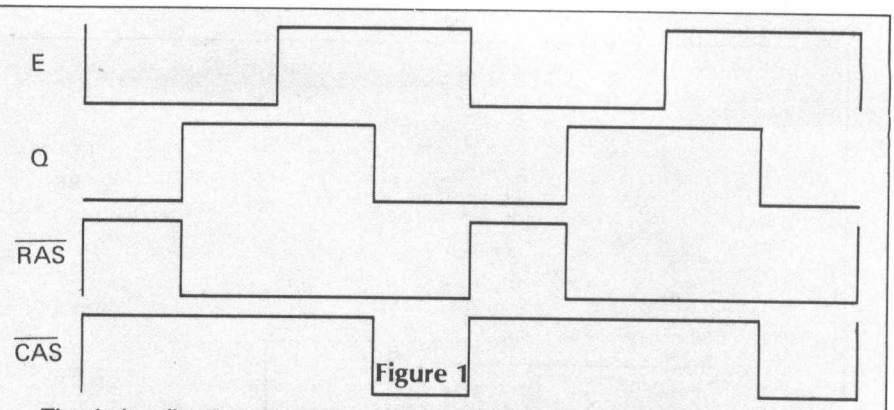


Figure 1

The timing diagram shows how the signals E and Q from the 6809 gate the column (CAS) and row (RAS) logic in the 6665 dynamic memory chips. This eliminates the need for delay circuits.

The Software

The software has two unusual requirements:

(1) The software must read every possible combination of the seven low-order addresses (A0–A6), at least one every 2 ms, to refresh the dynamic RAM.

(2) The software must bank-switch between the upper and lower halves of the ROM to provide the pause function.

Aside from this, the software is straightforward. The 6809 supports relocatable programs with its "long branch" instructions. Listing One (page 53) shows how these functions are implemented.

After a short routine to initialize the PIA, the program wakes up the printer with a null and sets up the X, Y, and U registers as pointers and counters. It next enters a loop that starts with a RAM refresh, which consists of stepping through 128 consecutive ADDA #0 instructions; this steps through our 128 consecutive addresses. The Motorola literature recommends 128 NOPs, which execute in 256μs, but I have elected to use 64 consecutive addresses in 128μs. During this sequence, once each byte, RAS on the dynamic RAMs is activated. Notice that the eight low-order addresses correspond to the 128 rows in the dynamic RAMs even though we are executing code reading the ROM. CAS remains high, and because the RAMs require both RAS and CAS for a read or write, the RAMs do not assert any data on the data bus. Refresh of the RAMs, however, does take place because the RAS signal alone sets off the refresh internally.

After the refresh sequence, the actual input, storage, and output of characters take place. It is critical that this routine not use any secondary loops that might prevent refresh. For example, if the printer is not ready, the routine continues on rather than waiting in a tight loop for it to get ready.

To implement a pause function with minimum hardware, we have put the same code in both halves of the ROM except for one instruction (see Listing Two, page 55): a BRA (branch always) operation in the active half of the ROM and a BRN (branch never) operation in the pause half. The change of a single bit ($20 vs. $21) causes the output portion of the routine to be bypassed when you close the pause switch. See the parts list (page below) for a source for a preprogrammed ROM if you do not care to program your own 2716.

Construction

The only critical aspect of the spooler board's construction is to provide heavy power connections and good bypassing for the dynamic RAM chips. I used the Vector 3677-2 board, which has wide ground and 5 volt busses. This 4.5 × 6.5-inch board will accommodate the entire circuit, except for the transformer and rectifier, if you position the IC sockets carefully. If you are not sure, use the 4.5 × 9-inch board (Vector 3677).

The RAM bypass capacitors must be close to the power pins of the RAM chips. I used stripped wire-wrap throughout, rather than Just-wrap or equivalent, because my technique with the shortcut methods leaves an occasional bad connection, which is tedious to debug. Because we do not use the edge-connector, there is space

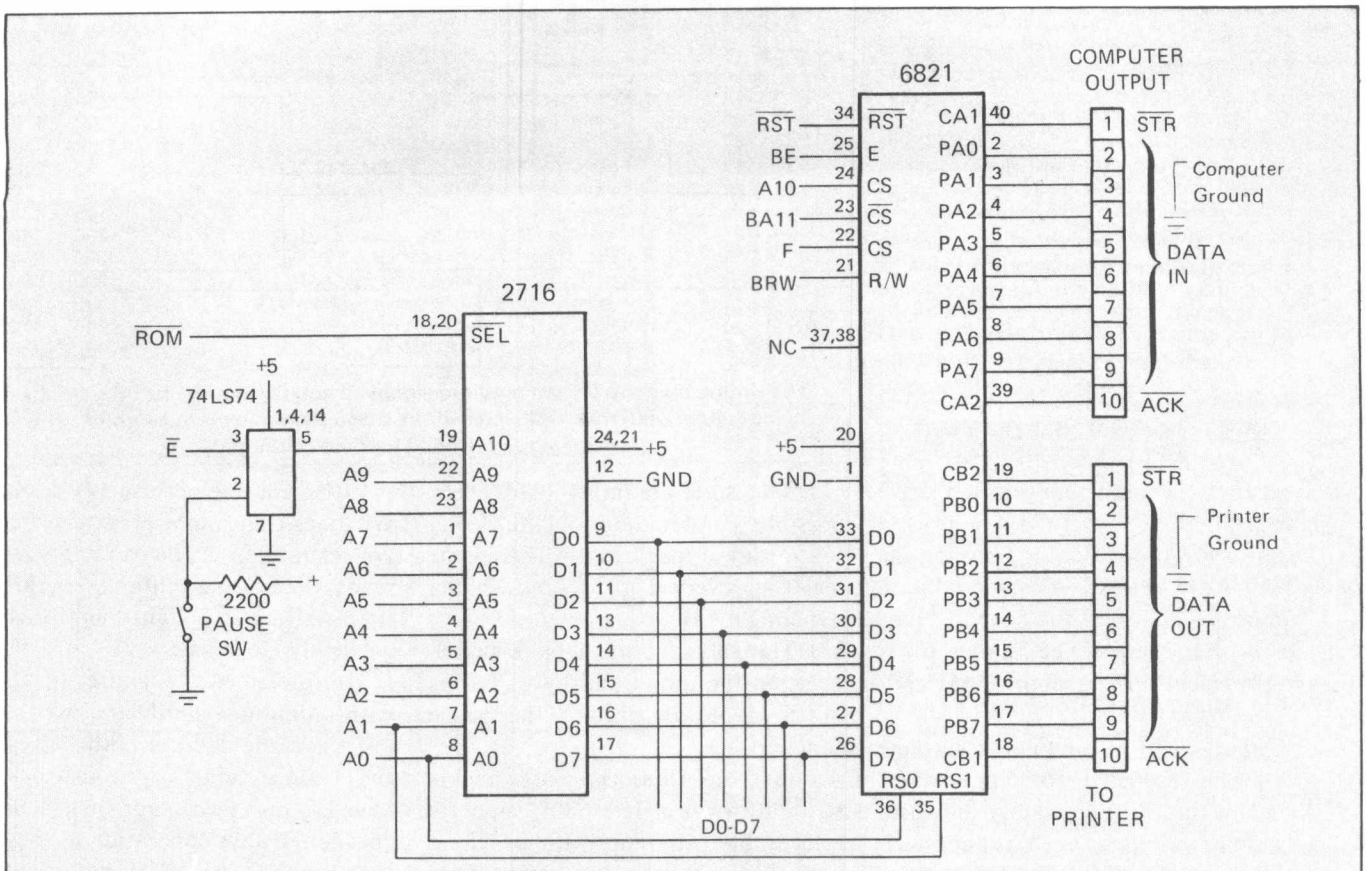


Figure 4.
Wiring of the 2716 EPROM and the 6821 parallel interface adapter.

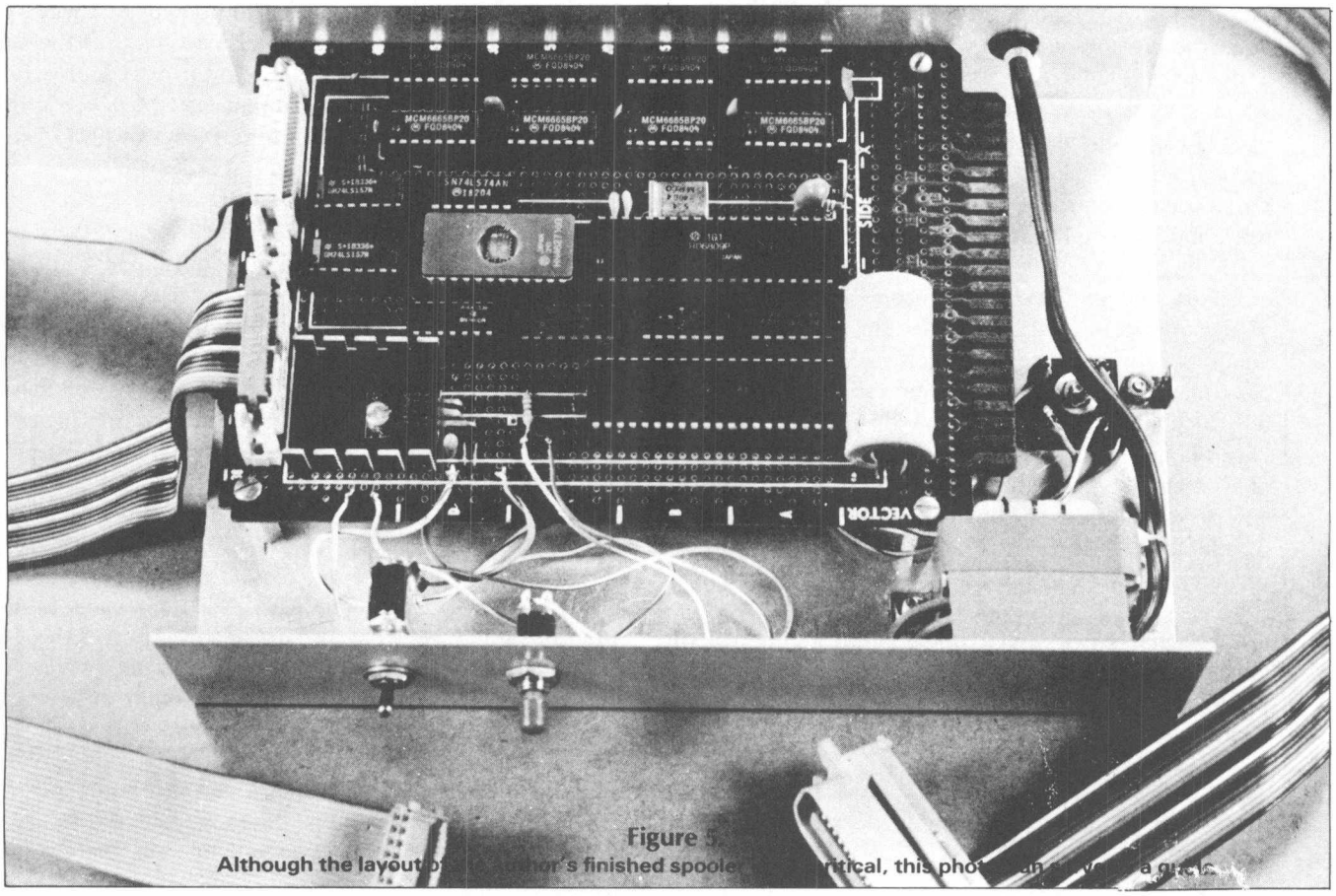


Figure 5.
Although the layout of the author's finished spooler is not critical, this photograph may be a guide.

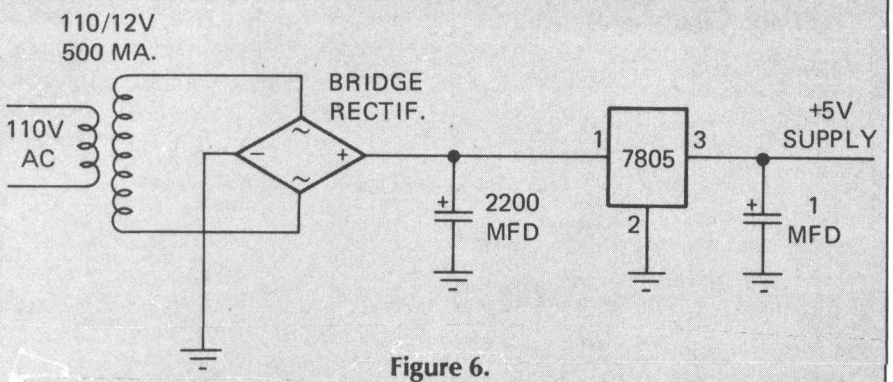


Figure 6.
The rectifiers, transformer, and switches are mounted in the enclosure, while the rest of the power-supply components fit on the circuit board (see Figure 5).

Parts List

Integrated Circuits

1 MC6809 microprocessor (do not use MC6809E)
1 MC6821 peripheral interface adapter
8 MCM6665A or 4164 64K dynamic RAM 200 ns
2 74LS157 quad 2-line to 1-line data selector
1 74LS00 quad NAND gate
1 74LS04 hex inverter
1 74LS20 dual 4-input NAND gate
1 74LS367 hex bus driver
1 74LS74 dual D-type flip-flop
1 2716 EPROM, 5-volt type. Programmed version available at the Bit Stop, 5958 Shenandoah Rd., Mobile, AL 36608 ($30 ppd)
1 7805 5-volt regulator TO-220 case

Sockets and Plugs

2 40-pin wire-wrap sockets
1 24-pin wire-wrap sockets
11 16-pin wire-wrap sockets
4 14-pin wire-wrap sockets
2 20-pin wire-wrap header, JDR Microdevices IDH20W or equivalent
2 20-contact ribbon header sockets, JDR IDS20 or equivalent, plus 1–2 feet 20-conductor ribbon cable

Miscellaneous

1 Vector 3677-2 4.5 × 6.5-inch board or Vector 3677 4.5 × 9.0-inch board
1 4.0 MHz crystal or TV color-burst crystal
1 Heat sink for TO-220 regulator
1 1-amp 50 v. bridge rectifier
1 Power transformer, primary 110 v. secondary 12 v. at 500 ma
1 Reset pushbutton, normally open
1 Pause switch, SPST
1 2200 mfd 15 v. electrolytic cap.
1 33 mfd 15 v. trantalum cap.
2 27 pf. disc cap.
8 .05 mfd disc cap.
3 2 mfd 15 v. trantalum cap.
1 1N4001 diode
8 33 ohm ¼ w. resistor
1 56K ½ w. resistor
1 6 × 8 × 2-inch aluminum enclosure or larger (Radio Shack)

Most of the required parts are available from these alternate sources:

JDR Microdevices, 1224 S. Bascom Ave., San Jose, CA 95128
ordering line 1-800-538-5000; in CA 1-800-662-6279
Priority One Electronics, 9161 Deering Ave., Chatsworth, CA 91311
ordering line 1-800-423-5922
DoKay Computer Products, 2100 De La Cruz Blvd., Santa Clara, CA 95050
ordering line 1-800-538-8800; in CA 1-800-848-8008

to place the large power-supply capacitor on the board. The only off-board components are the power supply transformer, the rectifier, and the control switches.

Possible Design Modification

You can alter the design for use with a serial printer by replacing the PIA with a 6850 ACIA and adding a suitable baud-rate generator. The ACIA has some built-in handshaking capabilities that can control the interfacing of the host computer and printer.

As to memory refresh, you can divorce the refresh from the mainline routine by interrupting the microprocessor at 500 Hz or faster (2 ms). The interrupt routine would read your 128 consecutive addresses and then either return or do any other short function required before returning from interrupt.

By further decoding, you could increase the buffer size from 60K to about 63.5K, using only 0.5K for the ROM space and only a few bytes for the PIA. Note that we do not use the upper 4K of RAM in our design. Also note that all of the low RAM space is available for character buffering because the software uses internal microprocessor registers exclusively as pointers and counters.

Because the 60K spooler is a "smart" device with its own microprocessor, you could use routines to add a variety of control functions, such as automatic form feed. If your software gets complex, you may wish to implement a stack in RAM and call a subroutine for refresh, e.g.:

```
LOOP BSR REFRESH
     (test a flag)
BPL LOOP
     (continue)
```

Note that the loop contains a call to the refresh routine. Each loop where a significant delay is encountered should contain such a call.

DDJ

Print Spooler (Listing Continued, text begins on page 46)

Listing One

An assembled listing of the ROM contents. The 2K EPROM occupies locations $F8000 to $FFFF.

```
                        1000 * SPOOLER64 - DUAL
                        1010         .OR $F800
                        1020         .TA $0800
                        1030 *       .TF SPOOLER64.OBJ5
F3FF-                   1040 STACK   .EQ $F3FF
F400-                   1050 PORTA   .EQ $F400
F400-                   1060 DDRA    .EQ $F400
F401-                   1070 CRA     .EQ $F401
F402-                   1080 PORTB   .EQ $F402
F402-                   1090 DDRB    .EQ $F402
F403-                   1100 CRB     .EQ $F403
0000-                   1110 SOM     .EQ $0000
F000-                   1120 EOM     .EQ $F000
F000-                   1130 MAX     .EQ EOM-SOM
                        1140 *-------------------------------
F800-                   1150         .BS $FB00-*  SKIP $0300 BYTES
                        1160 *-------------------------------
FB00- 7F F4 01          1170 START   CLR CRA      ACCESS DDR'S
FB03- 7F F4 03          1180         CLR CRB
FB06- 86 00             1190         LDA #$00     A - INPUT
FB08- B7 F4 00          1200         STA DDRA
FB0B- 86 FF             1210         LDA #$FF     B - OUTPUT
FB0D- B7 F4 02          1220         STA DDRB
FB10- 86 2F             1230         LDA #$2F     00101111
FB12- B7 F4 01          1240         STA CRA
FB15- 86 2F             1250         LDA #$2F     00101111
FB17- B7 F4 03          1260         STA CRB
FB1A- 86 00             1270         LDA #$00
FB1C- B7 F4 02          1280         STA PORTB    TOSS NULL
FB1F- B6 F4 00          1290         LDA PORTA    STROBE CA2
FB22- 8E 00 00          1300         LDX #SOM     X = INPTR
FB25- 10 8E 00
FB28- 00                1310         LDV #SOM     V = OUTPTR
FB29- CE 00 00          1320         LDU #0       U = CHRCNT
FB2C- 10 CE F3
FB2F- FF                1330         LDS #STACK
                        1340 *-------------------------------
                        1350 * REFRSH IS HERE. CONSISTS OF
                        1360 * 64 ADDA #0 CODES FOR REFRESH.
                        1370 * INSTRUCTION IS 8B 00

                        1480 *-------------------------------
FBB0- 11 83 F0
FBB3- 00                1490 CHKIN   CMPU #MAX    CHECK COUNT
FBB4- 27 14             1500         BEQ PAUSE    SKIP IF MAX.
FBB6- F6 F4 01          1510         LDB CRA      GET INPUT FLAG
FBB9- 2A 0F             1520         BPL PAUSE    SKIP IF CLEAR
FBBB- B6 F4 00          1530         LDA PORTA    GET INPUT DATA
FBBE- A7 80             1540         STA ,X+      STORE & INCR X
FBC0- 33 41             1550         LEAU 1,U     INCR. COUNT
FBC2- 8C F0 00          1560         CMPX #EOM    END OF MEM?
FBC5- 26 03             1570         BNE PAUSE    BRANCH IF NOT
FBC7- 8E 00 00          1580         LDX #SOM     WRAPAROUND
FBCA- 21 03             1590 PAUSE   BRN CHKOUT   NEVER BRANCH
FBCC- 16 FF 61          1600         LBRA REFRSH
FBCF- 11 83 00
FBD2- 00                1610 CHKOUT  CMPU #0      COUNT ZERO?
FBD3- 27 19             1620         BEQ GOREFR   SKIP IF 0
FBD5- F6 F4 03          1630         LDB CRB      GET PRTR READY
FBD8- 2A 14             1640         BPL GOREFR   BRANCH IF BUSY
FBDA- A6 A0             1650         LDA ,V+      GET DATA. INCR V
FBDC- B7 F4 02          1660         STA PORTB    PRINT IT
FBDF- B6 F4 02          1670         LDA PORTB    CLEAR IRQ FLAG
FBE2- 33 5F             1680         LEAU -1,U    DECR COUNT
FBE4- 10 8C F0
FBE7- 00                1690         CMPV #EOM    AT MEMORY END?
FBE8- 26 04             1700         BNE GOREFR   BRANCH IF NOT
FBEA- 10 8E 00
FBED- 00                1710         LDV #SOM     WRAPAROUND
FBEE- 16 FF 3F          1720 GOREFR  LBRA REFRSH  GO TO REFRESH
                        1730 *-------------------------------
FBF1-                   1740         .BS $FBFE-*  SPACE TO VECTORS
FBFE- FB 00             1750 RESET   .DA START    NEED ONLY RESET
                        1760 *-------------------------------
                        1770 *-------------------------------
FC00-                   1780         .BS $FF00-*  SKIP $300 BYTES
                        1790 *-------------------------------
FF00- 7F F4 01          1800 STARTX  CLR CRA      ACCESS DDR'S
FF03- 7F F4 03          1810         CLR CRB
FF06- 86 00             1820         LDA #$00     A - INPUT
FF08- B7 F4 00          1830         STA DDRA
FF0B- 86 FF             1840         LDA #$FF     B - OUTPUT
FF0D- B7 F4 02          1850         STA DDRB
FF10- 86 2F             1860         LDA #$2F     00101111
FF12- B7 F4 01          1870         STA CRA
FF15- 86 2F             1880         LDA #$2F     00101111
FF17- B7 F4 03          1890         STA CRB
FF1A- 86 00             1900         LDA #$00
FF1C- B7 F4 02          1910         STA PORTB    TOSS NULL
FF1F- B6 F4 00          1920         LDA PORTA    STROBE CA2
```

```
FF22- 8E 00 00 1930         LDX #SOM      X = INPTR
FF25- 10 8E 00
FF28- 00      1940          LDV #SOM      V = OUTPTR
FF29- CE 00 00 1950         LDU #0        U = CHRCNT
FF2C- 10 CE F3
FF2F- FF      1960          LDS #STACK
              1970 *--------------------------------
              1980 * REFRSH IS HERE. CONSISTS OF
              1990 * 64 ADDA #0 CODES FOR REFRESH.
              2000 * INSTRUCTION IS 8B 00

              2110 *--------------------------------
FFB0- 11 83 F0
FFB3- 00      2120 CHKINX CMPU #MAX     CHECK COUNT
FFB4- 27 14   2130          BEQ PAUSEX    SKIP IF MAX.
FFB6- F6 F4 01 2140         LDB CRA       GET INPUT FLAG
FFB9- 2A 0F   2150          BPL PAUSEX    SKIP IF CLEAR
FFBB- B6 F4 00 2160         LDA PORTA     GET INPUT DATA
FFBE- A7 80   2170          STA ,X+       STORE & INCR X
FFC0- 33 41   2180          LEAU 1,U      INCR. COUNT
FFC2- 8C F0 00 2190         CMPX #EOM     END OF MEM?
FFC5- 26 03   2200          BNE PAUSEX    BRANCH IF NOT
FFC7- 8E 00 00 2210         LDX #SOM      WRAPAROUND
FFCA- 20 03   2220 PAUSEX BRA CHKOUX
FFCC- 16 FF 61 2230         LBRA REFRSX
FFCF- 11 83 00
FFD2- 00      2240 CHKOUX CMPU #0       COUNT ZERO?
FFD3- 27 19   2250          BEQ GOREFX    SKIP IF 0
FFD5- F6 F4 03 2260         LDB CRB       GET PRTR READY
FFD8- 2A 14   2270          BPL GOREFX    BRANCH IF BUSY
FFDA- A6 A0   2280          LDA ,Y+       GET DATA. INCR Y
FFDC- B7 F4 02 2290         STA PORTB     PRINT IT
FFDF- B6 F4 02 2300         LDA PORTB     CLEAR IRQ FLAG
FFE2- 33 5F   2310          LEAU -1,U     DECR COUNT
FFE4- 10 8C F0
FFE7- 00      2320          CMPY #EOM     AT MEMORY END?
FFE8- 26 04   2330          BNE GOREFX    BRANCH IF NOT
FFEA- 10 8E 00
FFED- 00      2340          LDY #SOM      WRAPAROUND
FFEE- 16 FF 3F 2350 GOREFX LBRA REFRSX   GO TO REFRESH
              2360 *--------------------------------
FFF1-         2370          .BS $FFFE-*   SPACE TO VECTORS
FFFE- FF 00   2380 RESETX .DA STARTX    NEED ONLY RESET
```

End Listing One

Listing Two

A hexadecimal dump of the EPROM. Base address for this dump is $0800 (corresponds to the first byte of the EPROM).

```
)$0B00.0BFF

0B00- 7F F4 01 7F F4 03 86 00
0B08- B7 F4 00 86 FF B7 F4 02
0B10- 86 2F B7 F4 01 86 2F B7
0B18- F4 03 86 00 B7 F4 02 B6
0B20- F4 00 8E 00 00 10 8E 00
0B28- 00 CE 00 00 10 CE F3 FF
0B30- 8B 00 8B 00 8B 00 8B 00
0B38- 8B 00 8B 00 8B 00 8B 00
0B40- 8B 00 8B 00 8B 00 8B 00
0B48- 8B 00 8B 00 8B 00 8B 00
0B50- 8B 00 8B 00 8B 00 8B 00
0B58- 8B 00 8B 00 8B 00 8B 00
0B60- 8B 00 8B 00 8B 00 8B 00
0B68- 8B 00 8B 00 8B 00 8B 00
0B70- 8B 00 8B 00 8B 00 8B 00
0B78- 8B 00 8B 00 8B 00 8B 00
0B80- 8B 00 8B 00 8B 00 8B 00
0B88- 8B 00 8B 00 8B 00 8B 00
0B90- 8B 00 8B 00 8B 00 8B 00
0B98- 8B 00 8B 00 8B 00 8B 00
0BA0- 8B 00 8B 00 8B 00 8B 00
0BA8- 8B 00 8B 00 8B 00 8B 00
0BB0- 11 83 F0 00 27 14 F6 F4
0BB8- 01 2A 0F B6 F4 00 A7 80
0BC0- 33 41 8C F0 00 26 03 8E
0BC8- 00 00 21 03 16 FF 61 11
0BD0- 83 00 00 27 19 F6 F4 03
0BD8- 2A 14 A6 A0 B7 F4 02 B6
0BE0- F4 02 33 5F 10 8C F0 00
0BE8- 26 04 10 8E 00 00 16 FF
0BF0- 3F 00 00 00 00 00 00 00
0BF8- 00 00 00 00 00 00 FB 00
```

```
)$0F00.0FFF

0F00- 7F F4 01 7F F4 03 86 00
0F08- B7 F4 00 86 FF B7 F4 02
0F10- 86 2F B7 F4 01 86 2F B7
0F18- F4 03 86 00 B7 F4 02 B6
0F20- F4 00 8E 00 00 10 8E 00
0F28- 00 CE 00 00 10 CE F3 FF
0F30- 8B 00 8B 00 8B 00 8B 00
0F38- 8B 00 8B 00 8B 00 8B 00
0F40- 8B 00 8B 00 8B 00 8B 00
0F48- 8B 00 8B 00 8B 00 8B 00
0F50- 8B 00 8B 00 8B 00 8B 00
0F58- 8B 00 8B 00 8B 00 8B 00
0F60- 8B 00 8B 00 8B 00 8B 00
0F68- 8B 00 8B 00 8B 00 8B 00
0F70- 8B 00 8B 00 8B 00 8B 00
0F78- 8B 00 8B 00 8B 00 8B 00
0F80- 8B 00 8B 00 8B 00 8B 00
0F88- 8B 00 8B 00 8B 00 8B 00
0F90- 8B 00 8B 00 8B 00 8B 00
0F98- 8B 00 8B 00 8B 00 8B 00
0FA0- 8B 00 8B 00 8B 00 8B 00
0FA8- 8B 00 8B 00 8B 00 8B 00
0FB0- 11 83 F0 00 27 14 F6 F4
0FB8- 01 2A 0F B6 F4 00 A7 80
0FC0- 33 41 8C F0 00 26 03 8E
0FC8- 00 00 20 03 16 FF 61 11
0FD0- 83 00 00 27 19 F6 F4 03
0FD8- 2A 14 A6 A0 B7 F4 02 B6
0FE0- F4 02 33 5F 10 8C F0 00
0FE8- 26 04 10 8E 00 00 16 FF
0FF0- 3F 00 00 00 00 00 00 00
0FF8- 00 00 00 00 00 00 FF 00
```

End Listing

Designing a Real-Time Clock for the S-100 Bus

by Alan D. Wilcox

The author, with time on his hands and mind, builds a hardware clock for his CompuPro 816/Z.

Adapted by permission of the publisher, Prentice-Hall, Inc., from the forthcoming book, Designing and Troubleshooting a 68000 Microcomputer System, *by Alan D. Wilcox, available 1986. No part of this adaptation may be reproduced, in any form or by any means, without written permission from the publisher.*

After revising my latest program and printing it out, I set it aside to look over later. Naturally, it was in a stack of earlier versions of the program that didn't quite work right and looked almost the same except for a few hard-to-find bugs. You know the rest of the story: when I came back in a few weeks, I couldn't tell which was the latest and therefore correct copy of my program. If I had jotted down the time or at least the date on my latest copy, I could have identified it then. So I decided it was time to design a clock circuit to do the writing for me automatically.

The result is a clock circuit using the popular National MM58167A real-time clock IC. The circuit requires no changes in the operating system and has battery backup so you need not reset it after turning off the computer. Designed to meet S-100/IEEE-696, it performs easily with virtually any S-100 microprocessor system. Two assembly language programs set and print the time and date; a third program prints the time, date, and filename as a title header then prints out the file in 60-line pages.

Background

There are two basic ways to put a clock in a computer: either modify the system software to keep track of the time or provide a piece of hardware that will keep time independently of the computer. The software approach is easier in that you need not tamper with the computer hardware, but it has several drawbacks. First, when power is off, the clock is off; this requires resetting the clock every time you turn the computer on. Second, you must modify the *system* software to support a time-keeping program; procedures for this modification can be almost incomprehensible to the novice. Finally, the subsequent time-keeping isn't accurate anyway; for example, the clock must be off during disk read or write operations. The hardware approach avoids these problems.

The concept of a hardware clock is simply to have a clock available for the computer to read when it wants the time and date. The clock has its own battery power supply to keep it running when the computer is turned off. The only programming required is for setting the clock and reading it; these straightforward programs do not require an intimate understanding of the system software. Finally, accuracy is crystal controlled and is unaffected by other computer operations.

A number of articles have discussed how to implement a clock in a personal computer. In one design, Calaway and Hill presented what I call a software approach to

---

Alan D. Wilcox, 60 South Eighth St., Lewisburg, PA 17837.

time-keeping.[1] Their basic idea is to provide a small amount of hardware to generate an interrupt each second; their system responds to this interrupt by updating the total number of pulses counted and thereby updating the clock time. The operating system needs modification, however, which can range from straightforward to profoundly complex.

Hassebrock described a simple approach to keeping the time and date using the Hayes Chronograph.[2] The Hayes unit is a piece of hardware connected to the computer through a serial data port. When you want the time or date, you run a program that tells the clock to send the information to the computer for printing or display. Because the Chronograph is connected to a serial port, you must configure the port to match the required data rate, and so on. Therefore, you need a modem-type program to access the clock; the program may be either part of the time-setting/reading program or part of the operating system. Hassebrock chose the latter and modified his system somewhat. Overall, the concept of an external serial-data clock is appealing except that a spare serial port may not be available.

A hardware approach described by Ciarcia seemed initially to offer some distinct advantages.[3] First, the design uses hardware to keep time: no difficulty keeping time when the power is off, no system software modifications, and no accuracy problems. Second, the clock does not use a serial port, making port initialization unnecessary. The intended application was for use with the Z8-BASIC microcomputer as an intelligent clock. Although the approach has merit, as it stands, it is not a generally useful design.

Design Overview

The design requirement was for a real-time clock to operate in my CompuPro System 816/Z, a Z80B S-100 machine running at 6 MHz. The clock should keep both the time and date, as well as provide a means of interrupting the system either on a regular basis or at a predetermined time. It should maintain an accuracy within 10 sec/month, with or without system power. Except for initially setting the time and date, the operator should not have to interact with the clock in any way.

The clock should meet S-100/IEEE-696 and operate as a slave without affecting normal system operation; likewise, it should require no changes in the system software. Data transfer between clock and system should use I/O-mapped ports. The handshaking should use the S-100 RDY line, and because the processor operates at 6 MHz, some wait states should be available. Clock interrupts to the system should be switchable to any of the S-100 vectored interrupt lines.

Although all initial programs should poll the clock for the time, the clock should be able to use any of the vectored interrupt lines to implement a future real-time multitasking executive; in the long term, this could involve multiprocessing with a new 10 MHz 68000 CPU board in the system. For the near term, however, the software should at least set and display the time, as well as print files with a time/date header.

Hardware Design

The National MM58167A real-time clock IC[4] provided a solution to the design requirements. As shown in the block diagram (Figure 1, page 58), you can partition hardware using this clock IC into several major modules that include the clock, address decoder, data-bus interface, interrupt switches, and power supply. Once you divide the hardware into modules, the design can proceed in much the same way as in software development: top down, bottom up, or most critical first.

The appropriate design method in this case is to consider the most critical section first. Everything depends on the clock IC, and its requirements come before all else. The block diagram of the MM58167A (Figure 2, page 59) shows that the IC uses five address lines (32 different addresses) and a chip select and has a single bidirectional data bus with separate read/write controls, two interrupt outputs, and a power-down control. Designing to accommodate each of these requirements individually results in the implementation shown in the circuit schematic (Figure 3, page 60).

I decided to use I/O mapping rather than memory mapping for the clock data transfer. With a maximum of 256 ports, only the lower eight bits of the address bus need decoding, and five of these are decoded internally by the clock IC. For maximum flexibility, I chose to use DIP switches to select the three most significant bits of the I/O address. The output of the decoder is used as a chip select for the clock and by the read/write qualifier circuit.

The S-100 data bus in and out lines are buffered with a pair of 74LS244s connected directly to the clock IC. The buffer for data-in (DI from the processor's viewpoint) is strobed using the I/O read qualifiers pDBIN and sINP and the chip select CS. The data-out buffer is strobed using pWR*, sOUT, and the chip select CS. During a typical read bus cycle, for example, the system places the proper address on the address bus, asserts sINP, then asserts pDBIN to strobe the DI buffer and the clock RD* input. In a typical write bus cycle, the system puts the address on the bus, asserts sOUT then asserts pWR* to strobe the DO buffer and the clock WR* input.

Timing

A basic read or write bus cycle has three bus states (BS1, BS2, and BS3), each of which takes 167 ns in a 6 MHz system; consequently, a bus read or write normally completes within 500 ns. However, for I/O devices that require substantial access times, you can extend the bus cycle by adding a number of wait states. To determine whether wait states are required during any read or write bus cycle, the bus master samples the RDY line at the rising edge of the system clock in BS2. If the RDY line is low, then a wait state (BSw) is inserted immediately after BS2; if the RDY line is still low one bus state later, yet another BSw is inserted. The addition of wait states continues until RDY finally goes high; at that point, the bus cycle concludes with BS3.

The MM58167A clock requires the addition of wait states because of its slow access time. For a clock read

operation, the maximum specified time from valid address until valid data is 1050 ns, far longer than a normal read bus cycle with no wait states. The read bus cycle on my 6 MHz CompuPro system is shown in Figure 4 (page 62) with approximate times to scale. Asserting pDBIN at the beginning of BS2 leaves no time for the clock to bring the RDY line low. However, all Z80 microprocessors automatically insert a single wait state for I/O instructions (by carrying out a microcoded instruction in the Z80), so the clock really has until the beginning of the first wait state BSw (auto) to get RDY low.

The timing question becomes: "Can clock RD* be pulled low and can the clock respond with RDY low—all within 167 ns?" The specs for the clock indicate that read-strobe to ready-strobe (clock pin 4) is 150 ns maximum, but the 7406 easily could add another 40 ns in propagation delay to the system RDY line. Add these up. In the worst case, the clock board cannot get the RDY line down in time even by taking advantage of the Z80 automatic wait state. As indicated in Figure 4, the clock actually takes from about 100 ns (middle of BS2) to about 190 ns (middle of BSw). Actual timing measurements and read operations indicate that the clock sometimes is fast enough but usually is not. For reliable operation of RDY, we must add an extra wait state.

In my system, I used the Z80 processor board's option switch to add a single wait state to I/O operations in addition to the Z80 automatic wait. This required no additional hardware, although you could enhance a more general clock design by including an on-board wait state generator circuit. Libes and Garetz present a number of suitable circuits in their book.[6]

When the clock data is finally valid, the system allows the RDY line to go high, and the processor reads the data in BS3. To finish BS3, pDBIN is negated, which raises the clock RD* input, and the address is deselected.

The write bus cycle timing requirements for the clock are similar to the read timing requirements. As in the read, you must include a single wait state to allow sufficient time for RDY to be pulled low during the write operation.

Power-Down Design

The power-down design is critical in the actual operation of the clock. If POWER DOWN* is not asserted to a low-

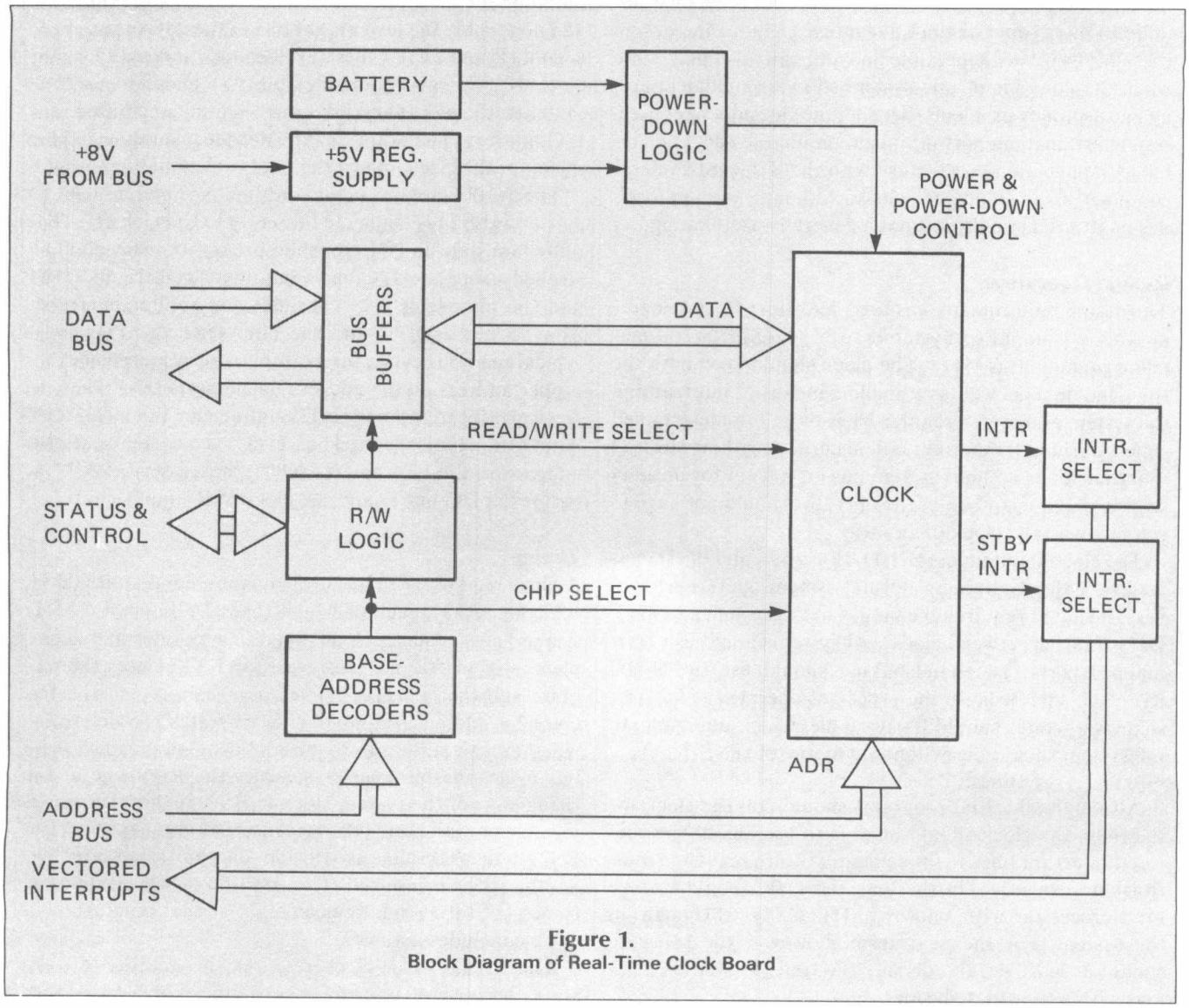


Figure 1.
Block Diagram of Real-Time Clock Board

voltage level at least one microsecond before system power removal, then the contents of the clock memory might be anything (or nothing) when you restore normal system power. Furthermore, when the power does come back on, POWER DOWN* must be held low until all bus signals are valid.

A common way of accomplishing this is to use a zener diode that sets a reference threshold voltage to control a transistor switch. Consider the circuit (Figure 3) when the system power is off: Q1 and Q2 are both off. When the system power comes on, nothing happens until the system bus voltage reaches about 6.6 V. Then the zener conducts, turning on Q2, which then turns on Q1. When Q1 goes on, voltage is applied to POWER DOWN*, which activates the clock chip for normal operation. Because a system bus voltage greater than 6.6 V is enough to allow near-normal output from all 5 V regulators, all bus signals should be valid by this time. When the system power goes off, the system voltage drops gradually enough so that, when it passes below 6.6 V, the zener and Q1/Q2 can drop the POWER DOWN* to a low voltage before the bus signals are no longer valid.

A small, 3.75 V, 20 mAh, NiCad rechargeable battery maintains clock operation when system power is off. Specified current consumption of the clock is on the order of 10 to 20 uA during power-down; my clock measured 14 uA while using the battery with a series-blocking diode. The 470 ohm resistor provides a trickle charge from ½ mA to about 2 mA, depending on the state of the battery and component tolerances. For an average use of several hours a day, this resistor keeps the battery charged between 3.8 and 4.15 V with no difficulties.

Logic Circuits

IEEE Std-696 (sec. 3.7) states that a card may not source more than 0.5 mA at 0.5 V nor sink more than 80 uA at 2.4 V on most system signal lines.[5] The effect of this is that you can connect only one LS TTL gate per card to each line of the address bus. My decoder uses the 74LS136 exclusive-or, which sources a worst-case maximum of 0.8 mA at 0.4 V. I expect this to cause no operational problems in any practical sense. I opted to use the 74LS136, rather than buffer the address bus with extra gates, to be strictly within the specification. I prefer to be conscious of trade-offs than to not be aware of them and have a problem occur later.

The pull-up resistors for the open-collector 7406 are selected to maintain proper logic levels and currents. The design trade-off is to let the various resistors be high for lower current consumption or to let them be low for more speed. I designed for as much speed as possible. Typical calculations are discussed in the standard *TTL Data Book* (page 6-6).[7]

Near the end of the project, I decided to implement the clock standby interrupt, which required an extra non-inverting gate. I used the spare 74LS10 and the last spare

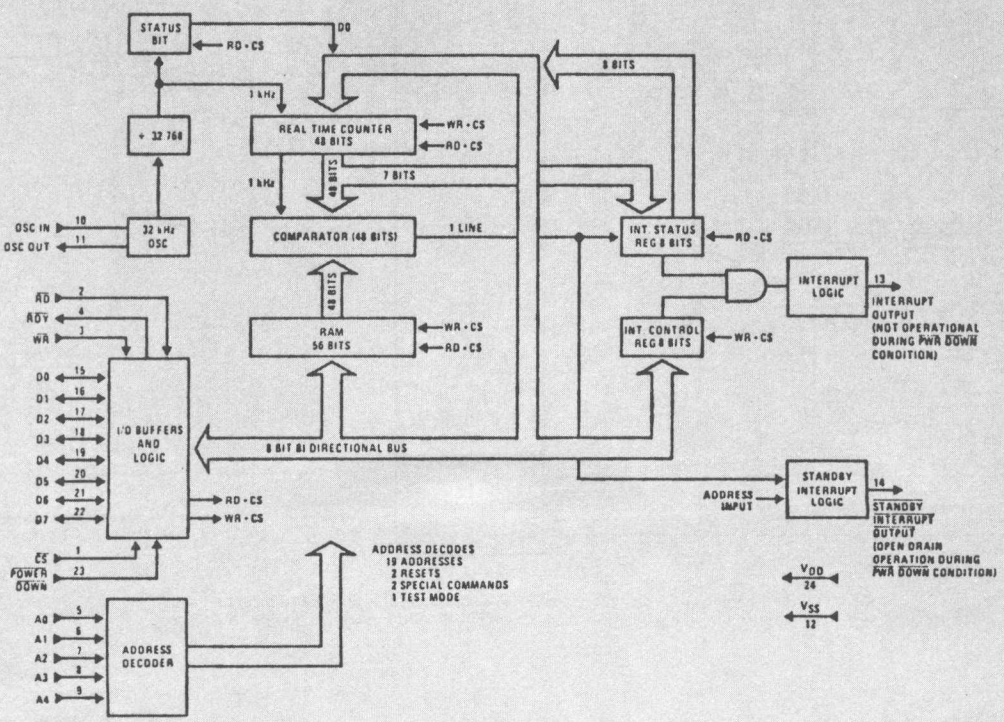
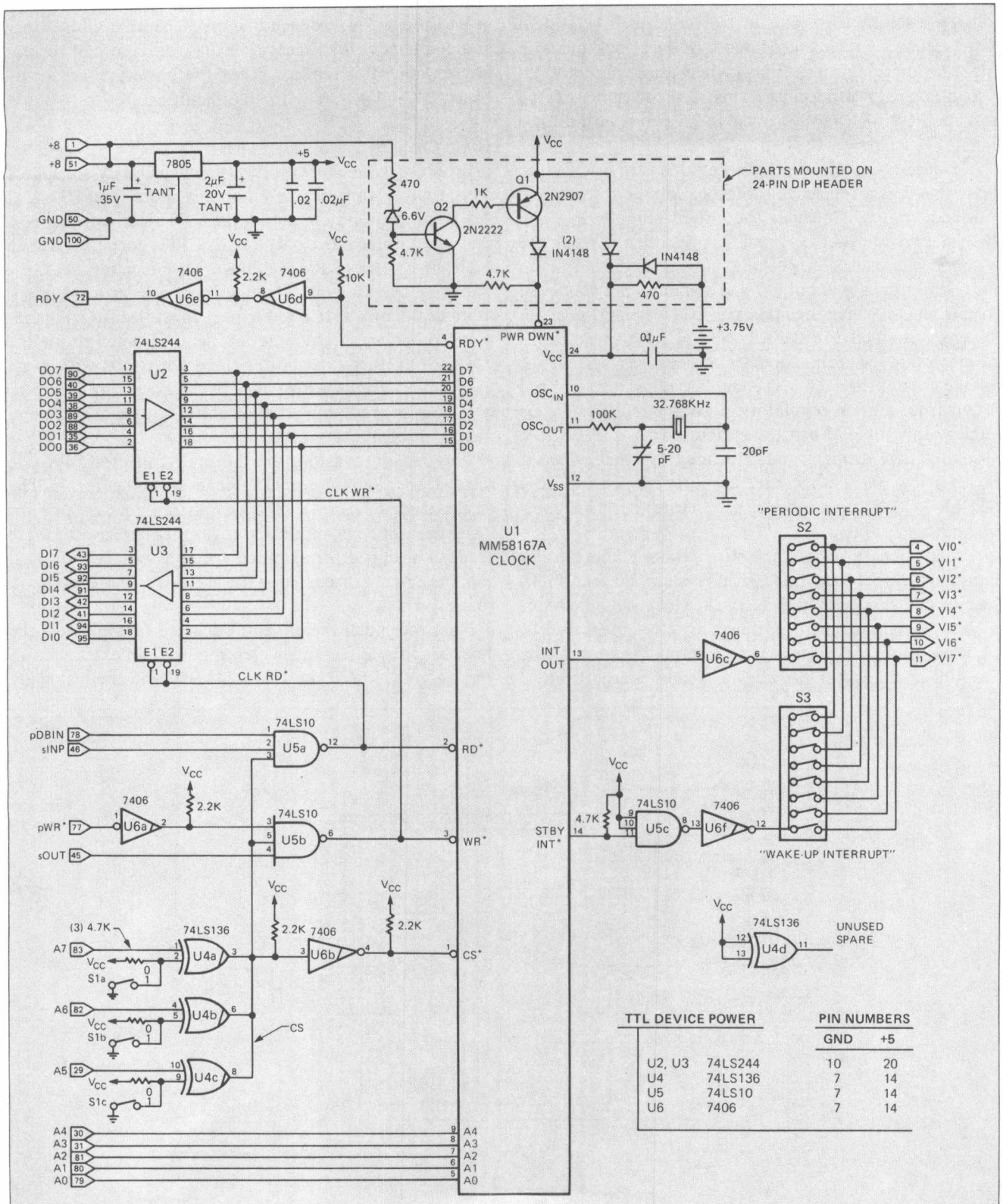


Figure 2.
Block Diagram of the MM58167A (material courtesy of National Semiconductor Corporation)



TTL DEVICE POWER		PIN NUMBERS	
		GND	+5
U2, U3	74LS244	10	20
U4	74LS136	7	14
U5	74LS10	7	14
U6	7406	7	14

*NOTE: Leave interrupt switches S2 and S3 all open unless interrupts are to be implemented. If interrupts are used, close only 1 switch in S2 and 1 switch in S3 at most.

Figure 3.
Schematic Wiring Diagram of S-100 Real-Time Clock

7406 gate to get this output from the clock board. This particular output does not have the flexibility of the programmable output, but you can use it to indicate a match between real time and a preset time for a wake-up alarm.

Construction and Installation

I prototyped the entire real-time clock on a Vector 8800V wire-wrap board, as shown in the photo (page 64). I added the interrupt switches (S3) for the wake-up interrupt after taking the photo; they are located beside the main interrupt switches (S2). All the circuits took less than a third of a standard S-100 board. I wired the transistor power-down circuit on the 24-pin header at the lower-left edge of the board by the battery. I used plastic "wrap ID" panels on the back of the board for each IC socket to make component and pin identification easier.

My system does not have any I/O port assignments in the range A0h to BFh, so I selected A0 as the base address for the clock. For this base address, S1 sections a and c are closed for a "1" and section b is left open for a "0." As shown in Figure 5 (page 63), the clock uses a total of 24 addresses rather than the whole block of 32 allocated to it. This permits additional design flexibility if port addresses are at a premium.

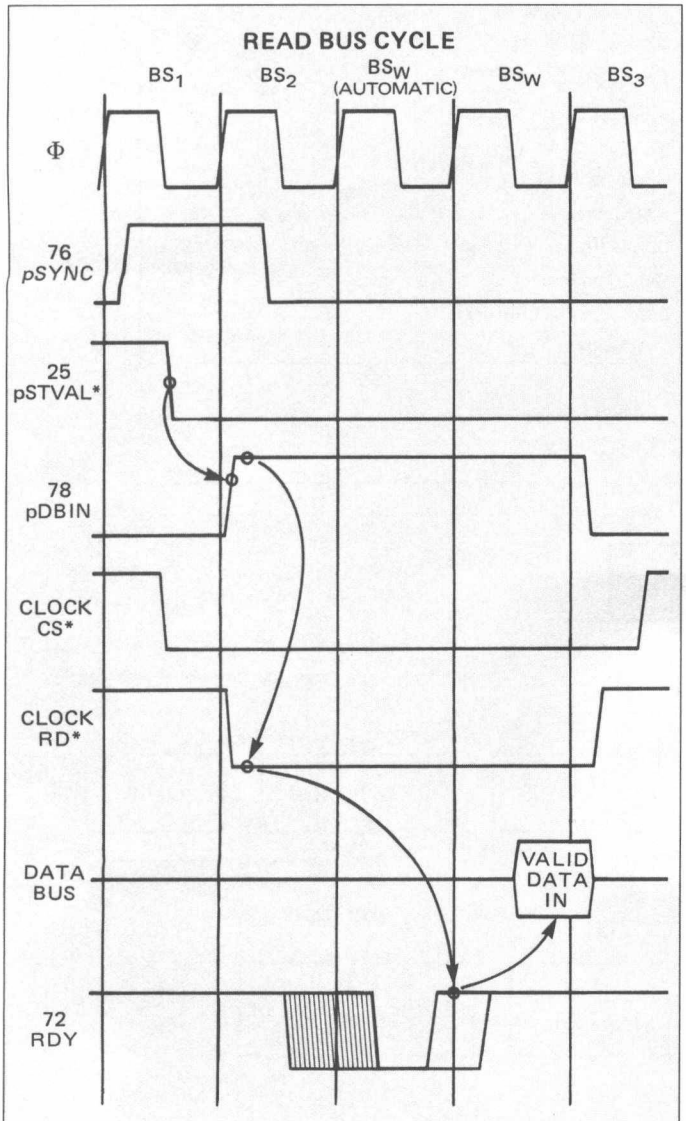


Figure 4.
Read Bus Cycle

You may install the board in any convenient location on the bus. Because of its wire-wrap construction, however, the board takes up the space of two normal boards and cannot be crowded. Depending on your needs, you can snip down the wire-wrap pins considerably (watch your eyes) to make the board thinner.

Software

When writing the programs to set the clock and print the time, I concentrated on writing clear, structured, modular code. Thus, you should have little difficulty making changes to transport the clock software to another system or to make modifications. Naturally, the resulting code perhaps is not as brief as might have been possible were I writing only for myself.

I have presented several programs here for using the clock. The first is CLOCKSET (see Listing One, page 68), a program to set either the day and date, the time, or both. The second is TIME (Listing Two, page 74), a program to display the time and date on the console. This same program will print the time and date plus a short text (such as a filename) to LST: if you include the text with the program invocation. This second feature of TIME is handy when you are using a text editor and need a date on a page about to be printed. The last is TIMELIST (see Listing Three, page 79), a program to print a time/date header with filename then list a text file with 60 lines per page.

Recall that I chose to use A0h as a clock base address for I/O. In all the programs, I used CLOCK as the base address and set its value equal to A0h. I identify other clock port addresses by a simple reference to CLOCK+1, CLOCK+2, and so on. For example, from Figure 5, CLOCK+3 is the address for "minutes" and CLOCK+7 is the address for "months." Using CLOCK in this way allows you to make an easy change to any base address just by changing one statement at the beginning of the program.

The first program, CLOCKSET, asks whether or not to set the date. If you select the default "No," then it asks whether to set the time. If you choose a second default "No," the program terminates without changing the clock. If, however, you do select the date-set option, then it asks for two digits for the day of the week: enter 01 for Sunday, 02 for Monday, and so on. You enter the day of the month the same way: 01 through 31. You enter the month itself as month 01 through 12. Select the year by entering just the last two digits; that is, 85 for 1985. Each January, you must change the year manually; the clock IC does not automatically change to a new year. When writing this program, I opted to keep it simple: no error checks and nothing fancy like entering literal names instead of numbers. If I make an error, I just invoke the program over again and do it correctly the next time through.

The second part of CLOCKSET is similar to the date-setting procedure. You enter the time in two-digit groups for the hours and minutes, and a carriage return sets the

Functional Description

TABLE II. ADDRESS CODES AND FUNCTIONS

A4	A3	A2	A1	A0	Function
0	0	0	0	0	Counter—Ten Thousandths of Seconds
0	0	0	0	1	Counter—Hundredths and Tenths of Seconds
0	0	0	1	0	Counter—Seconds
0	0	0	1	1	Counter—Minutes
0	0	1	0	0	Counter—Hours
0	0	1	0	1	Counter—Day of Week
0	0	1	1	0	Counter—Day of Month
0	0	1	1	1	Counter—Month
0	1	0	0	0	RAM—Ten Thousandths of Seconds
0	1	0	0	1	RAM—Hundredths and Tenths of Seconds
0	1	0	1	0	RAM—Seconds
0	1	0	1	1	RAM—Minutes
0	1	1	0	0	RAM—Hours
0	1	1	0	1	RAM—Day of Week
0	1	1	1	0	RAM—Day of Month
0	1	1	1	1	RAM—Months
1	0	0	0	0	Interrupt Status Register
1	0	0	0	1	Interrupt Control Register
1	0	0	1	0	Counters Reset
1	0	0	1	1	RAM Reset
1	0	1	0	0	Status Bit
1	0	1	0	1	GO Command
1	0	1	1	0	$\overline{\text{STANDBY}}$ $\overline{\text{INTERRUPT}}$
1	1	1	1	1	Test Mode

Figure 5.
Clock Address Codes (material courtesy of National Semiconductor Corporation)

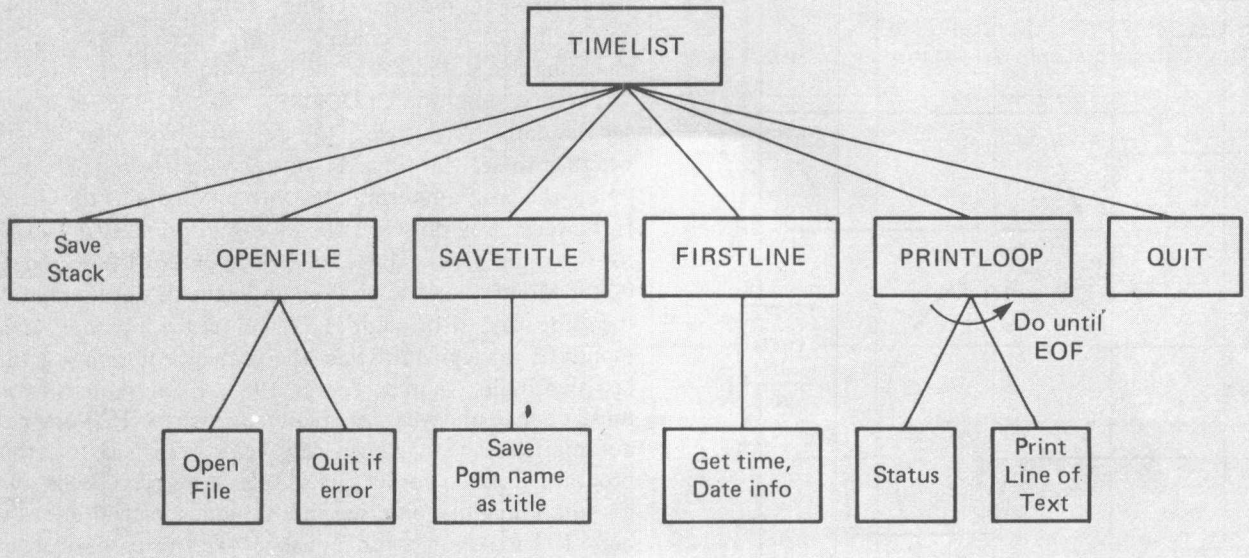


Figure 6
Structure Chart of the TIMELIST Program

seconds to zero. The clock chip keeps its time based on a 24-hour count, so if the time is 7:15 pm, for example, then the entry is 19 hours, 15 minutes. The carriage return to set seconds to zero does a write-data to CLOCK+21 (the GO command, see Figure 5). This write pulse resets the thousandths, hundredths, tenths, units, and tens of seconds counters. In the SETTIM subroutine, note that I reset the seconds (an early GO) before setting the minutes; this is because the clock chip will increment the minute counter upon GO if the seconds counter is greater than 40. For example, if the time is almost 7:15 pm and I enter 19 and 15 followed by <cr> to zero seconds, I might find the time erroneously set to 19:16; hence, the need to reset GO.

In the program TIME, unlike the simpler CLOCKSET program, I wanted more than just bare numbers from the clock: I wanted the literal names for days and months. The subroutine NTOLIT makes the conversion from the clock's packed BCD to a literal string. It does this by converting the number (say, 11 for November) to binary then indexing through the list of literals until it reaches the eleventh one. Then it moves the word "November" in memory to the space reserved for the message to be printed.

You can use TIME in two ways: either as a simple display to the console or as a display plus an output to the printer. To get the display, type TIME. To get the printer output in addition to the console display, type TIME <text>. The additional letters will send the time and date information plus the <text> to the currently defined LST: device connected to the computer. This is one way of time/date-stamping a page before sending some special output to the printer. Because the print output is aligned toward the right-hand side of the page, you have space for only about 15 characters before the printer wraps around. I use the space to put in a filename or some other text identification. Another way I use this feature is to print a disk name as <text> then send a disk directory to the printer; I keep this directory listing with the disk so I have a dated record of its contents.

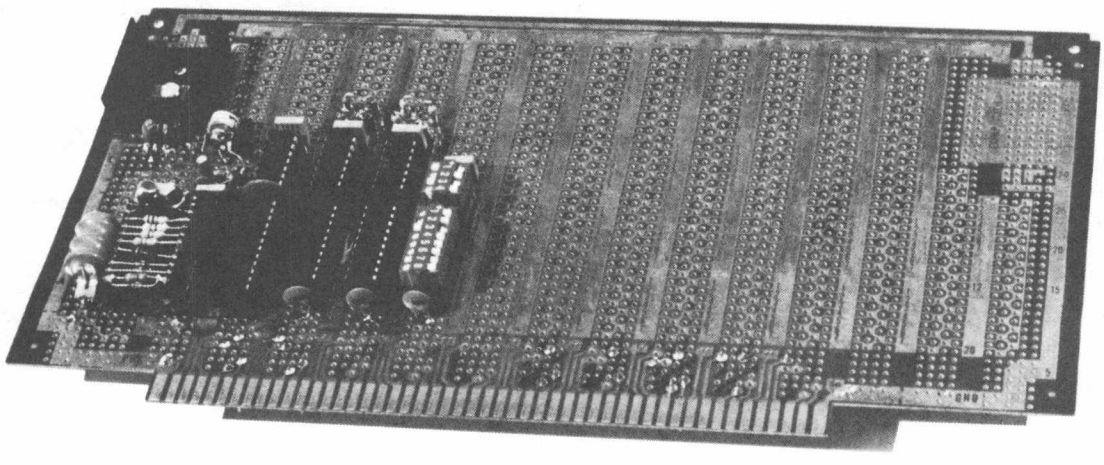
For convenient time/date-stamping of a full program listing, I chose the approach in the structure chart shown in Figure 6 (page 63). This program is invoked by TIMELIST <filename>. First, the title line is printed in the format: time, date, program name. Then, after skipping several lines, the printout of the text of the program begins, with a total of 60 lines on each page. All of the printing is done in subroutine PRINTLOOP, and you can interrupt it anytime from the keyboard; that is, a control-S pauses and any keypress continues, unless the printing is not paused, in which case any keypress aborts the program.

The clock-related aspects of this program are similar to TIME. The obvious difference is that a file must be opened, read, and printed. At the very beginning, then, because I used the 128 bytes between 80H and 100H for a disk buffer area, it is necessary to move the stack to a safe place. Then the first subroutine, OPENFILE, either opens the file successfully or calls QUIT if it encounters a problem. On exit, the stack pointer is restored to its previous position.

PRINTLOOP sends one character for each main loop within the subroutine. Before getting each new character, the program checks to see if a console break is present, to verify required spaces at the beginning of a line, and to determine if the maximum number of lines per page has been reached. The program as now assembled adds three spaces at the left margin for all printed text; this is because I usually notebook-punch my programs and my Okidata 82A prints too closely to the left-hand margin. To change this margin setting, adjust the SKIP equate. Likewise, to change from 60 lines per page, modify the LPAGE equate.

After getting a new byte by a call to GETNB, I check for the form-feed character and ignore it. To allow form feeds from the text, change the KEEPFF equate to TRUE. When accepting form feeds from the text, I disabled the line-counting within PRINTLOOP on the assumption that page length would have already been determined.

Before actually printing the character, the program checks for the proper line length. This is necessary because I skip several spaces at the left margin. If the text runs to 80 characters, but I've already printed three



The real-time clock on a Vector 8800 V wire-wrap board.

spaces, then three characters will have to wrap around to another line on the printer. To maintain the left margin, I need to insert the usual spaces before beginning to print the wrap-around letters.

If the character to be printed is a tab character (09), then TABTOSP does a conversion to spaces so that the printer lines up to every eighth column. The conversion is *not* simply to replace every tab with eight spaces: the program checks to see what the current column position is in relation to every eighth column then inserts just enough spaces to get the next eighth position. Thus, a tab positions the printer to columns 9, 17, 25, 33, and so on. Change the NRSPCS equate to the desired tab conversion if eight is not suitable.

As with all programs, a number of modifications always seem necessary. The CLOCKSET, TIME, and TIMELIST programs are certainly not the final versions. Improvements are always possible; it becomes an issue of whether the programs are "good enough to do the job" or *must* be revised. I chose to leave these three programs as they stand and go on to other more creative programs.

One program is to display the current time on the console, but I have a wristwatch that displays time quite well. Consider a "reminder" program to sound the console bell at a certain time. Also consider how to keep tax records of business use of the computer: a program to LOGON and LOGOFF that records date, time on/off, and subject of computer use. This program should create a file suitable for use by a spreadsheet program for tax records. These programs and others using the interrupt features of the clock are the subject of a future article.

Operation

When first starting up the clock, set the proper time and date using CLOCKSET. Then run the TIME program to see that the time and date are being displayed properly. Also check the printing feature of TIME by appending a small text to the TIME invocation to see that the time and text go to the printer. Test run TIMELIST with a two- or three-page text file to see that it prints the first page with time and date then prints 60 lines on each of the following pages.

Conclusions

The clock board has proven to be a useful addition to my computer system. I've found it especially convenient for dating disk directory prints, as well as for the normal dating of program printouts. Keeping the time and date on my program listings has certainly improved my program filing system: now when I discover a long-lost paper, I know when it disappeared!

I would like to thank our Bucknell Technician, Thomas J. Thul, Jr., for his help in laying out and constructing the clock circuit board. Program source code is available from the author on an 8-inch CP/M disk for $15 postpaid.

References

[1] Calaway, J. L., and B. Hill, "CP/M, Your Time Has Come," *BYTE*, May 1982, pp. 479-493.

[2] Hassebrock, Glen E., Jr., "The Hayes Chronograph, an H-89, and CP/M," *REMark*, No. 35, December 1982, pp. 9-14.

[3] Ciarcia, Steve, "Everyone Can Know the Real Time," *BYTE*, May 1982, pp. 34-58.

[4] *MM58167A Microprocessor Real Time Clock*, National Semiconductor Data Sheet, April 1982.

[5] *IEEE Standard 696 Interface Devices*, New York, NY: IEEE, 1983.

[6] Libes, Sol, and Mark Garetz, *Interfacing to S-100/IEEE-696 Microcomputers*, Berkeley, CA: Osborne/McGraw-Hill, 1981.

[7] *The TTL Data Book for Design Engineers*, 2nd ed., Dallas, TX: Texas Instruments, Inc., 1981.

Trademarks

Hayes Chronograph is a trademark of Hayes Microcomputer Products, and Z8 is a trademark of Zilog.

Hardware:	IEEE Standard 696-1983 compatible
	Operation to 6 MHz (using two wait states)
	Handshaking using RDY line (72) pulled low until clock output data valid
	DIP-switch selection of base address on any 32-byte I/O port boundary, 32 ports required
	DIP-switch selection of vectored interrupt 0 through 7 for clock interrupts 100 ms to 1 month or RAM and counter compares
	DIP-switch selection of vectored interrupt 0 through 7 for RAM and counter compares
	Accuracy 0.01%, but generally better than 5 sec/month
	Power requirement +8V < 200 mA during normal operation
	Rechargeable battery backup
	Construction using Vector 8800V and wire-wrap
Software:	24-hour time: hour, minute, second
	Calendar: day of week and month, month, year
	File-list utility with time/date header
	Clock time- and date-setting program

Table
Specifications of the S-100 Real-Time Clock

DDJ

Real-Time Clock (Listing Continued, text begins on page 56)

Listing One

```
                        ; CLOCKSET      Program to set date & time of real-time
                        ;
                        ; Program by    Alan D. Wilcox
                        ;               6 January 1985
                        ;
0100                            ORG     100H

00A0 =                  CLOCK   EQU     0A0H    ; Clock base address

0000 =                  FALSE   EQU     0
00FF =                  TRUE    EQU     0FFH

0005 =                  BDOS    EQU     5H      ; BDOS entry point

0001 =                  CONSIN  EQU     1       ; Console input
0002 =                  CONSOUT EQU     2       ; Console output
0009 =                  PRINTST EQU     9       ; Print string function
000B =                  GETSTAT EQU     11      ; Console status

000A =                  LF      EQU     0AH     ; Line Feed
000D =                  CR      EQU     0DH     ; Carriage Return
001B =                  ESC     EQU     1BH     ; Escape
                        ;
                        ;
                        ; ***** MAIN PROGRAM *****
                        ; *
                        ; *

0100 113202     START:  LXI     D,GREET
0103 CD1302             CALL    PRINTIT

0106 116502             LXI     D,DATESET       ; Set the date?
0109 CD1302             CALL    PRINTIT
010C CD1B02             CALL    GETANS          ; Get answer
010F FEFF               CPI     TRUE
0111 CC2901             CZ      SETDATE

0114 117F02             LXI     D,TIMESET       ; Set the time?
0117 CD1302             CALL    PRINTIT
011A CD1B02             CALL    GETANS          ; Get answer
011D FEFF               CPI     TRUE
011F CC6401             CZ      SETTIME

0122 119F03             LXI     D,CRLF
0125 CD1302             CALL    PRINTIT

0128 C9         DONE:   RET                     ; Return to CP/M

                        ; *
                        ; *
                        ;
                        ; ***** SUBR SETDATE *****
                        ;
                        ;       Set the current date
                        ;
0129 119902     SETDATE:LXI     D,DAY           ; What day of week?
012C CD1302             CALL    PRINTIT
012F CDDB01             CALL    GETBCD          ; Get it and store
0132 D3A5               OUT     CLOCK+5         ;  in clk counter

0134 11C402             LXI     D,DATE          ; What date is it?
0137 CD1302             CALL    PRINTIT
013A CDDB01             CALL    GETBCD          ; Get it and store
013D D3A6               OUT     CLOCK+6         ;  in clk counter

013F 11E002             LXI     D,MONTH         ; What month is it?
0142 CD1302             CALL    PRINTIT
0145 CDDB01             CALL    GETBCD          ; Get it and store
0148 D3A7               OUT     CLOCK+7         ;  in clk counter

014A 11FC02             LXI     D,YEAR          ; And the year?
014D CD1302             CALL    PRINTIT
```

(Continued on page 70)

Listing One

```
0150 CDDB01             CALL    GETBCD          ; Get it and store
0153 D3A9               OUT     CLOCK+9         ;  in clock RAM

0155 118A03             LXI     D,AGAIN         ; Do this over again?
0158 CD1302             CALL    PRINTIT
015B CD1B02             CALL    GETANS          ; Get answer
015E FEFF               CPI     TRUE
0160 CA2901             JZ      SETDATE         ; Yes, do it.

0163 C9                 RET
                ;
                ;
                ; ***** SUBR SETTIME *****
                ;
                ;       Set the current time
                ;
0164 CD9C01     SETTIME:CALL    DISPLAY         ; Show the time at start

0167 113503             LXI     D,SETHR         ; Set-hrs message
016A CD1302             CALL    PRINTIT
016D CDDB01             CALL    GETBCD          ; Get 2 ASCII, convert to BCD
0170 D3A4               OUT     CLOCK+4         ; Put hours into clock cnter

0172 115203             LXI     D,SETMIN        ; Set-min message
0175 CD1302             CALL    PRINTIT
0178 D3B5               OUT     CLOCK+21        ; Reset seconds
017A CDDB01             CALL    GETBCD          ; Get 2 ASCII, convert to BCD
017D D3A3               OUT     CLOCK+3         ; Put mins into clock counter

017F 116E03             LXI     D,SETSEC        ; Set-seconds message
0182 CD1302             CALL    PRINTIT
0185 CD0B02             CALL    INCHAR          ; Get key-press
0188 D3B5               OUT     CLOCK+21        ; "GO" command zero seconds

018A CD9C01             CALL    DISPLAY         ; Show the results

018D 118A03             LXI     D,AGAIN         ; Ask to do again
0190 CD1302             CALL    PRINTIT
0193 CD1B02             CALL    GETANS
0196 FEFF               CPI     TRUE
0198 CA6401             JZ      SETTIME         ; Yes, do it again

019B C9                 RET
                ;
                ;
                ; ***** SUBR DISPLAY *****
                ;
                ;       Displays current time
                ;
019C 111803     DISPLAY:LXI     D,TMSG          ; Time is ...
019F CD1302             CALL    PRINTIT

01A2 DBA4               IN      CLOCK+4         ; Get hours from clock
01A4 CDBC01             CALL    PRHEX           ; Print hours
01A7 3E3A               MVI     A,':'
01A9 CD0402             CALL    PCHAR

01AC DBA3               IN      CLOCK+3         ; Get minutes from clock
01AE CDBC01             CALL    PRHEX           ; Print minutes
01B1 3E3A               MVI     A,':'
01B3 CD0402             CALL    PCHAR

01B6 DBA2               IN      CLOCK+2         ; Get seconds
01B8 CDBC01             CALL    PRHEX           ; Print seconds

01BB C9                 RET
                ;
                ;
                ; ***** SUBR PRHEX *****
                ;
                ;       Converts binary value into two ASCII-hex characters
                ;       and prints on console
```

(Continued on page 72)

Listing One

```
                ;
01BC F5         PRHEX:  PUSH    PSW
01BD 0F                 RRC                     ; Put 4 MSB's into 4 LSB's
01BE 0F                 RRC
01BF 0F                 RRC
01C0 0F                 RRC
01C1 CDC901             CALL    PRNIB           ; Prnt hex equiv to 4 MSB's
01C4 F1                 POP     PSW
01C5 CDC901             CALL    PRNIB           ; Prnt hex equiv to 4 LSB's
01C8 C9                 RET
                ;
                ;
                ; ***** SUBR PRNIB *****
                ;
                ;       Prints a nibble of Reg A
                ;
01C9 E60F       PRNIB:  ANI     0FH             ; Mask out top 4 bits
01CB FE0A               CPI     10              ; Is it number or letter?
01CD D2D501             JNC     LETR            ; Must be a letter if jump.
01D0 C630               ADI     '0'             ; Add offset to make  ASCII
01D2 C3D701             JMP     PRNT
01D5 C637       LETR:   ADI     'A' - 10        ; Add offset to make binary
                                                ;  into ASCII letter
01D7 CD0402     PRNT:   CALL    PCHAR           ; Send ASCII to console
01DA C9                 RET
                ;
                ;
                ; ***** SUBR GETBCD *****
                ;
                ;       Gets 2 ASCII characters, converts them to packed BCD
                ;
                ;       EXIT:   A = 2 BCD characters
                ;
01DB CD0B02     GETBCD: CALL    INCHAR          ; Get character
01DE FE30               CPI     '0'
01E0 DADB01             JC      GETBCD          ; Ignore char if < ASCII 0
01E3 FE3A               CPI     '9'+1
01E5 D2DB01             JNC     GETBCD          ; Ignore char if > ACCII 9
01E8 D630               SUI     '0'             ; Make ASCII into binary
01EA E60F               ANI     0FH             ; Mask out 4 MSB's
01EC 07                 RLC
01ED 07                 RLC
01EE 07                 RLC
01EF 07                 RLC                     ; Put into MSB position
01F0 47                 MOV     B,A             ;  and save in B
01F1 CD0B02     LOBYTE: CALL    INCHAR          ; Get 2nd character
01F4 FE30               CPI     '0'
01F6 DAF101             JC      LOBYTE
01F9 FE3A               CPI     '9'+1
01FB D2F101             JNC     LOBYTE
01FE D630               SUI     '0'             ; Make ASCII into binary.
0200 E60F               ANI     0FH
0202 80                 ADD     B               ; Combine both BCD char in A.
0203 C9                 RET
                ;
                ;
                ; ***** SUBR PCHAR *****
                ;
                ;       Prints Reg A to console
                ;
0204 0E02       PCHAR:  MVI     C,CONSOUT
0206 5F                 MOV     E,A
0207 CD0500             CALL    BDOS
020A C9                 RET
                ;
                ;
                ; ***** SUBR INCHAR *****
                ;
                ;       Reg A gets ASCII value from keyboard
                ;
020B C5         INCHAR: PUSH    B               ; Have char saved in B
020C 0E01               MVI     C,CONSIN
```

(Continued on page 74)

Real-Time Clock (Listing Continued, text begins on page 56)

Listing One

```
020E CD0500             CALL    BDOS
0211 C1                 POP     B
0212 C9                 RET
                ;
                ;
                ; ***** SUBR PRINTIT *****
                ;
                ;       ENTRY:  DE = Start address of string
                ;
0213 F5         PRINTIT:PUSH    PSW
0214 0E09               MVI     C,PRINTST
0216 CD0500             CALL    BDOS            ; Print string to console
0219 F1                 POP     PSW
021A C9                 RET
                ;
                ;
                ; ***** SUBR GETANS *****
                ;
                ;       Ask question, then call this subroutine for answer.
                ;       Assume default answer is NO.
                ;
                ;       EXIT:   A = True = FF if YES
                ;               A = False= 0  if NO
                ;
021B 0E01       GETANS: MVI     C,CONSIN
021D CD0500             CALL    BDOS            ; Get console input
0220 FE59               CPI     'Y'
0222 CA2F02             JZ      ENDTRU
0225 FE79               CPI     'y'
0227 CA2F02             JZ      ENDTRU
022A 3E00               MVI     A,FALSE         ; Not Y or y. Assume NO.
022C C33102             JMP     GOTANS
022F 3EFF       ENDTRU: MVI     A,TRUE
0231 C9         GOTANS: RET
                ;
                ;
                ; ***** CONSOLE MESSAGES *****
                ;
0232 0D0A50726F GREET:      DB      CR,LF,'Program to set DATE & TIME of '
0252 7265616C2D             DB      'real-time clock.',LF,CR,'$'

0265 0D0A0A5365 DATESET:    DB      CR,LF,LF,'Set the date? <N> --> $'
027F 0D0A0A5365 TIMESET:    DB      CR,LF,LF,'Set the time? <N> --> $'

0299 0D0A0A5375 DAY:        DB      CR,LF,LF,'Sunday = 01.'
02A8 0D0A456E74             DB      CR,LF,'Enter day 01 to 07.  --> $'
02C4 0D0A456E74 DATE:       DB      CR,LF,'Enter date 01 to 31  --> $'
02E0 0D0A456E74 MONTH:      DB      CR,LF,'Enter month 01 to 12 --> $'
02FC 0D0A456E74 YEAR:       DB      CR,LF,'Enter year 84 to 99  --> $'

0318 0D0A0A4872 TMSG:       DB      CR,LF,LF,'Hrs : Min : Sec now  --> $'
0335 0D0A0A496E SETHR:      DB      CR,LF,LF,'Input HRs 00 to 23   --> $'
0352 0D0A496E70 SETMIN:     DB      CR,LF,'Input Min 00 to 59   --> $'
036E 0D0A507265 SETSEC:     DB      CR,LF,'Press <cr> to zero sec...$'

038A 0D0A0A446F AGAIN:      DB      CR,LF,LF,'Do over? <N> ... $'

039F 0D0A24     CRLF:       DB      CR,LF,'$'

03A2                    END
```

End Listing One

Listing Two

```
                ; TIME          Display time and date of real-time clock
                ;
                ; Program by    Alan D. Wilcox
                ;               18 December 1984
                ;
                ; Invoke by     A>TIME          Output to Console
                ;               A>TIME <text>   Output to Console & LST:
```

```
                    ;                                   (List time & text)
    0100                    ORG     100H

    00A0 =          CLOCK   EQU     0A0H    ; Clock base address

    0005 =          BDOS    EQU     5H      ; BDOS entry point
    0080 =          BUFF    EQU     80H     ; Buffer

    0005 =          LISTOUT EQU     5       ; List output
    0009 =          PRINTST EQU     9       ; Print string function

    000A =          LF      EQU     0AH     ; Line Feed
    000D =          CR      EQU     0DH     ; Carriage Return
    001B =          ESC     EQU     1BH     ; Escape
                    ;
                    ;
                    ; ***** MAIN PROGRAM *****
                    ; *
                    ; *
                    ;       Read current time and save in memory

  0100 DBA4      GETIME: IN      CLOCK+4         ; Get hours from clock cntr
  0102 11D701            LXI     D,HOURS
  0105 CD6D01            CALL    BCDTOASC        ; Convert to ASCII & save

  0108 DBA3              IN      CLOCK+3         ; Get minutes from clk cntr
  010A 11DA01            LXI     D,MINUTES
  010D CD6D01            CALL    BCDTOASC        ; Convert

  0110 DBA2              IN      CLOCK+2         ; Get seconds
  0112 11DD01            LXI     D,SECONDS
  0115 CD6D01            CALL    BCDTOASC

                ;       Read day of week and save literal equiv. in memory

  0118 DBA5      GETDAY: IN      CLOCK+5
  011A 11E201            LXI     D,DAY           ; Where to save the result
011D 211302            LXI     H,LITDAY        ; Location of literal weekday
0120 CD7F01            CALL    NTOLIT          ; Convert number to literal

                ;       Read date and save in memory

0123 DBA6      GETDATE:IN      CLOCK+6
0125 11EE01            LXI     D,DATE          ; Where to save result
0128 CD6D01            CALL    BCDTOASC        ; Convert to ASCII and save

                ;       Read month and save literal equivalent in memory

012B DBA7      GETMON: IN      CLOCK+7
012D 11F101            LXI     D,MONTH         ; Where to save result
0130 215902            LXI     H,LITMON        ; Location of literal months
0133 CD7F01            CALL    NTOLIT          ; Convert to literal

                ;       Read year from clock RAM & save in memory

0136 DBA9      GETYR:  IN      CLOCK+9
0138 11FD01            LXI     D,YEAR          ; Where to save result
013B CD6D01            CALL    BCDTOASC        ; Convert to ASCII and save

                ;       Send time message to console

013E 11CE01    CONSL:  LXI     D,MSGL          ; DE gets adr of message
0141 0E09              MVI     C,PRINTST
0143 CD0500            CALL    BDOS            ; Print string to console

                ;       Send time msg to printer if cmd string required it

0146 3A8000    PRINT:  LDA     BUFF            ; Was cmd 'TIME <text>'
0149 FE00              CPI     0               ; Done if nothing there
014B CA6C01            JZ      DONE

014E 4F                MOV     C,A             ; Put buffer count in C
014F 218100            LXI     H,BUFF+1        ; HL is source of text
0152 110102            LXI     D,TEXT          ; DE is destin for text

0155 7E        MOVETXT:MOV     A,M             ; Get byte
0156 12                STAX    D               ; Save it
```

(Continued on next page)

Listing Two

```
0157 23              INX     H
0158 13              INX     D
0159 0D              DCR     C
015A C25501          JNZ     MOVETXT          ; Do until C is 0

015D 3E24            MVI     A,'$'            ; End-text marker
015F 12              STAX    D

0160 21BD01          LXI     H,MSGR           ; Go ahead to printer
0163 CDA701          CALL    LISTIT
0166 21BA01          LXI     H,CRLF           ; Finish the line
0169 CDA701          CALL    LISTIT

016C C9        DONE: RET                       ; Return to CP/M

               ; *
               ; *

               ; ***** SUBR BCDTOASC *****
               ;
               ;       Converts binary value into two ASCII-hex characters
               ;
               ;       ENTRY:  Reg A  = 2 packed-BCD characters
               ;                   DE = Adr of where to store ASCII answers
               ;
               ;       EXIT:     (DE) = MS char
               ;                (DE+1) = LS char
               ;
               BCDTOASC:
016D F5                PUSH    PSW
016E 0F                RRC                     ; Put 4 MSB's into 4 LSB's
016F 0F                RRC
0170 0F                RRC
0171 0F                RRC
0172 E60F              ANI     0FH             ; Mask out top 4 bits
0174 C630              ADI     '0'             ; Add offset to make  ASCII
0176 12                STAX    D               ; Save the result
0177 F1                POP     PSW             ; Get the next character
0178 E60F              ANI     0FH
017A C630              ADI     '0'             ; Make ASCII
017C 13                INX     D
017D 12                STAX    D               ; Save it in next memory loc.
017E C9                RET


               ; ***** SUBR NTOLIT *****
               ;
               ;       Converts packed-BCD number (1 to 12) to its equiv
               ;       literal string and saves in specified memory.
               ;
               ;       ENTRY:  A  =  2 packed-BCD numbers from clock
               ;               DE =  Adr of where to store literal results
               ;               HL =  Pntr to 10-char literal strings
               ;
017F 47        NTOLIT: MOV     B,A             ; Save data
0180 E610              ANI     00010000B       ; See if have tens digit
0182 CA8B01            JZ      SMALLNR         ; Jump if not
0185 78                MOV     A,B
0186 C60A              ADI     10              ; Add 10 to low nibble so
0188 C38C01            JMP     BINARY          ;  it's binary now.
018B 78        SMALLNR:MOV     A,B

018C E60F      BINARY: ANI     0FH             ; Mask off top nibble
018E D601              SUI     1               ; Make number 0 to 11 max
0190 CA9B01            JZ      MOVE            ; At first already

0193 010A00            LXI     B,10            ; 10 literals in word
0196 09        NEXT:   DAD     B               ; Add it to HL for next word
0197 3D                DCR     A               ;  until get to proper one.
0198 C29601            JNZ     NEXT
```

(Continued on page 78)

Listing Two

```
019B 010A00     MOVE:   LXI     B,10            ; Move 10 letters only

019E 7E         MOVDAT: MOV     A,M             ; HL pts to desired word
019F 12                 STAX    D               ; DE pts to destination mem.
01A0 13                 INX     D
01A1 23                 INX     H
01A2 0D                 DCR     C               ; Proper literal word now
01A3 C29E01             JNZ     MOVDAT          ;  moved to memory to print

01A6 C9                 RET


                ; ***** SUBR LISTIT *****
                ;
                ;       Lists to printer until '$' encountered.
                ;
                ;       ENTRY:  HL  =  string address
                ;
01A7 7E         LISTIT: MOV     A,M             ; Get the letter to be sent
01A8 FE24               CPI     '$'             ; End of text yet?
01AA CAB901             JZ      LISTEND
01AD E5                 PUSH    H               ; Must retain HL!
01AE 5F                 MOV     E,A
01AF 0E05               MVI     C,LISTOUT
01B1 CD0500             CALL    BDOS            ; Print the char
01B4 E1                 POP     H
01B5 23                 INX     H
01B6 C3A701             JMP     LISTIT          ; Keep going until '$'
01B9 C9         LISTEND:RET


                ; ***** MESSAGE AREA *****
                ;
01BA 0D0A24       CRLF:     DB      CR,LF,'$'

01BD 2020202020   MSGR:     DB      '                '
01CC 2020                   DB      '  '
01CE 2020202020   MSGL:     DB      '          '
01D7              HOURS:    DS      2
01D9 3A                     DB      ':'
01DA              MINUTES:  DS      2
01DC 3A                     DB      ':'
01DD              SECONDS:  DS      2
01DF 202020                 DB      '   '
01E2              DAY:      DS      10
01EC 2020                   DB      '  '
01EE              DATE:     DS      2
01F0 20                     DB      ' '
01F1              MONTH:    DS      10
01FB 3139                   DB      '19'
01FD              YEAR:     DS      2
01FF 2020                   DB      '  '
0201 2020202020   TEXT:     DB      '                '
0210 0D0A24                 DB      CR,LF,'$'


                ; ***** LITERALS *****
                ;
                ;           Pack with NULLs for even spacing on print line
                ;
0213 53756E6461 LITDAY:     DB      'Sunday,',0,0,0
021D 4D6F6E6461             DB      'Monday,',0,0,0
0227 5475657364             DB      'Tuesday,', 0,0
0231 5765646E65             DB      'Wednesday,'
023B 5468757273             DB      'Thursday,',  0
0245 4672696461             DB      'Friday,',0,0,0
024F 5361747572             DB      'Saturday,',  0

0259 4A616E7561 LITMON:     DB      'January ',    0,0
0263 4665627275             DB      'February ',     0
026D 4D61726368             DB      'March ',  0,0,0,0
```

```
0277 417072696C             DB      'April ',  0,0,0,0
0281 4D61792000             DB      'May ',0,0,0,0,0,0
028B 4A756E6520             DB      'June ', 0,0,0,0,0
0295 4A756C7920             DB      'July ', 0,0,0,0,0
029F 4175677573             DB      'August ',   0,0,0
02A9 5365707465             DB      'September '
02B3 4F63746F62             DB      'October ',    0,0
02BD 4E6F76656D             DB      'November ',     0
02C7 446563656D             DB      'December ',     0

02D1                    END
```

End Listing Two

Listing Three

```
                        ; TIMELIST      Program to read a file and send it out
                        ;               to printer with time/date and program name
                        ;
                        ; Program by    Alan D. Wilcox
                        ;               29 October 1984
                        ;
                        ; Usage         A> TIMELIST  filespec.txt
                        ;

  0100                          ORG     100H

  00A0 =        CLOCK   EQU     0A0H    ; Clock base address

  0000 =        FALSE   EQU     0
  FFFF =        TRUE    EQU     NOT FALSE

  FFFF =        TOLPTR  EQU     TRUE    ; Make FALSE to send all output to
                                        ;  console rather than to line prntr

 0000 =         KEEPFF  EQU     FALSE   ; Make TRUE to use form feeds in
                                        ;  orig text. TRUE also cancels line
                                        ;  counting by this program.

 0005 =         BDOS    EQU     0005H   ; BDOS entry point

 0002 =         CONSOUT EQU     2       ; Console Output
 0005 =         LISTOUT EQU     5       ; List Output
 0009 =         PRINTST EQU     9       ; Print String Function
 000B =         GETSTAT EQU     11      ; Console Status
 000F =         OPENF   EQU     15      ; Open File
 0014 =         READF   EQU     20      ; Read Next Record

 005C =         FCB     EQU     5CH     ; File Control Block Address
 007C =         FCBCR   EQU     FCB+32  ; Current (next) record for r/w
 0080 =         BUFF    EQU     80H     ; Input Disk Buffer Address

 000C =         TSIZE   EQU     12      ; Title block size
 0003 =         SKIP    EQU     3       ; Added number of spcs at left margn
0050 =        COLS    EQU     80      ; Max possible columns per page
003C =        LPAGE   EQU     60      ; Lines to print per page
0008 =        NRSPCS  EQU     8       ; Convert tab to NRSPCS spaces

0009 =        TAB     EQU     09H     ; Horizontal Tab
000A =        LF      EQU     0AH     ; Line Feed
000B =        VT      EQU     0BH     ; Vertical Tab (alt FF)
000C =        FF      EQU     0CH     ; Form Feed
000D =        CR      EQU     0DH     ; Carriage Return
001A =        CNTLZ   EQU     1AH     ; End of File (cntl-z)
001B =        ESC     EQU     1BH     ; Escape
0020 =        SPC     EQU     20H     ; Space
              ;
              ;
              ; ********** MAIN PROGRAM **********
              ; *
              ; *

0100 210000   STACK:  LXI     H,0             ; Stack in safe place while
0103 39               DAD     SP              ;   using disk buffer space.
0104 227904           SHLD    OLDSTK          ; Save old stack position
```

(Continued on next page)

Listing Three

```
0107 319F04             LXI     SP,NEWSTK       ; Use new stack pointer

010A CD1B01             CALL    OPENFILE        ; Open program to be read

010D CD3B01             CALL    SAVETITLE       ; Save pgm name as the title

0110 CD7901             CALL    FIRSTLINE       ; Prnt time/date/title line

0113 CDD301             CALL    PRINTLOOP       ; Read & print file

0116 2A7904     QUIT:   LHLD    OLDSTK          ; Get the old stack pntr in
0119 F9                 SPHL                    ;  HL & put in on the stack.
011A C9                 RET                     ; Return to CP/M

                ; *
                ; *
                ;
                ; **********    SUBR OPENFILE     **********
                ;
                OPENFILE:       ; Open the file requested

011B AF                 XRA     A               ; Zero accumulator
011C 327C00             STA     FCBCR           ; Zero file record count
011F 115C00             LXI     D,FCB
0122 0E0F               MVI     C,OPENF         ; Open the file
0124 CD0500             CALL    BDOS            ; Get OFFH in Accum if error
0127 FEFF               CPI     OFFH
0129 C23501             JNZ     OKOPEN

012C 113D03             LXI     D,OPENERR       ; Had error in file opening
012F CD3503             CALL    PRINTIT         ; Print error message
0132 CD1601             CALL    QUIT            ; Restore stack and get out

0135 3E80       OKOPEN: MVI     A,BUFF          ; Set position of input
0137 327B04             STA     INBUFPT         ;  buffer pointer
013A C9                 RET
                ;
                ;
                ; **********    SUBR SAVETITLE    **********
                ;
                SAVETITLE:      ; Save name of program as title

013B 0E0C               MVI     C,TSIZE         ; Clear TSIZE space in memory
013D 21AA03             LXI     H,HEADER        ;  starting at HEADER.
0140 3620       CLRBLK: MVI     M,SPC           ; Fill with spaces
0142 23                 INX     H
0143 0D                 DCR     C
0144 C24001             JNZ     CLRBLK
0147 3A8300             LDA     BUFF+3          ; See if drive in cmd line
014A FE3A               CPI     ':'             ; Z set if true
014C C25A01             JNZ     NODRIV
014F 118400             LXI     D,BUFF+4        ; DE pts to pgm name only
0152 3A8000             LDA     BUFF            ; Get count of chars
0155 D603               SUI     3               ; Sub cnt, space, drive spec
0157 C36201             JMP     ADJSIZE

015A 118200     NODRIV: LXI     D,BUFF+2        ; DE points to pgm name
015D 3A8000             LDA     BUFF            ; Get count of chars
0160 D601               SUI     1               ; Subtr count & space

0162 FE0C       ADJSIZE:CPI     TSIZE           ; Adjust char cnt so it fits
0164 F26B01             JP      BIG             ; Jump if count > TSIZE
0167 4F                 MOV     C,A             ; Acceptable count into C
0168 C36D01             JMP     OKSIZE
016B 0E0C       BIG:    MVI     C,TSIZE         ; Title is max size

016D 21AA03     OKSIZE: LXI     H,HEADER        ; Destination for title move

0170 1A         MTITLE: LDAX    D               ; Get the character
0171 77                 MOV     M,A             ;  and save it as header
0172 23                 INX     H
0173 13                 INX     D
```

```
0174 0D             DCR     C
0175 C27001         JNZ     MTITLE          ; Title saved when zero
0178 C9             RET
                ;
                ;
                ; **********    SUBR FIRSTLINE  **********
                ;
                FIRSTLINE:      ; Read current time and save in memory

0179 DBA4           IN      CLOCK+4         ; Get hours from clock
017B 116B03         LXI     D,HOURS
017E CDE102         CALL    BCDTOASC        ; Convert to ASCII & save

0181 DBA3           IN      CLOCK+3         ; Get minutes from clock
0183 116E03         LXI     D,MINUTES
0186 CDE102         CALL    BCDTOASC        ; Convert

0189 DBA2           IN      CLOCK+2         ; Get seconds
018B 117103         LXI     D,SECONDS
018E CDE102         CALL    BCDTOASC

                ;       Read day of week and save literal equiv. in memory

0191 DBA5           IN      CLOCK+5
0193 118003         LXI     D,DAY           ; Where to save the result
0196 21BB03         LXI     H,LITDAY        ; Location of literal weekday
0199 CDF302         CALL    NTOLIT          ; Convert number to literal

                ;       Read date and save in memory

019C DBA6           IN      CLOCK+6
019E 118B03         LXI     D,DATE          ; Where to save result
01A1 CDE102         CALL    BCDTOASC        ; Convert to ASCII and save

                ;       Read month and save literal equivalent in memory

01A4 DBA7           IN      CLOCK+7
01A6 118E03         LXI     D,MONTH         ; Where to save result
01A9 210104         LXI     H,LITMON        ; Location of literal months
01AC CDF302         CALL    NTOLIT          ; Convert to literal

                ;       Read year from clock RAM & save in memory

01AF DBA9           IN      CLOCK+9
01B1 119A03         LXI     D,YEAR
01B4 CDE102         CALL    BCDTOASC        ; Convert to ASCII and save

                ;       Clear spaces at left margin before printing time

01B7 0E03           MVI     C,SKIP
01B9 216803         LXI     H,TOPLINE
01BC 3620       CLRSKP: MVI     M,SPC           ; Fill with spaces
01BE 23             INX     H
01BF 0D             DCR     C
01C0 C2BC01         JNZ     CLRSKP

                IF TOLPTR       ; Send time message to line printer
01C3 216803         LXI     H,TOPLINE       ; HL gets adr of message
01C6 CD1B03         CALL    LISTIT          ; To printer
                ENDIF
                ;
                IF NOT TOLPTR   ; Send time message to console
                    LXI     D,TOPLINE       ; DE gets adr of message
                    CALL    PRINTIT         ; Print to console
                ENDIF

                ;       Set line cntr to 3 to account for title block lines

01C9 3E03           MVI     A,3
01CB 327E04         STA     LINECNT

                ;       Initialize column counter

01CE AF             XRA     A               ; Zero accum
01CF 327D04         STA     COLCNT          ; Column counter zero
01D2 C9             RET
                ;
                ;
```

(Continued on next page)

Listing Three

```
                ; **********    SUBR PRINTLOOP  **********
                ;
                PRINTLOOP:      ; Will stay in this loop until either break
                                ;  from console or file is completely printed

                ;       Check console for break. Exit without losing stack

01D3 0E0B               MVI     C,GETSTAT
01D5 CD0500             CALL    BDOS            ; Accum LSB=1 if brk request
01D8 0F                 RRC                     ; Put LSB into carry
01D9 DA5702             JC      DONE            ; Leave program

                ;       Check to see if need spaces at start of left margin

01DC 3A7D04             LDA     COLCNT          ; Current column count
01DF FE03               CPI     SKIP            ; Sign set while COLCNT< SKIP
01E1 F2ED01             JP      CKLINE          ; Jump when COLCNT>=SKIP
01E4 3C                 INR     A               ; Increment col counter
01E5 327D04             STA     COLCNT
01E8 3E20               MVI     A,SPC
01EA C35102             JMP     GOPRNT          ; Go print space & continue

                ;       Check for proper lines per page. Send FF if needed

01ED 3A7E04     CKLINE: LDA     LINECNT         ; Current line count
01F0 FE3C               CPI     LPAGE           ; Max lines per page
01F2 C20102             JNZ     DATA            ; Get data if not equal yet

                IF KEEPFF
                        JMP     DATA            ; Don't bother counting.
                ENDIF                           ; Use FF's in source text.

01F5 AF                 XRA     A               ; Zero accum
01F6 327E04             STA     LINECNT         ; Reset line cntr when equal
01F9 327D04             STA     COLCNT          ; Reset column counter too
01FC 3E0C               MVI     A,FF
01FE C35102             JMP     GOPRNT          ; Print FF & continue

                ;       Get the data and filter out various characters

0201 CD5B02     DATA:   CALL    GETNB           ; Get the next byte.
0204 DA5702             JC      DONE            ; Carry set if EOF
0207 FE0D               CPI     CR
0209 CA5102             JZ      GOPRNT          ; Print the CR

020C FE0A       CKLF:   CPI     LF
020E C22102             JNZ     CKFF            ; Jump if not a LF
0211 3A7E04             LDA     LINECNT
0214 3C                 INR     A               ; Increment line counter
0215 327E04             STA     LINECNT
0218 AF                 XRA     A
0219 327D04             STA     COLCNT          ; Set col counter to zero
021C 3E0A               MVI     A,LF            ; Restore data
021E C35102             JMP     GOPRNT          ; Print the LF
                IF KEEPFF
                CKFF:   JMP     LNWID           ; Go and print FF & VT chars
                ENDIF
                ;
                IF NOT KEEPFF
0221 FE0C       CKFF:   CPI     FF
0223 CA0102             JZ      DATA            ; If FF, ignore. Get new data
0226 FE0B               CPI     VT
0228 CA0102             JZ      DATA            ; If VT, ignore it too.
                ENDIF

                ;       Check for proper width of line

022B F5         LNWID:  PUSH    PSW
022C 3A7D04             LDA     COLCNT          ; Get current column count
022F FE50               CPI     COLS            ; Max allowed per line
0231 C24202             JNZ     NOTMAX
```

(Continued on page 84)

Real-Time Clock (Listing Continued, text begins on page 56)

Listing Three

```
0234 CDC902             CALL    PRCRLF          ; If equal, do CR/LF, reset
                                                ;  the col counter to zero,
                                                ;  increment line counter.
0237 3A7B04             LDA     INBUFPT         ; Set buffer pointer back
023A 3D                 DCR     A               ;  to ignore this char.
023B 327B04             STA     INBUFPT
023E F1                 POP     PSW             ; Reset stack and go back
023F C3D301             JMP     PRINTLOOP       ;  thru read loop

0242 F1         NOTMAX: POP     PSW

0243 FE09       CKTAB:  CPI     TAB
0245 CC8602             CZ      TABTOSP         ; If TAB, expand it to spaces
0248 F5                 PUSH    PSW             ; Save data byte now in accum
0249 3A7D04             LDA     COLCNT
024C 3C                 INR     A               ; Increment column counter
024D 327D04             STA     COLCNT
0250 F1                 POP     PSW             ; Restore data for print

0251 CD2E03     GOPRNT: CALL    PCHAR
0254 C3D301             JMP     PRINTLOOP

0257 CDC902     DONE:   CALL    PRCRLF          ; Print CR/LF to wrapup
025A C9                 RET
                ;
                ;
                ; **********     SUBR GETNB     **********
                ;
                ;       Gets the next byte from memory or EOF flag if done
                ;
                ;       EXIT:   A = Next file byte fetched from memory
                ;               F = Carry set if End-of-File

025B 3A7B04     GETNB:  LDA     INBUFPT         ; Get current buff rel ptr
025E FE80               CPI     BUFF            ; Is it to end of rec yet?
0260 C27302             JNZ     GET             ; Get next byte if not.
                ;       Read a 128-byte record from disk

0263 115C00             LXI     D,FCB           ; DE gets FCB adr. Reads 128
0266 0E14               MVI     C,READF         ;  bytes into mem starting
0268 CD0500             CALL    BDOS            ;  at BUFF adr
026B B7                 ORA     A               ; A=0 if record read OK, so
026C CA7302             JZ      GET             ;  go get the info.
                                                ; Else if A#0, then is EOF.
026F 37         EOF:    STC                     ; Set carry to indicate EOF.
0270 C38502             JMP     GETEND

                GET:    ; Read the byte at BUFF + Reg A's relative offset

0273 5F                 MOV     E,A             ; DE contains rel position of
0274 1600               MVI     D,0             ;  pntr into disk buff space
0276 3C                 INR     A
0277 327B04             STA     INBUFPT         ; Incr and save new pointer.
027A 218000             LXI     H,BUFF          ; Start addr of buffer space
027D 19                 DAD     D               ;  plus rel offset ---> HL
027E 7E                 MOV     A,M             ; Get byte ptd to by HL
027F FE1A               CPI     CNTLZ           ; Check for EOF within record
0281 CA6F02             JZ      EOF             ;  & exit with carry bit set

0284 B7                 ORA     A               ; Reset carry bit
0285 C9         GETEND: RET
                ;
                ;
                ; **********     SUBR TABTOSP    **********
                ;
                TABTOSP:        ; Converts a tab to every NRSPCS columns

0286 3A7D04             LDA     COLCNT          ; Which column are we in?
0289 DE03               SBI     SKIP            ; Make relative offset
028B 5F                 MOV     E,A             ; Put it in E
028C 1608               MVI     D,NRSPCS
```

(Continued on page 86)

Listing Three

```
028E CDB002             CALL    MODULO          ; Do (COLCNT)mod NRSPCS
                                                ; Remainder is in Reg C
0291 3E08               MVI     A,NRSPCS
0293 91                 SUB     C               ; A has total # spaces to do
0294 3D                 DCR     A               ; Pickup space after RET
0295 4F                 MOV     C,A             ; C has # spaces to do now
0296 FE01               CPI     1               ; Set sign if need less than
0298 FAAD02             JM      ENDTAB          ;  one space printed.

029B 3E20       SPACES: MVI     A,SPC
029D C5                 PUSH    B
029E CD2E03             CALL    PCHAR
02A1 C1                 POP     B
02A2 3A7D04             LDA     COLCNT          ; Keep column count current
02A5 3C                 INR     A               ;  while sending spaces
02A6 327D04             STA     COLCNT
02A9 0D                 DCR     C
02AA C29B02             JNZ     SPACES          ; Loop until enough spaces

02AD 3E20       ENDTAB: MVI     A,SPC           ; Print the last space when
                                                ;  get to GOPRNT
02AF C9                 RET
                ;
                ;
                ; **********    SUBR MODULO     **********
                ;
                ;       Does division of 2 8-bit numbers: (Reg E)/(Reg D)
                ;       Result in Reg H
                ;       Remainer in Reg C = (Reg E) mod (Reg D)
                ;
02B0 210800     MODULO: LXI     H,00001000B
02B3 0E00               MVI     C,0
02B5 7B         NEWBIT: MOV     A,E
02B6 17                 RAL
02B7 5F                 MOV     E,A
02B8 79                 MOV     A,C
02B9 17                 RAL
02BA 92                 SUB     D
02BB D2BF02             JNC     NOADD
02BE 82                 ADD     D
02BF 4F         NOADD:  MOV     C,A
02C0 3F                 CMC
02C1 7C                 MOV     A,H
02C2 17                 RAL
02C3 67                 MOV     H,A
02C4 2D                 DCR     L
02C5 C2B502             JNZ     NEWBIT
02C8 C9                 RET
                ;
                ;
                ; **********    SUBR PRCRLF     **********
                ;
                PRCRLF:         ; Prints a CR/LF pair

02C9 F5                 PUSH    PSW
02CA AF                 XRA     A               ; Zero accum
02CB 327D04             STA     COLCNT          ; Set column counter to zero
02CE 3A7E04             LDA     LINECNT
02D1 3C                 INR     A               ; Increment line counter
02D2 327E04             STA     LINECNT
02D5 3E0D               MVI     A,CR
02D7 CD2E03             CALL    PCHAR
02DA 3E0A               MVI     A,LF
02DC CD2E03             CALL    PCHAR
02DF F1                 POP     PSW
02E0 C9                 RET
                ;
                ;
                ; **********    SUBR BCDTOASC   **********
                ;
```

Listing Three

```
                ;       Converts binary value into two ASCII-hex characters
                ;
                ;       ENTRY:  Reg A  = 2 packed-BCD characters
                ;                   DE = Adr of where to store ASCII answers
                ;
                ;       EXIT:     (DE) = MS char
                ;                 (DE+1) = LS char
                ;
                BCDTOASC:
02E1 F5                 PUSH    PSW
02E2 0F                 RRC                     ; Put 4 MSB's into 4 LSB's
02E3 0F                 RRC
02E4 0F                 RRC
02E5 0F                 RRC
02E6 E60F               ANI     0FH             ; Mask out top 4 bits
02E8 C630               ADI     '0'             ; Add offset to make  ASCII
02EA 12                 STAX    D               ; Save the result
02EB F1                 POP     PSW             ; Get the next character
02EC E60F               ANI     0FH
02EE C630               ADI     '0'             ; Make ASCII
02F0 13                 INX     D
02F1 12                 STAX    D               ; Save it in next memory loc
02F2 C9                 RET
                ;
                ;
                ; **********      SUBR NTOLIT     **********
                ;
                ;       Converts packed-BCD number (1 to 12) to its equiv
                ;       literal string and saves in specified memory.
                ;
                ;       ENTRY:  A  =  2 packed-BCD numbers from clock
                ;               DE =  Adr of where to store literal results
                ;               HL =  Pntr to 10-char literal strings
                ;
02F3 47         NTOLIT: MOV     B,A             ; Save data
02F4 E610               ANI     00010000B       ; See if have tens digit
02F6 CAFF02             JZ      SMALLNR         ; Jump if not
02F9 78                 MOV     A,B
02FA C60A               ADI     10              ; Add 10 to low nibble so
02FC C30003             JMP     BINARY          ;  it's binary now.
02FF 78         SMALLNR:MOV     A,B

0300 E60F       BINARY: ANI     0FH             ; Mask off top nibble
0302 D601               SUI     1               ; Make number 0 to 11 max
0304 CA0F03             JZ      MOVE            ; At first already

0307 010A00             LXI     B,10            ; 10 literals in word
030A 09         NEXT:   DAD     B               ; Add it to HL for next word
030B 3D                 DCR     A               ;  until get to proper one.
030C C20A03             JNZ     NEXT

030F 010A00     MOVE:   LXI     B,10            ; Cnt to move 10 letters only

    0312 7E         MOVDAT: MOV     A,M             ; HL pts to desired wor
0313 12                 STAX    D               ; DE pts to destination mem.
0314 13                 INX     D
0315 23                 INX     H
0316 0D                 DCR     C               ; Proper literal word now
0317 C21203             JNZ     MOVDAT          ;  moved to memory to print

031A C9                 RET
                ;
                ;
                ; **********      SUBR LISTIT     **********
                ;
                ;       Lists to printer until '$' encountered.
                ;
                ;       ENTRY:  HL  =  string address
                ;
031B 7E         LISTIT: MOV     A,M             ; Get the letter to be sent
031C FE24               CPI     '$'             ; End of text yet?
031E CA2D03             JZ      LISTEND
0321 E5                 PUSH    H               ; Must retain HL!
0322 5F                 MOV     E,A
0323 0E05               MVI     C,LISTOUT
0325 CD0500             CALL    BDOS            ; Print the char
```

```
0328 E1                 POP     H
0329 23                 INX     H
032A C31B03             JMP     LISTIT          ; Keep going until '$'
032D C9         LISTEND:RET
                ;
                ;
                ; **********     SUBR PCHAR     **********
                ;
                PCHAR:  ; Prints Reg A to console or printer

                IF TOLPTR
032E 0E05               MVI     C,LISTOUT       ; Output to printer
                ENDIF
                ;
                IF NOT TOLPTR
                        MVI     C,CONSOUT       ; Output to console
                ENDIF

0330 5F                 MOV     E,A
0331 CD0500             CALL    BDOS
0334 C9                 RET
                ;
                ;
                ; **********     SUBR PRINTIT   **********
                ;
                ;       ENTRY:  DE = Start address of string
                ;
0335 F5         PRINTIT:PUSH    PSW

0336 0E09               MVI     C,PRINTST
0338 CD0500             CALL    BDOS            ; Print string to console
033B F1                 POP     PSW
033C C9                 RET
                ;
                ;
                ; **********   CONSOLE MESSAGES   **********
                ;
033D 0D0A496E70 OPENERR:  DB      CR,LF,'Input file not found or not '
035B 7370656369           DB      'specified!',CR,LF,'$'
                ;
                ;
                ; ********** TIME/DATE/PGM HEADING **********
                ;
0368              TOPLINE:  DS      SKIP

036B              HOURS:    DS      2
036D 3A                     DB      ':'
036E              MINUTES:  DS      2
0370 3A                     DB      ':'
0371              SECONDS:  DS      2
0373 2020202020             DB      '              '
0380              DAY:      DS      10
038A 20                     DB      ' '
038B              DATE:     DS      2
038D 20                     DB      ' '
038E              MONTH:    DS      10
0398 3139                   DB      '19'
039A              YEAR:     DS      2
039C 2020202020             DB      '               '
03AA              HEADER:   DS      TSIZE
03B6 0D0A0A0A24             DB      CR,LF,LF,LF,'$'
                ;
                ;
                ; **********    LITERALS        **********
                ;
03BB 53756E6461 LITDAY:   DB      'Sunday,    '
03C5 4D6F6E6461           DB      'Monday,    '
03CF 5475657364           DB      'Tuesday,   '
03D9 5765646E65           DB      'Wednesday,'
03E3 5468757273           DB      'Thursday,  '
03ED 4672696461           DB      'Friday,    '
```

(Continued on page 90)

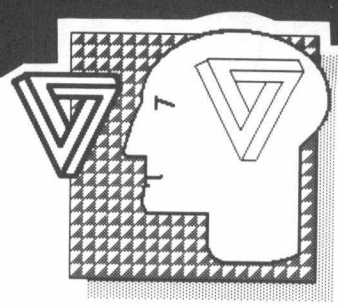
Real-Time Clock (Listing Continued, text begins on page 56)

Listing Three

```
03F7 5361747572              DB      'Saturday, '

0401 4A616E7561 LITMON:      DB      'January   '
040B 4665627275              DB      'February  '
0415 4D61726368              DB      'March     '
041F 417072696C              DB      'April     '
0429 4D61792020              DB      'May       '
0433 4A756E6520              DB      'June      '
043D 4A756C7920              DB      'July      '
0447 4175677573              DB      'August    '

0451 5365707465              DB      'September '
045B 4F63746F62              DB      'October   '
0465 4E6F76656D              DB      'November  '
046F 446563656D              DB      'December  '

                    ;
                    ; **********     VARIABLES      **********
                    ;

0479                OLDSTK:   DS    2               ; Stack pointer to get back
                                                    ;  to the CCP from this pgm.

047B                INBUFPT:  DS    2               ; Input buffer pointer

047D                COLCNT:   DS    1               ; Current length of line


047E                LINECNT:  DS    1               ; Keep count of lines printed
                                                    ;  on the page

                    ;
                    ; **********     STACK LOCATION     **********

047F                          DS      32            ; Reserve 16-level stack
                    NEWSTK:

049F                          END
```

End Listings

REALIZABLE FANTASIES



by Michael Swaine and Bob Albrecht

We announced this column of imagined and realizable projects in February in a piece called "Tiny Hackers" and actually launched it in March with Richard Stallman's "GNU Manifesto." But the point of writing about realizable fantasies is to encourage their realization, so this month we present progress reports on the fantasies presented to date.

Liberating the Mac

In January we published, though not in this column, Tom Lafleur's illustrated guide to fattening your Macintosh. Tom received over a hundred calls and letters shortly after the issue came out. By the time of the Macworld show, several people had set up business dedicated to Mac-fattening, at least some of which seem to have been spin-offs from Tom's article. Despite our warnings that doing it yourself voided your warranty, risked frying your Mac and was a tedious process, a lot of you opened your Macs and went at it. We think that one reason for the article's popularity is that a lot of you believe that Apple's decision to build the Mac as a closed system was a mistake, and one you're willing to rectify. ("Apple just ain't the company it once Woz."—Laran Stardrake.)

In April we followed up on this belief by reporting in this column on various ways in which people could be said to be liberating the Mac, including Lee Felsenstein's Hacker's Mac (aka Hackintosh), a project for learning about the Mac by redesigning it; and Jack Tramiel's Atari ST (aka Jackintosh), which, running DRI's GEM environment, may bring Maclike capabilities to the rest of us. As this goes to press, Atari has just previewed the ST in Germany, DRI has announced the first applications for GEM and Lee thinks that Tramiel may be doing some of the Hackintosh team's work for it, though he hasn't abandoned the project. Contact Lee at Golemics, 2600 Tenth Street, Berkeley CA 94710, (415) 486-8344.

Also in April, we described Steve Jasik's MacNosy disassembler for the Mac, but failed to give a full address. It's Free the ROM 64, 343 Trenton Way, Menlo Park CA 94025, and his telephone number is (415) 322-1386. Steve has fixed some bugs in Nosy and is now supplying a fast sort routine with the program. MacNosy's opening screen echos the opening lines of the cult classic television program The Prisoner:

Who are you?	I am number two.
Who is number one?	You are number six.
What do you want?	Information.

GNU Manifestations

As last month's letters indicated, reaction to March's Realizable Fantasy, a proposal by Richard Stallman to develop a free operating system that provides the capabilities of Unix, has been emotional. Stallman's mail has been overwhelmingly supportive; he's received many offers of help, and he called to offer us quarterly updates on the project's progress. We'll probably print the first of these next month. Stallman is reachable at 166 Prospect St., Cambridge MA 02139.

Tiny Hackers

Dragonsmoke, Bob Albrecht's newsletter, is the place to follow The Dragon's master plan to turn innocent children into programmers, and it seems that it will soon be carrying excerpts from Ron Jeffries' newsletter, the *Jeffries Report*. Dragonsmoke costs $12/year and you can get a sample issue by sending $1.00 or $.39 and an SASE to Dragonsmoke, P.O.Box 7627, Menlo Park CA 94026.

Check the June issue of *Rainbow* for a review of the CoCoMac, a low-budget way to get Maclike features. The same issue also contains information on CoCoMac RAM disk construction and a plan to get the Fujitsu dual-6809 machine distributed in the U.S. It's supposedly very powerful, even without the 68K you can drop in.

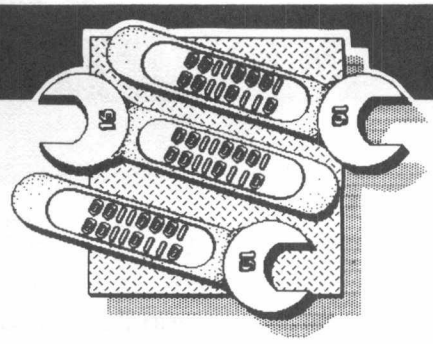
Subversives

In a guest essay in this space in May, Resident Intern Dave Cortesi gave his vision of this magazine, a realizable fantasy for *DDJ*: that *DDJ* is and should always be subversive. He said, "As a good revolutionary, I must believe that computer use is for everybody, and I really do. But programming the computers so they can be used is almost certainly a job for specialists. Whose specialists?" Not the establishment's, Cortesi hopes, but those who want to promulgate ideas to all who can grasp them, those who do not see information exchange as a zero-sum game, those who see themselves involved in a revolution, those with a mischievous, subversive desire "to snatch the tools of the establishment and apply them in the public domain." We never ignore Cortesi's advice, and we hope you won't either. You have, through your contributions, the power to steer this magazine away from the safe shores and keep it pushing out recklessly on journeys of discovery.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

MSDOS Installable Device Drivers

One of the more novel features added in Version 2 of MSDOS is the concept of "installable device drivers." This allows the user to attach new drivers for additional hardware devices or to supersede the system's existing built-in drivers by the simple expedient of putting the executable driver file on the boot disk and editing a line into the file CONFIG.SYS. This extremely powerful concept has made the lives of third-party mass storage device manufacturers much easier (previously they had to disassemble and patch the operating system to get their products to run).

But First . . . an Overview of Unix Device Drivers

Because the installable device drivers of MSDOS are patterned after Unix and much of the terminology used in the Microsoft driver documentation derives from Unix terminology, it is instructive to review the structure of the real McCoy. Much of the information presented here was gleaned from the article "Writing Device Drivers for Xenix Systems" by Jean McNamara, et al. (UniForum Conference Proceedings, January 1984).

Unix knows two types of devices: block and character. A block device is typically a mass storage medium such as a fixed or removable disk or magnetic tape drive. Such devices usually transfer data in chunks of fixed size, which are related (in the case of disks) to the characteristics and "format" of the physical media.

In contrast, a character device is a sequential device such as a CRT terminal or paper tape reader that supplies or accepts a stream of bytes. Although ideally the data stream of a character device has no inherent structure, in practice it usually is delimited by special control characters such as carriage returns, which the operating system also recognizes.

Under Unix, all drivers (whether controlling a block or character device) have the same general structure and contain two major parts: task time routines and an interrupt handler.

The task time routines are called at the time of the application program's (the task's) request for I/O. The Unix kernel transforms the request from its high-level, logical, device-independent character into addresses and parameters that are relevant to the physical device. When the driver's task time routines are executing, the application itself may be thought of as being in control of the system and active, although it is running in privileged or kernel mode.

The interrupt handler is entered asynchronously when the corresponding device generates a hardware interrupt. Interrupts are issued when an I/O operation has either finished or been terminated due to a hardware fault. If the system or the device does not support hardware interrupts, the interrupt handler is entered when the kernel, polling the device periodically, determines that it is no longer busy. Typically, the task requesting I/O from the device is inactive when the interrupt handler receives control; indeed, the task is usually in a suspended state pending I/O completion, and some other application is active and in control as far as the operating system is concerned.

A Unix block device driver comprises some tables and five major routines:

- *Init* is called once when the system is booted and the driver is first loaded into memory. It checks for the existence of the physical device and initializes it properly for future I/O, if present.
- *Open* is called by the Unix kernel when a user tries to access a file on the device. This routine should complete any initialization not performed by *Init* to prepare for I/O; it may be part of a mount sequence for removable media.
- *Close* is called by the Unix kernel to reset any flags or variables set by *Open* and to ensure that pending operations are completed. A typical action would be to flush buffered data to the physical device. This routine may be part of a dismount sequence for removable media.
- The *Strat* routine, an abbreviation for "strategy," is called by the Unix kernel when the application issues an I/O request for the block device. *Strat* is called with the address of a buffer header or parameter list, which defines the device unit number, request type (read, write, or format), where to find the data (logical block number), how much data it should transfer, and the memory address of its buffer. *Strat*'s responsibility is to validate the request then place it on the request queue for that device. Finally, it calls the internal routine *Start* (see below).
- *Intr* is called by the Unix kernel when the block device issues an interrupt. It is responsible for determining that the interrupt signal is valid (Was an operation on this device really in progress and are the appropriate completion flags set on the controller?). It determines success or failure status for the last operation by interrogating the controller and passes the appropriate flags back to the kernel. Finally, if additional requests are pending, it calls *Start* to try to initiate another I/O operation.

Start is an internal driver routine called by *Strat* or *Intr*; it removes the highest priority request from the request queue and initiates the I/O. If the device is busy or if no requests are waiting, *Start* simply exits.

The body of a Unix character device driver is similar to that of a block device driver. It is made up of seven major routines:

- *Init*, *Open*, and *Close* are similar in function and intent to those of the same name in a block device driver.
- *Read* transfers data from the device driver's input buffer to the application's buffer.
- *Write* transfers data from the application's buffers to the device driver's output buffers; it also starts output to the device if it is idle. If the output buffers are full, *Write* suspends the requesting process until they have emptied.
- *Ioctl* provides direct communication between the application and the driver. It allows the application to ask for device-specific information or to set the values of the driver's internal flags and variables (such as baud rate or number of stop bits for a serial I/O port).
- *Intr*, the asynchronous part of the driver, is called by the kernel when the device issues a hardware interrupt. This procedure performs the actual data transfer between the physical device's hardware controller and the driver's input/output buffers.

MSDOS Device Drivers

MSDOS installable device drivers bear a strong resemblance to Unix device drivers, both structurally and functionally. Like Unix, they fall into two major classes.

Character device drivers control devices that perform I/O one byte at a time, similar to a traditional TTY terminal. MSDOS has built-in drivers for the console device, serial port, and list device, named CON, AUX, and PRN, respectively. You can access these drivers using the traditional CP/M-like character I/O calls, or you can open them by name, like a file for input and output, using the new "handle" function calls of MSDOS 2.0 and above. A character device driver can support one hardware unit.

Block device drivers control random access storage devices such as flexible disk drives or fixed disks. A block device driver can support more than one hardware unit and/or may map a single physical unit onto two or more logical drives. Each device unit for a given block driver is assigned a drive designator (such as A:, B:, etc.); the first drive letter for a given block device driver is determined by that driver's position in the overall chain of drivers.

Once a driver is written and assembled, you may load and link it into the operating system simply by placing an entry of the form

```
device=filename.ext
```

in the CONFIG.SYS file on the system boot disk. You can override the default system driver for a character device with an installed driver simply by giving it the same logical device name in the device header. When processing an I/O request, DOS always scans the list of installed drivers before the default devices and takes the first match.

Structure of an MSDOS Device Driver

MSDOS device drivers always have an ORIGIN of zero but are otherwise assembled, linked, and converted into an executable module as though they were COM or EXE files. A device driver consists of three major parts.

A Device Header (Table 1, below) contains the linkage to the next driver in the chain, a set of attribute flags (Table 2, below) for the device, offsets to the executable strategy and interrupt routines for the device, and the logical device name (if it is a character device such as PRN or COM1). The linkage address of the last driver in a file is

Offset	Contents
0	Offset, pointer to next device header
2	Segment, pointer to next device header
4	Device attribute word (see Table 2)
6	Pointer to device strategy code (offset)
8	Pointer to device interrupt code (offset)
10	Logical name (8 bytes) if character device or number of units (1 byte) if block device followed by 7 bytes that may contain a name or nothing at all

Table 1
Device driver header.

Bit	Meaning
15	=1 if character device =0 if block device
14	=1 if IOCTL is supported
13	=1 if non-IBM format (block only)
12	#
11	=1 if open/close/RM is supported*
10	#
9	#
8	#
7	#
6	#
5	#
4	#
3	=1 if current clock device
2	=1 if current NUL device
1	=1 if current standard output device
0	=1 if current standard input device

Currently undefined and should be 0
* DOS 3.0 and above only; should be 0 for DOS 2.x

Table 2
Device attribute word, found in device driver header; only bits 11, 13, and 14 have significance on block devices.

always set to −1 when the driver is created; the drivers are chained together, and the linkage fields are updated by MSDOS at system initialization.

The Strategy Routine for the device is entered by DOS via a Far Call when the driver is first loaded and installed and whenever an application program issues an I/O request for the device. DOS uses ES:BX to pass the Strategy routine a double-word pointer to a data structure, which is called a Request Header; this structure contains information about the type of operation to be performed. According to MSDOS conventions, the Strategy code never actually performs an I/O operation but simply saves the pointer to the Request Header.

The last and most complex part of a device driver is the misnamed Interrupt Routine. This code implements the device driver proper; it performs the actual operation based on the function code and other information passed in the Request Header. Status and completion information may be passed back to DOS in the same Header.

When an I/O request is issued, the Interrupt Routine entry point is called by DOS immediately after the call to the Strategy Routine. Unlike in Unix, the Interrupt Routine is never entered asynchronously (as is the case on an I/O completion interrupt). The division of function between the Strategy and Interrupt Routines is completely artificial, at least under the currently available nonmultitasking versions of MSDOS.

```
;
; MS-DOS Request Header structure definition
;
Request   struc          ; request header template
                           structure
Rlength   db  ?          ; 0  length of request header
Unit      db  ?          ; 1  unit number for this request
Command   db  ?          ; 2  request header's command
                           code
Status    dw  ?          ; 3  driver's return status word
Reserve   db  8 dup (?)  ; 5  reserved area
Media     db  ?          ; 13 media descriptor byte
Address   dd  ?          ; 14 memory address for transfer
Count     dw  ?          ; 18 byte/sector count value
Sector    dw  ?          ; 20 starting sector value
Request   ends           ; end of request header template
```

Table 3
Format of request header. Only the first 13 bytes are common to all driver functions; the number and definition of the following bytes vary depending on the function type. The structure shown here is the one used by the read and write subfunctions of the driver.

Bit(s)	Significance
15	Error
10–14	Reserved
9	Busy
8	Done
0–7	Error code if bit 15 = 1

Table 4
Return status word of request header.

The Request Header

The MSDOS BDOS uses a data structure called the Request Header to give the driver the necessary information to perform an I/O operation. The address of the Request Header is passed to the Strategy Routine and saved in a local variable; the Interrupt Routine, which is called immediately afterward, uses this address to access information from the Header.

The first 13 bytes of the Request Header are the same for all device driver functions and therefore are referred to as the "static" portion of the Header. The number and contents of the following bytes vary according to the type of function requested (Table 3, at left). The Request Header's primary components are a Command Code that selects a driver subfunction (such as Read, Write, or Status) and a Return Status word that informs the BDOS about the driver's success with the request I/O operation (Table 4, below left, and Table 5, page 98). Other information passed in the Header to the driver includes minor unit numbers, transfer addresses, sector or byte counts, and so on.

When an I/O function is completed, the device driver uses fields in the Request Header to pass status, sector or byte counts, and other information back to the operating system.

The Driver Command Code Routines

The Command Code for the requested driver subfunction is passed in the third byte of the Request Header. The Interrupt Routine extracts it from the Request Header using the double-word pointer saved during the call to the Strategy Routine. Typically, the Command Code is used as an index into a jump table that points to the proper service code.

In the descriptions below, RH refers to the Request Header whose address was passed to the Strategy Routine in ES:BX. DWORD refers to a long address, of which the first two bytes contain the offset and the last two bytes contain the segment.

Function 0—Driver Initialization

This initialization code for the driver is called only once, when the driver is loaded. It is responsible for performing any necessary hardware initialization of the device, setup of interrupt vectors, and so on. It returns the address of the start of free memory after the driver, so that DOS knows where it can load the next installable driver. If it is initializing a block device driver, it must also return the number of units and the address of the BIOS parameter block (BPB) pointer array; if all units are the same, all pointers in the array can point to the same BPB.

If the initialization routine finds that the device is missing or defective and wants to abort without using memory, it should set the number of units to zero and set the ending address to CS:0000.

The operating system services that the initialization code can invoke at load time are limited; the code can call only MSDOS services 01–0CH and 30H. This is adequate to check the DOS version number and display a driver identification message, but not much else.

Many programmers position the initialization code at the end of the driver and return its address as the location of the first free memory; this allows the memory occupied by the code to be reclaimed after it is finished with its work.

This routine is called with:

RH+18	DWORD	Pointer to the character after the = on the CONFIG.SYS line that loaded driver; this information is read only
RH+22	BYTE	Drive letter for first unit of a block driver: 0=A, 1=B, etc. (MSDOS 3.x only)

This routine returns:

RH+3	WORD	Return status
RH+13	BYTE	Number of units (block devices only)
RH+14	DWORD	Address of first free memory above driver
RH+18	DWORD	BPB pointer array (block devices only)

Code	Error Definition
0	Write protect violation
1	Unknown unit
2	Drive not ready
3	Unknown command
4	CRC error
5	Bad drive request structure length
6	Seek error
7	Unknown media
8	Sector not found
9	Printer out of paper
10	Write fault
11	Read fault
12	General failure
13–14	Reserved
15	Invalid disk change (MSDOS 3.x)

Table 5
Driver error codes returned in bits 0–7 of return status word in request header.

Byte(s)	Contents
0–1	Bytes per sector
2	Sectors per allocation unit (must be power of 2)
3–4	Number of reserved sectors (starting at sector 0)
5	Number of file allocation tables (FATs)
6–7	Maximum number of root directory entries
8–9	Total number of sectors in media
10	Media descriptor byte
11–12	Number of sectors occupied by a single FAT

Table 6
Format of BIOS parameter block; a copy of this block is always found in the boot sector of an initialized disk.

Function 1—Media Check

The media check function is used on block devices only and should be a NOP in character device drivers. This routine is called first by BDOS for a block device transfer, passing the current media descriptor byte (Table 7, page 99). If feasible, the routine returns a code indicating whether the media has changed since the last transfer. This feature requires a machine that provides a door interlock hardware status flag or something similar. Determining the media change status improves performance because MSDOS does not need to reread the file allocation table (FAT) for each directory access.

This routine is called with:

RH+1	BYTE	Unit code
RH+13	BYTE	Media descriptor byte

This routine returns:

RH+3	WORD	Return status
RH+14	BYTE	Media change code: –1 Media has changed 0 Don't know if media changed 1 Media has not changed
RH+15	DWORD	Pointer to previous volume ID, if device attribute bit 11 = 1 and media has changed (MSDOS 3.x only)

Function 2—Build BIOS Parameter Block

The build BPB function is supported on block devices only and should be a NOP for character devices. The BDOS uses it to get a pointer to the valid BPB (Table 6, at left) for the current media. This routine is called when the media check routine returns a "Media has changed" code or when it returns a "Don't know if media changed" code and there are no dirty buffers (buffers with changed data that have not yet been written to disk). Thus this call indicates whether the media has legally changed. Under MSDOS 3.x, this function should also read the volume ID off the disk and save it.

The build BPB call also receives a pointer to a one-sector buffer in the address field of the request header. If the "non-IBM-format" bit in the device attribute word is

zero, the buffer contains the first sector of the FAT, including the media descriptor byte, and should not be altered by the driver. If the "non-IBM-format" bit is set, the buffer may be used as scratch space.

This routine is called with:

RH+1	BYTE	Unit code
RH+13	BYTE	Media descriptor byte
RH+14	DWORD	Buffer address

This routine returns:

RH+3	WORD	Return status
RH+18	DWORD	Pointer to new BPB

Function 3—I/O Control Read

This function allows control information to be passed directly from the application program to the device driver. It is called only if the IOCTL bit is set in the device header attribute word. No error check is performed on IOCTL I/O calls.

This routine is called with:

RH+1	BYTE	Unit code (block devices only)
RH+13	BYTE	Media descriptor byte
RH+14	DWORD	Transfer address
RH+18	WORD	Byte/Sector count
RH+20	WORD	Starting sector number (block devices only)

This routine returns:

RH+3	WORD	Return status
RH+18	WORD	Actual bytes or sectors transferred

Function 4—Read

This function transfers data from the device into the specified memory buffer. Under MSDOS 3.x, the routine can use the reference count of open files maintained by the open and close routines (functions 13 and 14) and the media descriptor byte to determine whether the media has changed illegally.

This routine is called with:

RH+1	BYTE	Unit code (block devices only)
RH+13	BYTE	Media descriptor byte
RH+14	DWORD	Transfer address
RH+18	WORD	Byte/Sector count
RH+20	WORD	Starting sector number (block devices only)

This routine returns:

RH+3	WORD	Return status
RH+18	WORD	Actual bytes or sectors transferred
RH+22	DWORD	Pointer to volume ID if error 0FH is returned (MSDOS 3.x only)

Function 5—Nondestructive Read

This function is available on character devices only. If an input status request returns a busy bit = 0 (characters waiting), the next character that would be read is returned to DOS but stays in the input buffer. This basically provides DOS with the capability to look ahead by one character.

This routine returns:

RH+3	WORD	Return status
RH+13	BYTE	Character

Function 6—Input Status

This function is available on character devices only and returns the current input status for the device. MSDOS

Media Descriptor Byte

Bit(s)	Significance
3–7	Always set (=1)
2	1=removable 0=not removable
1	1=8 sector 0=not 8 sector
0	1=2 sided 0=not 2 sided

Current Valid MSDOS Descriptor Bytes (5¼-inch Disks)

0F9H	2 sided, 15 sector
0FCH	1 sided, 9 sector
0FDH	2 sided, 9 sector
0FEH	1 sided, 8 sector
0FFH	2 sided, 8 sector
0F8H	(fixed disk)

Table 7
Media descriptor byte of request header, assuming "non-IBM-format" bit in attribute word of device header is zero.

Addr	Attr	Str	Int	Type	Units	Name
00E3:0111	8004	0FD5	0FE0	C		NUL
0070:0148	8013	008E	0099	C		CON
0070:01DD	8000	008E	009F	C		AUX
0070:028E	8000	008E	00AE	C		PRN
0070:0300	8008	008E	00C3	C		CLOCK
0070:03CC	0000	008E	00C9	B	02	
0070:01EF	8000	008E	009F	C		COM1
0070:02A0	8000	008E	00AE	C		LPT1
0070:06F0	8000	008E	00B4	C		LPT2
0070:0702	8000	008E	00BA	C		LPT3
0070:0714	8000	008E	00A5	C		COM2

End of device chain.

Table 8
Example listing of device chain under MSDOS 2.1: "plain vanilla" PC with no hard disks or user device drivers.

assumes all character devices have a type-ahead buffer. If the device does not have a type-ahead buffer, it should always return a busy bit = 0 so MSDOS will not hang.

This routine returns:

RH+3	WORD	Return status Busy bit = 1 Read request goes to physical device Busy bit = 0 Characters already in device buffer; read request returns quickly

Function 7—Flush Input Buffers

This function is available on character devices only. It terminates all pending requests; i.e., the input buffer is emptied.

This routine returns:

RH+3	WORD	Return status

Function 8—Write

This routine transfers data from the specified memory buffer to the device. Under MSDOS 3.x, the routine can use the reference count of open files maintained by the open and close routines (functions 13 and 14) and the media descriptor byte to determine whether the media has changed illegally.

This routine is called with:

RH+1	BYTE	Unit code (block devices only)
RH+13	BYTE	Media descriptor byte
RH+14	DWORD	Transfer address
RH+18	WORD	Byte/Sector count
RH+20	WORD	Starting sector number (block devices only)

This routine returns:

RH+3	WORD	Return status
RH+18	WORD	Actual bytes or sectors transferred
RH+22	DWORD	Pointer to volume ID if error 0FH is returned (MSDOS 3.x only)

Function 9—Write with Verify

This routine transfers data from the specified memory buffer to the device. If feasible, a read-after-write verification of the data is performed to confirm that the data was written correctly. Otherwise, it is exactly like function 8.

Function 10—Output Status

This routine returns the current output status for the device. This function is available on character devices only.

This routine returns:

RH+3	WORD	Return status Busy bit = 1 Write request waits for completion of current request Busy bit = 0 Device idle; write request starts immediately

Function 11—Flush Output Buffers

This function is available on character devices only and terminates all pending output requests. The output buffer, if any, is emptied.

This routine returns:

RH+3	WORD	Return status

Function 12—I/O Control Write

This function allows control information to be passed directly from the driver to the application program. It is called only if the IOCTL bit is set in the device header attribute word. No error check is performed on IOCTL I/O calls.

This routine is called with:

RH+1	BYTE	Unit code (block devices only)
RH+13	BYTE	Media descriptor byte
RH+14	DWORD	Transfer address
RH+18	WORD	Byte/Sector count
RH+20	WORD	Starting sector number (block devices only)

This routine returns:

RH+3	WORD	Return status
RH+18	WORD	Actual bytes or sectors transferred

Function 13—Open

This function is available on MSDOS version 3.0 and above only and is called only if the open/close/RM bit is set in the device attribute word.

Block devices may use the open function to manage local buffering and to increment a reference count of the number of open files on a device.

Character devices can use this function to send a device initialization string, which in turn can be set by IOCTL Write. Note that the predefined CON, AUX, and PRN devices are always open.

This routine is called with:

RH+1	BYTE	Unit code (block devices only)

This routine returns:

RH+3	WORD	Return status

Function 14—Device Close

This function is available on MSDOS version 3.0 and above only and is called only if the open/close/RM bit is set in the device attribute word.

On block devices, this function can manage local buffering and decrement a reference count keeping track of the number of open files on the device; when the count reaches zero, all files have been closed, and the driver should flush buffers because the user may change disks.

On character devices, this function can send a device-dependent post-I/O string such as a form feed, which in turn can be set by an IOCTL Write. Note that the CON, AUX, and PRN devices are never closed.

This routine is called with:

RH+1 BYTE Unit code (block devices only)

This routine returns:

RH+3 WORD Return status

Function 15—Removable Media

This function is available on MSDOS version 3.0 and above only and is called only if the open/close/RM bit is set in the device attribute word and the device is a block type.

This routine is called with:

RH+1 BYTE Unit code (block devices only)

This routine returns:

RH+3 WORD Return status
Busy bit = 1 Media is non-removable
Busy bit = 0 Media is removable

DDJ

16-Bit Toolbox Listing

(Text begins on page 94)

```
1                                       name    driver
2                                       page    55,132
3                                       title   'DRIVER --- installable driver template'
4
5                                     ;
6                                     ; This is a "template" for a MS-DOS installable device driver.
7                                     ; The actual driver subroutines are stubs only and have
8                                     ; no effect but to return a non-error "done" status.
9                                     ;
10                                    ; Ray Duncan, April 1985
11                                    ; Laboratory Microsystems Inc.
12
13    0000                            code    segment public 'CODE'
14
15    0000                            driver  proc    far
16
17                                            assume  cs:code,ds:code,es:code
18
19    0000                                    org     0
20
21    = 000F                          Max_Cmd equ     15              ; driver command code maximum
22                                                                    ; 12 for MS-DOS 2.x,
23                                                                    ; 15 for MS-DOS 3.x
24
25    = 000D                          cr      equ     0dh             ; ASCII carriage return
26    = 000A                          lf      equ     0ah             ; ASCII line feed
27    = 0024                          eom     equ     '$'             ; end of message signal
28
29                                            page
30                                    ;
31                                    ; Device Driver Header
32                                    ;
33    0000  FF FF FF FF               Header  dd      -1              ;link to next device,-1= end of list
34
35    0004  8000                              dw      8000h           ;attribute word
36                                                                    ;bit 15=1 for character devices
37
38    0006  0056 R                            dw      Strat           ;device "Strategy" entry point
39
40    0008  0061 R                            dw      Intr            ;device "Intrrupt" entry point
41
42    000A  44 52 49 56 45 52               db      'DRIVER  '      ;char device name, 8 char, or
43          20 20
44                                                                    ;if block device, no. of units
45                                                                    ;in first byte followed by
46                                                                    ;7 don't care bytes
47
48                                    ;
49                                    ; local variables for use by driver
50                                    ;
51    0012  ????????                  RH_Ptr  dd      ?               ; pointer to request header
52                                                                    ; passed to Strat by BDOS
53
54    0016  0D 0A 0A                  Ident   db      cr,lf,lf
```

```
0019  45 78 61 6D 70 6C                db      'Example Device Driver 1.0'
      65 20 44 65 76 69
      63 65 20 44 72 69
      76 65 72 20 31 2E
      30
0032  0D 0A 0A 24                       db      cr,lf,lf,eom

                                        page

                                ;
                                ; Driver Command Codes dispatch table
                                ;
                                ; The "Intr" routine uses this table and the Command Code
                                ; supplied in the Request Header to transfer to the
                                ; appropriate driver subroutine.

0036                            Dispatch:

0036  00BA R                            dw      Init            ;  0 = init driver into system
0038  009C R                            dw      Media_Chk       ;  1 = media check on blk dev
003A  009E R                            dw      Build_Bpb       ;  2 = build BIOS param block
003C  00A0 R                            dw      Ioctl_Inp       ;  3 = I/O ctrl read from dev
003E  00A2 R                            dw      Input           ;  4 = normal destructive read
0040  00A4 R                            dw      Nd_Input        ;  5 = non-destructive read,no wait
0042  00A6 R                            dw      Inp_Stat        ;  6 = return current input status
0044  00A8 R                            dw      Inp_Flush       ;  7 = flush device input buffers
0046  00AA R                            dw      Output          ;  8 = normal output to device
0048  00AC R                            dw      Outp_Vfy        ;  9 = output with verify
004A  00AE R                            dw      Outp_Stat       ; 10 = return current output status
004C  00B0 R                            dw      Outp_Flush      ; 11 = flush output buffers
004E  00B2 R                            dw      Ioctl_Outp      ; 12 = I/O control output
0050  00B4 R                            dw      Dev_Open        ; 13 = device open      (MS-DOS 3.x)
0052  00B6 R                            dw      Dev_Close       ; 14 = device close     (MS-DOS 3.x)
0054  00B8 R                            dw      Rem_Media       ; 15 = removeable media (MS-DOS 3.x)

                                        page
                                ;
                                ; MS-DOS Request Header structure definition
                                ;
                                ; The first 13 bytes of the Request Header are the same
                                ; for all Command codes and are termed the "Static" part of
                                ; the Header.  The number and meaning of the following bytes
                                ; vary depending on the Command code.
                                ;
                                ; The Request Header shown here applies to Read & Write functions.
                                ;

                                Request struc                   ; request header template structure

0000  ??                        Rlength db      ?               ; length of request header
0001  ??                        Unit    db      ?               ; unit number for this request
0002  ??                        Command db      ?               ; request header's command code
0003  ????                      Status  dw      ?               ; driver's return status word
0005     08 [                   Reserve db      8 dup (?)       ; reserved area
                  ??
                     ]

000D  ??                        Media   db      ?               ; media descriptor byte
000E  ????????                  Address dd      ?               ; memory address for transfer
0012  ????                      Count   dw      ?               ; byte/sector count value
0014  ????                      Sector  dw      ?               ; starting sector value

0016                            Request ends                    ; end of request header template

                                        page

                                ; Device Driver "Strategy Routine"

                                ; Each time a request is made for this device, the BDOS
                                ; first calls "Strategy routine",  then immediately calls
                                ; the "Interrupt routine".

                                ; The Strategy routine is passed the address of the
                                ; Request Header in ES:BX, which it saves in a local
                                ; variable and then returns to BDOS.

0056                            Strat   proc    far
                                                                ; save address of Request Header
```

```
0056  2E: 89 1E 0012 R                mov     word ptr cs:[RH_Ptr],bx
005B  2E: 8C 06 0014 R                mov     word ptr cs:[RH_Ptr+2],es

0060  CB                              ret                     ; back to BDOS

0061                            Strat endp

                                      page


                                ; Device Driver "Interrupt Routine"

                                ; This entry point is called by the BDOS immediately after
                                ; the call to the "Strategy Routine", which saved the long
                                ; address of the Request Header in the local variable 'RH_Ptr".

                                ; The "Interrupt Routine" uses the Command Code passed in
                                ; the Request Header to transfer to the appropriate device
                                ; handling routine.  Each command code routine is responsible
                                ; for any necessary return information into the Request Header,
                                ; then transfers to Error or Exit to set the Return Status code.

0061                            Intr    proc  far

0061  50                              push    ax              ; save general registers
0062  53                              push    bx
0063  51                              push    cx
0064  52                              push    dx
0065  1E                              push    ds
0066  06                              push    es
0067  57                              push    di
0068  56                              push    si
0069  55                              push    bp

006A  0E                              push    cs              ; make local data addressable
006B  1F                              pop     ds

006C  C4 3E 0012 R                    les     di,[RH_Ptr]     ; ES:DI = Request Header

                                                              ; get BX = Command Code
0070  26: 8A 5D 02                    mov     bl,es:[di.Command]
0074  32 FF                           xor     bh,bh
0076  83 FB 0F                        cmp     bx,Max_Cmd      ; make sure its legal
0079  7F 06                           jg      Unk_Command     ; too big, exit with error code
007B  D1 E3                           shl     bx,1            ; form index to Dispatch table
                                                              ; and branch to driver routine
007D  FF A7 0036 R                    jmp     word ptr [bx+Dispatch]

                                      page


                                ; General collection of exit points for the driver routines.


0081                            Unk_Command:                  ; Come here if Command Code too big.
0081  B0 03                           mov     al,3            ; Sets "Unknown Command" error
                                                              ; code and "Done" bit.

0083                            Error:                        ; Transfer here with AL = error code.
0083  B4 81                           mov     ah,81h          ; Sets "Error" and "Done" bits.
0085  EB 03 90                        jmp     Exit

0088  B4 01                     Done: mov     ah,1            ; Come here if I/O complete and
                                                              ; no error, sets "Done" bit only.


008A                            Exit:                         ; General purpose exit point.
                                                              ; Transfer here with AX =
                                                              ; Return Status word to be
                                                              ; placed into Request Header.

008A  2E: C5 1E 0012 R                lds     bx,cs:[RH_Ptr]          ; set status
008F  89 47 03                        mov     ds:[bx.Status],ax

0092  5D                              pop     bp              ;restore general registers
```


568

```
0093  5E                        pop     si
0094  5F                        pop     di
0095  07                        pop     es
0096  1F                        pop     ds
0097  5A                        pop     dx
0098  59                        pop     cx
0099  5B                        pop     bx
009A  58                        pop     ax
009B  CB                        ret                     ; back to BDOS

                                page

                        ;
                        ; Function 1  Media Check
                        ;
009C                    Media_Chk:
009C  EB EA                     jmp     Done

                        ;
                        ; Function 2  Build BIOS Parameter Block
                        ;
009E                    Build_Bpb:
009E  EB E8                     jmp     Done

                        ;
                        ; Function 3  I/O Control Read
                        ;
00A0                    Ioctl_Inp:
00A0  EB E6                     jmp     Done

                        ;
                        ; Function 4  Read from Device
                        ;
00A2                    Input:
00A2  EB E4                     jmp     Done

                        ;
                        ; Function 5  Non-destructive Read
                        ;
00A4                    Nd_Input:
00A4  EB E2                     jmp     done

                        ;
                        ; Function 6  Return Input Status
                        ;
00A6                    Inp_Stat:
00A6  EB E0                     jmp     Done

                        ;
                        ; Function 7  Flush Input Buffers
                        ;
00A8                    Inp_Flush:
00A8  EB DE                     jmp     Done

                        ;
                        ; Function 8  Write to Device
                        ;
00AA                    Output:
00AA  EB DC                     jmp     Done

                        ;
                        ; Function 9  Write with Verify
                        ;
00AC                    Outp_Vfy:
00AC  EB DA                     jmp     Done

                        ;
                        ; Function 10  Return Output Status
                        ;
00AE                    Outp_Stat:
00AE  EB D8                     jmp     Done

                        ;
                        ; Function 11  Flush Output Buffers
                        ;
00B0                    Outp_Flush:
00B0  EB D6                     jmp     Done

                        ;
```

 (Listing continued, text begins on page 94)

```
                                        ; Function 12  I/O Control Write
                                        ;
00B2                                    Ioctl_Outp:
00B2  EB D4                                     jmp     Done

                                        ;
                                        ; Function 13  Device Open (MS-DOS 3.x)
                                        ;
00B4                                    Dev_Open:
00B4  EB D2                                     jmp     Done

                                        ;
                                        ; Function 14  Device Close (MS-DOS 3.x)
                                        ;
00B6                                    Dev_Close:
00B6  EB D0                                     jmp     Done

                                        ;
                                        ; Function 15  Removeable Media (MS-DOS 3.x)
                                        ;
00B8                                    Rem_Media:
00B8  EB CE                                     jmp     Done

                                                page

                                        ; The Initialization code for the driver is called only once
                                        ; when the driver is loaded.  In this example, it returns its
                                        ; own address to the DOS as the start of free memory after the
                                        ; driver, so that the memory occupied by INIT will be reclaimed
                                        ; after it is finished with its work.  Only MS-DOS services 01-0CH
                                        ; and 30H can be called by the INIT code.
                                        ;
                                        ; Block device drivers must also return the number of units and
                                        ; the address of the BIOS Parameter Block pointer array; if all
                                        ; units are the same, all pointers can point to the same BPB.
                                        ;

00BA                                    Init:                           ; Function 0
                                                                        ; initialize device driver
00BA  06                                        push    es              ; push Request Header addr
00BB  57                                        push    di
00BC  B4 09                                     mov     ah,9            ; print sign-on message
00BE  BA 0016 R                                 mov     dx,offset Ident
00C1  CD 21                                     int     21h
00C3  5F                                        pop     di              ; restore Request Header addr
00C4  07                                        pop     es
                                                                        ; set first usable memory addr.
00C5  26: C7 45 0E 00BA R                       mov     word ptr es:[di.Address],offset Init
00CB  26: 8C 4D 10                              mov     word ptr es:[di.Address+2],cs
00CF  EB B7                                     jmp     Done


00D1                                    Intr    endp

00D1                                    Driver  endp

00D1                                    code    ends

                                                end
```

Structures and records:

```
                N a m e                 Width   # fields
                                        Shift   Width   Mask    Initial

REQUEST. . . . . . . . . . . .          0016    0009
  RLENGTH. . . . . . . . . . .          0000
  UNIT . . . . . . . . . . . .          0001
  COMMAND. . . . . . . . . . .          0002
  STATUS . . . . . . . . . . .          0003
  RESERVE. . . . . . . . . . .          0005
  MEDIA. . . . . . . . . . . .          000D
  ADDRESS. . . . . . . . . . .          000E
  COUNT. . . . . . . . . . . .          0012
  SECTOR . . . . . . . . . . .          0014
```

```
Segments and Groups:

                N a m e                 Size    Align   Combine Class

CODE . . . . . .                        00D1    PARA    PUBLIC  'CODE'

Symbols:

                N a m e                 Type    Value   Attr

BUILD_BPB. . . . . .                    L NEAR  009E    CODE
CR . . . . . .                          Number  000D
DEV_CLOSE. . . . . .                    L NEAR  00B6    CODE
DEV_OPEN . . . . . .                    L NEAR  00B4    CODE
DISPATCH . . . . . .                    L NEAR  0036    CODE
DONE . . . . . .                        L NEAR  0088    CODE
DRIVER . . . . . .                      F PROC  0000    CODE    Length =00D1
EOM. . . . . .                          Number  0024
ERROR. . . . . .                        L NEAR  0083    CODE
EXIT . . . . . .                        L NEAR  008A    CODE
HEADER . . . . . .                      L DWORD 0000    CODE
IDENT. . . . . .                        L BYTE  0016    CODE
INIT . . . . . .                        L NEAR  00BA    CODE
INPUT. . . . . .                        L NEAR  00A2    CODE
INP_FLUSH. . . . . .                    L NEAR  00A8    CODE
INP_STAT . . . . . .                    L NEAR  00A6    CODE
INTR . . . . . .                        F PROC  0061    CODE    Length =0070
IOCTL_INP. . . . . .                    L NEAR  00A0    CODE
IOCTL_OUTP . . . . . .                  L NEAR  00B2    CODE
LF . . . . . .                          Number  000A
MAX_CMD. . . . . .                      Number  000F
MEDIA_CHK. . . . . .                    L NEAR  009C    CODE
ND_INPUT . . . . . .                    L NEAR  00A4    CODE
OUTPUT . . . . . .                      L NEAR  00AA    CODE
OUTP_FLUSH . . . . . .                  L NEAR  00B0    CODE
OUTP_STAT. . . . . .                    L NEAR  00AE    CODE
OUTP_VFY . . . . . .                    L NEAR  00AC    CODE
REM_MEDIA. . . . . .                    L NEAR  00B8    CODE
RH_PTR . . . . . .                      L DWORD 0012    CODE
STRAT. . . . . .                        F PROC  0056    CODE    Length =000B
UNK_COMMAND. . . . . .                  L NEAR  0081    CODE

49190 Bytes free

Warning Severe
Errors  Errors
0       0
```

End Listing

COMPUTER CALISTHENICS

by Michael Wiesenberg

I've received a lot of response to the first two "Computer Calisthenics" columns.

The first puzzle in the November 1984 issue was:

There are two world-famed mathematicians, Mr. P and Mr. S. A friend of theirs who likes puzzles tells both of them that he has two numbers in mind. The only thing he will tell both of them about these two numbers is that they are both integers, greater than 1, and they are not the same.

He then whispers to Mr. P the product of the two numbers.

He whispers to Mr. S the sum of the two numbers.

Mr. P knows that Mr. S knows the sum, but he doesn't know what that sum is. Similarly, Mr. S knows that Mr. P knows the product, but he doesn't know what that product is.

The following conversation then takes place:

Mr. P: "I don't know the numbers."

Mr. S: "I know that; I also don't know the numbers."

Mr. P: "Oh, then I know the numbers."

Mr. S: "Really, then I do also."

Your task, of course, is to discover the two numbers. It would take you a long time to figure them out by hand, so write a program to do it. A short *program.*

Many offered programmatic solutions that produced the correct answer, but, unfortunately, they also yielded other sets of answers that were wrong.

Let's see what's happening at each statement. I'm going to give Alan Tracht, of Cleveland Heights, OH, an honorable mention for coming up with the correct reasoning required to solve the puzzle programmatically, but he doesn't get the t-shirt because his program is one of those that yields too many answers:

(1) Mr. P: "I don't know the numbers." They are not both prime.

(2) Mr. S: "I know that . . . ." The numbers cannot be the sum of two primes.

(3) ". . . I also don't know the numbers." The sum is at least 7, so that it can be the sum of at least two different pairs of numbers.

(4) Mr. P: "Oh, then I know the numbers." Of all the factor-pairs of the product, only one sums to a number that cannot be the sum of two primes.

(5) Mr. S: "Really, then I do also." Of all the addend-pairs, only one multiplies to a product that gives Mr. P the answer (under the conditions of the previous statement).

Let's see why certain sets of numbers fail one or more of the tests.

(5, 7): P=35, S=12

Mr. P would not have made statement 1.

(5, 6): P=30, S=11

Knowing the product, Mr. P would have made his first statement. Knowing the sum, Mr. S would have made statements 2 and 3. At this point, however, Mr. P still would not know whether his product of 30 was formed from the pair (5, 6) or (2, 15), and so would not have made statement 4.

(4, 61): P=244, S=65

This was the most common incorrect answer. These two numbers do not satisfy statement 5, because (4, 61) and (8, 57) each satisfy statement 4.

Mike Meyer, of Norman, OK, provided an interesting extension of this problem, showing how this puzzle is the first of an infinite sequence of puzzles; that is, the conversation becomes:

Mr. P: "I don't know the numbers."

Mr. S: "I know that; I also don't know the numbers."

Puzzle 1:

Mr. P: "I still don't know the numbers."

Mr. S: "I still don't know the numbers, either."

Puzzle 2:

and so on. At each point, which we can label *n*, insert the sequence:

Mr. P: "Oh, then I know the numbers."

Mr. S: "Really, then I do *also.*"

to create puzzle number *n*.

Interesting. The question is: who can *solve* one of those now?

The award goes to Charles Wells of the Department of Mathematics and Statistics, Case Western University, Cleveland, OH. The answer spit out by his program (Listing One, page 114) after running all night is (4, 13). He ran the program up to number pairs including 40, at which point he says it ran very slowly. He also implies that there might be larger answers, but "I refuse to believe anyone, mathematician or otherwise, could work this out for numbers above 20 in his head." Agreed. Perhaps I should have added to the puzzle the proviso that neither number was greater than 100, but nowhere did I say that the two world-famed mathematicians were idiot savants. Each would know that the other could not come up with an answer in his head for numbers greater than 100.

Although scores of entrants tried their hand at the preceding problem in logic, only two essayed the much easier task of programmatically describing a horse. J. C. Williams, of Cincinnati, OH, wins the prize for his program, presented in its entirety in these two lines:

```
FUNCTION HORSE( ) RETURN
    BOOLEAN
RETURN FALSE
```

He includes this "proof": "The function (program) is quite obviously a neigh-sayer," and appends the comment "(Sorry)." I presume he is apologizing for the pun, but that sort of originality was precisely what I was hoping for.

December's problem inspired more than double the entries of November's:

Assign a numerical value to each letter of the alphabet, starting with 1 for A and going up to 26 for Z. Any word in the English language has a value obtained by multiplying the values for each of its letters. For example, the word hello *is worth 86,400, obtained by multiplying 8 × 5 × 12 × 12 × 15. Which English word is equal to exactly 1,000,000? If there is none, which is the closest? Only words found in* The Random House Dictionary of the English Language *(unabridged edition) can be used. No capitalized words, none with hyphens or other embedded punctuation, nor those designated as foreign.*

Well, how about it, folks? Can you devise a program that finds the right word? And, having done that, can you tell us what that word is? Your program must be short and elegant. The algorithms can be demonstrated in a good pseudolanguage if you wish or perhaps in flowcharts.

Several entrants sent me words that multiplied out to 1,000,000, but I did not award any of them a t-shirt for several reasons. Many discovered that the word *typey* has a value of 1,000,000 and told me what dictionaries they had found it in. Others offered *tetty*, defined in the *Oxford English Dictionary* (*OED*) as a form of "testy." (It's also in my *Webster's New International Dictionary*, second edition.) The *OED* also yields *tytte*, an old form of "teat."

Those who sent me only one of these words did not qualify for the prize, because I asked for a *program*, not just an answer. Those who included a program that yielded only one of these words also did not win, because they did not follow the rules. I specified the *Random House Dictionary* for a number of reasons, and it does not contain *typey, tetty,* or *tytte*.

Why this particular dictionary? It is available as a dictionary program and thus does not necessitate typing in a whole dictionary full of words. Also, it is free of what I call "crossword puzzle dictionary words"—obsolete words, archaic variants, and other words that are seldom found in use today. Furthermore, *not all the words in it are in the electronic dictionary*. No currently available electronic dictionary even comes close to having as many words as its "hard copy" counterpart does. It is an unusual spelling checker that has even 50,000 words, and yet the tome of over 2000 pages that is my main reference work contains in excess of 260,000 *entries*, and there are many more words than entries. For example, *tablet* is a separate entry, but *tableted* is not. *Kaleidoscopically* is not a separate entry; it is found at the end of the definition for *kaleidoscope*. Spelling checkers are even more limited: they usually have only one form of each word. The plurals of nouns, the progressive forms of verbs, and so on, are often not there. The point I wished to make, which most people did not get, was that perhaps brute force, in the form of examining every word in a given dictionary, is not the way to solve this puzzle.

I commend all those who wrote programs generating all possible candidates and embedded in their programs the logic that the letters forming the hypothetical word must be multiples of 2 or 5 or both, those being the only prime factors of 1,000,000; that is, if a word exists in the specified dictionary that multiplies out exactly to 1,000,000, it must consist of some combination of B, D, E, H, J, P, T, and Y. Any qualifying word can have,

in addition, any number of As, that being a "free" letter because its addition to a word merely multiplies the product by 1. In fact, Ulrich Sondermann, of Concord, CA, chastised me for specifying the values of the letters to be 1 through 26, rather than 2 through 27—which, he said, was the way the problem was originally stated in the puzzle magazine. (The best word in that contest was *ixodid*, a kind of tick; Ulrich submitted the correct answer in that contest.) Ah, but I did not wish to repeat a previously published puzzle, and I *did* want a "free" letter because I wanted to provide a different kind of puzzle.

Ulrich deserves mention for another reason. He wrote his program in BASIC on a Timex/Sinclair: quite a feat! It generated 123 candidates (using the values I specified) in four minutes. He also provided a good algorithm for finding out if a given electronic dictionary contains any words consisting of these letter combinations. He said, "The problem then consists of creating a dictionary with words that have letters sorted in descending order. A match can then be made with each of the 123 candidates." (But what about the 124th candidate, Ulrich?)

I commend those of you who wrote such programs, but I did not award you prizes because I am not yet convinced that the *Random House Dictionary* has a word multiplying out to exactly 1,000,000 and because, with one notable exception, none of the programs accounted for the fact that adding any number of As to a word does not change its value.

Among the preceding, I give honorable mention to both Ken Waldron, of Vancouver, BC, and Ray Gardner, of Englewood, CO, because each devised a program that produced the same 124 candidates (all sans As). I also include Ray's program (Listing Two, page 116), because he accomplished in 24 lines of C code (seven lines of which, in good C coding practice, consist merely of one brace) what took Ken a page and a half of Pascal. Ray claims his program, written in Lattice C, runs in under four seconds on his IBM PC. Ray also says that his wife found the word *typey* in 15 minutes, without the benefit of a computer. (So much for computers.)

I also mention Thomas Shores, of Lincoln, NB, because, while his C program does not specifically generate words with As, he did acknowledge in his accompanying letter that they should be inserted "appropriately" within generated candidates. Also, his program generates all possible candidates, not just those multiplying out to 1,000,000. The best word he found was *moors*, value 1,000,350, but he thought others might come up with better possibilities. He stated that if any better words were found, they would have one of these values: 999,702; 999,810; 999,856; 1,000,000; 1,000,188. Good figuring, particularly since the word *I* think is the best has a value of 1,000,188. (How come you missed *rooms*, Thomas, an anagram of *moors*, and *comers*, which is also value 1,000,350?)

Trent Garverick, of Columbus, OH, gets an honorable mention for his short and sweet self-documenting pseudocode algorithm, the implementation of which in UCSD Pascal on an Apple II yielded what I believe to be the word *closest* to 1,000,000 found in the specified dictionary. *Curing* multiplies out to 1,000,188.

My own program had come up with that very word. My program first generated candidates within plus or minus 1000 of 1,000,000. I then "eyeballed" the list for real words. When I found several within 350, I narrowed the variance. Each time I found a better word, I started again with the smaller offset. My program, however, was inefficient, it took forever to run, and it generated literally thousands of possibilities, mostly because it spit out all permutations of each "word," not just words sorted in alphabetic descending order.

Trent based his program on examination of a real dictionary, rather than on generation of candidates. I still don't think that is the best method, but his program at least does not assume that a word multiplying out to exactly 1,000,000 must exist nor does it refuse to account for words with As in them.

I also mention Tom Balon, of Apalachin, NY, and Bob Smith, of Windsor, NY, who jointly produced a Pascal program on a VAX 11-785 to examine a dictionary of 33,595 words, coming up with *curing*. They also found, tied at 1,000,188, *Nicaragua*, but that word is disqualified as being a proper noun. Their program, while relatively short, was not self-documenting. Joe Celko, of Los Angeles, also deserves mention for a short, relatively easy-to-follow C program that produces, in combination with a spelling checker, lots of possibilities, the best of which he concludes is *curing*.

And now for the winner. The envelope, Randy, please. Ah, yes, Fred Smith, of Stoneham, MA, wrote a short C program into which you input a list of words; the program then determines the value of each word. The

goatees	997500
soybean	997500
stogy	997500
suety	997500
alkaline	997920
budlike	997920
chuckle	997920
clinked	997920
driven	997920
flanker	997920
invader	997920
lurked	997920
pucker	997920
variant	997920
viaduct	997920
village	997920
mopped	998400
palmated	998400
banquet	999600
curing	1000188
nicaragua	1000188
comers	1000350
cozies	1000350
moors	1000350
rooms	1000350
seamier	1000350
arkansas	1000692
sulks	1000692
packets	1003200
shakeable	1003200
shelve	1003200
shoved	1003200
specked	1003200
stalked	1003200

Figure
Partial Listing of Words Produced by Fred Smith's Million.C

default is to print only those words whose values lie within 10,000 of 1,000,000, but it has a command line switch (great for debugging) that prints all values. The program also ignores words of less than five letters. And here is the clever part, the spark of originality for which Fred really deserves the t-shirt: he then used the powerful capabilities of Unix to examine an entire dictionary, one of over 90,000 words.

He used two Unix tools, each performing one task well. (One was his own word value-determination program.) His program, *million*, reads all the words in a 90,000-word dictionary called *words* and pipes its output to the *sort* utility, which performs a numeric sort on the second field of each line and writes the final sorted file to an output file called *m.w.sorted*. Fred's program is Listing Three (page 118). Here is the Unix command:

```
million < words sort -n +1
  > m.w.sorted
```

So why, if I'm not convinced that reading all the words of a dictionary—furthermore, one that does not contain all the possible words of the dictionary I specified—is the best method, do I award Fred the t-shirt? Two reasons. One, his program was the cleverest. Two, nobody did it the way I thought it should be done.

I was hoping that someone would devise a program that generates all candidates equal to and close to 1,000,000 and retains only those obviously or likely to be English words, coming up with a small list that *a human being* then could scan for "real" words. What I wanted to show was the necessary cooperation between machines and humans on problems of this sort.

Russ Nelson, of Potsdam, NY, also deserves mention because he was clever enough to figure out precisely at which sprawling Silicon Valley corporate octopus my curious collection of I-Q Industries eccentrics *actually* works, proving to me that his knowledge was no mere guess by replicating the convoluted programmatic word game with which I produced the name. I would have awarded him an honorary t-shirt if he had also figured out what classical bit of entertainment inspired the transposition.

Thanks for all the kind letters!

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 197.**

Computer Calisthenics (Text begins on page 110)

Listing One

```
PROGRAM DAVIS(INPUT,OUTPUT);

CONST LARGE=50;

VAR M,N:INTEGER;

(*AT EACH STAGE BELOW, N REFERS TO THE
  NUMBER THAT MATHEMATICIAN WAS TOLD.*)



FUNCTION STAGEA(N:INTEGER):BOOLEAN;

  (*AT STAGE A, MR. P DOESN'T KNOW.
    THAT MEANS N HAS MORE THAN ONE
    ACCEPTABLE FACTORIZATION.*)

    VAR COUNT,F,G:INTEGER;

      BEGIN
        F:=2; G:=N DIV 2; COUNT:=0;
        WHILE F<G DO
          BEGIN
            IF N MOD F = 0 THEN
              COUNT:=COUNT+1;
            F:=F+1; G:=N DIV F
          END;
        STAGEA:=(COUNT>1)
      END;



FUNCTION STAGEB(N:INTEGER):BOOLEAN;

  (*AT STAGE B, MR. S KNOWS MR P. DOESN'T
  KNOW.  THAT MEANS EACH ACCEPTABLE PARTITION
  S*T=N YIELDS A NUMBER S*T WHICH SATISFIES
  STAGE A.  HE ALSO DOES NOT KNOW HIMSELF,
    WHICH MEANS N HAS MORE THAN ONE ACCEPTABLE
    PARTITION, WHICH MERELY MEANS N>6.*)

      VAR S,T,COUNT:INTEGER;
      TEMP:BOOLEAN;

      BEGIN
        S:=2; T:=N-S; COUNT:=0; TEMP:=(N>6);
        WHILE S<T DO
          BEGIN
            COUNT:=COUNT+1;
            TEMP:=TEMP AND STAGEA(S*T);
            S:=S+1; T:=N-S
          END;
        STAGEB:=(COUNT>1) AND TEMP
      END;



FUNCTION  STAGEC(N:INTEGER):BOOLEAN;

  VAR F,G,COUNT:INTEGER;

  (*AT STAGE C, P KNOWS.  THIS MEANS THERE
  IS EXACTLY ONE ACCEPTABLE FACTORIZATION
  OF M*N FOR WHICH STAGE B IS CORRECT*)

      BEGIN
        F:=2; G:=N DIV F; COUNT:=0;
        WHILE F<G DO
          BEGIN
            IF (N MOD F = 0) AND STAGEB(F+G)
              THEN COUNT:=COUNT+1;
            F:=F+1; G:=N DIV F
          END;
        STAGEC:=(COUNT=1)
      END;
```

(Continued on page 116)

Computer Calisthenics (Listing Continued, text begins on page 110)

Listing One

```
FUNCTION STAGED(N:INTEGER):BOOLEAN;

  VAR S,T,COUNT:INTEGER;

  (*AT STAGE D, S KNOWS TOO. THIS MEANS
    THAT THERE IS EXACTLY ONE ACCEPTABLE
    PARTITION S+T OF N FOR WHICH STAGE
    C IS CORRECT FOR S*T.*)

      BEGIN
        S:=2; T:=N-S; COUNT:=0;
        WHILE S<T DO
          BEGIN
            IF STAGEC(S*T) THEN COUNT:=COUNT+1;
            S:=S+1; T:=N-S
          END;
        STAGED:=(COUNT=1)
      END;
```

```
BEGIN (*MAIN*)

FOR M:=3 TO LARGE DO
  BEGIN
    WRITE(M:3);
    IF (M MOD 12 = 0) THEN WRITELN;
    FOR N:= 2 TO M-1 DO
      IF STAGEA(M*N) THEN
        IF STAGEB(M+N) THEN
          IF STAGEC(M*N) THEN
            IF STAGED(M+N) THEN
              BEGIN
                WRITELN;
                WRITELN('THIS PAIR WORKS:',N:3,M:3);
              END;
  END;
END.
```

End Listing One

Listing Two

```
/*     solve  --  Computer Calisthenics solution
*/

char letter [] = "bdehjpty";
char solution[20];

main ()
{
   solve(0,0,1000000L);
}

solve (length, nextletter, goal)
int length, nextletter;
long goal;
{
   int i, value;
   if ( goal == 1 ) {                    /* found a solution, write it  */
      solution[length] = '\0';
      puts(solution);
   } else {                              /* partial solution, extend it */
      for ( i = nextletter; letter[i]; i++ ) {
         value = letter[i] - 'a' + 1;
         if ( goal % value == 0 ) {
            solution[length] = letter[i];
            solve(length+1, i, goal/value);
         }
      }
   }
}
```

End Listing Two

(Continued on page 118)

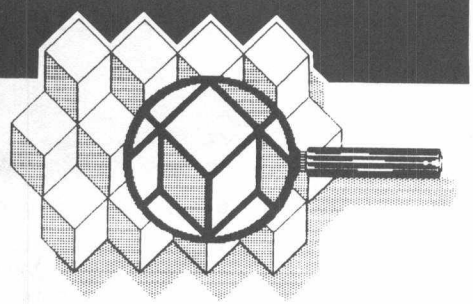
Computer Calisthenics (Listing Continued, text begins on page 110)

Listing Three

```
c  /*
c  --------------------------------> million.c <--------------------------------
c
c  Usage: million [-nl] < infile > outfile
c
c  I/O is from stdio. By default, only words with products >= 990000 and
c          <= 1010000 are output. If -nl switch is used ('no limits'), the
c          limit test is suppressed, and all words >= 5 letters in length are
c          printed.
c
c  */




   #include <stdio.h>
   #include <ctype.h>

   #define OPT     1000000                          /* optimum value        */
   #define MIN     OPT - 10000                      /* minimum acceptable   */
   #define MAX     OPT + 10000                      /* maximum acceptable   */
   #define BOOL    unsigned short
   #define MAXLINE 80              /* longest permissible input string     */
   #define TRUE    1
   #define FALSE   0

   main(argc,argv)
           int argc;
           char *argv[];
0----------  {
|          char word[MAXLINE];              /* input word to be examined            */
|          register char *pw;               /* register pointer thereto             */
|          register int wordlen;            /* length of string stored in word      */
|          register unsigned long prod;     /* working word value                   */
|          register BOOL print = TRUE;      /* ok to print word if TRUE             */
|          register BOOL limit = TRUE;      /* assume MIN-MAX range limits          */
|
|          if ((argc > 1) && !strcmp(argv[1], "-nl"))
|1---------    {                            /*  suppress limit test for printing    */
||              limit = FALSE;
|1---------    }
|          while ( gets(word))                      /* until end of input, get next word */
|1---------    {
||              if ((wordlen = strlen(word)) < 5)   /* filter short words */
||                  continue;
||              for (print = TRUE, prod = 1, pw = word ; *pw ; pw++)
||2--------c        {                        /* examine for non-alpha characters,
|||        c                                  calculate product                    */
|||                  if ( ! isalpha(*pw))
|||3-------              {
||||                      print = FALSE;
||||                      break;
|||3-------              }
|||                  prod *= (tolower(*pw) - 'a' + 1);
||2--------          }
||              if (((prod < MIN) || (prod > MAX)) && limit)
||2--------          {                              /* check limit conditions       */
|||                  print = FALSE;
||2--------          }
||              if (print)
||2--------          {                              /* is it ok to print? */
|||                  printf("%s",word);                      /* print word   */
|||                  for ( ; wordlen < 15 ; wordlen++)
|||3-------              {                          /* pad with blanks      */
||||                      putchar(' ');
|||3-------              }
|||                  printf("%u\n",prod);           /* print word value     */
||2--------          }
|1---------    }
0----------  }
```

End Listings

OF INTEREST



by Michael Swaine

The marketing of microcomputer technology is not an exact science. Lee Felsenstein, designer of the Osborne 1 and a founder of the Community Memory Project that wrote the **Sequitur** relational database management system, is now selling Sequitur through his company, Golemics, at a third of its original price. Sequitur was generally regarded as hot stuff when introduced some three years ago, but CM's faulty marketing kept it off the dealers' shelves, an article of furniture to which Sequitur will remain a stranger, since Sequitur will now be sold strictly by mail order at $250. "I kept my mouth shut [earlier] because I thought I didn't know enough to criticize. My Osborne experience made me realize that nobody else knows any better either," Lee says. He's wiser now; he turned forty this spring . . . Micro Craft, manufacturer of the **Dimension** computer, a pricey multiprocessor machine curiously positioned on release as a do-all end-user product, has also recently risen from the rubble of bad marketing. The company is now selling dual-68000 Unix machines and enjoying post-Chapter 11 prosperity . . . New Functions, the distributor of **Judy**, a concurrent personal assistant (aka gal/man Friday, sidekick, right-hand man, desktop accessory or pop-up), has one of the more novel distribution schemes extant. Judy owners can generate and give away Judy "clones"—limited-time free samples—and when the sampler buys his own Judy, the clonemaker gets a commission. New Functions recently raised that commission from 10% to 20%.

The largest computer show in the world is not held in the U.S. Atari introduced its 68000-based **ST** computer at the mammoth Hannover Fair in Germany in April. Despite the fact that the ST had a tendency to crash when asked to do anything much beyond the prepared demo, our correspondent on the scene found both the machine and its reception impressive, and advised us to look into software development for the ST under the GEM environment . . . Jack Tramiel's former company, Commodore, which has long stood strong in Europe, lost money this spring for the first time ever as dealers held out for the new **128** machine . . . A new **European Forth chip** could provide some competition for the **Novix NC4000 Forth chip**, although the Novix chip should actually be available by the time you read this. The NC4000 was designed by Charles Moore, who created Forth, and Robert Murphy; it can address four megabytes of memory and is intended to execute Forth code on the order of ten times the speed of any software implementation of Forth. Its designers expect it to find application in speed-critical areas like animation.

Boards

RAM price wars are behind price reductions on some hardware enhancements. The price of a do-it-yourself Macintosh upgrade is now below $200 . . . DFE has cut prices on its **NS32032 coprocessor board** for IBM PC/XT/AT and PC-compatible computers. The basic Tiger-32 was introduced last summer with a 6MHz clock, 32082 MMU, 512K onboard RAM and Xenix 3.0 for $2495, but RAM price-cutting has brought it down to $2095 . . . Univation's **Turbocharger** is a 9.54 MHz 8086 board for the IBM PC that sells for $1295 with 640K on-board RAM . . . Real Time Devices is selling a **GPIB instrumentation bus interface board** for PCs for $299, a per-channel cost of about $20.

Some new boards refine existing capabilities: Interstate Voice Products announced a **speaker-dependent IBM PC board** that has some limited facility for recognizing words in connected speech . . . We're examining STB's **video board**, which runs all IBM-compatible software, whether it was designed for a color or a monochrome monitor, by converting graphics displays into full-screen 16-level grey-scale images . . . ANEX Technology is selling a **banked 2-megabyte RAM** board for PCs that are running multi-user software.

For $61 (quantity one price), you can pick up a VLSI chip that lets you directly replace an 8088 microprocessor with an 80286. Edsun Labs' **EL286-88 Processor Converter** stands between the 286 and the peripherals, playing an 8088 to the peripherals, while accepting and converting all 16-bit transfer requests from the 286. The idea is to combine 16-bit main memory operations with low-cost 8-bit peripherals.

MSDOS

Alternatives or extensions to MSDOS/PCDOS are popping up like pop-ups. DRI has announced the first software to run under its Graphics Environment Manager (GEM): **GEM Desktop, Draw, Write, Paint, Wordchart and Graph**. Desktop lets you run up to six *utility programs* on top of applications, like the Mac's desktop accessories; all are supposed to be shipping as you read this . . . Digitrol's **pcShare** substitutes a multiuser system with file protection for PCDOS . . . RTCS's **PC/RTX** substitutes a real-time O/S (an implementation of Intel's iRMX-86) . . . Soft-

Logic's **DoubleDOS** makes PCDOS *multitask* for $99 ("multi" here means "two").

For software development under MSDOS/PCDOS, Phoenix software has a program development manager called **Pmaker** that they say is similar to the Unix "make" command, and a program performance analyzer caller **Pfinish** ... **Advance Trace86** from Morgan Computing is a debugger/assembler that lets you back up execution 20 instructions ... Genesis's **GeneScope** is a new full-screen symbolic debugger for the PC.

Whether developing expert systems on a micro is the Next Big Thing or just the latest fad, it's getting cheaper and easier. **Insight 2** from Level Five Research lets you incorporate Pascal *routines* and dBase II files into a knowledge base developed on a PC or compatible ... AI pioneer Donald Michie's **Expert Ease** package has undergone some changes of ownership and distribution (it's now distributed by Human Edge), the most visible consequence of which is a price cut from $2000 to $695.

C

"To boldly go where no C software distributor has gone before ...." Phoenix calls **Pre-C**, its lint superset, the industry's first program analyzer for the C language designed to support MSDOS and PCDOS ... Micro Software Developers says its **C Debugger** is the first true source debugger for C ... Catalytics has released what it calls the first complete C interpreter, its **Safe C** ... For Lattice C programmers who don't necessarily hanker to be first, or at least would like some company out there on the giddy edge, Bill Hunt, author of *The C Toolbox*, has started the **Lattice C User's Group**.

Whitesmiths has upgraded its 8086-family **C compiler** to support all 8086 memory models and for closer adherence to their best guess at the emerging ANSI standard ... Toolworks has moved its **C/80 compiler** to MSDOS and is selling it for $49.95 ... Smith & Smith Associates has released **CrossRefC**, a C cross-reference/listing generator, for $39.95 ... Vance Info Systems announced **C Lib**, a C function library of over 200 routines, for the MSDOS environment.

CP/M

CP/M users needn't feel left out of the pop-up fad: Poor Person Software's **Write-Hand-Man** (that's something like a sidekick, we suppose) includes a notebook, phonepad, desk calendar, file and directory viewing programs and a communications program; new functions can be added fairly easily. It sells for $49.95 and runs under CP/M 2.2 on all CP/M machines.

Also for CP/M, **MB+ Tools** is Minnow Bear Software's $175 set of programmer productivity tools for Pascal MT+ ... Soft Advances says that its **DSD80 full-screen symbolic debugger** is fully DDT-compatible and fully supports Z80 instructions using either extended Intel or Zilog mnemonics ... Selfware's **Convert** 3.1 lets you read, write and format disks, copy files and list directories in 105 CP/M formats on an MSDOS machine, and costs $99.

Mac

Just now, despite such attractions as the novelty of a Smalltalklike Forth extension (Kriya's **Neon**), some of the most interesting products for the Mac are to be found on user group disks. Not surprisingly, a high proportion of these public-domain or user-supported programs consists of communications software, like Dennis Brothers' **MacTEP** and Scott Watson's **Red Ryder**. You can pick more fruit if you have a ladder ... There are also a lot of fonts, desk accessories and graphics; the **IM Underground's graphics** consist of schematics for the machine ... Equally unsurprising, given the features of the Mac, is the number of attractively-designed Mac newsletters, including **MacTutor**, **The Mac Street Journal** and **MacBriefs**.

Apple][

Contrary to the impression given at its stockholders' meeting early this year, Apple makes another computer, the Apple bracket-bracket. Programmers are even continuing to write software for the bracket-bracket ... MicroSparc has released the *ProDOS* version of its **Ampersoft** utility package at $49.95 for Applesoft programmers ... There is, of course, much existing Apple][software: **InCider** is selling disks of software from the magazine at $21.47 ... **Pandora Software** has collected a library of 4000+ public-domain programs for the Apple][and is selling them at $5 per disk (at twenty programs per disk, that's $1000 for the library, but still).

A Visit to the Fab Lab

A blip of *deja vu* struck as I read in the March *IEEE Software* about Ivan Guzman de Rojas and his plan for doing natural-language translation via the peculiarly algebraic Andean Indian language Aymara. Yes, trivial recall was working perfectly: halfway down the Andean peak of press releases on my desk I found the announcement of the opening of the **Aymara Fab Lab** in Sunnyvale. A coincidence worthy of a drive down the peninsula.

Dean Norman, Director of the Aymara Fab Lab, greeted me at the door and began talking immediately

about Guzman. Norman explained that Guzman, discovering that Aymara, lacking irregular verbs and gender, was an unprecedentedly logical language (although its logic was not standard two-valued logic), had succeeded in codifying the algorithmic structure of its syntax. For the first time, someone had expressed a natural language in software. Wasn't Guzman, I asked, considering the application of his achievement in the design of translating machines, the notion being that computerized Aymara could serve as the bridge in a multilanguage translation system?

Right, Norman answered, although at AFL they were taking the process in a somewhat different direction. Did I recall the Sapir-Whorf hypothesis from linguistics? I did, more or less: that language delimited the thinkable thoughts, and thus our culture and perceptions. Under normal conditions, we see only those distinctions for which we have words. In cultures in which green is not a major linguistic division of the spectrum, it is also not a primary perceptual division.

Norman nodded. The principle can be applied to any language-processing system, natural or artificial. Curious as it might sound, the Aymara-speaking software would, to a certain extent, think like an Aymaran. With its multi-valued logic, it would make distinctions that would never occur to a New York stockbroker; with its lack of grammatical gender, it would fail to make distinctions the stockbroker would unconsciously make. And Aymaran is only the easiest language, not the only one, to which the principle can be applied. Employing Guzman's translation techniques, it would be possible to develop front-end packages that, with the proper filtering out of Aymaran values and perceptions, would embody pure upper-class British perceptions or ancient Greek thought processes. We could examine the way judges in ancient Sumeria examined evidence.

That, Norman explained, was what they were up to at AFL: just as the developers of expert systems were trying to capture the knowledge of selected individuals in software, AFL was trying to capture the style of thinking, the intellectual spirit of whole cultures. I mulled that over. Wouldn't there be a great advantage, I asked, in combining the two approaches, developing a system with a specifiable knowledge base and a specifiable style of thinking? Couldn't one develop, say, a machine with the knowledge of a high-energy physicist and the spirit of a 12th-century Mandarin? Or the knowledge of a modern statesman and the intellectual style of the first Continental Congress? But Norman suddenly looked uncomfortable and said that he couldn't discuss details of ongoing projects.

Reference Map

Golemics, Inc.
2600 10th St.
Berkeley, CA 94710
(415) 486-8347
Reader Service No. 101.

Micro Craft Corp.
4747 Irving Blvd., Ste 241
Dallas, TX 75247
(214) 630-2562
Reader Service No. 103.

New Functions
P.O. Box BB
Staten Island, NY 10302
(718) 273-2668
Reader Service No. 105.

Atari
1196 Borregas
Sunnyvale, CA 94086
(408) 745-2000
Reader Service No. 107.

Dr. Alan Winfield
Metaforth Computer Systems Ltd.
Unit 2 B The Newlands Center
Inglemire Lane
Hull HU6 7TQ
England
Reader Service No. 109.

Novix
10590 North Tantau Ave.
Cupertino, CA 95014
(408) 996-9363
Reader Service No. 111.

DFE Electronic Data Systems
USA: 5820 Stoneridge Mall Rd.
Suite 115
Pleasanton, CA 94566
(415) 847-2024
Europe: Singerstrasse 3,
7513 Stutensee
West Germany
07244-9011
Reader Service No. 113.

Real Time Devices, Inc.
1930 Park Forest Ave.
P.O. Box 906
State College, PA 16804
(814) 234-8087
Reader Service No. 115.

Interstate Voice Products
1849 W. Sequoia Ave.
Orange, CA 92668
(714) 937-9010
Reader Service No. 117.

STB Systems, Inc.
601 N. Glenville, Ste 125
Richardson, TX 75081
(214) 234-8750
Reader Service No. 119.

Univation
1037 North Fair Oaks Ave.
Sunnyvale, CA 94089
(408) 745-0180
Reader Service No. 121.

ANEX Technology, Inc.
151 North Route 9W
Congers, NY 10920
(914) 268-2400
Reader Service No. 123.

Edsun Laboratories, Inc.
7 Sears Rd.
Wayland, MA 01778
(617) 358-5667
Reader Service No. 125.

Digital Research Inc.
Box DRI
Monterey, CA 93942
(800) 443-4200
Reader Service No. 127.

Digitrol Computers Inc.
440 Phillip St.
Waterloo, Ontario
Canada N2L 5R9
(519) 884-4541
Reader Service No. 129.

RTCS Corp.

1390 Flynn Rd., Unit E
Camarillo, CA 93010
(805) 987-9781
Reader Service No. 131.

SoftLogic Solutions
530 Chestnut St.
Manchester, NH 03101
(800) 272-9900
Reader Service No. 133.

Phoenix Computer Products Corp.
1416 Providence Highway, Ste 220
Norwood, MA 02062
(617) 762-5030
Reader Service No. 135.

Morgan Computing Co., Inc.
P.O. Box 112730
Dallas, TX 75011
(214) 245-4763
Reader Service No. 137.

Genesis Microsystems Corp.
196 Castro St.
Mountain View, CA 94041
(415) 964-9001
Reader Service No. 139.

Level Five Research
4980 South A-1-A
Melbourne Beach, FL 32951
(305) 729-9046
Reader Service No. 141.

Human Edge Software Corp.
2445 Faber Place
Palo Alto, CA 94303
(415) 493-1593
Reader Service No. 143.

Poor Person Software
3721 Starr King Circle
Palo Alto, CA 94306
(415) 493-3735
Reader Service No. 145.

Minnow Bear Computers
P.O. Box 2233 Sta. A
Champaign, IL 61820-8233
(217) 398-6883
Reader Service No. 147.

Soft Advances
P.O. Box 49473
Austin, TX 78765
(512) 478-4763
Reader Service No. 149.

Selfware
3545 Chain Bridge Rd., Ste 3
Fairfax, VA 22030
(703) 352-2977
Reader Service No. 151.

Micro-Software Developers, Inc.
214½ West Main St.
St. Charles, IL 60174
(312) 377-5151
Reader Service No. 153.

Catalytix Corp.
55 Wheeler St.
Cambridge, MA 02138
(617) 497-2160
Reader Service No. 155.

Lattice C User's Group
P.O. Box 3072
Glen Ellyn, IL 60138
(312) 858-7950
Reader Service No. 157.

Whitesmiths
97 Lowell Rd.
Concord, MA 01742
(617) 369-8499
Reader Service No. 159.

The Software Toolworks
15233 Ventura Blvd., Ste 1118
Sherman Oaks, CA 91403
(818) 986-4885
Reader Service No. 161.

Smith & Smith Associates
P.O. Box 160
Hunt Valley, MD 21030
(301) 666-8129
Reader Service No. 163.

Vance Info Systems
2818 Clay St.
San Francisco, CA 94115
(415) 922-6539
Reader Service No. 165.

Kriya
505 N. Lake Shore Dr. Suite 5510
Chicago, IL 60611
(312) 822-0624
Reader Service No. 167.

Dennis Brothers' MacTEP
Berkeley Mac Users Group
1442A Walnut St., Ste. 153
Berkeley, CA 94709
Reader Service No. 169.

Red Ryder
The FreeSoft Co.
10828 Lacklink
St. Louis, MO 63114
(314) 428-8057
Reader Service No. 171.

The IM Underground
715 Church St. #16
Ann Arbor, MI 48104
Reader Service No. 173.

MacTutor
P.O. Box 846
Placentia, CA 92670
(714) 993-9939
Reader Service No. 175.

The Mac Street Journal
P.O. Box 6686
Yorkville Station
New York, NY 10128
Reader Service No. 177.

MacBriefs
P.O. Box 2178
Huntington Beach, CA 92647
(714) 842-0518
Reader Service No. 179.

MicroSparc
45 Winthrop St.
Concord, MA 01742
(617) 371-1660
Reader Service No. 181.

InCider
CW Communications/Peterborough
Peterborough, NH 03458
Reader Service No. 183.

Pandora Software
P.O. Box 55
Clearfield, UT 84015
Reader Service No. 185.

IEEE
345 East 47th St.
New York, NY 10017
Reader Service No. 187.

Aymara Fab Lab
1974 Gardner St.
Sunnyvale, CA 94086
Reader Service No. 189.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 198.**

Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS #106 AUGUST 1985 $2.95 ($3.95 CANADA

The Definitive C Compiler Review

Update on Small C

An Optimizer for C Code

An MSDOS "Make" Utility

CP/M Exchange Goes On Line



0 08

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Polly Koch*
Typesetter *Jean Aring*

Production
Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover Artist *William Cone*

Circulation
Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Snee*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Kathleen Boyd*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*
Administrative Coordinator *Kobi Morgan*

Advertising
Advertising Director
Shawn Horst (415) 424-0600
Advertising Sales
Walter Andrzejewski (617) 567-8361
Lisa Boudreau (415) 424-0600
Beth Dudas (714) 643-9439
Michele Beaty (317) 875-0557
Advertising Systems Manager
Ron Copeland (415) 424-0600
DDJ Classifieds
Tim Ortiz (415) 424-0600
Advertising Administrative Manager
Alice Abrams (415) 424-0600

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $46 for airmail to Canada, $62 for airmail to other countries. Foreign subscriptions must be pre-paid in U.S. Dollars, drawn on a U.S. Bank. For subscription problems, call: outside of CA 800-321-3333; within CA 619-485-9623 or 566-6947.

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 686-1520 TELEX: 620430 (WUI)

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

August 1985
Volume 10, Issue 8

CONTENTS

This Month

This month *DDJ* focuses its attention on C. The issue should contain a little something for everybody. One "little something" for all our readers is Jim Hendrix' "Small-C Update." After reading this latest chapter in the story of the compiler first published in the pages of *Dr. Dobb's*, MSDOS C programmers can go straight to the review of MSDOS C compilers, while CP/M C programmers can flip to David Fox' article on optimizing assembly language output from the Software Toolworks' C/80. Unix gurus, on the other hand, will probably want to turn to this month's C Chest, where they can find an MSDOS version of one of their favorite Unix utilities, make. We hope you will find something you like.

Writers' Guidelines

What are the topics on which *DDJ* would like to receive articles? What style of writing is appropriate to *DDJ*? How should I format my text? How should I format listings and how extensively should I comment them? Should I submit hard copy of my manuscript, or a diskette, or both? How much will I get paid for my contribution? The answers to these and other frequently-asked questions can be found in the latest revision of the Writers' Guidelines for *Dr. Dobb's Journal*. To obtain a copy along with the current editorial calendar, drop us a note.

This Month's Referees

Richard Relph, ELXSI
Richard G. Larson, University of Illinois at Chicago
James Hendrix, University of Mississippi

Dr. Dobb's Journal

ARTICLES

C Compilers for MSDOS — 30
by the PicoNet C-SIG and the SVCS
The C-SIG and SVCS have assembled a suite of 30 benchmarks, run them through the most popular MSDOS compilers and interpreters and tabulated the results. **(Reader Ballot No. 191).**

A Peephole Optimizer for Assembly Language Source Code — 56
by David L. Fox
Using regular expressions you can optimize the output of any compiler producing assembly language code. **(Reader Ballot No. 192).**

Small-C Update — 84
by James Hendrix
An update on the compiler first published in the pages of *DDJ*: bug fixes, improvements and MAC, a new Small-C macro assembler. **(Reader Ballot No. 193).**

Asynchronous Protocols — 92
by David W. Carroll
A description of the features, strengths and weaknesses of the most popular asynchronous protocols in use today. **(Reader Ballot No. 194)**

COLUMNS

Dr. Dobb's Clinic — 20
by D. E. Cortesi
DRI Generic Sales; the Z80; nesting MSDOS; quasi random number generators. **(Reader Ballot No. 190).**

C Chest — 96
by Allen Holub
Allen Holub reviews three MSDOS implementations of the Unix utility "make" and offers one of his own. **(Reader Ballot No. 195).**

16-Bit Software Toolbox — 116
by Ray Duncan
A bug in TEE; musings on DRI's GEM-Paint and Concurrent PCDOS; square roots; more Mac Feedback. **(Reader Ballot No. 196).**

CP/M Exchange — 124
by Robert Blum
The CP/M Exchange RCP/M; a catalog of free CP/M software; exiting properly to CP/M. **Reader Service No. 197).**

DEPARTMENTS

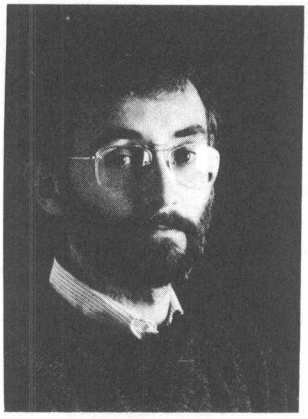
Editorial 6

Letters 10

DDJ Classified 89

Advertiser Index 128

EDITORIAL



We'd like to congratulate Microsoft, Lifeboat, and PC World for their jointly-sponsored C-85 conference in San Francisco in May. All the *DDJ* editorial staff attended (Allen Holub, our C columnist, and Richard Relph, coordinator of the C compiler review in this issue, were among the speakers), and we found it educational and well-planned.

One session was of particular interest to us. At a discussion of compiler and library evaluation and benchmarking, a speaker chastised computer publications for the inadequacy of their technical reviews. We sat in the audience nodding in agreement.

It's true: today's computer magazines typically do not provide adequate information to permit software developers to choose one compiler or programming tool over another. A relatively conscientious reviewer may (1) run three benchmark tests, usually a floating-point test, the Fibonacci and the Sieve of Eratosthenes; (2) list the product's features; and (3) give a clear picture of his own experiences in using the product. Most do less.

The C-85 conference session pointed out the need for a higher level of criticism, at least in the evaluation of technical products. This is an area in which the vendors and customers are both highly knowledgable. Programmers and system designers read reviews for solid information on which to base decisions affecting their livelihoods. Vendors deserve an accurate appraisal of their products in terms that matter to their customers.

A useful compiler review should, we think, assess completeness, correctness, adherence to standards, and use a widely-available suite of benchmarks applied in a repeatable manner to measure all the major aspects of compile- and run-time performance. Publication of a review should constitute a commitment to continue to evaluate the product, and to publish updates and corrections when appropriate. And no review should be the work of a single individual, no matter how knowledgable or above reproach.

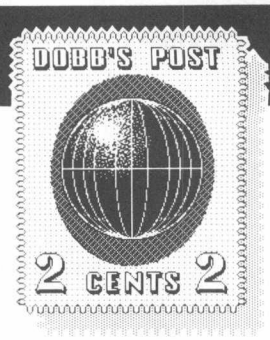
We've tried to provide such a useful review in this month's evaluation of 13 C compilers, done by over a dozen programmers using several dozen benchmarks, with several pairs of eyes reading the manuscript. But we don't claim we brought it off flawlessly, and we'll be following up on this review with updates, corrections (if necessary), and comparable benchmark results for other compilers.

For us, and we think for computer publications generally, this is new ground. we don't believe that any computer magazine has been doing technical reviews the way they should be done, and that includes *Dr. Dobb's Journal*. We hope to change that, and this issue's C compiler review is one step in that direction.

We plan to take some more steps in the direction of useful reviews in the coming months. One desirable feature that this month's C compiler review lacks is a single benchmark test providing a real-world mix of functions; we're looking into some tests that purport to do this. We welcome your feedback and suggestions.

Michael Swaine

Michael Swaine



Tiny BASIC

Dear *Dr. Dobb's*:

I read with great interest the article "Tiny BASIC for the 68000," which ran in the February 1985 issue of your magazine. I have been a long-time 68000 programmer and have written a full BASIC interpreter for the machine. But, dear Doctor, a 68000 BASIC only about 10% faster than a Z80? I quickly got a copy of the source from the bulletin board and started probing. Mr. Brandly did a nice job of coding, but he missed a few tricks. It is relatively easy to do a substantial speed-up without increasing size significantly or changing any outward functionality.

The main problem in this BASIC implementation is the TSTC subroutine. TSTC compares the current character against a test value and branches to a specified location if they do not match. TSTC gets used a lot in even simple statements; 10 X=1 (crlf) calls it 7 times. I count 98 clocks for TSTC if the characters match, and 102 clocks if they do not match. Inline code to replace TSTC could be made a macro as follows:

```
TSTC   MACRO
       BSR IGNBLK
       CMPI.B #'&1',(A0)
       BNE &2
       ADDQ.L #1,A0
       ENDM
```

which executes in 46 clocks (for match) and 40 clocks (no match). This gives a savings of 52 to 62 clocks per use of TSTC . . . or almost 400 clocks in a simple assignment statement. Any statement involving an expression will see a similar (or greater) improvement.

The second improvement also occurs in the expression evaluator routine EXPR. (This is where most improvements are to be found, since expression evaluation is key to the interpreter's performance.) EXPR uses the EXEC subroutine to check for relational operators (=,<,>,<=,>=). EXEC is a table-searching routine normally used to check for keywords. It includes a lot of overhead to check for shorthanded keywords that can never occur in relationals. Also, the most usual case—no relational—requires searching the six possible relationals first before discovering the default condition. My suggested improvement treats relational operators as a special case, removing them from the EXEC tables and substituting the code in Listing One (page 16) for the EXEC call at the top of EXPR. Note that the default case (no relational) now takes only three compares. This is significant because it is used every time an expression (number, variable, etc.) is encountered.

The third improvement is found in the find-lines routines: (FNDLN, FNDLNP, FNDNXT, and FNDSKP.) These routines find heavy usage in GOTOs, GOSUBs, and IFs. The point here is that the original coding resets register A2 to the end of the program text for each line. This 24 clock overhead per line can be saved by a little re-ordering and a few extra bytes Listing Two, page 16.)

My final improvements do not affect the execution speed of the interpreter, but rather speed up program editing. The block-move subroutines MVUP and MVDOWN can be speeded up quite a bit at the cost of some additional code. I'll illustrate with MVUP. It is currently coded:

```
MVUP   CMP.L A1,A3
       BEQ MVRET
       MOVE.B
         (A1)+,(A2)+
       BRA MVUP
*
MVRET RTS
```

This requires 36 clocks per byte moved. If one calculates the number of bytes to be moved outside the loop, a much tighter inner loop (22 clocks per byte) is possible (see Listing Three, page 17). The set-up overhead is 48 clocks, so it takes only 4 cycles through the inner loop to outweigh it. This version of MVUP can handle a maximum of 65,536 bytes (the original had no restriction), but this should not prove much of a problem in Tiny BASIC programs.

A final note to the Doctor: please keep articles like this coming! There is nothing that improves the "breed" of programmers like quality examples. Thank you.

Sincerely yours,
Robert D. Grappel
28 Buckmaster Drive
Concord, MA 01742

Ellipses

Dear *DDJ*,

Many thanks to Tom Hogan for his excellent derivation of the DV (decision variable) method for plotting ellipses. His explanation of this potentially confusing subject is very clear. This general approach, sometimes called 'displacement comparison' appears elsewhere in the literature, but this is the first time I have seen it applied to ellipses.

I would like to point out a minor error in the derivation, and then show how the full potential of the method can be achieved by one more refinement that eliminates all multiplication within the loops.

The error is in Equation 10. If the x-axis is the major axis, the initial point is at

$$(0, \sqrt{\frac{\beta}{\alpha}}\ R_m), \text{ not } (0, \frac{\beta}{\alpha}\ R_m)$$

Equation 10 is correct if the first term is changed to reflect this. The

situation is clarified if one considers a different form of the equation of the ellipse:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

where a and b are the radii along the x the y axes, respectively.

If we let $\alpha = a^2$ and $\beta = b^2$, then Mr. Hogan's Equation 1 is equivalent. It is clear that the aspect ratio is b/a, and Mr. Hogan's use of the aspect ratio is consistent, except for equation 10. If the x-axis is the major axis, then $R_m = a$, and the initial point is at b, which is given by $R_m \times$ aspect ratio, or $R_m \times (b/a)$. (We are assuming that the center of the ellipse is at (0, 0)). This is a fairly obvious error, but it could confuse the unwary (me, for example). The statement in the program which initializes d is correct, so I assume Mr. Hogan found the error and simply overlooked this one instance.

Much more interesting is an examination of the operations that are performed within the loops for computing successive dot locations. An example is the computation of 'd' at the end of the first loop

```
(d += two_beta *
   (3 + (rel_x << 1)) )
```

Note that every term in the computation is a constant except for rel_x, and further, that rel_x is incremented by exactly one every time the loop is executed. So let's rewrite that line for clarity:

```
(d+= two_beta*3 +
   two_beta*rel_x*2)
```

or in human rather than computer math

$$(d += 6\beta + 4\beta \text{xrel})$$

Now we will cleverly initialize our added variable d_norm to 2β before entering the loop, so every pass we add 4β to d_norm before adding it, in turn, to the former value of d.

All of the multiplication operations within the loops can be replaced by additions in a similar way, as demonstrated in Listing Four (page 18) (we use x, y in place of rel_x, rel_y to denote the local coordinates relative to the center of the ellipse, and we don't bother to maintain the global coordinates row and col, but the result is the same).

I will be happy to discuss this with anyone who wants to call me at (206) 226-3916, or via IEEE COMPMAIL—user name eric.therkelsen.

Again, many thanks to Mr. Hogan for a derivation that is a significant contribution to the literature.

Sincerely,
Eric E. Therkelsen
ETTS, Incorporated
19224 Southeast 164th
Renton, WA 98058

Prolog

Sirs:

I enjoyed your March, 1985, issue on Prolog, but thought I should write to warn your other readers of possible support problems with Programming Logic Systems, Inc., the U.S. marketers of MicroPROLOG. Over the past 5 months I have attempted, with no success whatsoever, *to obtain the* simple information whether there have been any changes or upgrades to the software in the year or so since I purchased it. Neither the U.S. or British offices will respond to repeated mail inquires (the British office simply sent a brochure, in no way answering my very simple question). If these people can't respond to a question as simple as that, it does not bode well for their willingness to respond to substantive inquiries, and suggests to me that one of the other fine implementations of the language would be a better choice.

Sincerely,
George Carey
2048 Winding Creek Ln.
Marietta, GA 30064

Listing One

```
        BSR IGNBLK          skip blanks
        MOVE.B (A0)+,D2     get first char.
        CMPI.B #'=',D2      equals?
        BEQ XP15            yes, process it
*
        CMPI.B #'>',D2      >, or >= ?
        BNE XXP1            no
*
        CMPI.B #'=',(A0)    >=?
        BNE XP13            no, process >
*
        ADDQ.L #1,A0        skip past =
        BRA XP11            process >=
*
XXP1    CMPI.B #'<',D2      <, <=, or <>?
        BEQ XXP2            yes
*
        SUBQ.L #1,A0        no, default case
        BRA XP17            handle default
*
XXP2    CMPI.B #'=',(A0)    <=?
        BNE XXP3            no
*
        ADDQ.L #1,A0        yes, skip past =
        BRA XP14            process <=
*
XXP3    CMPI.B #'>',(A0)    <>?
        BNE XP16            no, process <
*
        ADDQ.L #1,A0        skip past >
        BRA XP12            process <>
```

End Listing One

Listing Two

```
FNDLN   CMPI.L #$FFFF,D1
        BCC QHOW
        MOVE.L TXTBGN,A1
        MOVE.L TXTUNF,A2        note change
        SUBQ.L #1,A2
*
FNDLNP  CMPA.L A1,A2
        BCS FNDRET
```

```
*
        MOVE.B (A1),D2
        LSL.L #8,D2             note change
        MOVE.B 1(A1),D2
        CMP D1,D2
        BCC FNDRET
*
F1      ADDQ.L #2,A1            note change
F2      CMPI.B #CR,(A1)+
        BNE F2
*
        BRA FNDLNP
*
FNDNXT  MOVE.L TXTUNF,A2        note change
        SUBQ.L #1,A2
        BRA F1
*
FNDSKP  MOVE.L TXTUNF,A2        note change
        SUBQ.L #1,A2
        BRA F2
```

End Listing Two

Listing Three

```
MVUP    MOVE.L D0,-(SP)         preserve D0
        MOVE.L A3,D0
        SUB.L A1,D0             compute no. of bytes
        BLE MVRET               no bytes?
*
        SUBQ.L #1,D0            adjust for DBRA
*
MV2     MOVE.B (A1)+,(A2)+
        DBRA D0,MV2
*
MVRET   MOVE.L (SP)+,D0         restore D0
        RTS
```

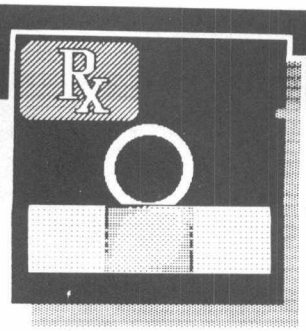
End Listing Three

Listing Four

```
ellipse_write ( xc, yc, a, aspect )
    int xc, yc, a;
                /* xc, yc are coords of center; a is radius along x-axis */
                /* we chose to input a rather than Rm */

    float aspect;
    {
    long alpha, beta;
    long four_alpha, four_beta, alpha_y, beta_x, d, d_exc, d_norm;
    int b, Rm;      /* b is radius along y-axis */
        /* since a is not necessarily the major radius, we will compute Rm */

    int  x, y;          /*    local coords relative to center of ellipse    */
                                .
                                .
            [ initialization similar to Mr. Hogan's ]

                                .
                                .
    alpha_y = alpha * y;     /* new vbls to avoid multiplication in loop...*/
    beta_x = 0L;
    d_exc = - four_alpha * y;
    d_norm = beta << 1;

    d = 2*alpha*(y*(y-1)) + alpha + 2 * (beta - beta*Rm - alpha*beta);
            /* This is Mr. Hogan's original statement - it is correct
               even though equation 10 is not.    */

    while ( alpha_y > beta_x )
        {
        put_dots ( xc, yc, x, y );
        if ( d >= 0 )
            {     /*    exception condition - decrement y occasionally    */
            d += ( d_exc += four_alpha );
            alpha_y -= alpha;
            y--;
            }
        d += ( d_norm += four_beta ) ;     /*    always increment x    */
        beta_x += beta;
        x++;
        }

    /*    slope is now < -1, so proceed with rest of ellipse    */
                                .
                                .
                        [ 'nuff said? ]
                                .
                                .
```

End Listings

DR. DOBB'S CLINIC



by D. E. Cortesi

DRI Generic Sales Again

Barry Watzman writes to contradict us regarding Digital Research's policy on selling generic operating systems. He claims that DRI is "not only not selling generic systems any longer, they are no longer licensing value-added vendors either. The new rule is, if you're not the manufacturer of the computer on which the implementation will run, forget it."

By way of proof, he sent along a photostat of a letter to him from Alice Clark, Contract Administrator at DRI. It includes the sentence: "We are not currently licensing companies to modify and redistribute our software unaccompanied by hardware or an application program."

The dispute seems to be over how much and what kind of value a "value-added" vendor must add to be acceptable. Watzman wants to make a business of configuring DRI systems to Zenith hardware (he does it very well from all we hear). DRI's position, he says, prevents it.

What Z800?

Roger Smith was turning over back issues of *DDJ* when his eye was caught by the confident words of L. Barker in this column for November 1981. "When the Z800 comes out next year," Barker averred, "with the Z80 instruction set and multiply and divide, this [worrying about the speed of arithmetic] will look like a waste of time."

That pin-compatible Z80 superchip never appeared, so far as we know. "Does anybody know what happened?" Smith wonders, and so do we.

Nesting MSDOS

Remember how we finally got CP/M batch command processing pretty well fixed up? If you don't, you might like to refer to these old Clinics:

June 1982 for PAUSE and BEEP commands
August 1982 for null-line, lowercase, and control character fixes
September 1982 for the QUITIF program
October 1982 for nested submits
January 1984 for a PAUSE that works in CP/M Plus

You might think that batch execution in MSDOS Version 2 wouldn't need our help, but you'd be wrong: MSDOS won't nest batch files. When you invoke a batch file in a batch file, the called file replaces its caller instead of being its subroutine.

We just discovered how to do it. You nest batch files by nesting command shells. Command.com, the shell, can be named as a command. If given a /c switch, it executes the single command that follows on the command line and ends. If that command is a batch file, it will be executed in full, and control will return to the calling batch file. Type in our demo file, recurse.bat (Listing One, page 24), and try it out with a line like

```
recurse 0 1 2 3 4 5
```

What's nice is that each level of command.com has its own environment. The nested batch file can do all the sets and paths it likes; on exit, the prior environment is restored. Add a chkdsk command to the end of the file to see how much space each nesting level takes up.

Random Error

In February, we published a quasi-random number generator (QRNG) credited to persons named Talley and Metropolis (about whom we know nothing more). Ever since then, we've been backing away from it, but we have the subject more or less under control now, mostly thanks to David Ross and Michael Barr.

We showed the QRNG in C. It had two main parts. The first, rand0( ), purported to produce quasi-random numbers uniformly distributed between zero and MAX. Take that at face value; we'll come back to it.

The second part, randi(u), used the first to produce what purported to be quasi-random numbers uniformly distributed between 1 and u. It did not. Here's the essence of how we coded it:

```
randi(u)
unsigned u;
{
    return(1 + rand0( ) % u);
}
```

The percent sign is the C sign for modulus. Can you see what's wrong with this?

Michael Barr and Peter K. Pearson did. Barr put it best: "Imagine you adopted the following method to find random digits from 0 to 9: throw a dart at the face of a clock and take the low-order digit of the number nearest to where the dart hits. It is evident that you will get the digits 1 and 2 one time in six and the other digits once in twelve. Thus a random event is translated into one that is still random, but not uniformly distributed. [Your procedure randi(u)] does that. It shows up most clearly when u is a sizable fraction, say 2/3, of MAX. In that case, you will get approximately twice as many values between 0 and u /2 as between u /2 and u."

We devised randi( ) with the vague mental image of how the mod function folds the real number line, like a carpenter's rule, into a heap of u-sized segments. And so it does, but

the short last piece improperly enriches the low end of the range. What randi( ) really wants to do is to rescale the range 1 . . . MAX to fit in the interval 1 . . . *u*. Rescaling calls for, not mod, but division, like this:

```
randi(u)
unsigned u;
{
  return(1+ rand0( )*u/MAX);
}
```

But there are problems implementing that. The main charm of these routines was that they worked with 16-bit integers. The expression *u* /MAX is a fraction; if we do rand0( )*u first, its result will exceed 16 bits.

Now, rand0( ) makes use of two constants: MAX0 and MAX. MAX0 is usually 2^w, while MAX = MAX0–1. In our original version, *w* was 15, but clearly it should be the word size of the computer, usually 16. Supposing that to be the case, the expression

```
rand0( )*u/MAX0
```

can be implemented cheaply enough by doing a 16-bit unsigned multiply and taking the most significant 16 bits of the 32-bit result. Picture it as using multiplication by *u* to squeeze out some of the high-order bits of rand0( ).

The right scaling factor is *u*/MAX not *u*/MAX0. But David Ross says that "to adjust *xu*/MAX0 to make it equal *xu*/MAX, just add one whenever *xu*/(MAX0*MAX) is greater than 0.5, *i.e.* when 2*xu* exceeds MAX0*MAX."

All these binary manipulations make it impractical to present the revised rand0( ) and randi( ) in any language but assembly language. Versions for the 8086 and Z80 follow. If we had versions for the 6xxxx machines, we'd print 'em and will if you send us some.

Considering QRNGs

The strongest impression we've gained from this exercise in QRNGs is of our own ignorance. A *lot* is known about the design of QRNGs, and most of it is strongly mathematical. The best source we know is Donald Knuth's *Seminumerical Algorithms*, pages 1–178. There oughta be a law that nobody can write about QRNGs until they pass a midterm on this material—in which case, this column would be on another topic even though we spent several hours recently poring over that chapter.

We learned quite a bit about linear congruential QRNGs, but the Talley-Metropolis routine is not one of those. Mathematically, it can be stated as:

$$X_n = (((X_{n-1} + X_{n-2}) \bmod \text{MAX0}) * 2) \bmod \text{MAX}$$

which is not identical to any of the QRNG formulae Knuth treats (although it bears an uncomfortable resemblance to the Fibonacci sequence, whose output Knuth says "is definitely *not* random").

We also learned that you should *never* base an application on a QRNG that hasn't been formally justified and thoroughly tested. Therefore you shouldn't rely on this one. We publish it because readers might enjoy working *on* it, as opposed to *with* it.

We also learned a lot about testing QRNGs; Knuth is particularly good on that subject. In particular, we know that you can spend about as much time as you like coding, running, and analyzing tests. The time we could spend on this project was severely limited. But here are what we and our correspondents have learned.

Period Length

A major point of interest is the length of the sequence of numbers the QRNG will produce. Michael Barr's approach to this was interesting and could be the basis of more extensive theoretical and practical work. "You can think of it as tracing an orbit among the sets of pairs of numbers (*a*,*b*) with *a* and *b* in the range 0 . . . MAX. And you can ask the question, what is the length and nature of that orbit?

"Clearly, (0,0) is an orbit of its own. In particular, the generator can never produce two zeroes in a row [*really? DEC*]. This is, by itself, a failure of randomness, perhaps not a serious one, but nonetheless a failure.

"When I tried it with MAX0 = 8, I discovered that the other 63 pairs were all in one orbit! This was surprising, but when I tried out other values of MAX0 (all even) I found that this is a fluke. For no other small value of MAX0 do all the nonzero pairs make up a single orbit. For example, for MAX0 = 6, there is an orbit of length 14, one of 16, and two of length 1—(0,0) as always and (4,4). As a matter of fact, if MAX0 is divisible by 6, there is always that second fixed point."

We'd been operating under the assumption that a generator that produced 16-bit numbers would have a maximum period of 65,536, but that proved not to be the case. We hacked together a version of what Knuth calls the collision test. This program cleared 64K bits (8K bytes) then called Rand0 256 times and set the selected bits to one. Then it began getting and counting Rand0 values and testing the bits they indexed. It counted successive hits of 1-bits.

We had presumed that, if the QRNG were working right, it would generate 64K numbers and then start repeating, hitting all 256 1-bits in a row; or it might hit them all again sooner, indicating a short sequence from the original seeds; or it might get into a loop hitting 0-bits and never return to the original 256 numbers.

Nothing of the sort happened. It behaved instead like a well-distributed generator with a much, much longer period. It made single hits at intervals of roughly 256 probes; it made double hits at intervals of roughly 64K; and when we stopped after 1.5 million probes, it had yet to hit three 1-bits in a row (intuitively, that should happen around 24 million), nor had it looped.

This is an intriguing result. A QRNG with such a long period that codes so tightly in a 16-bit machine is a rare bird. We only wish that somebody would analyze its period from a theoretical standpoint. It would be almost as nice if somebody would carry out a longer collision test.

A question related to period length is: What is a good initial seed? The QRNG is initialized with two numbers; what should they be for a maxi-

mal period? It's all very well to stick in two 16-bit prime numbers; if we had any theoretical understanding, we might be able to specify rules for the selection of good seeds.

Distribution

Of equal concern is the distribution. A few short tests have been made. Everett Carter generated 1000 numbers using randi(100) with the flawed randi( ) function. He subjected them to a Chi-squared test with 99 degrees of freedom; the result was a probability of 0.82 that the sample was drawn from a uniform distribution. Says Carter, "if I needed a uniform distribution, I'd try for better than 0.82."

Barr notes that, "provided MAX0 is even, it is possible to run the generator backwards. That is, if you think of it as producing a sequence X_1, X_2, ... X_n, you can calculate X_{n-2} from X_n and X_{n-1}."

Peter Pearson made the remarkable assertion that "sets of numbers drawn three at a time from rand0 are strongly interdependent. Specifically, if p, q, and r are the results of three successive calls on rand0, the quantity $2(p+q)-r$ can have one of only four values: 0, 32767, 65536, and 98303."

But on the other hand, Carter said, "On the correlation test, this generator does very well over short spans (I never tried long spans). This test measures the independence of the numbers by calculating the correlation between one number and a number later in the series. In the tests that I made with a lag distance of up to 20 intervals, the correlations are below 0.1 and most are well under 0.05. While one might hope for more independence, these values are very good relatively."

Clearly many more tests of this QRNG should be done. Knuth recommends the runs test and the collision test as particularly powerful and describes algorithms for them and several others.

The Mitchell-Moore QRNG

There is another simple QRNG, one that has guaranteed good characteristics and a ridiculously long period. Its only drawback is that it needs 55 words of auxiliary space and a second QRNG to seed it. Knuth credits the method to J. G. Mitchell and D. P. Moore; its formula is:

$$X_n = (X_{n-24} + X_{n-55}) \bmod \text{MAX0}$$

It is easily expressed in C; see Listing Four (page 27). It uses a 55-integer array initialized with numbers not all of which are even; usually these 55 seeds are drawn from a simpler QRNG (Talley-Metropolis will do). The constants 24 and 55 are not arbitrary; they were chosen because they produce a period of length 2^{55}. Nor are they unique; Knuth gives a table of other good pairs. Use 27 and 98 if you think you need a period of 2^{98}.

Random 8086

The routines in 8086 assembly language may be found in Listing Two (page 24). The seeding routine is omitted, as are details of segment definitions. Pay close attention to the consumer warning in the comments. It was Michael Barr who spotted the elegant equivalence between the C statements

```
c *= 2; if (c>MAX) c -= MAX;
```

and the single 8086 instruction

```
rol ax,1
```

Boy, if you ever see a C optimizer that can find that transformation, buy it.

Random Z80

The QRNG in Z80 assembly language is in Listing Three (page 24). The most complicated part of the Z80 (or any other 8-bit) version is the simulation of a 16-bit unsigned mul-

Dr. Dobb's Clinic

(Text begins on page 20)

Listing One

```
rem recurse.bat called with %1 %2 %3 %4 %5 %6 %7 %8 %9
if (%1)==() exit
prompt command level %1$g
command /c recurse %2 %3 %4 %5 %6 %7 %8 %9
```

End Listing One

Listing Two

```
; Talley/Metropolis routines for 8086.  Consumer Warning: This
; generator is experimental and lacks a sound mathematical
; basis or statistical validation -- USE AT YOUR OWN RISK.
;
; Rand0: return a 16-bit quasi-random number in AX.
;       Preserves all registers except AX, flags.
Rand0:
        push    dx
        mov     dx,Xminus2
        mov     ax,Xminus1
        mov     Xminus2,ax
        add     ax,dx
        rol     ax,1
        mov     Xminus1,ax
        pop     dx
        ret
;
; RandI: return in AX a number in 0..CX-1.
;       Preserves all except AX, flags.
RandI:
        push    dx
        call    Rand0     ; ax = rand0()
        mul     cx        ; dxax = u*rand0()
        mov     ax,dx     ; ax = u*rand0()/MAX0
        pop     dx
        test    ah,80h
        jz      RandIx
        inc     ax        ; ax = u*rand0()/MAX
RandIx: ret
```

End Listing Two

tiply. The one in Listing Three is a standard routine straight out of that classic work, *Z80 Programming for Logic Design* by Osborne, Kane, Rector, and Jacobson, published in the distant year of 1978 by Osborne and Associates and still one of our most useful references.

The only change we've made in Osborne's multiply is to use long jumps (JP opcodes) instead of short relative ones. A JP takes 10 clock cycles to do regardless, while a JR takes either 7 ticks to fall through or 12 to jump. Ordinarily, a JR is the opcode of choice because it is a byte shorter and, under the assumption that it will fall through and be taken with equal frequency, should consume half a clock cycle less on average.

However, if we can predict anything about the frequency with which a branch will be taken, we should make a choice between jump types. The last JP in Listing Three, for instance, will be taken 15 times and fall through but once. Clearly it ought to be a JP, since a JR would cost 30 more cycles.

The first of the three jumps should be a JR. It tests successive bits of the multiplier. In this application, those are bits of a quasi-random number, and the 50-50 assumption certainly ought to hold, indicating the use of JR. In a general-purpose multiply routine, we might expect that both multiplier and multiplicand will be positive binary numbers, but even if we expect only positive numbers less than 2^{15}—so that we expect the jump to be taken 9 of 16 times (i.e., always taken for the first two bits and taken for half of the other 14 bits)—the JR still squeaks out two-tenths of a clock cycle faster than a JP.

The middle jump is not so obvious, but because it will be executed in any manner on half or fewer of the iterations, it is also less important. In effect, we choose here between a 5-cycle JP or a 3.5/6 cycle JR. The jump will be taken when no carry results from adding the multiplier into the low word of the partial product. At first we thought that would be 3 out of 4 times, a clear win for JP over JR; on reflection, we ain't so sure. What do you think?

DDJ

Listing Three

```
; Talley/Metropolis routines for Z80.  Consumer Warning: This
; generator is experimental and lacks a sound mathematical
; basis or statistical validation -- USE AT YOUR OWN RISK.
;
; Rand0: return a 16-bit quasi-random number in HL.
;       Preserves all except HL, flags.
Rand0:
        push    de
        ld      de,(Xminus2)
        ld      hl,(Xminus1)
        ld      (Xminus2),hl
        add     hl,de
        add     hl,hl
        jr      nc,Rand0q
        inc     hl
Rand0q: ld      (Xminus1),hl
        pop     de
        ret
;
; RandI: return in HL a number in 0..BC-1.
;       Preserves all except HL, flags.

;       Requires Mul16 to do DEHL = BC*HL
RandI:
        call    Rand0   ; HL = rand0()
        push    de
        call    Mul16   ; DEHL = BC*rand0()
        ex      de,hl   ; HL = BC*rand0()/MAX0
        pop     de
        bit     0,h
        ret     z
        inc     hl      ; adjust to .../MAX
        ret

; a usable 16-to-32-bit multiply subroutine, producing
; DEHL as the 32-bit result of HL * BC, preserves BC, AF.
Mul16:
        push    af
        ld      a,16    ; loop count
        ex      de,hl   ; multiplier to DE
        ld      hl,0    ; clear partial product
Mul16A:
        ex      de,hl   ; next multiplier bit
        add     hl,hl   ; ..shifted to carry
        ex      de,hl   ; ..and 0-bit appended to hi prod
        push    af      ; (save m'ier bit)
        add     hl,hl   ; next product bit to carry
        jr      nc,Mul16B
        inc     de      ; adjust bit shifted earlier
Mul16B: pop     af      ; if m'ier bit was 1,
        jr      nc,Mul16C
        add     hl,bc   ; add multiplicand to product
        jp      nc,Mul16C
        inc     de      ; ..propogating carry
Mul16C: dec     a
        jp      nz,Mul16A
        pop     af
        ret
```

End Listing Three

(Continued on page 83)

C Compilers for MSDOS

by the C Special Interest Group of PicoNet and the Silicon Valley Computer Society under the direction of Richard Relph

Our reviewers found that some of the fastest compilers overall were relatively slow in doing screen I/O, and uncovered order-of-magnitude differences in execution time and generated-code size.

What would you consider the ideal credentials for a reviewer of computer language compilers? Would you want a software designer, or perhaps even a compiler designer? Should the reviewer be a professional who earns his or her living at the keyboard? Or should the review be conducted by the weekend programmer who really understands the value of user-friendly software as well as the frustration of user-hostile documentation?

The reviewer for this project is all of the above. This report on 13 C compilers for MSDOS was conducted by the joint C Special Interest Group (C-SIG) of PicoNet (a non-profit, general microcomputer users' group) and the Silicon Valley Computer Society (SVCS), an IBM PC users' group.

This combined C-SIG is composed of a broad cross-section of microcomputer users. Some are language-compiler designers, some design and maintain systems software, still others make their living developing commercial software such as spreadsheet or data base packages. There are several novice programmers in the group, but most are seasoned programmers, if not expert software designers. The "reviewer," therefore, is well qualified for the job.

Background

The first C-SIG was organized in 1983 by a group of PicoNet members. Their purpose was to provide a forum for discussion and study for those interested in the C programming language. In the beginning, classes were organized, and after 10 or 12 sessions the language was covered fairly thoroughly. The classes brought all members of the group to a common basic level of understanding of C.

The group soon graduated to developing software utilities: those tools that are commonly found on the larger systems for which C was originally developed, but that are not commonly available for the microcomputer environment. The C-SIG also organized group purchases of compilers, other software packages, and hardware at discounts, so that all members could own a quality C compiler, participate in the various group activities, and contribute to group projects.

Invariably, at the monthly meetings someone would ask what was the best current compiler (or version thereof), opening a debate over which compilers were the fastest, which produced the optimum code, which conformed most to "standard" C. We soon recognized from these discussions that the group was somewhat inbred; that is, among the 20 or 30 members, only two or three C compilers were represented (partly due to group purchases). Then, near the beginning of 1985, the PicoNet and SVCS C-SIGs merged.

SVCS was made up of IBM PC users. PicoNet was traditionally a CP/M group, but a large number of the original C-SIG members had recently acquired MSDOS-based PCs through a group purchase. Out of the group's new interest in MSDOS came increased interest in C compilers that ran under MSDOS. We soon discovered that the number of compilers available for the PC environment

was large indeed, and we quickly identified some 24 implementations. The questions of which was fastest, which produced the tightest code, and which was closest to the standard continued to come up, but we learned that no one had the answers. We decided to find them.

Given the magnitude of the task, the breadth of the group, and the number of individuals involved in the project, responsibilities had to be divided among the participating members. Each joined a group that was assigned a specific task critical to the overall project. The groups had responsibilities in the following areas:

- Standards: responsible for the evaluation of compilers' and libraries' adherence to the standard C language as defined by Kernighan & Ritchie (K&R) in *The C Programming Language*. Also responsible for qualifying all benchmarks written or acquired to measure compiler performance for universal compatibility and for assessing each compiler's ease of use (Richard Relph, Don Lawrence, Andi Pisiali, Stan Peters).
- Benchmarks: responsible for writing and/or acquiring C compiler benchmark programs and for conducting benchmark tests (Fred Viles, Steve Mahn, Scott Thomas, *George Giomousis*, Bill Overstreet, Fletcher Johnson).
- Documentation: responsible for reviewing documentation for clarity, completeness, style, and presentation. Also responsible for evaluating customer service for responsiveness and technical proficiency (Chuck Rulofson, Don Hockler, Bob Stephan, Peter Vutz).
- Writers: to put it all together (Irwin Diehl, Don Dodge).

That's the background—who we are, what we did, why we did it, and how. Oh yes, we should mention the "where." We've been most fortunate to have been provided a regular meeting place by ZeroOne Systems (formerly Technology Development of California). ZeroOne has been a most gracious host for the regular monthly meetings and the special weekend benchmark sessions. Their continuing support is appreciated.

Benchmarks

The benchmark group was charged with measuring the performance of the code produced by the various compilers. In the course of several meetings, we decided on our general approach and on the specific benchmark functions to be produced. We first discussed the possibility of producing a real-world benchmark program, one that would perform some useful function using an appropriate mix of language and library features. This plan, although attractive, was problematic since we could not agree on an appropriate mix. We settled on a "surgical" approach: many small functions, each intended to test a single feature of the compiler or its library.

We started by drawing up a list of every aspect of compiler and library performance that any one of us had ever wanted to see in other compiler reviews; not surprisingly, the list was enormous and impossible. In the interest of completing this project at all, we pared the list down, ending with 30 separate benchmarks to test performance in the areas of code size; time to compile and link; storage class support; math; array and pointer referencing; function calling; optimization; screen and disk I/O; and general performance. We ran some of these functions for different memory models and some with the 8087 options available.

To get accurate and consistent results, we used a common format for all the execution-time benchmarks. Each benchmark was written as a function that accepted a loop count and returned the measured time in tenths of seconds. We wrote a pair of functions to measure an interval of time using the system clock, avoiding the problems inherent in the stopwatch approach. All compilers except Digital Research's provided a library function giving access to DOS interrupts, so the timer functions could be written easily in C for all but DRI. (So we wrote an assembly routine.) We then verified that the overhead in the timer functions was negligible for all compilers. The skeleton for all benchmark functions is in the Figure (below).

We adjusted each loop count to force the benchmark into a reasonable time frame (about 1 minute), so as to minimize the percent error in the timer functions. Although the system-clock interrupt returns a measurement of the time down to hundredths of a second, the clock is actually updated only 18 times per second, so the time reported for a given function can vary by as much as 0.2 seconds.

Except for the 8087 benchmarks, all the tests were performed on a Leading Edge PC. running MSDOS 2.11 and equipped with 640K RAM and a 10 Mb hard disk. These machines are close PC-compatibles running at a clock rate of 7.16 MHz; consequently, CPU-bound benchmarks will take 50 percent longer on an IBM PC. The Norton Utilities (V3) confirm this performance factor.

We selected the Leading Edge PC because we had made a group buy of several identically configured systems. We ran the compile-time and link-time benchmarks on the same machine in two different ways: using the hard disk, with care taken to ensure that files were read and written to the same location for each compiler, and on a 512K JDrive RAM disk. We ran the disk I/O benchmarks on the hard disk, again ensuring that the same disk loca-

```
benchfn(loop)
int loop;
{
    /* Initialization */

    time_0( ); /* Start clock */
    for ( ; loop ; loop-- )
    {
        /* Code to be timed */
    }
    return(time_n( )); /* As sec/10 */
}
```

Figure

tions were used. Before you moan "but I only have a plain IBM PC with floppies," remember that the goal here was to compare the compilers, not the machines. The ratio of the timings should be the same anywhere.

Benchmark Design

So much for background. Now we will discuss the benchmark functions, grouped by aspect to be tested. We have listings for all but the compile- and link-time programs, which were not actually executed. For a diskette (5¼-inch MSDOS) with the listings send a check for $6.00 payable to M&T Publishing to: *DDJ* Compiler Review, 2464 Embarcadero Way, Palo Alto, CA 94303.

Code Size

We wanted to get an idea of the granularity of the library and the minimum .EXE file size that could be produced. Granularity, for those not familiar with the term, refers to how finely the library is structured. A highly granular library allows the bare minimum of object code to be linked for each function. Four programs were written for these tests:

- minmain.c—Main program with no executable code.
- minputs.c—Main program with one puts( ) call.
- minprtf.c—Main program with one printf( ) call.
- minfio.c—Main program with calls on fopen( ), fgets( ), fputs( ), and fclose( ).

We compiled and linked, but did not execute, each program. We were especially interested in minprtf.c, since using printf( ) usually means dragging in the whole floating-point package. Some compilers provided an option to link with an integer-only version of printf( ). In those cases we linked minprtf.c twice, once with the integer-only and once for the full floating-point versions of printf( ). See Table 1 (page 34) for the results.

Clocking Compile and Link Time

Since most of the performance benchmarks are very short and not representative of typical programs, we used separate programs ranging from 1 to 1000 lines for this test. They were:

- doc1.c—A one-liner, to show the minimum overhead.
- doc10.c—A 10-liner, about the smallest module you'd ever compile.
- doc100.c—This 100-liner is the most significant since it represents a typical size of source file.
- doc1000.c—A 1000-line program. Basic overhead should be minimal here.

Doc100.c was produced from a preexisting program belonging to one of the group. Doc1000.c is a stripped-down version of the DIFF program available from DECUS (Digital Equipment Corp. Users' Society).

Since modular compilation tends to be the rule with C programming, we timed the compile and link steps separately. The compile step included all phases of the compilation, including assembly, if any. The compile and link process was controlled by a batch file since we weren't interested in benchmarking the testers' typing speeds. Some compilers provided a control program that allowed the linker to be run automatically after the compile with a single command. In these compilers, we defeated that option and issued a second command to do the link so we could derive separate compile and link times. To produce the time estimate, we used the Norton Utilities timemark program, invoking it from the batch file at the appropriate points. See Tables 2a and 2b (page 34) for these results.

Testing Screen and Disk I/O

We wanted to test how fast the generated code could write to the console using puts( ) and printf( ), as well as how fast it could read and write to the disk. We wrote six functions to find out.

- The fillscr.c function used puts( ) to write a string of 1248 characters to the screen without scrolling. Scrolling was prevented by using a bare carriage return every 78th character of the string. The screen gradually fills because puts( ) adds its own newline at the end of the string, and the puts( ) call is repeated <loop> times. The screen was cleared by the DOS CLS command prior to running this test.
- The scroll.c function was almost the same as fillscr( ), except the string had a newline every 78th character so each puts( ) produced 16 lines of characters plus a blank line on the screen. So the time difference between scroll( ) and fillscr( ) measured the scrolling overhead.
- The prtf.c function produced the same number of characters and lines as scroll( ), except it used printf( ) instead of puts( ) and executed each basic format conversion several times. Comparing the time for prtf( ) to that for scroll( ) shows how efficiently the compiler's printf( ) function performs its conversion duties.
- The cpychr.c function copied a 10,000-byte file using fgetc( ) and fputc( ).
- The cpyblk.c function copied a 10,000-byte file using fread( ) and fwrite( ) to copy in 1024-byte blocks. The difference between the times for these functions reveals the advantage blocked I/O has in a particular compiler's library.
- The diskio.c function was designed to test random character I/O to a reasonably large data file. It used fseek( ), fgetc( ), and fputc( ) to randomly read and write individual characters in a previously created data file 240,000 bytes long.

We argued about whether to use fopen( ), fread( ), fwrite( ), and the like, rather than open( ), read( ), write( ), and other corresponding functions. The latter functions were provided with many of the compilers and might have produced faster times, but ultimately we decided to use the f functions, primarily on portability considerations. The February 1985 preliminary draft of the proposed ANSI standard does not include open( ), et al., on the grounds that they are too closely tied to the Unix

Vendor:	minmain	minputs	minprtf	minfio	DOC1	DOC10	DOC100	DOC100
Aztec (Manx)	1856	3136	**4336**	**3952**	1600	4416	**6784**	**10928**
Control C	6698	6714	6714	7846	6698	6794	12198	14352
C Systems	12032	12032	19456	12544	11934	19440	25018	28464
Computer Innovations	4352	4480	9344	5632	9408	9408	12789	16956
Datalight	3386	3603	5644	**3927**	2020	7732	12148	NA
DeSmet	1536	1536	6656	8192	1536	6656	14848	19456
Digital Research	12800	13440	18816	15616	12800	18944	24704	28416
EcoSoft	**692**	**2452**	7616	4026	**708**	**770**	11528	NA
Lattice	10278	10306	11702	10812	10294	11786	14654	19688
Mark Williams	6698	6714	6714	7846	6698	6794	12198	14352
Microsoft	1912	4296	5750	5896	1928	5802	9090	14684
Software Toolworks	3918	5286	6850	7422	8556	10964	14180	22192
Wizard	**716**	**2440**	8168	4278	**732**	8236	12234	16336

Table 1
.EXE File Size

Vendor:	DOC1		DOC10		DOC100		DOC1000	
	compile	link	compile	link	compile	link	compile	link
Aztec (Manx)	**2**	**2**	3.5	**3**	16.5	**4**	103	5.5
Control C	**2**	12	**3**	12	12	19	69	21
C Systems	6	11	9	14	39.5	17.5	233	19
Computer Innovations	6	10	7	14	19	16	105	20
Datalight	2.5	3	**3**	5.5	12	9	NA	NA
DeSmet	3	3.5	**3**	4	**6**	**4.5**	**28**	**5**
Digital Research	3	16	4	20	17	25	120	29
EcoSoft	3	3	4	13	13	18.5	NA	NA
Lattice	2.5	9.5	4	10	18.5	11.5	126	15.5
Mark Williams	**2**	12	**3**	12	12	19	69	21
Microsoft	4	3	6	5.5	20	7	131.5	9.5
Software Toolworks	3	11	6	13	28	15	243.5	20
Wizard	3	**2**	**3**	7	10	9	54	12.5

Table 2a
Compile and Link Times: RAM Compilation

Vendor:	DOC1		DOC10		DOC100		DOC1000	
	compile	link	compile	link	compile	link	compile	link
Aztec (Manx)	**8**	**5.6**	9.5	**8**	24.5	**10**	116	**12**
Control C	13	20	14	20	26.5	27.5	93	30
C Systems	17	18	20	22	52	25	255	27
Computer Innovations	21.5	15	22	20	37	23	128	28
Datalight	8.5	7	9	11.5	20.5	15	NA	NA
DeSmet	10	7	10	8.5	**14.5**	**10**	**43**	**11**
Digital Research	13	24	14	29	28	36	135	40
EcoSoft	14	7	15	20.5	27	26	NA	NA
Lattice	9	15.5	10	16	27.5	18	149	22.5
Mark Williams	13	20	14	20	26.5	27.5	93	30
Microsoft	19	8	21.5	13	38	15	159.5	20
Software Toolworks	9	18	11.5	21	35	24	254	30
Wizard	15	5.5	17	13	27	16	78	20

Table 2b
Compiler and Link Times: Hard-disk Compilation

Vendor:	Overall performance and misc.:						
	looptst	optim	fibtest	sieve	rsieve	nsieve	isort
Aztec (Manx)	37.8	35.7	56.2	**99.0**	58.0	53.3	42.0
Control C	62.2	40.9	63.0	103.4	59.1	54.8	60.0
C Systems	100.8	73.1	69.7	173.8	132.6	190.7	51.5
Computer Innovations	66.1	39.8	54.0	117.5	117.4	119.3	49.0
Datalight	63.0	35.3	**49.8**	**99.9**	100.0	101.3	42.6
DeSmet	63.0	46.4	52.9	108.2	108.4	103.4	53.8
Digital Research	58.8	66.9	51.7	104.8	58.8	57.0	49.9
EcoSoft	63.0	50.7	52.0	115.3	115.4	138.1	85.7
Lattice	63.0	**25.0**	70.0	100.3	100.2	96.9	77.2
Mark Williams	61.8	40.8	63.0	103.3	59.0	54.7	59.7
Microsoft	58.8	59.0	59.0	99.4	54.9	**50.0**	44.3
Software Toolworks	74.2	31.8	56.4	133.6	110.3	176.8	123.6
Wizard	58.8	68.8	52.6	101.0	**55.7**	51.1	60.0

Table 3a
Execution Times

operating system and that they are redundant. We dodged the potentially sticky problem of conversion between newline and CR-LF by making sure that none of the data files contained these characters.

Testing String Library Functions

We wanted to test some library functions beyond the usual I/O functions. We were concerned, though, that many of the functions we could test would not be provided with all of the compilers, so we wound up with a single benchmark function, isort.c, to test strcmp( ) and strcpy( ). This function used strcmp( ) and strcpy( ) to perform a simple sort of an array of up to 1024 eight-character strings. (In this case, the <loop> parameter was used to control the number of elements to sort; it does not represent the number of sorts performed.) The isort.c test tells little about compiler performance per se, but it does give a clue as to how well the runtime library was written.

Math Tests

We wrote four functions to test basic arithmetic functions for various data types.

- The tint.c function includes 1500 add, 1600 subtract, 200 multiply, and 200 divide operations, <loop> times. Inner loop overhead amounts to <loop> * 100 test-for-zero and decrement operations. Since there are not many ways to perform 16-bit integer arithmetic on an 8088, we expected most of the compilers to perform about equally on this one.
- The tlong.c function was exactly the same as tint( ), except that the variables involved in the adds, subtracts, multiplies, and divides were all declared long instead of int. Significant differences between compilers were more likely here.
- The tfloat.c function tested floating-point arithmetic with 40 each adds, subtracts, and multiplies and 20 divides, all done <loop> times.
- The tdouble.c function was the same, except that the variables were declared double instead of float. For most compilers, tdouble( ) should be somewhat faster than tfloat( ) since K&R calls for all floats to be converted to double before being used in an expression. Tfloat( ) and tdouble( ) were run with whatever flavors of floating-point support (including 8087 options) were available for each compiler. The results of these tests appear in Table 3b (page 38) and Table 4 (page 43).

Array and Pointer Tests

We wrote two functions to explore how efficiently the compilers handled array- and pointer-referencing tasks. The function array.c simply initialized a three-dimensional array of 1000 ints, and pointer.c did the same, but using a pointer to a pointer to a pointer to an int. The unnecessary extra levels of indirection were added to increase the amount of pointer dereferencing relative to the overhead; i.e., to increase the ratio of data to noise.

Function Call Tests

To test the overhead due to function call and return, we

Vendor:	Math functions:				Array/ptr:
	tint	tlong	array	ptr	
Aztec (Manx)	**34.0**	125.4	**30.7**	**50.4**	
Control C	36.4	176.1	96.4	72.5	
C Systems	39.3	247.9	90.0	85.0	
Computer Innovations	36.9	215.8	95.1	71.9	
Datalight	35.4	**87.7**	61.4	65.2	
DeSmet	36.0	178.9	93.6	68.0	
Digital Research	**34.1**	808.4	96.8	68.0	
EcoSoft	36.0	298.8	91.7	68.0	
Lattice	36.3	192.8	60.2	66.8	
Mark Williams	36.7	175.4	96.4	72.5	
Microsoft	**33.5**	102.9	76.2	**50.4**	
Software Toolworks	59.0	N.A.	131.5	116.8	
Wizard	**34.0**	173.0	78.4	**50.4**	

Vendor:	Screen I/O:		Disk I/O:				Storage class support:		
	fillscr	scroll	prtf	cpyblk	cpychr	diskio	autost	stattst	regtst
Aztec (Manx)	50.5	57.8	61.7	98.2	**39.7**	62.8	**70.6**	**79.3**	**54.2**
Control C	**22.3**	**29.0**	37.6	104.2	59.5	47.6	72.3	81.1	**54.1**
C Systems	31.0	35.5	42.1	182.6	82.2	**26.7**	126.9	136.0	108.8
Computer Innovations	27.8	32.6	58.1	147.0	**42.0**	50.9	79.3	86.6	79.3
Datalight	49.9	54.7	63.9	123.6	53.3	N.A.	79.4	86.5	79.4
DeSmet	27.8	31.5	48.1	128.8	94.6	48.1	79.4	86.5	79.4
Digital Research	37.5	41.9	**35.9**	176.0	83.3	36.2	**70.5**	**79.3**	**52.2**
EcoSoft	31.3	37.8	40.8	N.A.	N.A.	N.A.	79.4	86.5	79.4
Lattice	62.4	67.5	71.0	129.0	63.5	66.1	79.3	86.5	79.4
Mark Williams	**22.3**	**29.0**	37.5	156.1	85.0	42.4	72.2	81.1	**54.0**
Microsoft	47.7	55.0	59.1	**73.0**	53.0	57.0	**70.5**	**79.3**	**52.2**
Software Toolworks	62.5	67.6	67.9	726.0	334.2	55.3	112.6	136.0	112.5
Wizard	49.7	54.7	65.5	139.7	60.4	55.5	**70.6**	**79.3**	**52.3**

Table 3b
Execution Times continued

created a version (fibtest.c) of the fib benchmark from the August 1983 issue of *Byte* magazine. (This is a good test of function-call overhead because it's composed entirely of negligible arithmetic and recursive calls.) As with tint( ), the optimum code for function calls and returns is well known, so we did not expect to see much variation between compilers on this test, although significant differences would be interesting.

Storage Class Tests

We were curious about the efficiency of the code generated for variables of various storage classes and in particular about how well register variables were supported. To find out, we wrote three almost-identical functions: autotst.c, stattst.c, and regtest.c. Each contained four empty **for** loops. The only difference among the functions is that the four loop variables were declared with automatic, static, and register storage classes, respectively.

Looking For Optimization

We thought it would be a nice idea to try to determine just how extensive each compiler's optimization algorithms were. To that end we wrote optimize.c, a mishmash of statements that a compiler could reasonably be expected to optimize. The idea was that the more cases a compiler noticed and optimized, the better its performance on the test would be.

Assessing General Performance

The looptst.c function did nothing at all; it was just an empty loop. We included it so that we could get an idea of how much overhead was being introduced by our outer loop in each of the benchmarks. Looptst( )'s inner loop did nothing 10,000 times, so there were 5,000,000 total loop iterations.

We knew we could not get away with not running the infamous *Byte* sieve benchmark, so we decided to go overboard and produce three, starting with sieve.c, our version of the standard sieve benchmark as published in the January 1983 *Byte*. We added rsieve.c, identical to sieve( ) except that the two most frequently referenced variables were declared register. This is the way some vendors run the sieve when producing numbers for their advertising copy. And we wrote psieve.c, a pointerized version of the sieve. This function may not prove very useful in comparing compilers, but writing it was an interesting exercise. It showed us that it is not true that using pointers is always better than using array references. Psieve( ) is considerably less readable and performs only marginally better than rsieve( )—on some compilers it actually ran at a slower rate!

Testing Memory Models

All the benchmarks mentioned were compiled and run under the small-memory model. To see what kind of performance penalty you pay to use other models, we compiled and ran array( ), pointer( ), and psieve( ) with every other memory model option available with each compiler. In retrospect, we probably should have included fibtest( ) and maybe sieve( ) in the set.

Benchmark Results

You can see the results for all these benchmarks in Table 3a (page 34) and Table 3b (page 38). Since there are some surprises and discoveries that may not jump out at you from all those numbers, we want to discuss some of the more interesting (in some cases, bizarre) results. We'll discuss them by functional grouping.

In the area of screen I/O, none of the compilers was so fast as to indicate that the library routines were writing directly to screen memory (which could be a portability problem), but there was a wide range in performance. The compilers seemed to separate into two groups. Mark Williams' compiler was the fastest in a group clustered around 30 seconds, with MicroSoft's the fastest in a group clustered around 55 seconds. This suggests the use of two divergent methods of handling screen I/O under MSDOS.

Also, we were surprised to find that for most compilers, prtf( ) was not much slower than scroll( ). We had expected the conversion overhead to be significant, but this was true only with Computer Innovations' compiler and, to a lesser extent, with DeSmet's. A real anomaly was Digital Research's compiler, whose prtf( ) time was faster than scroll( ) and even faster then fillscr( )! That means that in DRI's compiler there is some overhead associated with using puts( ) that is avoided in using printf( ); this made no sense to us.

The storage class tests basically tested the efficiency of loading data from memory. Since there are only a limited number of ways to load a two-byte quantity from memory, it is not surprising that there were many-way ties in this category. In fact, except for the two slowest compilers (c-systems' and Software Toolworks'), all the compilers were closely ranked for all three tests. There were four compilers tied for first, followed by two more just a couple of seconds slower and five more tied a bit further off. Mark Williams' and Control C's compilers were the two tied in the second group, but that's no surprise; they tied in every test. They are really the same compiler.

If we compare the regtest( ) and autotst( ) results, it becomes clear that all the faster compilers implemented register variables, while generally the slower compilers did not. The c-systems compiler, the slowest for these tests, did implement register variables, but apparently to no great advantage. Out of curiosity, we also ran a version of regtest( ) that declared only two of the four loop variables to be of register class. For all compilers, the results were the same as for regtest( ). This was educational; it means that, at most, only the first two variables declared were actually kept in registers. This has implications for programming practice: use register declarations, but take care that variables be declared with the most frequently used first.

For all compilers, the automatic storage class is more efficient than the static. This may come as a surprise to those accustomed to C on 8-bit machines with no stack-relative addressing mode.

Most compilers performed about the same on the looptst( )benchmark—about 60 seconds. The Aztec compiler, however, was significantly faster at 38 seconds. It is

unusual that no one else came close to this time, since most benchmarks had several compilers within a few seconds of the fastest time. It turned out that the Aztec compiler generated only two instructions for the empty inner loop, whereas the best the others could do was three instructions.

The Aztec compiler repeated this performance on the array( ) test, coming in twice as fast as the nearest competitor. Here Aztec's performance was due to avoidance of the **imul** instruction in doing subscript arithmetic. In general, the Aztec compiler avoided **imul** in favor of shifts and adds wherever possible. The Aztec compiler also shared the lead in the pointer( ) test with MicroSoft's and Wizard's.

There were no surprises in the tint( ) test. Except for Software Toolworks' C, all the compilers were in a very narrow range, clustered around 35 seconds. There are not many ways to add or subtract two 16-bit integers on a 16-bit machine, and we did not use register declarations in tint( ), which would have given a slight edge to the compilers that support them.

Tlong( ) is another story altogether. The Datalight compiler had its only win here (by a significant margin) at 88 seconds, followed by MicroSoft's and then the Aztec compiler. The real standout in this benchmark was Digital Research's compiler, which came in last by an astounding margin with a benchmark of 808 seconds! It is hard to imagine how it could execute this slowly, over nine times as long as the fastest time. It is especially puzzling because the DRI compiler was a respectable performer otherwise, in fact fastest or essentially tied for fastest on a number of the other tests. If you use longs very much, DRI's is clearly not the compiler of choice!

The results of the optimize( ) benchmark are difficult to interpret. Some of the "lightweight" compilers ran this test quite well, while MicroSoft's did rather poorly, even though we had observed that its optimization was generally very good. Also, the Lattice compiler was fastest by a large margin on this test but was not an outstanding performer on any of the other benchmarks. We looked at the code generated by some of the compilers for a clue and discovered a partial answer. Optimize( ) was especially unkind to those compilers that do not recognize that multiply operations can be replaced by left shift instructions. One saved **imul** instruction can make up for a lot of redundant load and store operations! Actually, this kind of test should always be approached with caution. One of these days a compiler may come along that is so good at global optimizations that it notices when our benchmark functions produce no side effects visible outside the function. It might then be free to replace an entire function with a single RET—very fast indeed!

The disk I/O benchmarks were really testing part of the library, not the compiler, so we expected a wide range of numbers. What we did not expect was that for many of the compilers, cpychr( ) would be faster than cpyblk( ). When you look at Table 3b, don't overlook the loop count: cpyblk( ) moved twice as many bytes as cpychr( ). The MicroSoft compiler had the fastest time for cpyblk( ), with no other compiler coming close. The Aztec compiler was fastest for cpychr( ), with Computer Innovations' the only close second. The Software Toolworks compiler was abysmally slow on both of these, about four times slower than the generally poor-performing compiler from c-systems. On the other hand, the Toolworks compiler gave a reasonable time for diskio( ), while the fastest time, by a wide margin, was turned in by c-systems' compiler. In fact, the compilers that generally did the best on the various compile-time benchmarks (those from Aztec, MicroSoft, and Wizard) were among the slowest on the random-access diskio( ) test. You figure it out!

The min–( ) benchmarks were intended to test the granularity of the library: we wanted to see how much unwanted "stuff" would be dragged into the .EXE file along with the library functions actually being called. Minmain( ) contains no function calls at all, so the .EXE file size for this test is an indication of the best you can do. The DRI, c-systems, and Lattice compilers were the big losers here, all requiring more than 10,000 bytes of library code to do nothing at all. Not surprisingly, these three were also the worst on the minputs( ), minprtf( ), and minfio( ) tests, although the Lattice compiler's .EXE size remained about the same. The EcoSoft and Wizard compilers both produced amazingly small .EXE files for minmain( ), since the startup code should at least initialize the stdin, stdout, and stderr files. DeSmet's, in third place at 1536 bytes, already includes all the library code needed to support the puts( ) function. That made DeSmet's the smallest .EXE file for minputs( ) by a fair margin, although Wizard's and EcoSoft's were also very small.

Minprtf( ) was the acid test, because the formatting supported by printf( ) usually requires the floating-point package as well as I/O support. Some of the compilers provide an integer-only version of printf( ), however, and we used that option whenever possible since we believe the majority of C applications do not require floating-point support. The compilers that provide this option are giving you more flexibility with an effectively more granular library. They are the compilers from Aztec, Mark Williams (and Control C), Lattice, MicroSoft, and Datalight.

The MicroSoft compiler deserves special mention among these. The other compilers made you specify whether floating-point support was desired, either with a command-line option or via library-search order. MicroSoft's handled this automatically by detecting whether any float or double variables or pointers had been declared in your program, and then linking the math library only if needed. We found this sort of feature to be typical of the Microsoft product.

Using the floating-point version of printf( ) typically added from 2000 to 3000 bytes to the .EXE file size. Choosing the compile option to leave the float library out, Aztec C produced the smallest .EXE file by a large margin, even over the integer-only competitors. Datalight's and MicroSoft's came next and then DeSmet's. The DeSmet compiler produced the smallest .EXE file with the floating-point library included, although the Aztec file was only a few hundred bytes larger. Also, it should be noted that the Software Toolworks compiler is a subset

Floating-point and 8087 Results

Vendor	tflt	tdbl	EXE size	8087	inline	Notes
8087 series:						
Aztec (Manx)	119.1	141.0	7584	Y	N	m87.lib
Aztec (Manx)	149.1	168.5	10000	N	N	m87s.lib
c-systems	145.0	153.7	22424	Y	N	
c-systems	59.9	68.3	20956	N	Y	
Computer Innovations	59.2	68.3	10027	N	Y	
DeSmet	119.0	128.3	12288	N	N	stdio7.s
Digital Research	67.3	76.5	22528	N	Y	
Ecosoft	290.5	161.7	10828	Y	N	
Lattice	1675.8	1256.0	17452	Y	N	didn't use 8087
Mark Williams	53.7	62.9	9632	N	Y?	-rndp -vsmall
Microsoft	81.0	89.7	21172	Y	N	/fpc
Microsoft	81.0	89.6	14148	N	N	/fpc87
Microsoft	55.7	64.2	20932	Y	Y	/fpi
Microsoft	55.6	64.2	13908	N	Y	/fpi87
Wizard	350.0	209.0	11788	Y	N	
Wizard	368.7	72.0	11916	N	Y	-f
Floating-point series:						
Aztec (Manx)	78.8	81.0	9136	N		m87.lib
Aztec (Manx)	80.6	83.2	10000	Y		m87s.lib
c-systems	95.1	78.6	22424	Y		
Computer Innovations	173.1	166.0	11453	N		
Datalight	121.8	71.1	10068	N		
DeSmet	97.8	79.2	14848	N		
Digital Research	83.6	61.7	22016	N		
EcoSoft	129.6	105.8	10878	Y		
Lattice	112.0	84.2	17452	Y?		
Mark Williams	98.8	94.3	13137	N		
Microsoft	91.7	101.9	21172	Y		/fpc
Microsoft	25.6	45.6	16580	N		/fpa
Microsoft	92.6	102.6	20932	Y		/fpi
Wizard	23.1	71.4	11788	Y		

The floating-point series are with loop counts of 600, run at 7.16MHz.
The 8087 series are with loop counts of 6000, run at 4.77MHz.
The rightmost column indicates whether in-line code is supported. The column headed "8087" indicates, for floating-point results, whether the test runs faster with an 8087 present, and, for the 8087 results, whether it works if an 8087 is not present.

Table 4
Floating-point and 8087 Results

Documentation and Support:

Vendor	Usability	Readability	Support
Aztec (Manx)	2.83	**3.72**	1.88
Control C	3.17	2.87	1.56
c-systems	2.50	2.55	2.19
Computer Innovations	3.17	2.83	2.50
Datalight	1.00	1.03	.63
DeSmet	2.33	2.17	3.44
Digital Research	3.33	2.58	**4.06**
EcoSoft	2.17	2.05	1.56
Lattice	3.67	2.97	2.81
Mark Williams	3.17	2.87	1.88
Microsoft	**4.67**	**4.02**	3.13
Software Toolworks	1.83	1.33	1.56
Wizard	3.33	2.85	**4.38**
Interpreters:			
C-terp	2.50	1.83	.63
Instant C	3.00	1.25	1.56
Run/C	2.50	**3.87**	.94

Table 5
Documentation and Support

compiler with no floating-point support at all.

For the minfio( ) benchmark, the Datalight, Aztec, and EcoSoft compilers shone the brightest, with Wizard's not far behind. Here DeSmet's .EXE file was a lot larger than the one generated for minprtf( ), even though the float library was not required. This was viewed by all as very strange.

The min–( ) benchmarks, except for minmain( ), are all significant because they give you an idea of how small a small program might be. Since there are very few useful functions to be performed that do not require some console and disk I/O, minmain( ) was included primarily to show the losers in this category in the unkindest light possible. The c-systems and DRI compilers did not fare well at all, requiring 19K and 18K, respectively to say "hello world" via printf( ). Lattice's compiler, a very well thought of product generally, also looked none too good here. The Aztec compiler was the overall winner in this category. The Aztec compiler also had the smallest execution-time benchmark files, requiring only 14K for benchall.exe versus an average of 55K–65K for the others.

Memory Models

The 8086's segmented architecture leads to some tradeoffs between execution speed and addressing range. The way a compiler views the 8086's address space is called the memory model. All the compilers reviewed here support the so-called "small" memory model as the default, and many of them allow no other. The small model allows up to 64K for instructions and 64K total for all kinds of data: heap, stack, statics, and globals. Automatic variables are allocated on the stack, with space dynamically allocated via malloc( ), etc., coming from the heap.

Other possible models are called different things by different manufacturers. Lattice, Aztec, c-systems, Wizard, and Microsoft all provide a model in which code (instructions) can run up to a megabyte (the maximum allowed by the 8086), but data is limited as in the small model. Lattice, Aztec, c-systems, and Wizard provide a model that limits code as in the small model, but the heap may extend to a megabyte, the stack can grow to 64K, and statics and globals together can take up 64K. DRI and Computer Innovations, in addition to the just-mentioned four, provide a model allowing code up to a megabyte and data as in the preceding model. Wizard and Microsoft have a model that extends this last model by allowing more than 64K of combined statics and globals (although no single item may be greater than 64K), in addition to the one-megabyte code and heap spaces; the stack is still limited to 64K.

The Wizard and Lattice compilers each have an option that permits the compiler to generate much better code for the large models, if the programmer is willing to live with certain restrictions regarding pointer arithmetic. These two compilers also allow very small programs to be changed to .COM files. Such programs must fit all code and data into 64K.

Microsoft is currently the only one of the 13 reviewed manufacturers supporting "mixed-model" programming, although others indicate that it is high on the to-do list. Mixed-model programming allows certain pointers to be declared "near" or "far," allowing you to use a pointer from a model different from the model of the rest of the program. You can write a program that uses the small model with its speed benefits and make only certain pointers of the time-consuming "far" variety to gain access to very large arrays.

Floating-Point and 8087 Support

Although C is presently most often applied to problems that don't require any floating-point arithmetic, the language does support and define the use of "real" numbers. Of the 13 compilers reviewed (see Table 4, page 43), all but one, Software Toolworks C, support floating-point math, and Toolworks expects to have it available by the time you read this.

The remaining 12 all support both float and double data types as 32- and 64-bit quantities, respectively. Nearly all use the IEEE floating-point standard to represent float and double. This makes it relatively easy to take advantage of an 8087 math chip (which also uses the IEEE standard). Datalight's is the only compiler we reviewed that did not offer any way to take advantage of the 8087. Lattice's compiler documentation indicated that its library was "sensing," but our results indicated that it did not use the available 8087.

A sensing library is a library of floating-point functions that use the 8087, if one is present, and do the required operations in software if not. This means that you, the programmer, don't have to know in advance if your program is going to run on a machine with an 8087.

Another way of supporting an 8087 is to use a library that requires the presence of an 8087. Compared to the sensing library, this buys you a little time (the time the sensing library takes to decide whether there's an 8087 for it to use) and some 2K–7K of code (the software floating-point routines). Or you can generate 8087 instructions in-line like integer operations. This saves the time needed to make the function call.

It is possible to execute an 8087 instruction in a machine not having an 8087. What happens is that the 8086 traps to a user-provided interrupt routine. This routine may emulate the 8087 and return. This allows the compiler to generate in-line instructions and still not require an 8087, but to use one if it's available. Only Microsoft's compiler implements this type of an emulation scheme.

Documentation

We reviewed all documentation and packaging for each compiler. To be consistent, we developed "benchmarks" for documentation, with a grading system consisting of two main categories: usability and readability.

Usability

The usability category covered the overall packaging of the documentation: the organization and quality of presentation that make the documentation easy or difficult to use. The items that contributed to the usability score

were the following:

- Packing list (P/L): Was a list of all manuals, disks, etc., included so that you could confirm that you had received everything?
- Disk contents (DISK): Was a list of all disks and their contents included? Often one was included, but buried in the manual and hard to find.
- Binding (BIND): How was the documentation received from the vendor? We saw everything from loose pages (not even clipped or stapled) to multiple 3-ring binders in attractive cases.
- Table of contents (TOC): Was there one? How complete was it?
- Index (INDEX): Was an index included, and how complete was it? Believe it or not, some vendors don't want you to locate anything in the documentation, as proven by the fact that they don't index their manuals!
- Error codes (ERR): Was a list of error codes generated by the compiler present, and how useful was it?
- Startup (START): Was a getting-started section supplied for those who need this level of support? Also considered here: the quality of instructions on batch files for installation, sample "test" programs to try out the compiler, and sample batch files for operation.
- Operation (OPER): Was a compiler-operation section included, how complete was it, and could you find each compiler option without rereading the entire section?
- Libraries (LIB): Was a section describing the libraries supplied, and how easy was it to find a particular function? This is where a good index helps, but at least the functions ought to be in some order, not at random! The CI-86 compiler included a brief summary of the functions in its table of contents—bravo!

Readability

Readability pertained to clarity of presentation and to the ease with which the material could be understood. To establish some method of consistent grading, we picked three library functions and subjectively graded the presentation on each function across all compilers. We picked fopen( ), fread( ) (where supported), and printf( ). We looked for such things as good examples, clear definition of all parameters, clear definition of return values, error returns, notice of deviation from standards, and a good explanation of the function. We also gave credit if only one function was printed on a page, making it easier to find and read. Cop-outs such as "see K&R for description of this function" sat very poorly with us.

Table 5 (page 43) shows the numerical scores we arrived at for documentation. The Microsoft manuals were the best on virtually every point. The Wizard, Lattice, and DRI manuals should be usable over the long run, and the Aztec book was very readable.

User Support

The best software product is of marginal value if it is not well supported. We attempted to review the kind and quality of user support available for the compilers reviewed. This included contacting support groups for most of the compiler publishers to evaluate their responsiveness.

Although you should not expect the same level of support for an MSDOS compiler as you might for a compiler running on a large mainframe, some of the same support features should be available. Ideally, support should reflect some modularity, so those who need full support can obtain it, while those who need only occasional assistance don't have to pay for support they won't use. We attempted to evaluate user support based upon several criteria that we determined to be important. The numerical results are shown in Table 5. Here is a discussion of the criteria.

Bug List

Most of the compilers did not have available a *current* list of known bugs. Some did indicate problems known but not yet fixed as of a particular release, either in a read.me file or in the documentation. Ideally, what we wanted to see was a regular update describing current bugs, the expected fix date, and any work-arounds. This would be a big help at two in the morning when you are under pressure to complete a big project, but are stopped dead by a compiler bug. It also ought to help the customer service group by reducing the number of times the same bug is reported.

Registration

Most vendors supplied a means to register the compiler so that the buyer could receive notices of future releases. Some were even postpaid. We had no means of verifying that new release notices would appear in the mail, but past experience indicates that sometimes such notices are sent and sometimes not, depending on the vendor.

Update Subscription

Only one vendor (Wizard) offers a fixed-price annual subscription service for all bug fixes and updates. Many others indicated that updates were available but were charged on a per-update basis. The Wizard approach has appeal: it's simple, and you should receive updates faster, with less paperwork at both ends.

Phone Consultation

Phone-in consulting service was available from all vendors, but some vendors required a serial number to gain access and others required money before giving the buyer a phone number to call. Although it is certainly not unreasonable to charge for support, many vendors bundle phone support into the basic product. We found that some vendors restricted calling hours.

Support Quality

All the reviewers made notes when they called vendors for support, and the documentation group also attempted to contact the support personnel for each vendor. We found a wide range in quality of support. Sometimes we were referred to the author of a compiler. That should get you qualified technical support, but we felt it to be a negative; it

meant the technical support was dependent on one person, and if that person is unavailable, you're stuck. On the other hand, talking to a "support person" who doesn't know C makes one very uneasy. We found that the support team at Wizard offered by far the best technical assistance.

In Table 5 you see an attempt to assign numbers to our experiences with the vendors in these areas of technical support. The clear winners here were Wizard's, DRI's, DeSmet's, and Microsoft's compiler support.

Compiler-specific Comments

The following comments, arranged by compiler, elaborate on the major items not already discussed or easily discernible from the tables: the quality of the documentation, standards issues, user support, overall benchmark performance, and summary remarks.

Aztec C86

Aztec C86 was easily one of the fastest compilers overall and handled array referencing more efficiently than any other compiler. Our results indicated that its library provides a lot of flexibility, allowing you to generate small .EXE files. It was the best in this area. The Aztec compiler provides an integer-only version of printf( ). There are several large-memory models to choose from. A make utility is included.

There was a general problem with the benchmarks: the main( ) routines used a cast to produce a null pointer to function returning an int — (int (*)( )) 0. Some compilers could handle it; many others couldn't. Aztec C86 couldn't handle this function cast in main( ), but it gave an excellent error message when it tried.

The Aztec manual had one major problem: it has no index. Otherwise it rated above average, though we thought it needed more white space for readability. Aztec did include a section on making your code ROMable, very useful to some, and it had good examples in the library descriptions, although these tended to describe more than one function at a time. This does save repeating the same text over again, but it makes it a little difficult to extract specifics about a particular function. A new manual is available (received too late for review); among the utilities included are editor, debugger, make, Aztec to .OBJ format converter, CRC utility, archiver, squeezer, grep, and hex generator.

Control C CC-86

Control C's compiler is a copy of Mark Williams C (MWC), documentation and all, so we'll refer you to our comments on MWC. Apparently, Control C is doing the port of the Mark Williams compiler to CP/M-86 and, as part of the arrangement, is able to provide the MSDOS version to the outside world. The only comments we'll make specific to the Control C compiler are that one of the six disks in the original package we received had a defect and that the Control C package did not include the function-key pad overlay for the debugger, as the Mark Williams package did.

c-systems 8088 Software Development Package

Although generally slow, the c-systems compiler turned in the fastest time in the disk I/O test. It did not do well in tests of generated-code size. c-systems provides several large-memory models. The c-systems compiler does not support fgetc( ) in stdio.h, even though fputc( ) is supported. Consequently, fgetc( ) was coded as a module, rather than as a macro, using getc( ).

The c-systems manual contained a neat trick for compiling multiple modules using wildcards. Other compilers support wildcards in the command line; c-systems uses the **for** batch command from the command line. For example:

```
A>for %f in (min*.c) do cc %f
```

This technique can be used with any compiler. c-systems' compiler was about average in the area of documentation. It didn't include a binder, and the print quality was only fair. The installation section was too brief, but the operation section was well done. It did include a detailed list of differences from the standard, something many did not. The library descriptions lacked examples and could use more information on return and error values. There were problems using drivers; occasionally they would not run for no apparent reason and they only used the first 128 bytes of the environment string so the INCLUDE variable was not always findable; the compiler has PL/M compatibility.

Computer Innovations Optimizing CI-86

The CI-86 compiler performed in the middle range on nearly all the benchmarks, walking away with the copychr disk I/O test. One nice feature of the CI-86 compiler is that numerous paths are provided for interfacing to the operating system. For instance, you can initialize and release interrupt routines such as might be required for writing a communications program.

Computer Innovations provides a large-memory model, although when we used it, it would not allow pointer arithmetic on a multiply-dimensioned array across dimensions. The documentation explains this strange behavior, stating that without the functions that convert pointer to long (ptrtoabs( )) and long to pointer (abstoptr( )), "big-model pointer arithmetic assumes the data segments are the same for the pointers." The compiler did give a reasonable error message when an input data file was not found. The CI-86 compiler doesn't allow ++ or — — on float or double variables. For example:

```
double d; d++;
```

did not work, while

```
double d; d += 1;
```

did work.

We judged the Computer Innovations manuals above average. They included a good approach in the table of contents: a one-line description of each library function, serving as a quick-reference guide. Their library descrip-

tions were well done, with examples and notes on DOS applicability. The compiler-operation section was too brief, and no listing of compiler error messages was included.

Datalight C

The old MSDOS Small-C compiler distributed by Datalight was a port of the public domain Small-C compiler for CP/M. The new C Compiler, which is a full implementation of the C language, stood out in a number of areas. It came in first or tied for first on several tests, standing in a class by itself on the long-arithmetic benchmark. Datalight provides an integer-only version of printf( ).

The Datalight compiler appears to implement real block scope for declarations. Function xmalloc( ) was declared as extern char *xmalloc( ) inside main, and this declaration was not in force in later functions, leading to a "definition inconsistent with prior use" error when the function was called. We view this as appropriate, but it is not clear what behavior K&R calls for, and most of the other compilers did not catch this error.

The Datalight compiler could not compile the global definition

```
char null_strg[ ] = { "" };
```

It had no problem when it was changed to

```
char null_strg[1] = { "" };
```

The documentation that came with the new Datalight compiler was a significant improvement over the previous version, as was the compiler itself. The packaging was still low-budget, like the compiler—56 8½-by-11 spiral-bound pages with small, difficult-to-read print. All the essentials were there, though. There was a usable table of contents and an index, the few supported error messages were all described, there was a setup section with a sample program, and some of the function definitions included examples. Overall, not bad for a $50 compiler.

DeSmet C

The DeSmet compiler was generally mid-range in speed, coming in above average in the screen I/O tests and losing the floating-point math test. It produced the smallest .EXE files in certain of the code-size tests. The compiler is supported by a good user group, and there was a bug list.

No error message was produced when an input data file was not found—the compiler just went to sleep! DeSmet's compiler does not like really long comments; an error message was generated at line 28 after the opening /*.

DeSmet's manual is a low-budget product. Its pages were not bound nor even stapled; they lacked an installation section, and although the DeSmet package supplied a lot of software, nowhere is there a list of what you should have received in the package. The manual also omitted an index and didn't organize the functions as we would have liked, with one function on a page. The descriptions were brief, but they did include examples.

Digital Research DRI C

Although the DRI compiler was among the faster overall, it dawdled through long arithmetic. The compiler performed poorly on the tests of library granularity. DRI provides a large-memory model, and DRI's support got high marks.

DRI's compiler had no problem with the function cast in main( ). No timer function and no access to DOS interrupts were provided; an assembler is required, but rasm86 was included, and the manual documented the assembly and linkage process very well, including an example. Capital letters on the command line were converted to lower case before being passed to main( ).

DRI's compiler was the only one to report an error for two statements in doc1000.c that defined and initialized global arrays. Both initialization lists included a trailing comma, as in:

```
int array[ ] = { 1, 2, 3, 4,
        5, 6, 7, 8,
    };
```

The output file produced by cpyblk( ) was too small because fread( ) returned zero instead of the actual number of bytes read on the final block. The block size was 1024 bytes, so the last block would have been a short one.

Digital Research put a good documentation package together. They provided a reprint, in their own binding, of K&R. The index was good, and a nice error list appeared in an appendix. The getting-started and operation sections were well done, with good tables and examples. The library function descriptions could have been better organized, though, and should have included examples instead of directing the reader to K&R for descriptions of functions.

EcoSoft-C

The EcoSoft compiler was not dazzlingly fast nor distressingly slow on the benchmarks. It did well in the code-size tests, producing a remarkably small .EXE file in the minmain( ) test. The library had no fgetc( ) nor fputc( ). These should have at least been defined as macros in stdio.h. We appreciated the fact that EcoSoft included the source for CC, its driver program. Given that they had written a fairly good book on C, we expected good documentation from EcoSoft. We were disappointed to find a very short table of contents and no index. We found the documentation difficult to use. The getting-started and compiler-operation sections were too brief. The library functions did not include examples and were not terribly well organized for easy reference. The compiler was hard to use relative to others in our environment without recompilation of the CC driver.

Lattice C

The Lattice compiler had some strengths and weaknesses (e.g., screen I/O) in execution speed. Although one benchmark is inadequate to assess a compiler's optimization, it should be noted that Lattice won the optimization test hands down. Lattice didn't do particularly well on the

code-size tests, but it is one of the few vendors providing the flexibility of an integer-only version of printf( ). Lattice also allows small programs to be changed to .COM files and provides several large-memory models and an option that allows the programmer to trade some pointer-arithmetic facility for better generated code.

The LC program could not handle directory path names in its file list, even though LC1 and LC2 allowed this. The math library was required to do any floating-point stuff, not just transcendentals, etc. Also, the math library had to precede the std library in the search order. The manual documented neither of these facts. Having long supported C for MSDOS, Lattice has produced above average documentation for its compiler and included a good index into the functions. While the installation and getting-started section was brief, it was adequate. The compiler-operation section was good, but finding the utilities operations was a little difficult; these should have had their own sections. The function descriptions were fairly well done but lacked examples. The function descriptions did include a caution paragraph, which is good. This was the only vendor to supply a packing list with the package.

Compilers

Wizard C V. 2.1
Wizard Systems Software
11 Willows Ct.
Arlington, MA 02174
(617) 641-2379

DeSmet C V. 2.4
C Ware Corp.
P.O. Box C
Sunnyvale, CA 94087
(408)720-9696

Lattice C V. 2.14
Lattice
P.O. Box 3072
Glen Ellyn, IL 60138
(312) 858-7950

Datalight C V. 1.0
Datalight
11557 8th Ave NE
Seattle, WA 98125
(206) 367-1803

8088 Software Dev Pkg V. 2.08a
c-systems
P.O. Box 3253
Fullerton, CA 92634
(714) 637-5362

Optimizing CI-86 V. 2.20k
Computer Innovations
980 Shrewsbury Ave. #310
Tinton Falls, NJ 07724
(201) 542-5920

C Programming System V. 2.0
Mark Williams Co.
1430 W. Wrightwood
Chicago, IL 60614
(312) 472-6659

Microsoft C V. 3.0
Microsoft
10700 Northup Way
Bellevue, WA 98004
(206) 828-8088

Aztec C86 V. 3.20c
Manx Software Systems
Box 55
Shrewsbury, NJ 07701
(201) 780-4004

DRI C V. 1.1
Digital Research
P.O. Box 579
Pacific Grove, CA 93950
(408) 649-3896

CC-86 V. 2.0
Control C Software
6441 SW Canyon Ct.
Portland, OR 97221
(503) 297-7153

Ecosoft-C V. 2.0
6413 N. College Ave.
Indianapolis, IN 46220
(317) 255-6476

Software Toolworks C V. 3.2
Software Toolworks
15233 Ventura Blvd., #18
Sherman Oaks, CA 91403
(818) 986-4885

Interpreters

Instant C V. 1.00
Rational Systems
P.O. Box

Run/C V. 1.11
Age of Reason
318 E. 6th #123
New York, NY 10003

C-terp V. 1.04
Gimpel Software
3207 Hogarth Lane
Collegeville, PA 19426
(215) 584-4261

Table 6
Addresses of Compiler and Interpreter Vendors

Mark Williams C Programming System

The Mark Williams compiler was the fastest compiler in the screen I/O tests. MWC provides an integer-only version of printf( ). We couldn't determine from the manual what the memory model options were when using MWC object format (−vsmall and −vlarge also force Microsoft object format). There appeared to be no way to specify the desired memory model with MWC object format, implying that the small model was the only one available. An attempt to cast a long to a char led to the following compiler message:

```
113: At 591: Fatal error: no match, op = 62
```

This was not too helpful, and the C syntax *was* legal. And there was no list of errors in the manual, with or without probable cause.

This compiler had a fairly well done manual, with a sample program to compile and run during installation. The function descriptions were reasonably well done and readable. There could have been more examples, and many functions were packed into one description, making it hard to extract pertinent information.

Microsoft C

The Microsoft compiler was fast in nearly all categories, although it didn't do well on the random-access diskio( ) test. We found it to be a carefully designed compiler, very well documented, and with some well-thought-out features. It provides an integer-only version of printf( ) and a variety of large-memory models, plus mixed-model programming. This last feature and 8087 emulation are two features only Microsoft offers.

Microsoft's compiler set a very high standard for software documentation. Two full binders were included: a complete language reference manual plus the compiler and library reference manual. The accompanying slip case folded down to form a book stand. We found it difficult to fault any aspect of Microsoft's documentation package, except for the slight jitter in their laser printer. One experienced C programmer in the group noted that he had never spent more than a couple of minutes with the documentation, because he was able to find what he needed very quickly.

The Microsoft compiler includes the Bessel functions; has excellent Unix/Xenix portability; and is very easy to use.

Software Toolworks C

The Software Toolworks compiler for MSDOS is a port of their C80 compiler for CP/M. The original compiler was a descendant of Small C, adding such features as multi-dimensional arrays and structures. The MSDOS version does not currently support the long, float, or double data types. Longs and floats will be available in an optional "mathpack," currently under development. The lack of these data types meant that we could not run some benchmarks, and the prtf( ) benchmark had to be modified so that the same number of lines were produced, but all longs, floats, and doubles were changed to ints.

Although the compiler does not support longs, it does support fseek for files in excess of 64K. The file position has to be passed as an array of two ints. As a result, a special version of the diskio benchmark was written; the principal changes were to make all longs arrays of int and to use a file created with the Lattice compiler to obtain the random position to seek to. The seek addresses were identical to those used in other benchmarks. The compiler also did not support initialization of data of class auto. Several of the benchmarks had to be ch ıged to account for this feature.

Other unimplemented features included: typedef, bit fields, and line control (#line). Major restrictions included: function calls had to have the same number of arguments as the called function definition, which resulted in a kludge in printf, fprintf, sprintf, scanf, fscanf, and scanf; declarations were allowed only at the start of a function; #define could not have arguments. The printf kludge required that stdio.h be included for minprtf. The compiler supported the small model only (64K of code and 64K of

C Interpreters

In addition to the compilers, we evaluated three C interpreters: C-Terp, Instant C, and Run/C. We were fairly demanding of the interpreters, and our comments should probably be taken as an experienced programmer's view. A beginner might be happier than we were with these products. They were generally faster than the compilers when it came to making a change and seeing what the results were, and that's important when you're learning a language. We did, however, have some complaints.

All three forced some rewriting of our benchmark programs. All three lacked some run-time support modules, in particular fread and fwrite. (C-Terp supplied them, but not in the standard release versions.) All three were memory-hungry, requiring some 300K to execute. We found at least one bug in each of the interpreters in the process of running our benchmarks, once we got the benchmark code in a form each compiler would accept. Of the three, Run/C was the least expensive, but also the least flexible. Block comments had to start in column one with only the "/*" on the line; the terminating "*/" had to meet similar constraints. Run/C was also slow in loading and executing. When we specified a program from the command line, Run/C loaded it, then checked for errors; when one was found, Run/C aborted interpretation, and we lost the (significant) load time. Run/C could not handle pointers to functions.

Instant C was fastest and had most of the features we felt we couldn't get along without,but we found that its editor got in the way. We were frustrated by its limitation to two array indices and by its small work area. It had nice, APL-like immediate-mode variable output that was very handy during the debug phase.

C-Terp was the most flexible and came closest to implementing the complete language. It had the ability to interface to functions written in assembly language, although requiring a compatible compiler to implement the interface. Assembly-language interface is promised for the near future for the other two interpreters. Unfortunately, C-Terp's documentation was sparse, a problem for beginners.

These tools are useful for checking out a short algorithm quickly, but not as the sole language implementation for developing complex pieces of software. You can learn C from them, although a purist might argue that you don't know C until you have worked with the full language. Of these interpreters only C-terp offered one of C's greatest strengths, the flexibility of using pointers as data.

There are three questions to ask yourself in considering buying one of these interpreters or recommending it to others: Can your programming needs be met by an interpreter alone? If you're planning to develop significant software in C, they probably can't. Is it worthwhile to you to purchase both an interpreter and a compiler? It may well be; they address different needs. And if you intend to buy an interpreter, which one? We've provided some material here for the beginning of your evaluation process, but note that we only looked at three C interpreters. And this is an experienced programmer's view of these products. For the benchmark results see Table 8 (below).

Interpreter:	tint	tlong	array	pointer	looptst	opt	fibtest	sieve
C-terp	.2	.2	22.1	28.0	41.7	126.4	54.5	329.2
Instant C	<0.1	<0.1	<0.1	<0.1	.8	1.6	19.2	4.4
Run/C	2.4	2.5	115.2	113.1	205.3	815.5	252.5	407.3

Table 8
Addresses of Compiler and Interpreter Vendors

data). The compiler included a function profiler that recorded the total number of calls to a function and the time spent in each function. In contrast to the C80 compiler, this compiler did not support in-line assembly code, although details of the assembly language interface were provided in the manual. This was clearly a subset compiler, with poor documentation. There was an index, though there were many functions to a page, making it difficult to find and read the descriptions. The function descriptions were terse and devoid of examples, again making it difficult for a new user.

Wizard C

Wizard's was generally a fast compiler. It supports a variety of large-memory models, lets the programmer trade some pointer-arithmetic capabilities for better generated code, and allows very small programs to be changed to .COM files. Wizard's got the highest marks for support.

Field width specification did not work for character conversions in printf( ); e.g., printf(%17c,c) produced one character, not 17. Careful reading of K&R leaves us with small doubt that Wizard is in the wrong. It is certainly safe to say that if Wizard maintains that field width is not supposed to work with %c, then they stand alone.

The Wizard compiler was an obvious port down from Unix, as the documentation reflected Unix throughout. While the getting-started section was small (one page) and no installation section was included, we appreciated the 22 pages of error messages. The function descriptions were readable, but like functions were grouped together and there were no examples.

Excellent diagnostics were provided for portable code; to increase portability from Unix, the library included Unix functions that have little meaning but are a pain to remove (e.g., getpid). The compiler supports anachronisms (=+ ...); supports the use of libraries for other compilers by changing call and return specifics; and includes lint. It has PL/M compatibility and is very easy to use.

Aztec	$199 – Pro Package $500
Microsoft	$279
Computer Innovations	$395
DRI	$350
Mark Williams	$495
Control-C	$500
c-Systems	$199
Wizard	$450
DeSmet	$109 + $50 (debugger) + $35 (DOS LINK)
Datalight	$60
Ecosoft	$49.95
Lattice	$425
Software Toolworks	$49.95 + ($29.95 float)
C-terp	$300
Instant C	$495
Run/C	$149.95

Table 7
Compiler and Interpreter Prices

Miscellaneous Comments

Microsoft, Wizard, Mark Williams, Control C, Ecosoft

These all had drivers that allowed the compilation of several files (assembly of assembly language files) and optional linkage of the files. This proved to be a great boon.

Mark Williams, Control C

Excellent source level debuggers and very nice examples of usage in the manual and on disk; does not interfere with screen I/O.

DeSmet, Aztec, c-systems

Included source or symbolic level debuggers, but we did not test them.

Microsoft, Lattice, Wizard

Support use of third party or optional source level debuggers, such as SYMDEB.

Aztec, Wizard, Lattice, DeSmet

Provide for generation of ROMable code.

Microsoft, Aztec, Mark Williams, Control C, c-systems, DeSmet, Wizard

The first six used environment to pass "usual" options to compiler; Wizard used a file and this approach was slightly preferred. DeSmet and c-systems had bugs in usage.

Microsoft, Aztec, Wizard, Contol C, Mark Williams

Support new structures (passing, assigning, and returning) and enum and void.

Aztec, DRI

Could not locate math functions in table of contents; includes overlay support.

Mark Williams, Control C

Unable to determine from the documentation what the specifics of the large model were.

Aztec, CI-86, Wizard, Software Toolworks

Library source included or available.

Summary

Were we objective? We all had our preferences and unequal experience with the compilers going into the review; we can't expect that we entirely overcame them. But large amounts of data can swamp out fairly extreme priors, and it was hard to deny the numbers we were getting.

So, who won? If you pressed us to declare a winner, we'd probably say the Microsoft or Aztec compiler. But the Wizard compiler had excellent diagnostics; it would be easier writing portable code with it than with any other compiler we tested. And the Mark Williams compiler had the best debugger. Our data would support different results, depending on how you chose to weight the features we assessed. No two programmers will weight them the same way. And we took almost no account of price. For addresses of compiler vendors see Table 6 (page 54); for compiler prices see Table 7 (page 50).

DDJ

A Peephole Optimizer for Assembly Language Source Code

by David L. Fox

The need for small fast programs is a constant of computer programming. Unfortunately, small fast programs tend to be difficult to write and maintain. As a result a programmer often has to make a choice between assembly language and a high-level language.

D. E. Cortesi, who has discussed this problem in several "Dr. Dobb's Clinic" columns,[1] has pointed out that an optimizing compiler is a powerful incentive to choose a high-level language. The compilers currently available for microcomputers, however, fall far short of the state of the art in terms of the size and speed of the code they produce. In fact, many of these compilers produce code that could be improved by the simple technique of peephole optimization.

In this article, I describe a standalone peephole optimizer that can post-process the output of any compiler producing assembly language source code. The program is written in C and has been compiled with Software Toolworks' C/80 compiler. According to a published benchmark comparison,[2] in terms of both code size and execution speed this is one of the better C compilers available for CP/M. Nevertheless, peephole optimization can improve its output, and I will use it for the examples in this article.

David L. Fox, 2140 Braun Dr., Golden, CO 80401

Program Optimization

The term *optimization,* as it is usually applied to programs, is actually something of a misnomer: it derives from the word *optimum,* meaning the best possible, and no program optimizer or optimizing compiler can seriously claim to produce the best possible code. A more realistic goal is merely to improve the code.

The greatest improvement in a program usually comes from using a better algorithm. If you are not convinced of the importance of choosing an efficient algorithm, I suggest that you write two routines to sort an array of numbers, one using a bubble sort and another using a Shell sort. Use them to sort an array of 1000 numbers. The bubble sort, which requires a time proportional to the square of the number of items sorted, will run tens or even hundreds of times longer than the Shell sort, which requires time proportional to $n^{1.5}$ to sort n items. Unfortunately, the choice of algorithm is not something that we can easily mechanize. If you want a substantial improvement in the performance of a program, rewriting it using a better algorithm may be the only choice available.

The more modest improvements obtained by the means discussed below should not be dismissed, however, because they frequently lend themselves to implementation by a program (an optimizing compiler or a separate optimizer) and require little

Algorithms for finding and replacing regular expressions are well known. Common inefficiencies in compiler output can be specified as regular expressions. So . . .

effort on the part of the programmer. Many of these improvements involve recognizing specific constructions in the input and replacing them with more efficient code in the output. This process is easily mechanized.

One rich source of potential improvement in compiler output is removing the recalculation of values that are already available. For example, a straightforward compilation of a statement such as

```
a[i] = b[i] + c[i]
```

will calculate the byte offset associated with the subscript *i* three times. (If a, b, and c are arrays of 16-bit integers, this offset is *2*i.*) Substantial savings would be possible if this type of situation were recognized and the offset calculated only once.

An even simpler example of this is illustrated in Listing One (page 68) where the compiler produces the pair of instructions:

```
SHLD i
LHLD i
```

After execution of SHLD i, the contents of i are still in the HL register pair and the LHLD i may be omitted. Note, however, if the LHLD instruction is labeled, it may not be omitted.

Another example in Listing One is the following:

```
MOV A,H
ORA L
JNZ L1
LXI H,0
```

The zero flag is set if HL contains zero, in which case the jump to L1 is not taken and the LXI H,0 may be removed.

Simple compilation of conditionals and loops often results in multiple jump instructions that can be easily improved. A jump to a jump can be replaced by a single jump instruction, as shown in Listing One. Another commonly found construction is the jump over a jump, such as

```
    JZ  L1
    JMP L2
L1: ...
```

which may be replaced by

```
    JNZ L2
L1: ...
```

Any unlabeled instruction following an unconditional jump can never be executed and may be removed. Also candidates for removal are unreachable instructions created by the jump optimization described above. For example:

```
    JMP L1
    .
    .
    JMP L2
L1: JMP L3
```

After we replace JMP L1 with JMP L3 and remove the label L1, JMP L3 becomes unreachable and may also be removed. Thus, one improvement can create an opportunity for further improvement, making it worthwhile to pass the program through the optimizer more than once.

The examples so far have not involved any trade-offs between code size and execution speed. In the following situation, speed or size can be optimized, but not both. Pointer dereferencing (obtaining the value stored at a given address) may involve the following subroutine:

```
deref:  MOV A,M
        INX H
        MOV H,M
        MOV L,A
        RET
```

Excluding the RET instruction, the above subroutine is only four bytes long. Because any program that uses pointers or arrays does quite a bit of dereferencing, use of the subroutine deref will produce a smaller program. If speed is more important, however, replacing the three-byte instruction CALL deref with four bytes of in-line code will result in faster execution. Some compilers allow the user to choose whether smaller or faster code is more important, but most compilers for microcomputers do not provide this option.

The details of the compiler and the machine instruction set may offer other opportunities for improvement. One common possibility arises from the fact that many compilers for CP/M produce 8080 code. If you have a Z80 processor, you may find ways to replace combinations of 8080 instructions with Z80 instructions. A good example is a 16-bit subtraction, which requires six instructions on an 8080 but only a single Z80 instruction. Identifying the possible improvements for your compiler will require careful study of its output and runtime library.

These examples illustrate the types of improvements that the peephole optimizer can make. Many other improvements, which require some knowledge of the organization and structure of the program such as recognizing constant expressions and removing them from loops, are beyond the scope of the optimizer presented here.

If the optimizer is specific to one compiler, it can be quite simple; see the optimization section of Small-C version 2.[3] On the other hand, if you want to use the program with different compilers or have some flexibility in what changes to make, the optimizer will require most of the capabilities of a general text editor. For that reason, I have borrowed heavily from Chapters Five and Six of Kernighan and Plauger's book *Software Tools in Pascal,*[4] which give Pascal listings for a fairly complete editor. Listing Five (page 76) contains the needed routines (translated into C and slightly extended) from Kernighan and Plauger's editor.

Regular Expressions

Once you have identified output from your compiler that may be improved, you must communicate descriptions of the improvable constructions to the optimizer program. The notation must be precise and unambiguous—capable of specifying constructions such as:

```
first line:    SHLD  (any label)
second line:   LHLD  (same label
                      as above)
```

A notation sufficiently powerful to describe such strings is used widely

within the Unix environment. This notation, adopted here, is that of regular expressions.

A regular expression is a description of a set of strings. The simplest case is when the regular expression describes a single string: the regular expression is just the string itself. For example, the regular expression for the 8080 store HL instruction mnemonic is SHLD. Nonprinting characters are represented by the standard C escapes:

\b	backspace
\n	newline
\r	carriage return
\t	tab
\\	backslash
nnn	byte with octal value *nnn*

An alternative way of looking at a regular expression is as a pattern. A string matches the pattern if, starting at the left, each character of the string matches the corresponding character of the pattern. Because the optimizer is concerned with identifying lines of assembly language code, we will say that any line containing a string described by a regular expression matches that regular expression.

To specify more complex patterns, we need an element that can match more than one character in the input string. One such useful element is the wild card character (.), which will match any single character. We can also specify one of a group of characters by using a character class, which consists of a [, followed by a list of characters, and ends with a], to match a single occurrence of any of the characters enclosed in the square brackets. For example, [SL]HLD will match SHLD or LHLD.

We may use a shorthand notation for all consecutive letters or digits: [a-z] matches any lower-case letter, and [A-Z0-9] matches any upper-case letter or any digit. If the first character of the character class is ~, it negates the character class: it will match any character *except* those in the list following the ~. For example, [~aA] matches any character except a or A.

Character classes are essential to matching constructions such as "any label." The standard C identifier consists of any string of letters, digits, or underscore characters that does not begin with a digit; thus, the character class that matches the first character of an identifier is [a-zA-Z_], and the remainder of the characters match the class [a-zA-Z0-9_].

To fully describe a label with a regular expression, we need a construction for "the remainder of the characters" that does not limit us to a specific number of characters. We achieve this by following any character (or character class) by an asterisk, *. The result, called a closure, will match zero or more occurrences of the single character preceding the *. For example, aa* matches any string consisting of one or more occurrences of the letter a. The regular expression describing a C identifier can now be given as:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

This actually matches an identifier of unlimited length, but it serves our needs.

We now need some way of specifying "same label as above." For this, we draw on the ability to group portions of a pattern and refer to the group(s) later. A portion of a pattern that appears between the pairs of characters \(and \) constitutes a group. A group of characters matched by a parenthetical pattern may be identified by the notation \1 for the first group in the pattern, \2 for the second, up to \9 for the ninth. For example,

```
\(mat\).*\1
```

matches mathematician while

```
\(...\).*\1
```

matches barbarian, mathematician, and any string that repeats a group of three letters. The notation developed so far allows us to describe the example at the beginning of this section with two regular expressions:

```
SHLD\t\([a-zA-Z_]
  [a-zA-Z0-9_]*\)
LHLD\t\1
```

Finally, it is sometimes useful to require that the string matching a regular expression occur at the beginning or end of a line. The characters to do this are ^ for the beginning of a line and $ for the end of a line. Thus, ^A matches any line that begins with A, and ^$ matches only empty lines. Any characters that have special meanings (e.g., [ ]*.\^$~-) may have the special meaning turned off by preceding the special character with \. The pattern \.$ matches any line that ends with a period.

This flexible and powerful notation suffices to describe most constructions appropriate for improvement with a peephole optimizer. There are a few traps for the unwary, especially when using closures. Remember that x* matches zero or more occurrences of x, so that the regular expression x* matches any line, even those with no x's at all. When you want to match one or more x's, use xx*. Another point is that any closure matches the longest possible string: e.*e matches the entire word *emergence,* not just *eme* or *erge.* Table 1 (below) gives some more examples of regular expressions and the strings they describe. For a more extensive discussion of regular expressions, see Chapter 3 of Aho and Ullman's *Principles of Compiler Design.*[5]

Using the Optimizer

The description of each improvement in the compiler output has two parts: a list of regular expressions describing the code to be replaced and a list of lines of improved assembly language code. A group of lines is replaced only if each line in turn matches the corresponding regular expression. The constructions \1, \2, and so on in a replacement cause the replacement to contain the same string matched by the corresponding "\(\)" group in the list of regular expressions.

The list of potential improvements is placed in a file that is read by the optimizer. Listing Six (page 82) shows such a file for use with C/80 compiler output. Each improvement consists of the following: one or more lines containing regular expressions identifying the construction that may be improved, a separator line that contains a % as its first character,

zero or more lines that are to replace the text matched by the regular expressions, and a terminator line starting with %%. The file may contain any number of potential improvements. In Listing Six, comments describe each improvement; they make the output somewhat larger than the input and may be removed if desired.

The program command line syntax is

```
opt options infile outfile
```

where infile is the name of the file containing the assembly language source code to be optimized, outfile is the name of the file to contain the optimized output (must be different from infile), and options represents zero or more of the following optional parameters:

-b*n* Store *n* lines in the internal FIFO buffer. Default is 500.

-f*file* Read the list of improvements from *file*. Default is opt.dat.

-j*re* A jump instruction described by the regular expression *re*. Default is \tJ[CEMNOPZ]*\t.

-l*re* A label described by the regular expression *re*. Default is [a-zA-Z_\.][a-zA-Z_0-9]*.

-o Omit jump optimization. With this option selected, the optimizer can be used as a general purpose stream editor.

-u*re* An unconditional jump instruction described by the regular expression *re*. Default is \tJMP\t.

-v Turn off verbose output. Normally opt displays the last line of every improvable construction on the console; selecting this option omits that output.

The internal organization of the program is quite simple. Each line is read into a FIFO buffer in memory. The size of this buffer determines the range of jump instructions that may be optimized. The input is compared with the regular expressions of the improvable constructions, which are contained in a linked list. If a match is found, the appropriate replacement is made. A symbol table containing all the labels in the input is maintained for jump optimization.

Because this article is on optimization, it is appropriate to discuss the optimization of the optimizer. As given in Listings Two to Six (pages 68–83), the program processes approximately 200 lines per minute. Using the optimizer on its own assembly language code saved 199 bytes and increased speed by about 2%. Adding to the data file some constructions specific to the optimizer and reoptimizing saved an additional 86 bytes and increased speed by another 1%, for a total improvement of 285 bytes and about 3%. A third run, made with a data file that replaced calls to the pointer dereferencing routine with inline code, increased program size by 473 bytes and speed by 10%.

We might wish to improve two separate aspects of the peephole optimizer's performance. First, consider the amount of improvement the optimizer makes in the programs it processes. We could increase performance in this area by finding more improvable constructions in the compiler output and adding them to the data file opt.dat. Unless we have missed some extremely poor and very common compiler output, however, this approach will likely produce only a small increase in performance. As is usually the case, substantial improvement requires the use of an improved algorithm.

Second, consider the speed of the optimizer in accomplishing its task—presently about 200 lines per minute. Once again, small improvements may

^. . .$	Matches any line with exactly three characters
(.*)	Matches any string enclosed in parentheses
/ *[a-zA-Z]**/	Matches any C comment containing only blanks and letters
[~]*	Matches any string of nonblank characters
\([a-z][a-z]*\)\tJMP\t\1	Matches "loop JMP loop" but not "loop1 JMP loop" or "abc JMP xyz"
\(..*\)\1	Matches any string containing two adjacent repeated substrings such as *beriberi*, *vacuum*, and *tintinnabulation*
\(..*\)\(..*\)\2\1	Matches *affair*, *accommodate*, and *this is the end* (\1="th", \2="is ")

Table 1
Examples of Regular Expressions and Matching Strings

ROUTINE	CALLS	TICKS	% TIME (Approx.)
amatch	81309	9370	27
omatch	92501	6548	19
fgets	3149	3707	11
outline	3103	3541	10
match	32504	3011	9
locate	36934	2504	7
patsize	42331	2331	7
findsym	1316	1363	4
strcmp	34942	812	2
free	3374	654	2
optimiz	3103	376	1
malloc	3692	287	1
main	1	112	0

Table 2
Profiler Output for Optimizing a Test File

be made by tweaking the present code, but major improvements will require a change to better algorithms. By using the profiler included in the Software Toolworks' C/80 package, we can identify those subroutines of the program that are executed most frequently. The profiler output for a run of the optimizer is shown in Table 2 (page 66).

Table 2 shows that the optimizer spends over two-thirds of its time matching regular expressions to strings (match( ) and subsidiary routines). Clearly, any significant increase in speed will require an improvement in the pattern-matching code. I have made a small effort along these lines by replacing the C version of the routine locate( ) with an assembly language version. Because better algorithms for regular expression matching are known, this seems like the most profitable course for increasing the speed of the program. In contrast, replacing the linearly searched linked list used for the symbol table with a more efficient hash table cannot increase the speed of the program by more than 4% because only 4% of the execution time is spent in the routine findsym( ) looking up symbols in the table.

Installation

Although I have endeavored to make the program as flexible as possible so it will be useful with a variety of compilers, I have tested it only with the C/80 compiler. Users of other compilers will have to construct a data file like that in Listing Six for the constructions peculiar to their compiler.

Listings Two to Five are "standard" C and should be portable to other C compilers. My library follows the usage of the Unix (version 7) library and differs only slightly from the library distributed with C/80 and some other C compilers. Listing Seven (page 83) shows the library functions used by the optimizer.

Although I have not tested it, the new Small-C library[6] appears to contain suitable routines, with the exception of the memory management routines malloc( ) and free( ). The optimizer frees blocks of memory without regard to the order in which they were allocated. The C/80 library (version 3) also contains suitable routines, provided that the error return value is changed to 0 (NULL) from -1. Listings for suitable memory management routines are also given in Chapter 8 of Kernighan and Ritchie.[7]

Availability

I will return a copy of the source code and an executable .COM file (for CP/M 2.x) to anyone who sends me a postage-paid disk mailer, a formatted 5¼-inch disk, and $10.00. I can write a large number of single-sided, soft-sectored formats but do not have the hardware for hard-sectored or 8-inch disks.

References

[1]D. E. Cortesi. *Dr. Dobb's Journal.* No. 83 (November 1983) p. 10, No. 87 (January 1984) p. 18, and No. 91 (May 1984) p. 12.

[2]Jim Gilbreath and Gary Gilbreath. *Byte.* (January 1983) p. 283.

[3]J. E. Hendrix. *Dr. Dobb's Journal.* No. 74 (December 1982) p. 16, and No. 75 (January 1983) p. 48.

[4]Brian W. Kernighan and P. J. Plauger. *Software Tools in Pascal.* Addison-Wesley, 1981. See also Brian W. Kernighan and P. J. Plauger's *Software Tools* (Addison-Wesley, 1976), which gives the same tools in Ratfor, a Fortran-based language.

[5]Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1979.

[6]James Hendrix and Ernest Payne. *Dr. Dobb's Journal.* No. 91 (May 1984) p. 50, and No. 92 (June 1984) p. 56.

[7]Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

DDJ

Peephole Optimizer

(Text begins on page 56)

Listing One

```
int i,j,m,n;
while (i == 0)
{       i = j + n;
        m = i + j;
        if (j == 1)
                j = 0;
}
```

(a) Fragment of C code

```
                        ; start of loop
.f:     LHLD    i       ; evaluate i == 0
        MOV     A,H
        ORA     L
        JNZ     .g      ; jump out of loop if not true
        LHLD    j       ; i = j + n
        XCHG
        LHLD    n
        DAD     D
        SHLD    i
        LHLD    i       ; m = i + j
        XCHG
        LHLD    j
        DAD     D
        SHLD    m
        LHLD    j       ; evaluate j == 1
        DCX     H
        MOV     A,H
        ORA     L
        JNZ     .h      ; jump if not true
        LXI     H,0     ; j = 0
        SHLD    j
.h:     JMP     .f      ; loop
.g:     DS      0       ; continue with rest of program
```

(b) Assembly language compiler output
(Comments added)

```
                        ; start of loop
.f:     LHLD    i       ; evaluate i == 0
        MOV     A,H
        ORA     L
        JNZ     .g      ; jump out of loop if not true
        LHLD    j       ; i = j + n
        XCHG
        LHLD    n
        DAD     D
        SHLD    i       ; redundant LHLD deleted
        XCHG
        LHLD    j
        DAD     D
```

Peephole Optimizer (Listing continued, text begins on page 56)

```
        SHLD    m
        LHLD    j       ; evaluate j == 1
        DCX     H
        MOV     A,H
        ORA     L
        JNZ     .f      ; JNZ to JMP improved
                        ; LXI H,0 removed following in line test
        SHLD    j
        JMP     .f
.g:     DS      0       ; continue with rest of program

(c) Optimized assembly language code
```

End Listing One

Listing Two

```
/* optdefs.h -- Definitions and declarations for peephole optimizer. */

/*****************************************
*    Copyright 1984 by David L. Fox
*    All rights reserved.
*    Permission is granted for unlimited personal, non-commercial use.
*****************************************/

/* Standard definitions (stdio.h) */

#define EOF     -1
#define NULL    0
#define TRUE    1
#define FALSE   0
#define MAXSTR  256     /* Maximum string length */
#define FILE    int     /* File descriptor type */

/* Definitions for optimizer */

#define MAXLINES        500     /* Lines of source in memory  */
#define JTSIZE          10      /* Entries in jump list table */
#define LBLLEN          15      /* Maximum label length */
#define OPTIONS         "B:F:J:L:OU:V"  /* Legal options */
#define LBLTERM         ':'     /* Character which terminates a label */
#define SYMLEN          8       /* Symbol (label) length */

#define UNDEF           0       /* Undefined symbol */
#define INBUF           1       /* Symbol defined and in buffer */
#define OUTBUF          2       /* Symbol written to output, removed from
                                   buffer */
#define UNJMPD          3       /* Symbol is label of JMP, replaced with
                                   destination */

/* Structure declarations */

struct lkline {         /* Linked list of lines */
        struct lkline *nextln;  /* Link to next line */
        char *ln;               /* This line */
};

struct trfmn {          /* Transformation from old to new code */
        int nold;       /* Number of lines in old construction */
        int nnew;       /* Number of lines in new construction */
        struct lkline *old;     /* Linked list of old reg. exprs. */
        struct lkline *new;     /* Linked list of new substitutions */
        struct trfmn *nexttr;   /* Link to next transformation */
};

struct jmp_lst {        /* List of lines with jumps to a label */
        int njmp;
        int jlines[JTSIZE];     /* Line nos. containing jump to label */
        struct jmp_lst *jlnext;
};

struct symt {           /* Symbol table entry */
        char sname[SYMLEN+1];
        int lineno;             /* Line number in buffer */
        char scode;             /* Status: undefined, in buffer,
                                   written out, replaced */
        struct jmp_lst *jmplst;
        struct symt *snext;
};

/* Definitions for regular expression code */

#define MAXPAT  (2*MAXSTR)
#define CLOSIZE 1       /* Size of a closure entry */
#define CLOSURE '*'
#define BOL     '^'
#define BOLSTR  "^"     /* String consisting of BOL */
#define EOL     '$'
#define ANY     '.'
#define CCL     '['
#define CCLEND  ']'
#define NEGATE  '~'
#define NCCL    '!'     /* Cannot be the same as NEGATE. */
#define LITCHAR 'c'

#define ENDSTR  '\0'
#define NEWLINE '\n'
#define ESCAPE  '\\'
#define DASH    '-'
```

```
#define GRPST   '('     /* Codes for grouping and subpatterns */
#define GRPEND  ')'
#define GRPPAT  '*'
#define DITTO   -1

/* #define      Z80             /* Insert this define if you have
                                   a Z80 processor. */
```

End Listing Two

Listing Three

```
/* optglob.c -- Global variable declarations for optimizer. */

/*****************************************
*    Copyright 1984 by David L. Fox
*    All rights reserved.
*    Permission is granted for unlimited personal, non-commercial use.
*****************************************/

char **linep;           /* Array of pointers to lines */
int nlines = 0;         /* Number of lines */
int frstln;             /* Index of oldest line in buffer */
int curln;              /* Current line in buffer */
FILE infd;              /* Input file descriptor */
FILE outfd;             /* Output file descriptor */
unsigned maxlines =     /* Lines in buffer */
        MAXLINES;
int verbose = TRUE;     /* Verbose output flag */
int jmpopt = TRUE;      /* Flag for jump optimization */
int optind = 0;         /* Index of next argument */
char *optarg;           /* Pointer to option argument */
struct trfmn *trans;

char *patfile = "opt.dat\0\0\0\0\0\0\0\0";      /* File for optimization
                                                   data */
char *lbl = "[a-zA-Z_\.][a-zA-Z0-9_]*";         /* Label template */
char *jmpins = "\tJ[CEMNOPZ]*\t";               /* Jump instruction
                                                   template */
char *ucjmp = "\tJMP\t";                /* Unconditional jump */

char *lblpat, *jmppat, *ucjpat;         /* Pointers to patterns made
                                           from reg. expr. */

struct symt frstsym = {
        ('\0'),         /* sname */
        -1,             /* lineno */

        0,              /* scode */
        NULL,           /* jmplst */
        NULL};          /* snext */

struct symt *symtab = &frstsym;         /* Start of symbol table */

/* Global variable declarations for regular expression code */

char pat[MAXPAT];       /* Space for pattern */
int lparen, rparen;     /* Parenthesis counts */
char *gstart[10];       /* Pointers to start of grouped text */
char *gend[10];         /* Pointers to end of grouped text */
char *lgp;              /* Pointer to start of last group */
```

End Listing Three

Listing Four

```
/*  opt.c -- Assembly language optimizer */

/*****************************************
*   Copyright 1984 by David L. Fox
*   All rights reserved.
*   Permission is granted for unlimited personal, non-commercial use.
*****************************************/

/*
Performs peephole and jump optimization of an assembly language
source file.  Transformations for peephole optimizer are specified
in file opt.dat as follows:

        regular expressions
        for matched lines
        %
        replacement lines
        %%

Command line syntax is

opt [ -bnn -ffile -o -lre -jre -ure -v ] infile outfile

where the options are

-bnn    Use nn lines in the FIFO buffer (default=1000).
-ffile  Read peephole optimization information from file instead
        of opt.dat.
-o      Do not do jump optimization.
-lre    Labels defined by regular expression re.
-jre    Jump instructions defined by regular expression re.
-ure    Unconditional jump instruction defined by regular
        expression re.
-v      Verbose output off.
*/

#include        "b:optdefs.h"
#include        "b:optglob.c"
#include        "b:restuf.c"

/* Main program */

main(argc, argv)
int argc;
char **argv;
{
char inline[MAXSTR], *s, *lines;
int c, i;
char *p, *q;
FILE patfd;
char *fgets(), *newline(), *makepat(), *strcpy();
struct symt *symp, *chksym();

/* Initialization */
  frstln = 0;
  curln = maxlines - 1;

  /* Create patterns for jumps and labels from regular expressions */
  /* Allocate memory. */
  if ((lblpat = sbrk(MAXSTR)) == -1 ||
      (jmppat = sbrk(MAXSTR)) == -1 ||
      (ucjpat = sbrk(MAXSTR)) == -1)
          error("Not enough memory");

  /* Create patterns. */
  p = lblpat;
  makepat(lbl, '\0', lblpat, &p);
  p = jmppat;
  makepat(jmpins, '\0', jmppat, &p);
  p = ucjpat;
  makepat(ucjmp, '\0', ucjpat, &p);

  /* Allocate memory for groups. */
  for (i = 1; i < 10; ++i)
          gstart[i] = sbrk(MAXSTR);
  if (gstart[10] == -1)
          error("Not enough memory");

  /* Process options. */
  while ((c = getopt(argc, argv, OPTIONS)) != EOF)
  {       switch(c | 0x20) {
          case 'b':       /* Set FIFO buffer size. */
                  maxlines = atoi(optarg);
                  break;
          case 'f':       /* Set data file name. */
                  strcpy(patfile, optarg);
                  break;
          case 'l':       /* R.e. for labels */
                  p = lblpat;
                  makepat(optarg, '\0', lblpat, &p);
                  break;
          case 'j':       /* R.e. for jumps */
                  p = jmppat;
                  makepat(optarg, '\0', jmppat, &p);
                  break;
          case 'o':       /* Omit jump optimization. */
                  jmpopt = !jmpopt;
                  break;
          case 'u':       /* R.e. for unconditional jumps */
                  p = ucjpat;
                  makepat(optarg, '\0', ucjpat, &p);
                  break;
          case 'v':       /* Switch verbose output. */
                  verbose = !verbose;
                  break;
          default:
                  error("usage: opt [ -bnn -ffile -lre -jre -ure -v ]\
   [ infile [ outfile ] ]");
          }
  }

  /* Read optimization data file. */
  patfd = mustopen(patfile, "r");
  gettmpl(patfd);
  fclose(patfd);

  /* Allocate memory for FIFO buffer. */
  if ((linep = sbrk(maxlines*sizeof(char *))) == -1)
          error("Not enough memory");

  /* Set up i/o channels. */
  outfd = stdout;
```

Peephole Optimizer (Listing continued, text begins on page 56)

Listing Four

```
        )
        if (n > 0)      /* No match, skip to next transformation. */
                continue;
        if (verbose)    /* Print last line matched. */
                fprintf(stderr, "MATCH---%s", linep[ln]);
        /* Replace matched lines with improved code. */
        if (!subst(ln,p))
                error("substitute failed");
}
}

/* unjump -- Route jumps to jumps directly to target */
/*      syp is pointer to symbol table entry for label on JMP */
unjump(syp)
struct symt *syp;
{
char *p, dpat[MAXSTR];
int i, errflg;
struct symt *dsyp, *isjmp();
struct jmp_lst *jp;

if (syp->scode == OUTBUF) {
        /* Target is not in memory. */

        fprintf(stderr, "%s not in buffer\n", syp->sname);
        for (jp = syp->jmplst; jp != NULL; jp = jp->jlnext)
                free(jp);       /* Release memory used by jump list. */
        errflg = TRUE;
}
else {  /* Target is in memory. */
        errflg = FALSE;
        if ((dsyp = isjmp(linep[syp->lineno])) == NULL)
        {       message(linep[syp->lineno]);
                error("Not a jump");    /* "Can't happen." */
        }

        /* Make literal pattern for substitutions */
        for (p=syp->sname, i=0; *p != '\0'; ++p)
        {       dpat[i++] = LITCHAR;
                dpat[i++] = *p;
        }
        dpat[i++] = CLOSURE;
        dpat[i++] = LITCHAR;
        dpat[i++] = LBLTERM;
        dpat[i] = '\0';
        /* Replace destination label for every entry in jump list. */
        jp = syp->jmplst;
        for (jp = syp->jmplst; jp != NULL && jp->njmp >= 0; jp = jp->jlnext)
        {       for (i=0; i < jp->njmp; ++i)
                {       if (!subline(jp->jlines[i], dpat, dsyp->sname))
                        {       errflg = TRUE;
                                fprintf(stderr,
                                   "Cannot substitute %s in %s\n",
                                   dsyp->sname, linep[jp->jlines[i]]);
                        }
                        /* Add new reference to jump list */
                        addjmp(dsyp->sname, jp->jlines[i]);
                }
                free(jp);
        }
}
syp->jmplst = NULL;
if (!errflg)    /* Delete label from target. */
{       if (!subline(syp->lineno, dpat, ""))
                fprintf(stderr,
                   "Failed to delete %s in %s\n",
                   dpat, linep[syp->lineno]);
        syp->scode = UNJMPD;
}
}

/* I/O routines for optimizer */

/* outline -- Output a line from FIFO buffer. */

outline()
{
struct symt *s, *islab(), *isjmp();
struct jmp_lst *jp;
if (linep[frstln] != NULL) {    /* If line not deleted */
        fputs(linep[frstln], outfd);
        /* Check for labels and jumps. */
        if ((s = islab(linep[frstln])) != NULL)
                s->scode = OUTBUF;
        if ((s = isjmp(linep[frstln])) != NULL)
        {       s->scode = OUTBUF;
                /* Delete jump list. */
                for (jp=s->jmplst; jp != NULL; jp = jp->jlnext)
                        free(jp);
                s->jmplst = NULL;
        }
        --nlines;
        free(linep[frstln]);
}
/* Check for buffer wrap around. */
if (++frstln >= maxlines) {
        frstln = 0;
}
}

/* gettmpl -- Get templates for changes. */

gettmpl(fd)
FILE fd;
{
int n;
struct trfmn *t;
struct lkline *l, **ll, *mm, *newlk();
char line[MAXSTR], *s, *p;
char *fgets(), *makepat(), *makesub(), *strcpy();

/* Allocate memory for first transformation. */
t = trans = (struct trfmn *) sbrk(sizeof(struct trfmn));
t->nexttr = NULL;

while ((s = fgets(line, MAXSTR, fd)) != NULL)
{       lparen = rparen = 0;    /* Initialize parenthesis counters. */
        ll = &t->old;
        t->nold = t->nnew = 0;
        while (*s != '%' && s != NULL)
        {       p = pat;
                if (makepat(s, '\n', pat, &p) == NULL)
                {       message(line);
                        error ("bad r.e.");
                }
                l = newlk(ll, pat);
                ll = &l->nextln;
                ++t->nold;      /* Increment line count. */
                s = fgets(line, MAXSTR, fd);
        }
        if (s != NULL && strncmp(s, "%%", 2) != 0)
        {       mm = &t->new;
                while ((s = fgets(line, MAXSTR, fd)) != NULL &&
                   strncmp(s, "%%", 2) != 0) {
                        /* Get replacement strings. */
                        if (makesub(line, '\n', pat) == NULL)
                        {       message(line);
                                error("bad sub string");
                        }
                        l = newlk(mm, pat);
                        mm = &l->nextln;
                        ++t->nnew;
                }
        }
        /* Allocate memory for next transformation. */
        t->nexttr = (struct trfmn *) sbrk(sizeof(struct trfmn));
        t = t->nexttr;
        t->nexttr = NULL;
}
}

/* newlk -- Create a new lkline structure. */

struct lkline *
newlk(lkp, str)
struct lkline **lkp;    /* Address of pointer to next member of
   linked list. */
char *str;
{
char *strcpy();
struct lkline *l;

if ((l = (struct lkline *)sbrk(sizeof(struct lkline))) == NULL)
        error("memory full");
*lkp = l;
if ((l->ln = sbrk(strlen(str)+1)) == NULL)
        error("memory full");
strcpy(l->ln, str);
l->nextln = NULL;
return(l);
}

/* sttecpy -- Copy string between start and end pointers. */
char *
sttecpy(dest, start, end)
char *dest, *start, *end;
{
char *p;

for (p = dest; start < end; )
        *p++ = *start++;
*p = '\0';
return(dest);
}

/*****************************************/
/* Symbol table routines for optimizer. */
/* addsym -- Add a label to the symbol table. */

struct symt *
addsym(name)
char *name;
{
char *p, *q, *strcpy();
static struct symt *s = &frstsym, *sret;

strcpy(s->sname, name);
```

(Continued on page 74)

Peephole Optimizer (Listing continued, text begins on page 56)

Listing Four

```
s->lineno = -1;
s->scode = INBUF;
if ((s->snext = malloc(sizeof(struct symt))) == NULL)
        error("symbol table overflow");
sret = s;
s = s->snext;
s->snext = NULL;
s->jmplst = NULL;
return(sret);
}

/* addjmp -- Add line number to list of jumps to label. */

addjmp(name, lineno)
char *name;
int lineno;
{
char *p, *q;
struct symt *symp, *findsym(), *addsym();
int n;
struct jmp_lst **jp;

if ((symp = findsym(name)) == NULL)
{       symp = addsym(name);    /* Symbol not found, add it to table. */
        symp->scode = UNDEF;
}
jp = &symp->jmplst;
/* skip to end of jump list */
while (*jp != NULL && (*jp)->njmp >= JTSIZE)
        jp = &(*jp)->jlnext;    /* Skip to last block of jump list. */
if (*jp == NULL)        /* Allocate space for new block. */
{       if ((*jp = malloc(sizeof(struct jmp_lst))) == NULL)
                error("jump list overflow");
        else    /* Initialize new block. */
        {       (*jp)->njmp = 0;
                (*jp)->jlnext = NULL;
        }
}
(*jp)->jlines[(*jp)->njmp++] = lineno;
}

/* findsym -- Linear search of symbol table. */

struct symt *
findsym(name)

char *name;
{
struct symt *s;

for (s = &frstsym; s->snext != NULL; s = s->snext)
        if (strcmp(s->sname, name) == 0)
                return(s);
return(NULL);
}

/* chksym -- Check for a labeled line or line with a jump. */

struct symt *
chksym(curln)
int curln;
{
char *sttecpy();
struct symt *syp, *s,  *addsym(), *islab(), *isjmp();

        if (syp = islab(linep[curln]))
                syp->lineno = curln;
        if (syp->lineno = -1)   /* First time for this line. */
                syp->lineno = curln;
        if ((s = isjmp(linep[curln])) != NULL)
                addjmp(s->sname, curln);
        if (syp != NULL)
        {       if (match(linep[curln], uc;pat) != NULL)
                        return(syp);    /* Labeled unconditional jump. */
        }
        return(NULL);
}

/* islab -- If str points to a valid label return pointer to its
            entry in symbol table, else return NULL. */

struct symt *
islab(str)
char *str;
{
```

Peephole Optimizer (Listing continued, text begins on page 56)

Listing Four

```
char *p, *q, name[LBLLEN], *amatch(), *sttecpy();
struct symt *syp, *addsym();

p = lblpat;
if ((q = amatch(str, str, &p)) != NULL) {
        if((syp = findsym(sttecpy(name, str, q))) == NULL) {
                        syp = addsym(name);
                }
        }
return(syp);
}

/* isjmp -- If line contains a jump instruction
        return pointer to its destination label symbol table entry
        otherwise return NULL. */
struct symt *
isjmp(line)
char *line;
{
char *p, *q, *r, name[LBLLEN];
struct symt *syp, *findsym();
char *amatch(), *sttecpy();

syp = NULL;
if ((p = match(line, jmppat)) != NULL) {
        q = lblpat;
        if ((r = amatch(p,p,&q)) != NULL) {
                sttecpy(name, p, r);
                if ((syp = findsym(name)) == NULL)
                        syp = addsym(name);
        }
}
return(syp);
}

/*************************************/
/* Utility routines */

/* getopt -- Unpack options from arg list. */
/*      Argc and argv are argument count, and values from main.
        Optstr is a string of legal option letters.
        Getopt returns the next option letter from argv.
        If the letter in optstr is followed by ':' the external
                pointer optarg is set to point to the argument
                following the option letter.
        The external variable optind is set to the index of the
                next argument in the argv array to be processed.
        If an illegal option is found, getopt returns '?'
        If the next argument in argv is not an option
                getopt returns EOF

*/

getopt(argc, argv, optstr)
int argc;
char *argv[], *optstr;
{
static char *cp;
char *p;
char *strchr();

if(optind == 0)
        cp = argv[++optind] + 1;        /* First call */
if(*argv[optind] != '-' || optind >= argc)
        return(EOF);            /* No more options */
if(*cp == '-')
{       ++optind;
        return(EOF);            /* End of options */
}
if((p = strchr(optstr, *cp++)) == NULL)
{       fputs("unknown option: ", stderr);

        putc(*(cp-1), stderr);
        putc('\n', stderr);
        if(*cp == '\0')
                cp = argv[++optind] + 1;
        return('?');
}
if(*(p+1) == ':')
{       if(*cp == '\0')         /* Get argument */
                optarg = argv[++optind];
        else
                optarg = cp;
        cp = argv[++optind] + 1;
}
else
        if(*cp == '\0')
                cp = argv[++optind] + 1;        /* Set up for
                                                   next argv. */
return(*p);
}

/* message -- Send a message to the console (stderr) and return. */
message(s)
char *s();
{
fputs(s, stderr);
putc('\n', stderr);
return(0);
}

/* mustopen -- Attempt to open a file, exit on failure. */

FILE
mustopen(name, mode)
char *name, *mode;
{
FILE fd;
if((fd = fopen(name, mode)) == NULL)
{       fputs(name, stderr);
        error(": cannot open file");
}
return(fd);
}

/* message -- Send a message to the console (stderr) and exit. */
error(s)
char *s;
{
fputs(s, stderr);
putc('\n', stderr);
fclose(stderr);
exit();
}
```

End Listing Four

Listing Five

```
/* restuf.c -- Regular expression routines. */

/* These routines are translations into C of the regular expression
*       matching and pattern generation routines in
*       B.W. Kernighan and P.J. Plauger, "Software Tools in Pascal"
*       chapters 5 and 6.
*   The major extension is the addition of code to allow matching and
*       substitution of parts of expressions using \(...\) for
*       grouping and \n for replacement.
*/

/* match -- Find match anywhere on line. */

match(lin, pat)
char *lin, *pat;
char *amatch();
{
char *pos, *lst, *p;

lst = lin;
for (pos = NULL; *lin != ENDSTR && pos == NULL; ++lin)
{       p = pat;
        pos = amatch(lst, lin, &p);
}
return(pos);
}

/* amatch -- Anchored match */
/* Look for match between lin and pat. */
/* Return pointer to next unmatched character in line,
   or NULL if no match. */
/* lst is pointer to start of line. */
/* lin is pointer to current position in line. */
/* *ppat is pointer to current position in pattern,
   updated if match found. */

char *
amatch(lst, lin, ppat)
char *lst, *lin, **ppat;
{
auto char *lp, *mp, *tp;
auto int level, skip;
char tpat[MAXPAT];
char *strcpy();

while (**ppat != ENDSTR)
{       switch(**ppat) {
        case CLOSURE: {
                *ppat += patsize(*ppat);        /* Step over closure. */
                for (lp = lin; *lp != ENDSTR; )
                        if (! omatch(lst, &lp, *ppat))
                                break;
                if (*(tp = *ppat + patsize(*ppat)) == GRPEND)


                {       skip = patsize(tp);     /* At end of group, skip
                                                   over group closure. */
                }
                else
                        skip = 0;
                /* lp points to input character that made us fail. */
                /* Match rest of pattern against rest of input. */
                /* Shrink closure by 1 after each failure. */
                while (lp >= lin)
                {       tp = *ppat + patsize(*ppat) + skip;
                        if (skip)
                                gend[lparen] = gstart[lparen] +
                                   (int)(lp - lgp);
```

```
                if ((mp = amatch(lst, lp, &tp)) != NULL)
                        if (skip == 0)
                        {       *ppat = tp;
                                return(mp);
                        }
                        else
                        {       *ppat += patsize(*ppat) + skip;
                                return(lp);
                        }
                else
                        lp--;
        }
        return(NULL);
        /* If mp == NULL failure, else success. */
}
case GRPST: {   /* Start of group */
        mp = *ppat;
        *ppat += patsize(*ppat);
        level = ++lparen;
        strcpy(gstart[level], lin);
        lgp = lin;
        if ((lin = amatch(lst, lin, ppat)) == NULL)
        {       *ppat = mp;
                --lparen;
                return(NULL);
        }
        gend[level] = gstart[level] + (int)(lin - lgp);
        break;
}
case GRPEND: {  /* End of group */
        *ppat += patsize(*ppat);
        ++rparen;
        return(lin);
}
case GRPPAT: {  /* Match previously found group. */
        if ((level = (*ppat)[1] - '0') < 1 || level > 9)
        {       message(*ppat);
                error(" amatch: can't happen");
        }
        *ppat += patsize(*ppat);
        if (gend[level] > gstart[level] + MAXSTR/2)
                return(NULL);   /* Group is too long for tpat. */

                /* Make temporary pattern from matched group. */
                for (tp = tpat, lp = gstart[level]; lp < gend[level]; )
                {       *tp++ = LITCHAR;
                        *tp++ = *lp++;
                }
                *tp = ENDSTR;
                tp = tpat;
                if((lin = amatch(lst, lin, &tp)) == NULL)
                        return(NULL);
                break;
        }
        default:
        if (omatch(lst, &lin, *ppat))
        {       *ppat += patsize(*ppat);
        }
        else {
                return(NULL);
        }
}
}
return(lin);
}

/* omatch -- Match one pattern element at pat. */

omatch(lin, linp, pat)
char *lin, **linp, *pat;
{
char *lpos;
int advance;

advance = -1;
lpos = *linp;

if (*lpos == ENDSTR)
        return(FALSE);
switch(*pat)
{
case LITCHAR:
        if (*lpos == pat[1])
                advance = 1;
        break;
case BOL:
        if (lpos == lin)
                advance = 0;
        break;
case ANY:
        if (*lpos != NEWLINE)
                advance = 1;
        break;
case EOL:
        if (*lpos == NEWLINE)
                advance = 0;
        break;
case CCL:

         if (locate(*lpos, &(pat[1])))
                 advance = 1;
         break;

case NCCL:
        if (*lpos != NEWLINE && ! locate(*lpos, &(pat[1])))
                advance = 1;
        break;
default:
        message(pat);
        error("in omatch: can't happen");
}
if (advance >= 0)
{       *linp += advance;
        return(TRUE);
}
else
        return(FALSE);
}

/* patsize -- Returns size of pattern entry at pat. */

patsize(pat)
char *pat;
{
switch (*pat)
{
case LITCHAR:
case GRPPAT:
        return(2);
case BOL:
case EOL:
case ANY:
case GRPST:
case GRPEND:
        return(1);
case CCL:
case NCCL:
        return(pat[1] + 2);
case CLOSURE:
        return(CLOSIZE);


default:
        message(pat);
        error("in patsize: can't happen");
}
}

/* locate -- Look for c in character class at pat. */

locate(c, pat)
char c, *pat;
{
/* int i;
/*
/* /* Size of class is at pat, characters follow. */
/* for(i = *pat; i > 0; --i)
/* {    if (c == pat[i])
/*      {       return(TRUE);
/*      }
/* }
/* return(FALSE);
 C code commented out. */
#asm
; Assembly language version for speed
        POP     B       ;return address
        POP     H       ;pat
        POP     D       ;c
        PUSH    D       ;restore
        PUSH    H       ; the
        PUSH    B       ;  stack
        MOV     C,M     ;get count (*pat)
        MVI     B,0
        MOV     A,E     ;c into A
        INX     H       ;point to first character
#ifdef Z80
        DB      0EDH,0B1H       ;Z80 CPIR instruction
        JZ      locfnd  ;jump if c found
#else
loclp:  MOV     E,M     ;get char

        CMP     E       ;compare with c
        JZ      locfnd  ;jump if match
        INX     H       ;increment pointer
        DCX     B       ;decrement count
        MOV     D,A     ;save A
        MOV     A,B
        ORA     C       ;test for BC == 0
        MOV     A,D     ;restore A
        JNZ     loclp   ;loop if BC != 0
#endif
        LXI     H,FALSE ;return FALSE if not found
        JMP     locend  ;exit from end for profiler
locfnd: LXI     H,TRUE  ;return TRUE if found
locend: DS      0
#endasm
}

/* makepat -- Make pattern from arg, terminate at delim. */

char *
makepat(arg, delim, stpat, ppat)
char delim, *arg, *stpat, **ppat;
{
char  *lp, *starg;
char *lastp, *tmp, *esc();
```

Peephole Optimizer (Listing continued, text begins on page 56)

Listing Five

```
starg = arg;
lastp = *ppat;
while (*arg != delim && *arg != ENDSTR)
{       lp = *ppat;
        if (*arg == ANY)
                addpat(ANY,stpat,ppat);

        else if (*arg == BOL && arg == starg)
                addpat(BOL,stpat,ppat);
        else if (*arg == EOL)
                addpat(EOL, stpat,ppat);
        else if (*arg == CCL)
        {       if (! getccl(&arg, ppat, stpat))
                        return(NULL);
        }
        else if (*arg == CLOSURE && arg > starg)
        {       if (*(lp = lastp) == BOL || *lp == EOL || *lp == CLOSURE)
                        return(NULL);   /* Closure on a null string
                                           not allowed. */
                stclose(ppat,lastp,stpat);
        }
        else if (*arg == ESCAPE && arg[1] == '(')
        {       addpat(GRPST, stpat, ppat);
                arg += 2;
                if (++lparen > 9) return(NULL);
                if ((arg = makepat(arg, delim, pat, ppat)) != NULL)
                        --arg;
                else
                        return(NULL);
        }
        else if (*arg == ESCAPE && arg[1] == ')')
        {       addpat(GRPEND, stpat, ppat);
                ++rparen;
                return(arg+2);
        }
        else if (*arg == ESCAPE && ('0' < arg[1] && arg[1] <= '9'))
        {       addpat(GRPPAT, stpat, ppat);
                addpat(arg[1], stpat, ppat);
                ++arg;
        }


       else
       {        addpat(LITCHAR, stpat, ppat);
                tmp = esc(arg);
                addpat(*arg, stpat, ppat);
                arg = tmp - 1;
        }
        lastp = lp;
        arg++;
}
if (*arg != delim)
        return(NULL);
if(! addpat(ENDSTR, stpat, ppat))
        return(NULL);
if(lparen != rparen)
        return(NULL);
return(arg);
}

addpat(c, stpat, ppat)          /* Add c at ppat, check for overflow. */
char c, **ppat, *stpat;
{
if (*ppat - stpat > MAXPAT)

        return(FALSE);
*(*ppat)++ = c;
return(TRUE);
}

/* getccl -- Expand char class at arg into pat. */

getccl(argp, patp, stpat)
char **argp, **patp, *stpat;
{
char *ccst;
(*argp)++;      /* skip [ */
if (**argp == NEGATE)
{       addpat(NCCL, stpat, patp);
        (*argp)++;
}
else
        addpat(CCL, stpat, patp);
ccst = *patp;
addpat(0,stpat, patp);  /* space for count */
dodash(CCLEND, argp, stpat, patp);
*ccst = *patp - ccst - 1;
return(**argp == CCLEND);
}

/* stclose -- Insert closure entry at *patp. */

stclose(patp, lastp, stpat)
char **patp, *lastp, *stpat;
{
char *pp, *q;

for (pp = *patp - 1; pp >= lastp; pp--)
{       q = pp + CLOSIZE;
        addpat(*pp, stpat, &q);
}
*patp += CLOSIZE;
*lastp = CLOSURE;
}
```

```
/* esc -- Map *str into escaped character, return pointer to
          next character. */

char *
esc(str)
char *str;
{
int n, c;
char *p;
char *strcpy();

if (*str != ESCAPE)
        return(++str);
else if (*(p=(str+1)) == ENDSTR)        /* ESCAPE not special at end. */
        return(++str);
else
{       if (*p == 'N' || *p == 'n')
        {       *str = '\n';
                return(strcpy(++str,++p));
        }
        else if (*p == 'T' || *p == 't')
        {       *str = '\t';
                return(strcpy(++str,++p));
        }
        else if (*p == 'B' || *p == 'b')
        {       *str = '\b';
                return(strcpy(++str,++p));
        }
        else if ((*p | 0x20) == 'r')
        {       *str = '\r';
                return(strcpy(++str,++p));
        }
        else if (*p == '0' && (*(p+1) | 0x20) == 'x')   /* Hex */
        {       p += 2;
                *str = 0xff & xtoi(p);
                return(strcpy(++str,p += 2));
        }
        else if (*p >= '0' && *p <= '7')        /* Octal representation */
        {       for (n=0,c=0; (*p >= '0' && *p <= '7') && n<=3; ++n,++p)
                {       c = (c<<3) + (*p - '0');
                }
                *str = c;
                return(strcpy(++str,p));
        }
        else
        {       *str = *p;
                return(++p);
        }
}
}

/* dodash -- Expand set at src[i] into dest[j],
        stop at delim. */

dodash(delim, srcp, stdest, destp,)
char **srcp, **destp, *stdest, delim;
{
char *src, *cp, *tmp;
int k, junk;

src = cp = *srcp;
while (*src != delim && *src != ENDSTR)
{       if (*src == ESCAPE)
        {       tmp = esc(src);
                addpat(*src, stdest, destp);
                src = tmp-1;
        }
        else if (*src != DASH)
                addpat(*src, stdest, destp);
        else if ( src == cp || src[1] == ENDSTR)

                addpat(DASH, stdest, destp);    /* literal - */
        else if (isalnum(*(src-1)) && isalnum(src[1])
                  && *(src-1) <= src[1])
                {       for(k = *(src-1) + 1; k <= src[1]; ++k)
                                addpat(k, stdest, destp);
                src++;
                }
        else
                addpat(DASH, stdest, destp);
        src++;
}
*srcp -= *srcp - src;
}

/* makesub -- Make substitution string from arg in sub. */

char *
makesub(arg, delim, sub)
char *arg, delim, *sub;
{
char *p, *q, *stsub, *tmp;

stsub = sub;
while (*arg != delim && *arg != ENDSTR && *arg != '\n')
{       if (*arg == '&')
        {       addpat(DITTO, stsub, &sub);
                addpat('\0', stsub, &sub);
        }
        else if (*arg == ESCAPE && arg[1] > '0' && arg[1] <= '9')
        {       addpat(DITTO, stsub, &sub);
```

```
                addpat(arg[1], stsub, &sub);
                ++arg;
        }
        else
        {       tmp = esc(arg);
                addpat(&arg, stsub, &sub);
                arg = tmp - 1;
        }
        ++arg;
}
if (*arg != delim)
        return(NULL);   /* Missing delimiter */
if (! addpat('\0', stsub, &sub))
        return(NULL);   /* No space for sub string */
else
        return(arg);
}

/* subst -- Substitute "sub" for occurrences of pattern. */
/* Modified for optimizer. */

subst(ln, tr)
int ln;
struct trfmn *tr;
{


struct lkline *oo, *nn;
char *newline();
int line, n;

lparen = rparen = 0;
nn = tr->new;
oo = tr->old;
for (n = tr->nold; n > 0; --n)
{       if ((line = ln - n + 1) < 0)
                line += maxlines;       /* Buffer wrap around. */
        if (n <= tr->nold - tr->nnew) {
                delln(line);    /* Not enough replacements. */
                oo = oo->nextln;
        }
        else {
                if (!subline(line, oo->ln, nn->ln))
                        return(FALSE);
                else
                {       oo = oo->nextln;
                        nn = nn->nextln;
                }
        }
}
for (n = tr->nnew; n > tr->nold; --n) {         /* More lines to add. */
        newline("\n");
        if (!subline(curln, BOLSTR, nn->ln))
                return(FALSE);
        else
                nn = nn->nextln;
}
return(TRUE);
}

/* subline -- Do substitution on one line. */
subline(line, pat, sub)
int line;
char *pat, *sub;
{
char new[MAXSTR];
char *p1, *p2, *p3, *last, *pp;
int subbed;
char *amatch(), *strcpy(), *malloc();

p2 = new;
subbed = FALSE;
last = NULL;
p1 = linep[line];
while (*p1 != ENDSTR)
{       if (!subbed)
        {       pp = pat;
                p3 = amatch(linep[line], p1, &pp);
        }
        else
                p3 = NULL;
        if (p3 != NULL && last != p3)


        {       subbed = TRUE;          /* replace matched text */
                catsub( p1, p3, sub, new, &p2);
                last = p3;
        }
        if (p3 == NULL || p3 == p1)
        {       addpat(*p1, new, &p2);
                ++p1;
        }
        else    /* Skip matched text. */
                p1 = p3;
}
if (subbed)
{       if (!addpat(ENDSTR, new, &p2))
        {       return(FALSE);
       }
       else {  /* Copy new line into linep array. */
                free(linep[line]);
                if ((linep[line] = malloc(strlen(new)+1)) == NULL)
                        error("not enough memory");
                strcpy(linep[line], new);
       }
}
return(subbed);
}
```

```
/* catsub -- Add replacement text to end of new. */

catsub(p1, p2, sub, new, pp3)
char *p1, *p2, *sub, *new, **pp3;
{
char *p, *q, *esc();

for (p = sub; *p != ENDSTR; ++p)
{       if (*p == DITTO)
        {
                if (p[1] > '0' && p[1] <= '9')
                {       p1 = qstart[p[1]-'0'];
                        p2 = qend[p[1]-'0'];
                }
                for (q = p1; q < p2; ++q)
                {       addpat(*q, new, pp3);
                }
                ++p;
        }
        else
        {       addpat(*p, new, pp3);
        }
}
}

/* delln -- Delete a line. */

delln(ln)

int ln;
{
free(linep[ln]);
linep[ln] = NULL;
}
```

End Listing Five

Listing Six

```
SHLD\t\([\.a-zA-Z_][\._a-zA-Z0-9+\-]*\)
^\tLHLD\t\1
%
SHLD\t\1\t; redundant LHLD deleted
%%
\tMOV\tA,H
^\tORA\tL
\tJNZ\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\tLXI\tH,0
%
\tMOV\tA,H
\tORA\tL
\tJNZ\t\1\t; LXI H,0 removed following in line test
%%
\tCALL\t\([cen]\.\)
\tJNZ\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\tLXI\tH,0
%
\tCALL\t\1
\tJNZ\t\2\t; LXI H,0 removed following library test
%%
\tCALL\t\([cen]\.\)
\tJZ\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\tLXI\tH,0
%
\tCALL\t\1
\tJZ\t\2\t; LXI H,0 replaced following library test
\tDCX\tH
%%
\tCALL\t\([cen]\.\)
\tJZ\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\tLXI\tH,1$
%
\tCALL\t\1
\tJZ\t\2\t; LXI H,1 replaced following library test
%%
^\tJZ\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\tJMP\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\1\(:\t.*\)
%
\tJNZ\t\2\t; JZ around jump removed
\1\3
%%
^\tJNZ\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\tJMP\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\1\(:\t.*\)
%
\tJZ\t\2\t; JNZ around jump removed
\1\3
%%
PUSH\tH
^\tLXI\tH,\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\tPOP\tD

%
XCHG\t;PUSH H POP D  around LXI H changed to XCHG
\tLXI\tH,\1
%%
PUSH\tH
^\tLHLD\t\([\._a-zA-Z][\._a-zA-Z0-9]*\)
^\tPOP\tD
%
XCHG\t;PUSH H POP D  around LHLD changed to XCHG
\tLHLD\t\1
%%
STA\t\([\.a-zA-Z_][\._a-zA-Z0-9+\-]*\)
^\tLDA\t\1
```

Peephole Optimizer (Listing continued, text begins on page 56)

```
%
STA\t\1\t; redundant LDA deleted
%%
\tJMP\t
^\(^\t[^D][^BSW]\)
%
\tJMP\t
; unreachable code:   \1
%%
\tRET$
^\t
%
\tRET
\t; unreachable code:
%%
\tJMP\t\([\._a-zA-Z][\._a-zA-Z0-9]$\)
^\1\(:\t.*\)
%
\tDS\t0\t; jump to next location removed
\1\2
%%
```

End Listing Six

Listing Seven

```
/* Library routines used by optimizer. */

atoi(str)                   /* Convert str to integer. */
     char *str;
exit()                      /* Return to CP/M */
fclose(fd)                  /* Close file with descriptor fd. */
     FILE fd;
char *fgets(str, nbytes, fd)  /* Read one line (maximum of nbytes) */
     char *str;               /* from file into str. */
     int nbytes;
     FILE fd;
FILE fopen(name, mode)      /* Open file name with mode="r" or "w". */
     char *name, *mode;
fprintf(fd, format [, arg ] ... )  /* Formatted output routine. */
     FILE fd;
     char *format;
fputs(str, fd)              /* Output str to file fd. */
     char *str;
     FILE fd;
free(block)                 /* Release block of memory allocated by */
     char *block;           /* malloc. */
isalnum(c)                  /* Returns non-zero (true) if c is a */
     char c;                /* letter or digit. */
char *malloc(nbytes)        /* Return pointer to block of nbytes */
     int nbytes;            /* of available memory. */
putc(c, fd)                 /* Output character c to file fd. */
     char c;
     FILE fd;
char *sbrk(nbytes)          /* Return pointer to nbytes of memory. */
     int nbytes;            /* Cannot be freed. */
char *strchr(str, c)        /* Return pointer to first occurrence of c */
     char *str, c;          /* in str. */
strcmp(str1, str2)          /* Compare strings 1 and 2. Return 0 if */
     char *str1, *str2;     /* str1==str2, negative if str1<str2, or */
                            /* positive if str1>str2. */
char *strcpy(dest, source)     /* Copy string at source to dest. */
     char *dest, *source;
strlen(str)                 /* Return length of str. */
     char *str;
strncmp(str1, str2, n)      /* Compare first n characters of str1 and */
     char *str1, *str2;     /* str2.  Return 0 if identical, negative */
     int n;                 /* if str1<str2 or positive if str1>str2. */
xtoi(str)                   /* Return integer value of hexadecimal */
     char *str;             /* representation in str. */
```

End Listings

(Continued from page 29)

Dr. Dobb's Clinic (Listing continued, text begins on page 20)

Listing Four

```
/* Mitchell-Moore QRNG in C.  Presumes the existence of a */
/* simpler QRNG named r0rand() and its seeder, r0seed(x). */

extern void r0seed();
extern unsigned r0rand();

static unsigned xa[55];
static int j,k;

rseed(x)
unsigned x;
{
     r0seed(x); /* initialize rand0() predictably */
     for(j = 0; j < 55; ++j) xa[j] = r0rand();
     j = 23;
     k = 54;
}

rand()
{
unsigned x;
     x = xa[k] += xa[j];
     if (j==0) j = 55; --j;
     if (k==0) k = 55; --k;
     return(x);
}
```

End Listings

Small-C Update

by J. E. Hendrix

What's the author of Small-C V2 been up to lately? Improvements, bug fixes, and a macro assembler.

It has been about a year since the last *Dr. Dobb's* article on the Small-C compiler, and many readers may be wondering about further developments. I have received a number of queries, mostly about structures and additional data types, but for several reasons I have not developed Small-C further.

For one thing, additions would affect nearly every part of the compiler, so it makes sense to clean up the overall structure of the compiler before proceeding. Rather than bloating the compiler with code, I would like to streamline its design, reduce its size, and improve its efficiency before adding more features. Then there is the need to free Small-C from dependence on other packages for assembling and linking its output. I consider this capability more important than additional compiler features. All of this amounts to a considerable project, and I just haven't had the time recently to spend on it.

I also wonder whether Small-C really needs to become a full C compiler to be a useful tool. It is already suitable for many applications; text processing and system utilities are obvious examples. With plenty of full-featured C compilers available at reasonable prices, I haven't felt a great urge to take Small-C beyond its present entry-level stature.

I have sent the compiler to individuals and companies involved in auto-ignition control; manufacturing of CRT terminals, electronic games, microcomputers, and circuit cards; mental health; publishing; materials management; telecommunications; engineering research; control of devices like transmitters and telescopes; and education. I have noticed an increasing interest in Small-C in education circles—high school, vocational-technical, and college. Because it is small, is written in a popular high-level language, and generates readable assembly language code, Small-C is ideal for studying what compilers do and how they work. Its one-pass algorithm and the fact that input and output default to the console make it especially convenient for studying the relationship between compiler input and output. The Small-C compiler is also a great guinea pig for term projects, giving hands-on experience in compiler construction to would-be computer scientists. And *The Small-C Handbook* makes it easy for users to quickly familiarize themselves with the language and the compiler.

I maintain a mailing list of Small-C users whom I try to keep informed of bug fixes and improvements. But because Small-C may be distributed freely (on a noncommercial basis), many people have copies that they obtained indirectly and that may be out of date. If you have version 2.1 with the new library that was introduced in May and June 1984 (*DDJ* No. 91 and No. 92), then the fixes reported below will bring you up to date.

I am still distributing Small-C for $25, *The Small-C Handbook* for $17.95, and a package of Small-C software tools for $35. Naturally, complete source code is included. To order, send a check or money order to me at the address listed at the end of the article. Include $4 for post-

---

J. E. Hendrix, Box 8378, University, MS 38677.

age and packaging plus $6 for overseas air mail. Send a self-addressed, stamped envelope for information.

People frequently ask for diskette formats other than 8-inch SSSD and North Star 5¼-inch, making it necessary for me to refer them elsewhere, and many people to whom I have sent Small-C and the tools have machines with different diskette formats, CPUs, and operating systems. In most cases, however, I do not know where they stand in porting the software or how willing they are to take referrals. So if you are running Small-C 2.1 (with the new library) on anything except an 8-inch or North Star CP/M machine and are willing to help distribute it, please let me know. Indicate (1) your machine, (2) your operating system, (3) the diskette formats you can supply, and (4) whether or not you have the Small-Tools package.

Redundant Code

David Bookbinder has observed that Small-C generates redundant code for deallocating local variables at the end of a function when the last statement is a return. The same is true if a return is the last statement in any compound statement that declares local variables. This is harmless, but it unnecessarily adds to the size of programs and it really is untidy.

Likewise, if the last statement in a function is a goto, unnecessary code for deallocating local variables and a RET is generated: because a goto would block the exit path from a function, there is no need for such code. (The compiler already eliminates the RET if the last statement is a return.) There is no problem with goto's terminating nested compound statements because goto's are not allowed if variables are declared anywhere after the opening brace of the function body—so there are no variables to deallocate.

The following patch eliminates this redundant code:

(1) Add the following line to the end of the file CC.DEF:

```
#define STLABEL 14
```

(2) Change the line in newfunc( ) (file CC12.C) that contains a call to statement( ) to read:

```
        statement( );
    #ifdef STGOTO
        if(laststst != STRETURN && laststst != STGOTO)
            ffret( );
    #else
        if(laststst != STRETURN) ffret( );
    #endif
```

(3) Change the line in statement( ) (file CC13.C) that contains a call to dolabel( ) to read:

```
    else if(dolabel( )) laststst=STLABEL;
```

(4) Change the line in compound( ) (file CC13.C) that contains a call to modstk( ) to read:

```
    #ifdef STGOTO
        if(laststst != STRETURN && laststst != STGOTO)
    #else
        if(laststst != STRETURN)
    #endif
            modstk(savcsp, NO); /* delete local variable
                space */
        csp=savcsp;
```

(5) Recompile, assemble, and link the compiler.

FPUTC Placement

A problem has been reported concerning the placement of the module FPUTC in the relocatable library CLIB.REL. Its placement between FPUTS and FREAD makes it precede PUTCHAR, which makes reference to it. Because L80 can't load a module that it has already passed, programs (like "words" on page 31 of *The Small-C Handbook*) that reference putchar( ) but not fputc( ) will not link properly. The correction for this condition is to move FPUTC down in CLIB.REL so that it falls between FOPEN and FREE. This is done by invoking LIB80 as follows:

```
A>LIB80
*NEWLIB=
*CLIB<. . FPUTS>
*CLIB<FREAD . . FOPEN>
*CLIB<FPUTC>
*CLIB<FREE . . >
*/E
```

You should also change the placement of FPUTC in the file NEWLIB3.SUB so it will order the modules correctly if you ever build a new library from scratch. As a precaution, you should have LIB80 list the contents of the new library (see notes below) then verify that the order exactly matches NEWLIB3.SUB. Finally, delete CLIB.REL and rename NEWLIB.REL to CLIB.REL.

fflush( ) and Auxiliary Buffers

The function fflush( ) attempts to flush an auxiliary buffer even if the file were opened for reading only. In this case, the auxiliary flush routine returns an error condition when a normal return without taking any action would have been appropriate. Before auxiliary buffering was added, it was not necessary to check the open mode because the "dirty buffer" status was being checked and it could not have been set for read-only files. This fix verifies the open mode before any attempt to flush a buffer. An attempt to flush a closed file now gives a normal return. Modify fflush( ) as shown below, compile, assemble, and replace in CLIB.REL:

```
FFLUSH.C
    . . .
    fflush(fd) int fd; {
        if(Umode(fd) & WRTBIT) {
            if((Uauxsz && Uauxsz[fd] && Uauxfl(fd))||
                (!isatty(fd) && Udirty[fd] &&
```

```
            Usector(fd, WRTRND))) {
            Useterr(fd);
            return (ERR);
            }
        }
    return (NULL);
    }
```

Reset Auxiliary Buffer Pointers

When an fd that has an auxiliary buffer (because of a previous call to auxbuf( )) is closed and reopened, the next-byte and end-of-data auxiliary buffer pointers are not reset. This causes improper reading and writing. You must revise two library modules to correct this problem: CSYSLIB and AUXBUF. These modules should be revised as indicated below, compiled, assembled, and replaced in CLIB.REL:

CSYSLIB.C

```
...
int
   *Uauxsz,   /* addr of Uxsize[ ] in AUXBUF */
    Uauxin,   /* addr of Uxinit( ) in AUXBUF */
    Uauxrd,   /* addr of Uxread( ) in AUXBUF */
    Uauxwt,   /* addr of Uxwrite( ) in AUXBUF */
    Uauxfl,   /* addr of Uxflush( ) in AUXBUF */
...
Uopen(fn, mode, fd) char *fn, *mode; int fd; {
    char *fcb;
    if(!strchr("rwa", *mode)) return (ERR);
    Unextc[fd] = EOF;
    if(Uauxin) Uauxin(fd);
    if(strcmp(fn, "CON:")==0) {
       Udevice[fd]=CPMCON; Ustatus[fd]=
          RDBIT|WRTBIT; return (fd);
       }
...
```

AUXBUF.C

```
...
extern int *Uauxsz, Uauxin, Uauxrd, Uauxwt,
   Uauxfl, Ustatus[ ];
...
auxbuf(fd, size) int fd; char *size; { /* fake
   unsigned */
   if(!Umode(fd)||!size||avail(NO) <
         size||Uxsize[fd])
      return (ERR);
   Uxaddr[fd] = malloc(size); Uxinit(fd);
   Uauxin = Uxinit;
         /* tell Uopen( ) where Uxinit( ) is */
   Uauxrd = Uxread;
         /* tell Uread( ) where Uxread( ) is */
   Uauxwt = Uxwrite;
         /* tell Uwrite( ) where Uxwrite( ) is */
   Uauxsz = Uxsize;
         /* tell both where Uxsize[ ] is */
   Uauxfl = Uxflush
         /* tell fflush( ) where Uxflush( ) is */
   Uxsize[fd] = size;
         /* tell Uread( ) that fd has aux buf */
   return (NULL);
   }
/*
** Initialize aux buffer controls
*/
Uxinit(fd) int fd; {
   Uxnext[fd] = Uxend[fd] = Uxaddr[fd];
   Uxeof[fd] = NO;
   }
...
```

Optimizing Tests Against Zero

Small-C optimizes tests against the value zero. However, in its enthusiasm, it overlooks certain unary operators that might spoil its efforts. So it optimizes if(!(i==0)) . . . ; as though it were if(i==0) . . . ;. You can fix this by making the changes indicated below to file CC32.C of the compiler:

CC32.C

```
...
hier13(lval) int lval[ ]; {
...
   else if (match("~")) {        /* ~ */
      ...
      return (lval[7]=0);
      }
   else if (match("!")) {        /* ! */
      ...
      return (lval[7]=0);
      }
   else if (match(" - ")) {      /* unary - */
      ...
      return (lval[7]=0);
      }
   ...
```

Lexcmp( )

Lexcmp( ) erroneously returns zero, indicating a match, when the first bytes that are not identical are upper and lower cases of the same letter; for instance, "Happy" and "HAY" appear to match. Revise lexcmp( ) as indicated below, recompile, assemble, and replace in CLIB.REL:

```
LEXCMP.C
  ...
  lexcmp(s, t) char *s, *t; {
     while(lexorder(*s, *t) == 0)
        if(*s++) ++t;
        else return (0);
     return (lexorder(*s, *t));
     }
  ...
  char Ulex[128] = {
          /**** NUL '- / ****/
            0,  1,  2,  3,  4,  5,  6,  7,  8,
            9, 10, 11, 12, 13, 14, 15, 16, 17,
           18, 19, 20, 21, 22, 23, 24, 25, 26,
           27, 28, 29, 30, 31, 32, 33, 34, 35,
           36, 37, 38, 39, 40, 41, 42, 43, 44,
           45, 46, 47,
          /**** 0-9 ****/
           65, 66, 67, 68, 69, 70, 71, 72, 73,
           74,
          /**** : ; < = > ? @ ****/
           48, 49, 50, 51, 52, 53, 54,
          /**** A-Z ****/
           75, 76, 77, 78, 79, 80, 81, 82, 83,
           84, 85, 86, 87, 88, 89, 90, 91, 92,
           93, 94, 95, 96, 97, 98, 99, 100,
          /**** [ \ ] ^ U ` ****/
           55, 56, 57, 58, 59, 60,
          /**** a-z ****/
           75, 76, 77, 78, 79, 80, 81, 82, 83,
           84, 85, 86, 87, 88, 89, 90, 91, 92,
           93, 94, 95, 96, 97, 98, 99, 100,
          /**** { ¦ } ~ ****/
           61, 62, 63, 64,
          /**** DEL ****/
           101
          };
...
```

This patch also makes a minor change for improved efficiency and removes leading zeroes from the values in Ulex[ ] so they will not look like octal values to full C compilers.

Macro Storage

Experience has shown that the amount of storage space allocated for macro replacement strings is probably too small for the number of macros allowed. Also, the number of macros allowed may not be large enough for some programs.

The following parameters in CC.DEF have been increased to help this situation:

```
#define     MACNBR     130
#define     MACQSIZE   (MACNBR*7)
```

You will need to recompile the compiler to effect this change. However, if you are not having problems with overflowing macro storage space, you may ignore this change.

fread( ) and fwrite( )

The functions fread( ) and fwrite( ) do not return proper values. They each take a pair of arguments indicating how many items to transfer and the size (in bytes) of an item. They are supposed to return the number of "items" actually transferred, but instead they are returning the number of "bytes" transferred.

This edit fixes these problems and makes some changes for improved efficiency in write( ). Fortunately, these changes are easily made without recompiling the compiler. Look for these functions in the files FREAD.C and FWRITE.C. Using AR.COM, extract these files from CLIB.ARC and change them as indicated below:

```
FREAD.C
  ...
fread(buf, sz, n, fd) char *buf; int sz, n, fd; {
     return (read(fd, buf, n*sz)/sz);
     }
  ...

FWRITE.C
  ...
fwrite(buf, sz, n, fd) char *buf; int sz, n, fd; {
     if(write(fd, buf, n*sz) == -1) return (0);
     return (n);
     }
  ...
write(fd, buf, n) int fd, n; char *buf; {
     char *cnt; /* fake unsigned */
     cnt = n;
     while(cnt--) {
        Uwrite(*buf++, fd);
        if(Ustatus[fd] & ERRBIT) return (-1);
        }
     return (n);
     }
```

Then compile them and put them in CLIB.REL as indicated in the notes at the end of this article. Finally, using AR.COM, put them back into CLIB.ARC.

Uputsec( )

The function Uputsec( ) in CSYSLIB always assumes that the sector being written is at the end of the file. Consequently, it always initiates the next sector with the value 1A. This is bad if you are using random access techniques, because it overwrites existing data with 1A bytes.

This fix makes Uputsec( ) realize that it is not necessarily at end of file. Using AR.COM, extract CSYSLIB.C from CLIB.ARC and change it as indicated below:

```
CSYSLIB.C
   . . .
Uputsec(fd) int fd; {
   if(fflush(fd)) return (NO);
   Uadvance(fd);
   if(Ustatus[fd]&EOFBIT||Usector(fd, RDRND))
      pad(Ubufptr[fd], CPMEOF, BUFSIZE);
   return (YES);
   }
```

Then compile it and put it in CLIB.REL as explained in the notes. Finally, using AR.COM, put it back into CLIB.ARC.

hier1( )

A problem in hier1( ) (file CC31.C) causes the operators += and -= in expressions like i += p (where p is a pointer) to erroneously assign to p instead of i. If p were a local pointer, then the assignment is made to the address corresponding to the value of i, thus corrupting the system. This problem occurred because, in the effort to assign attributes to a subexpression from included primitives and subexpressions, nothing was done to isolate the attributes of the receiving field.

This is fixed by allocating a short temporary array lval3[2] for passing to store( ). Using AR.COM, extract everything from CC.ARC (or just CC3.C, CC31.C, CC32.C, and CC33.C if you already have CC?.REL files). Change hier( ) in CC31.C as indicated below:

```
CC31.C
   . . .
hier1(lval) int lval[ ]; {
   int k,lval2[8], lval3[2], oper;
   . . .
   if(k==0) {
      needlval( );
      return 0;
      }
   lval3[0] = lval[0];
   lval3[1] = lval[1];
   if (lval[1]) {
      if(oper) {
         push( );
         rvalue(lval);
         }
   . . .
   store(lval3);
   return 0;
   }
```

Then compile the compiler (only part 3 if you already have CC?.REL files for the other parts) and link the parts with CLIB.REL, giving a new CC.COM. Finally, replace CC31.C in CC.ARC.

Leading Blanks

Frank Hayes has reported that because CP/M Plus doesn't load the command tail into its buffer (at 80H) with a leading blank, as CP/M 2.2 does, the Small-C function Uparse( ) misses a character, often causing it to fail to recognize a redirection specification.

This correction to Uparse( ) removes the assumption of a leading blank, making Uparse( ) work correctly with either CP/M 2.2 or CP/M Plus. Using AR.COM, extract CSYSLIB.C from CLIB.ARC. Change the first few lines in Uparse( ) to read as follows:

```
CSYSLIB.C
   . . .
Uparse( ) {
      char *count, *ptr;
      count = 128;
      ptr = Ualloc((count = *count&255)+1, YES);
      strncpy(ptr, 129, count);
      Uvec[0]=Uarg1;
   . . .
```

Compile and assemble CSYSLIB and replace it in CLIB-.REL. Using AR.COM, also replace the source in CLIB-.ARC. Finally, compile, assemble, and link any Small-C programs that run under CP/M Plus.

Notes

(1) In the source statements shown above, the capital letter "U" prefixes many global variable and function names. Older copies of Small-C used the underscore character "_" in this position to avoid conflicts with user-declared names. Because some versions of Macro-80 will not handle a leading underscore on external references, underscores were changed to the letter "U" to make them acceptable to Macro-80. There is no significance to the use of upper case except that it stands out in the source listing as a uniqueness prefix instead of as part of the name proper. In making corrections, you should use "U" or "_" according to the usage in your existing runtime library.

(2) The commands to replace a module in CLIB.REL are:

```
LIB80
*NEWLIB=
*CLIB<. . prevmodule>
*module
*CLIB<nextmodule . . >
*/E
```

"Prevmodule" is the name of the module preceding the one being replaced, "module" is the one being replaced, and "nextmodule" is the one following the one being replaced. This will leave the old library named CLIB.REL and create a new one named NEWLIB.REL. When you are sure the new library is okay, delete the original library and rename NEWLIB.REL to CLIB.REL. To find the order of the modules in the original library or to verify the new library, execute the following commands:

```
LIB80
*libname/L
*^C
```

"Libname" is the name of the library being checked.

Addendum—Small-Mac Assembler

A companion assembler to the Small-C compiler is now available. Small-Mac is a complete relocatable macro assembler package written in Small-C, for use with the Small-C compiler. It runs on 8080/Z80 machines under CP/M. Its features include low cost, ease of use, CPU adaptability, source code distribution, and educational value.

The following programs are included:

MAC	macro assembler
LNK	linkage editor
LGO	load-and-go loader
LIB	library manager
CMIT	CPU configuration utility
DREL	dump relocatable files

MAC is a two-pass, table-driven, relocatable macro assembler. It "learns" the target machine from a machine instruction table (MIT), which is created with a text editor and compiled with the CMIT configuration utility. It generates relocatable object modules in the 8-bit Microsoft format. MAC is invoked with a simple command syntax and issues descriptive error messages.

LNK combines specified object modules with modules found in specified libraries to create executable programs. Its default output is a standard .COM file. However, it can generate "load-and-go" (.LGO) files for execution at any desired address.

LGO loads and optionally executes .LGO files. It provides a convenient way to invoke operating system extensions at cold start time.

LIB builds, maintains, and lists the contents of LNK compatible libraries.

CMIT compiles machine instruction tables, lists them, and optionally configures the assembler with the resulting object table.

DREL produces a formatted dump of .REL and .LIB files.

Small-Mac instruction operands may contain expressions of any complexity. Expression operators and precedence rules follow the C language. The Small-Mac macro facility is easy to learn, remember, and use. Conditional assembly and repeat pseudo-ops are not supported at this time.

Small-Mac comes on two 8-inch SSSD diskettes containing both source and object files. A 64-page manual is included. Send $30 (plus $4 for postage and handling, plus $6 for overseas air mail) to:

J. E. Hendrix
Box 8378
University, MS 38677

Asynchronous Protocols

by David W. Carroll

A status report on what Bell Labs, with some help from Ward Christensen and others, hath wrought.

From its quiet beginnings in 1977 in Ward Christensen's MODEM program for data exchanges between dissimilar CP/M 8-bit computers, the XMODEM communications protocol has grown until it is now supported by almost every commercial communications package for 8- and 16-bit micros, by numerous public domain terminal programs, by most remote bulletin board and file exchange systems, and even by a recent implementation on CompuServe. But is XMODEM the only answer? This article focuses on the state of currently available asynchronous protocols and on the reasons for their development.

Why Protocols?

Software communications protocols are often necessary to allow communications between computer systems. Predefined protocols are required in many cases to allow reliable, error-free data communications between dissimilar computer systems. Lack of hardware DMA or interrupt capability, error checking and correction of data, binary file transmission, use of different microprocessors and disk operating systems, and provision for micro-to-mainframe links, multiple file transfers, and unattended operation are several specific reasons for using protocols.

David W. Carroll, P.O. Box 699, Pine Grove, CA 95665.
Copyrght © 1985 by David W. Carroll.

Background

Most microcomputer systems support asynchronous serial communications. This means that each byte of data is sent as a self contained unit; the data flow is made up of discrete characters, each completely independent from the others. To allow the receiving computer to detect and interpret incoming data, a start bit is sent, followed by a given number of data bits making up the character (usually 5, 7, or 8), followed by one or two stop bits. Bell Laboratories developed this technique in 1924 for use in teletypewriter communications systems. The receiving system must resynchronize with each character sent.

In the 1950s, military and commercial teletypewriter systems implemented a simple method of error checking by adding a bit to the data to indicate the parity of the data bits making up each character. Parity checking indicates to the receiver that the number of "1" bits in the character should be either odd or even (hence "odd" and "even" parity schemes). Parity checking provides about a 94 percent probability of detecting an error in received data. There is no method for correcting a detected parity error.

A 5-bit code (like Baudot) allows for a total of 32 characters (58 characters with the figures/letters mode-shifting scheme used in telex systems) and is still used in telex communications today. Many other character-coding schemes have been developed, but the 7-bit United

States American Standard Code for Information Interchange (USASCII or simply ASCII), defined in the ANSI X3.1 standard of 1967, is the most widely used standard for micros. This code has 128 possible characters and includes numbers, upper- and lower-case English alphabet, punctuation, and control codes.

The 7-bit ASCII code, with or without parity checking, has long been a standard for time-shared mainframe remote terminal communications, mainly because the Teletype ® model 33 machine was the most common time-sharing terminal during the 1960s and 1970s.

Simple Protocols

The advent of microcomputers and low-cost modems in the past 10 years has generated a need for a simple, inexpensive and reliable way to exchange program and data files containing 8-bit binary data, as well as text files made up of standard ASCII characters. The IBM PC further expanded this need with its extended 8-bit character set made up of 256 characters. As more and larger data files are being transmitted over ordinary telephone lines, more elaborate error detection and automatic correction are also required.

In asynchronous time-sharing remote links, data is sent arbitrarily by the transmitting system, without regard for the capability of the receiving system to accept it. When the character is sent on to a terminal simply for display or for storage in a memory buffer, this does not present a problem. However, when files of data are transferred, the receiving system periodically must dump the data to a disk file. In many hardware configurations, this requires all of the system's resources; it can no longer handle incoming data during the dump period. Thus, to avoid data loss the receiver must have some way to halt the data flow from the sender.

Some simple protocols have been developed to allow unformatted text transmission between mainframes and microcomputers. Three basic methods control the data flow: handshaking, delay, and interruption. Use of handshaking involves sending a character (usually a carriage return) at the end of each line and requires the return of a character (usually a line feed) before flow resumes. The second alternative is for the sending system to delay a predetermined number of seconds after transmitting each line of text to allow the receiving system to empty its data buffer when required; this method works, but it can slow down communications dramatically. The interruption method has the system send data until it receives an interrupt character then pause until it receives the resume character. This protocol is perhaps the most efficient because it allows continuous transmission until the buffer is full, ensuring maximum throughput.

CR-LF Protocol

One common file transfer method used to prevent buffer overflow in time-sharing systems is to transmit one ASCII line of text at a time, including the carriage return character, then to wait for the receiver to send a line feed as acknowledgement. Often a delay of several seconds is added before the next line is transmitted. CR-LF is the most common example of the handshaking technique, although prompt sensing is another popular method used for uploading to an editor or BASIC interpreter.

XON—XOFF File Capture Protocol

Possibly the most common text file transfer protocol in use today is the file capture or XON–XOFF protocol offered on almost all time-shared mainframe computers. This protocol detects two characters—usually ASCII DC1 and DC3 (^Q and ^S). XOFF or ^S pauses transmission and XON or ^Q resumes transmission. This protocol, another holdover from teletypewriter terminals, was used on teleprinters for paper tape operations.

Today, this file capture mode is used to transfer text files between dissimilar computer systems. Data is

captured to a memory buffer, and XOFF is sent when the buffer is nearly full. When transmission stops, the data in the buffer is written to disk. Care must be taken to ensure that the text file does not contain any ^Q or ^S characters (as in WordStar files).

This protocol is supported by CompuServe, The Source, all three major data networks, and most time-sharing host computers and bulletin board systems. It supports only parity checking and 7-bit ASCII file transfers on most systems.

HEX File Conversions

Many older host computers and front-end communications processors do not support 8-bit data transfers over asynchronous links. How then can these systems (like CompuServe until recently) send binary data files?

A program converts the binary data to ASCII characters representing the hexidecimal value of the data. Thus, each 8-bit data byte ends up as two ASCII characters. For example:

Binary		ASCII
0110 1011	→	6B
1110 0001	→	E1
0011 1111	→	3F

These "characters" can be sent easily as text—but what about errors? To resolve this problem, a line format is used that includes a checksum of the characters, similar to Intel Hex format (CP/M-80, etc.). Obviously, this method is both inefficient and time consuming: it involves translations at both ends and more than doubles the transmitted size of the original file.

CompuServe supports this protocol, but it is becoming less popular than some others, like XMODEM.

XMODEM

Determined to solve the problem of file transfer, Ward Christensen, a skilled programmer in Chicago, designed a block-oriented file transfer technique that allows for text *and* binary file transfer with simple, yet effective, error checking. His first version of the program, called MODEM, was written in September 1977 and appeared on CP/M User Group Disk 25. It allowed two microcomputer users, each running the CP/M 80 operating system, to exchange both ASCII text files and 8-bit binary .COM files by modem or direct connect. (In the early days, there were many different types of CP/M systems, and most disk formats were not compatible, so directly transferring files over a communications link was often the only way to move programs from one type of computer to another.)

MODEM protocol transfers are made up of blocks of 132 bytes: four header bytes, consisting of a start of header (SOH) ASCII character, the 8-bit block number, the one's complement of the block number, and the 8-bit checksum, plus 128 data bytes. The receiver controls the sending of each block and any necessary retransmissions by sending an acknowledge (ACK), negative acknowledge (NAK), or cancel (CAN) ASCII code in response to each block. Timeouts are used as a further check of transmission accuracy and for resynchronization.

By 1979, BYE, an unattended system control program, and XMODEM, the unattended version of MODEM, were operating in bulletin boards across the country. Most file transfer systems were based on MODEM, BYE, and XMODEM, as well as on the programs written (in part by Ward Christensen) for one of the first microcomputer-based Computer Bulletin Board System (CBBS) ® in Chicago.

A large number of "improvements" to the first MODEM program have been written, and dozens of commercial and public domain programs now support the XMODEM protocol. Recently, the basic protocol definition was expanded to include optional Cyclic Redundancy Check (CRC) error checking, an improvement on the simple checksum method used in the original programs. CRC ensures a 99.99 percent probability of detecting errors. The CRC polynomial is specified by the CCITT international telecommunications standards organization.

MODEM7 Protocol

Mark M. Zeigler and James K. Mills wrote the MODEM7 CP/M 80 program in 1980. The MODEM7 protocol is an adaptation of the original MODEM/XMODEM protocol defined by Ward Christensen. The MODEM7 program adds the ability for wildcard and multiple file transfers to the basic XMODEM protocol, as well as optional CRC error checking.

Bruce R. Kendall translated the MODEM7 program to 8086 code for CP/M 86 and MSDOS in 1983; it is still used by many as a primary communications program. The MODEM7 protocol is supported by the FIDO bulletin board software.

Interestingly, Crosstalk and Crosstalk XVI (probably the most popular commercial communications program) do not properly support XMODEM transfers to all systems and do not support CRC mode at all. A public domain subprogram is available for 16-bit systems to correct this problem.

KERMIT Protocol

KERMIT (the protocol, not the frog) was developed at Columbia University and released to the public domain to allow efficient transmission of 8-bit binary data on systems that are hardware limited to 7-bit ASCII. The KERMIT method encodes the binary data into acceptable ASCII characters for transmission, but it uses more than four bits of each character (see the HEX scheme above) and uses a block rather than a line error-checking protocol to achieve much greater efficiency. KERMIT also allows multiple file transfers and master/slave operations.

KERMIT is available for most microcomputers and many mainframes at little or no cost. It presently is supported by the FIDO bulletin board system and Telois commercial micro communications software. It may become a widely used standard in the near future.

Microcom Networking Protocol

The Microcom Networking Protocol (MNP) is an interesting attempt by a commercial company to define an industry standard. Microcom, maker of modems and communications software, defined their protocol in 1982. It has been accepted by IBM, Tymnet, Telenet, and some others as a standard. Also, the data networks accept it as a way to provide an error-

free "local" link to their access nodes.

MNP is used in Microcom's ERA 2 communications software and in IBM's Personal Communications Manager program (written by Microcom). It is also supported by the latest version of RBBS-PC (12-4A1).

CompuServe A & B Protocols

CompuServe has defined two protocols for use with its VIDTEX service and provides software to support them on several computers, including most TANDY (TRS-80) computers, the Apple, and the IBM-PC.

CompuServe A protocol, originally developed to allow binary file transfers from the mainframes used by the service, was later updated to the current protocol. It is used to upload and download binary files. CompuServe B protocol is used primarily with Tandy (Radio Shack) TRS-80 computers for file transfers and support of full-color video graphics (VIDTEX) capabilities.

CompuServe also supports BIN and HEX file 7-bit ASCII transfers in modified Intel Hex format.

ANSI X3.28 Protocol

With the demand for reliable communications, it is surprising that the ANSI X3.28 protocol (originally defined in 1971) was not implemented on microcomputers until this past year. The X3.28 protocol is similar to MODEM7 and KERMIT. It provides block transmission with CRC-type error checking, remote system master/slave control, and multiple file transfers.

Artisoft Inc. of Tucson, Arizona, has included X3.28 protocol support in its Envoy-PC communications software, as well as file capture and XMODEM. Recently, X3.28 has received attention as a possible protocol for use in local area networks.

Specialized Protocol Software

Several software vendors have developed their own protocols for use between systems running their programs. Some of these include the following.

Crosstalk

Crosstalk and Crosstalk XVI by Microstuf both support a 256-byte block, error-checking protocol (also called CLINK) similar to MODEM7 that allows multiple file transfers. Later versions of Crosstalk also support XMODEM.

Smartcom

Smartcom and Smartcom II support the Hayes Verification Protocol, a special error-checking protocol designed by Hayes Microcomputer products for file transmission. Smartcom II also supports the XMODEM protocol.

Telink

The Telink protocol is supported by the Telink program and Minitel public domain communications program, written by Tom Jennings, and by the FIDO bulletin board system, also authored by Jennings. All programs support MODEM7 and XMODEM as well.

Move-It

Move-It from Woolf Software uses a proprietary "packet" protocol with 16-bit checksums, message numbering, and remote multiple file transfer capability. Move-It does not support XMODEM.

MITE

MITE (Mycroft Intelligent Terminal Emulator) from Mycroft Labs supports all of the above protocols (except Telink) plus the following:

- XMODEM checksum and CRC
- MODEM7 checksum and CRC (called XMODEM/Batch)
- MITE batch block protocol (see *DDJ*, August 1982)
- IBM PC (text protocol from IBM Asynchronous Support Program)
- TEXT (simple Hex transfer protocol)

ASCOM

The ASCOM program from Dynamic Microprocessor Associates supports XMODEM (called CPMUG) as well as two checksum block protocols, BLOCK and BLOCKV.

Summary

A number of asynchronous protocols have evolved to fill the need for reliable file transfers of both text and binary data between various types of mainframes and microcomputers. The most widely used standard is the XMODEM protocol originally defined by Ward Christensen, but even this protocol has four possible formats (checksum/CRC and single file/batch). Most commercial and public domain file transfer programs support at least the basic checksum, single file version of the XMODEM protocol. XMODEM is simple, fairly unambiguous, and easily implemented in a variety of languages, from C to BASIC.

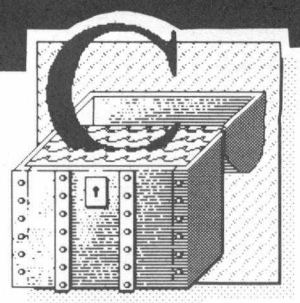
Transfers from 7-bit mainframe host systems are still a clumsy and time-consuming process. The KERMIT protocol, which improves the efficiency of these transfers, is now available in a number of formats for many different computers.

Several other proprietary protocols have been developed, but none has received the widespread acceptance of XMODEM. With XMODEM so well entrenched, it is unlikely to be replaced by another protocol for several years, if at all.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 194.**

C CHEST



by Allen Holub

The main topic of this month's column is the Unix[1] "make" utility. We'll talk about how make works, review a few versions that run under MSDOS, and present the complete source for a rather stupid, but nonetheless functional, version of make for MSDOS.

However, before leaping into make, I'd first like to thank everyone who sent in sort routines. Several of the routines are quite spiffy; we'll definitely have another sorting column in the next few months.

Next, some good news for Lattice C users: they've finally cleaned up their manual. The old manual was so poorly organized as to make the compiler almost useless to someone who wasn't thoroughly familiar with Unix. Critical information was scattered over several appendixes and a myriad of technical bulletins. This wouldn't have been a problem if Lattice had published a new index with each addendum to the manual, but no such luck. As far as I can tell, the new manual uses the same text as the old one; however, it's now organized into a coherent whole with a real index (they even highlight function names in blue). If you own version 2.1x or later of the compiler, you can get a free update to the new version (2.15) and a copy of the new manual by sending Lattice your original release disks. Older versions of the compiler (including Microsoft and Lifeboat versions) can be updated for a fee ranging from $50 to $100, depending on the version. If you have an older version of the compiler and just want the new manual, send Lattice the table of contents and index from the old manual and they'll send you the new one (but no software update).

Finally, while rooting through a friend's back issues of the *PC Tech Journal*, I discovered MSDOS's undocumented SWITCHAR feature in an article by J. Eric Roskos (August 1984, page 73). It's a constant annoyance to me that command.com insists on using \ to separate directory names in a path and / as a command line switch designator (Unix programmers are used to / and -, respectively). This seemed all the more onerous when I discovered that MSDOS itself accepts either a \ or / in a path name. Only command.com and a few utility routines insist on specializing the two.

Well, it turns out that if you include the line

```
SWITCHAR = -
```

in your config.sys file, command.com will use a - instead of a / as its command line switch designator (you can replace the - with any printing character). Because / is no longer special, command.com passes it through to DOS where it's accepted as a legal path name designator. Moreover, the path printed by the "prompt $p" command will now use slashes instead of back-slashes when it displays the path!

I've no idea how this mechanism actually works, though I suspect it operates in ways similar to an environment variable. If anyone can enlighten me, please write. On page 113 of the same issue of the *PC Tech Journal*, Daniel Frank shows how to use DOS function 37 (also undocumented) to get the current switch character. Frank also presents a few SWITCHAR caveats. To summarize: RESTORE can get very confused if the SWITCHAR is set when you run BACKUP, so don't use the SWITCHAR if you're doing a backup. The same article gives a patch to RESTORE to fix this problem; the interested reader can find it there.

Unfortunately, in a fit of perversity, Microsoft elected not to support SWITCHAR in DOS version 3. The operating system spits out the message: "Unrecognized command in CONFIG.SYS." Argh. All is not lost, though. Unipress, the manufacturer of Ps-make, reviewed below, sells a program that changes the switch character, and this program *does* work with version 3.

Make

Make is one of those deceptive utilities: you can't figure out why anyone would bother to use it until you actually do use it, then you can't figure out how you got along without it. Make takes as input a makefile, which describes all the modules in a large program and the relationships between the modules. Using the makefile, make determines which modules must be recompiled at any given moment and then compiles them (and only them). It's like an intelligent batch utility that knows how C programs and C compilers work and does only those actions that are absolutely necessary.

The best way to explain make is with an example. You've an executable program called farm.exe. The original source is split up into three modules: cow.c, pig.c, and farm.c. All three modules have an #include <stdio.h> statement in them. In addition, cow.c and pig.c have an #include <animals.h> statement. Now, if you change something in cow.c, you need to recompile cow.c and then relink the new cow.obj into farm.exe. If you change something in animals.h, you need to recompile both cow.c and pig.c then relink. If you change stdio.h, you need to recompile everything. Make describes the relationships between these various files as "dependencies." The

makefile is a list of these dependencies. A makefile for the program just described is shown in Figure 1 (below).

The # designates a comment line. Farm.exe "depends on" cow.obj, pig .obj, and farm.obj. This means that if any of the files cow.obj, pig.obj, or farm.obj have been changed more recently than farm.exe, then farm.exe must be remade—by executing the line: link \lc\s\c cow pig farm,farm, con,\lc\s\lc. Similarly, if any of the files pig.c, stdio.h, or animals.h have been changed more recently than pig.obj, then pig.obj must be remade (using the command lc -ms -i\lc\ -i\lc\s\ pig). The actions associated with each set of dependencies can extend to several lines. To activate the process, you just type the command make. Make reads in the makefile and, using the information found there, figures out what to do and does it.

Make supports several options that make your life a little easier. You may use macros to save some typing. A macro is created by placing the following definition at the head of the makefile:

name = stuff

where name is the macro name and stuff is the text to be substituted. In the makefile the macro is expanded by $(name). Our original makefile, rewritten using macros, is shown in Figure 2 (page 97).

You can simplify the makefile even further. Notice that all .c files are converted to .obj files with the same set of actions. Make provides a mechanism for doing this sort of repetitive action, called a "generic dependency". Consider the makefile shown in Figure 3 (page 98).

The lines:

```
.c.obj:
        lc -ms -i\lc\ -i\lc\s\ $*
```

say "to turn a .c file into an .obj file, execute the line lc -ms -i\lc\ -i\lc\s\ $*" where $* is a predefined macro that evaluates to the root portion of the file being made (the one to the left of the : on the dependency line). For example, given the *dependency* line:

```
foo.obj: foo.c rat.h
```

$* will evaluate to the string foo. In the case of this generic dependency, make assumes that all .obj files depend on a .c file having the same root name. So, when foo.obj is being made, a dependency on foo.c is assumed, and it need not be listed on the dependency line.

There are several commercially available make utilities that run on MSDOS. (See the review of MSDOS C compilers on page 30 for compilers that include a make utility in the standard distribution package. You can't have a make for CP/M because you can't time and date stamp a file.) I looked at three of these: Polymake (by Polytron), Ps-make (by Unipress Software), and Pmaker (by Phoenix).[2] Of these, Polymake (at $99) is both the least expensive and the most complete implementation of make. Ps-make (at $179) is missing some of the features of Polymake. Pmaker, the Phoenix product (at $195), is both the most expensive and the most limited of the three versions. The Pmaker manual is extremely sparse.

```
#
# Make farm using the Lattice C compiler.
#

farm.exe:       cow.obj pig.obj farm.obj
                link \lc\s\c cow pig farm,farm,con,\lc\s\lc

cow.obj:        cow.c stdio.h animals.h
                lc -ms -i\lc\ -i\lc\s\ cow

pig.obj:        pig.c stdio.h animals.h
                lc -ms -i\lc\ -i\lc\s\ pig

farm.obj:       farm.c stdio.h
                lc -ms -i\lc\ -i\lc\s\ farm
```

Figure 1.
A Simple Makefile

```
#
# Make farm using the Lattice C compiler.
#

INCLUDES = stdio.h animals.h
OBJECTS = cow.obj pig.obj farm.obj
COMPILE = lc -ms -i\lc\ -i\lc\s\

farm.exe:       $(OBJECTS)
                link \lc\s\c $(OBJECTS),farm,con,\lc\s\lc

cow.obj:        $(INCLUDES) cow.c
                $(COMPILE) cow

pig.obj:        $(INCLUDES) pig.c
                $(COMPILE) pig

farm.obj:       stdio.h farm.c
                $(COMPILE) farm
```

Figure 2.
A Makefile with Macros

Pmaker supports virtually no command line options and only a stripped-down set of internal utilities (no predefined macros, no generic dependencies). It does come with a utility that creates makefiles (by going into your source code and looking for #include statements), but because my makefiles are often a little weird, I found this utility pretty useless. Frankly, I don't think that Pmaker is worth the money. In fact, the version of make presented at the end of this column is almost as powerful as the Phoenix product—and it's free. I'll therefore limit my remarks to Polymake and Ps-make.

Polymake

Polymake supports all the make features described above. It has 13 command line options:

-a
Rebuild all targets, even if time and date say that you don't have to.

-b <file>
Look for a file called builtins.mak—Polymake supports a default actions file in which you define things like .c.obj. or macros that are used a lot.

-c
Create a batch file rather than doing the action—this lets you use Polymake with DOS version 1.

-d
Echo an execution trace as actions are performed—debug mode.

-f <file>
Use <file> as the makefile instead of "makefile."

-i
Ignore errors returned from the programs.

-k
Keep working and ignore error codes returned from a process. (Like the Unix version, if any program invoked by Polymake returns a value other than 0, Polymake terminates. This can be annoying if you want to compile 85 modules and go to lunch, redirecting all the error messages into an error log file. The -i option suppresses this automatic termination. From reading the manual, I'm not clear about the differences between -i and -k. I've been using -k.)

-n
Print out the various actions but don't actually execute them—this is useful for debugging a makefile.

-p
Print out a representation of the makefile after it's been interpreted.

-q
Don't print out error messages from commands.

-r
Don't use any builtins.mak files (see option -b above).

-s

```
#
#  Make farm using the Lattice C compiler.
#

INCLUDES = stdio.h animals.h
OBJECTS = cow.obj pig.obj farm.obj

.c.obj:
                lc -ms -i\lc\ -i\lc\s\ $*

farm.exe:       $(OBJECTS)
                link \lc\s\c $(OBJECTS),farm,con,\lc\s\lc $*

cow.obj:        $(INCLUDES)
pig.obj:        $(INCLUDES)
farm.obj:       stdio.h
```

Figure 3.
A Generic Makefile

```
#
#  Make farm using the Lattice C compiler.
#

farm.exe:       cow.obj pig.obj farm.obj
                link \lc\s\c cow pig farm,farm,con,\lc\s\lc

cow.obj:        cow.c stdio.h animals.h
                lc -ms -i\lc\ -i\lc\s\ cow

pig.obj:        pig.c stdio.h animals.h
                lc -ms -i\lc\ -i\lc\s\ pig

farm.obj:       farm.c stdio.h
                lc -ms -i\lc\ -i\lc\s\ farm

#
#  Null targets not needed in a real makefile:
#

animals.h:

stdio.h:

pig.c:

cow.c:

farm.c:
```

Figure 4.
An Mkfile

Don't echo commands (silent).

-t

Touch. (Perhaps the most useful option supported by make, -t is for the situation where you change a comment in stdio.h and now make thinks that all 97 modules of your program monster.exe have to be recompiled; make -t modifies all the times and dates for files to be made so that they look as if they've been recompiled, but it doesn't actually recompile them. Polymake also comes with a touch utility (invoked from MSDOS with "touch <file list>") that sets the time and date of the indicated files to the current time and date.)

There's also a rich set of predefined macros:

$(name)	Expand macro "name"
$$	$ sign
$@	Name of current target
$*	Root of current target
$<	Source being used to make current target
$?	All sources that have been modified more recently than current target
$( )	Null string
$(mflags)	Command line flags (so that make can invoke itself recursively)
$(cwd)	Current working directory

Finally, Polymake supports several internal options, as well as generic dependencies. Although I won't describe all of them here, one is quite useful: the local input script. This feature is for programs, such as librarians and linkers, that sometimes need to use input files rather than the command line. For example, you can have so many files to link together that they won't all fit on a DOS command line. In this case, you put the command line into a file then call link with @file given on the command line. Local input scripts let Polymake create these input files for you. The example given in the manual illustrates the process pretty well:

```
text.exe : t1.obj t2.obj t3.obj
  link <@<
t1+
t2+
t3
test.exe
test.map
\lc\s\lc.lib
<
```

All text between the second and third < sign is put into a file called lis_qqq.tmp. Link is invoked with the line link @lis_qqq.tmp. The @ in this example can be replaced by any printing character. For example,

```
text.exe: t1.obj t2.obj t3.obj
  link <<<
text
<
```

will execute link <lis_qqq.tmp; lis_qqq.tmp will contain the single line "text." The temporary file is deleted when Polymake is done with it.

As you can probably tell, I like Polymake. I've been using it for a month now and haven't found any problems yet. My only complaint is that the manual doesn't have an index (which will make it hard to find anything if you're not familiar with the Unix make). On the other hand, Polymake sticks pretty close to the Unix model, so you may not need to consult the manual very often.

Ps-make

As I said earlier, Ps-make is more limited that Polymake, but it does do everything I consider to be essential: generic dependencies, $@, $*, $<, and macros. Several command line options are supported (there is some overlap with Polymake):

-f<file>

Use <file> as the makefile.

-o<file>

Use <file> as name of batch file (see -b).

-b

Create a batch file instead of executing commands.

-B

Beep before exiting.

-c

Don't convert to lower case when evaluating rules.

-d

Print file modifications times used in comparisons.

-e

Ignore various environment variables—the name of the default rules file (builtins.mak in Polymake) is kept in an environment variable.

-p

Print the generated dependency tree.

-s

Don't execute commands as they're executed.

-t

Touch.

-v

Force Ps-make to verify that a generated file exists before using it.

A MAKE Utility in C

One of the first utilities I wrote for MSDOS was the version of make given here. The program's called mk (i.e., a make with half of it missing). Mk is pretty stupid; it doesn't support macros or command line switches. On the other hand, it does do the critical job: recompile only those parts of a large modular program that need it. I was glad to have it before I got my copy of Polymake, and it is, after all, free. Mk was designed to help maintain large C programs, and it works quite well in this application, but it won't do everything that the real make can do.

Mk uses a makefile called "mkfile," which, like the real makefile, lists all the dependencies needed to make a program. If mk is invoked with a name as an argument (i.e., mk foo.exe), it will make the indicated file; that is, it looks in the mkfile for the indicated filename to the left of a colon on a dependency line and starts the make from that point. If no file is specified on the command line, the first file listed in the mkfile is made.

The mkfile itself has a more restricted syntax than a real makefile:

(1) A # sign in column 1 designates a comment line. The # must be in the leftmost column.

(2) The dependency line comes first. It uses the format "target: dependencies". The colon is required (an error message is printed if it's not there). The dependencies are delimited from one another with white space (any combination of tabs and blanks). The dependencies must all be listed on a single line; however, any line terminated with a back-slash (\) will be continued to the next line. For example:

```
This is\
one line\
of text.
```

(3) All files listed as dependencies must be used as targets somewhere in the mkfile (i.e., to the left of a colon). A null target (one consisting of a single filename followed by a colon but no dependencies, this in turn followed by a blank line) is permitted.
(4) All lines following the dependency line, up to a blank line, are the actions executed to make the target. The blank line is required to terminate the block of actions. Any number of blank lines are permitted as a terminator, but any nonblank line following a blank line is assumed to be a new dependency line.
(5) Targets having no dependencies will always be made. For example,

```
listing:
    print foo.c bar.c rat.c
```

will always execute the print command.
(6) For the sake of comparison, all nonexistent files are considered to exist and to be very old, so a nonexistent file listed to the left of a colon will always be made. The dates and times associated with a target are updated as soon as the target is made.

Figure 4 (page 98) is a mkfile version of the makefile in Figure 1. Here the blank lines are required, as are the various null targets at the end of the file (for animals.h, stdio.h, etc.).

The listing for mk starts on page 104; mydos.h (#included on line 3) was presented last month. There are several macros at the head of the listing. #including the #define on line 6 will cause mk to generate a trace as it parses the mkfile, but it won't actually try to do anything. MAXLINE (on line 8) limits the number of characters in a line (lines continued with a terminating \ are considered to be one line). MAXBLOCK and MAXDEP (lines 9 and 10) are the maximum number of actions that can follow a dependency line and the maximum number of dependencies, respectively. MAKEFILE (line 12) is the name used for the makefile. The three macros on lines 24–26 have horrible side effects, so be careful with them (don't ever say skipwhite(++s)).

I suspect that the real make sets up a dependency tree, a multi-way tree that has the major file to be made as its root and dependencies as children. Mk doesn't work that way. Mk reads in the entire mkfile before it tries to do anything. If it gets lost (or finds a syntax error), an error message is printed and mk terminates.

The makefile is organized as a binary tree, ordered alphabetically by target filename; the tree nodes are typedefed on lines 38–47. Lnode and rnode are pointers to children in the tree. The target filename (the one to the left of the equals sign) is used as the search key and stored in the being_made field.

The dependencies and the actions following the dependencies are put into two argv-like arrays of pointers to strings. The fields depends_on and do_this are used like argv to access these arrays. Both arrays are terminated with a null entry, so the equivalent of argc isn't required.

Finally, the time field holds the file create time and date returned by DOS. The time and date are concatenated (with the date in the most significant word). This enables the time field to be used as a single number in comparisons.

Most of the routines are commented well enough to be self-documenting, but there are some exceptions. Gtime( ) (lines 97–137) gets the time and date for a file from DOS, concatenates the time and date together, and returns the concatenated result. The functions gregs( ) and dos( ), used to access DOS, were presented in this column last month.

The assignment on lines 129–30 merits some attention. Regs.x.dx and regs.x.bx are both shorts. A short, shifted left 16 bits, evaluates to 0, so dx has to be cast into a long before the shift. Similarly, if regs.x.cx isn't cast into a long, the | will behave unpredictably, depending on whether sign extension is active. However, now we have to AND the result of the type conversion with 0xffff to defeat the sign extension (or else we may end up setting the entire top 16 bits to 1).

The function getline( ) (lines 200–251) is useful enough to merit some notice, too. It works something like gets( ) except that it recognizes line continuation (lines ending in \) and it allocates buffer space using malloc( ). I've used it in several other programs.

The final routine to look at here is make( ) (lines 302–381); all the real work performed by mk is done here. Make( ) is recursive. To make a single object, you have to make the dependent files. If any of the dependent files were remade, you then have to perform the action on the current file. When make( ) returns, the dates on the remade files will have been modified to the current date. Consequently, the time and date comparisons for the target and the dependent file (on line 339) must follow the recursive make( ) call (on line 327). If any of the dependencies were younger than the target, doaction is incremented (line 350). We don't actually do the action on line 350 because we don't want to do it several times.

Finally, mk uses a system( ) call (line 368) to execute each action line. In Unix, system( ) creates a parallel process running the shell then passes its argument through to the spawned shell. Because Unix supports true multitasking, the only real penalty involved in a Unix system( ) call is the time it takes to activate the shell.

MSDOS is another matter. System( ) actually reads a *second copy* of command.com into memory then executes the command line with this second copy. Because command.com uses about 21K and because the program being invoked takes up some space, as does mk itself (about 15K in

my version), memory can disappear pretty fast. I've a bunch of memory, so this limitation hasn't been a problem, but it may be for you.

There are two solutions. The better of these works only if you aren't going to use any functions internal to command.com (more importantly, you can't run a batch file from inside mk). The Lattice compiler supplies several system( )-like procedures. In particular, the various fork( ) functions (forkv, forkl, forkvp, forklp) will execute another program directly, without the additional invocation of command.com. These functions are something of a misnomer as they don't behave like the Unix fork( ) function; however, at the cost of a little overhead (you have to extract the program name from the action), you can use a fork( ) function instead of a system( ) call to do the action. If your compiler doesn't have a fork( ) or a system( ) call, you can create one using the DOS version 2 EXEC function (0x4b). I can print a system( ) function in this column at a later date if enough people need one, but most compilers seem to come with this function.

The second solution to the space problem is more involved. Instead of doing the action from inside mk, the program can create a .bat file and exit—you can then execute the batch file from outside mk. If you're running DOS version 1, you have to use this procedure because the EXEC function isn't supported. The problem here is that the time and date associated with a target are dynamically updated by mk when the target is remade (on line 374 of the listing). This new time is used by previous invocations of the subroutine make( ) to see if they need to do an action. Because gtime( ), the routine that does the updating, looks at the real file, it can't be used for updating if this file hasn't actually been modified. So, in addition to replacing the system( ) call on line 368 with a fprintf( ) call, you need to replace the gtime( ) call on line 374 with either a representation of the current time and date or with some constant that looks like a very young time and date (i.e., 0x7fffffffL).

One final caveat: system( ) prepends the argument /c to the command when it executes EXEC. (The process is described in section 7 of the version 3 *DOS Technical Reference* and in the version 2 manual somewhere.) Anyway, if you've used the SWITCHAR feature described earlier, DOS will be confused by the /, so set the switch character back to / before you use mk.

Erratum: I'm embarrassed to admit that my complaints last month about the erase-screen and erase-inline functions of ANSI.SYS were unfounded. I was using puts( ) to do my output and the extra newline character came from puts( ), not from ANSI.SYS.

Notes

[1]Unix is a registered trademark of Bell Labs.

[2]Polymake, Ps-make, and Pmaker are registered trademarks of Polytron, Phoenix, and Unipress Software, Inc., respectively.

C Chest Listing (Text begins on page 96)

```
 1: #include <stdio.h>
 2:
 3: #include <mydos.h>

 4: /*---------------------------------------------------------------------------*/

 5: #ifdef  NEVER
 6: #define DEBUG 1          /* Include for debug diags in make()     */
 7: #endif

 8: #define MAXLINE  (80*10)  /* Maximum input line length            */
 9: #define MAXBLOCK 64       /* Max number of lines in an action     */
10: #define MAXDEP   32       /* Max number of dependancies           */
11: #define COMMENT  '#'      /* Delimits a comment                   */
12: #define MAKEFILE "mkfile" /* Name of makefile                     */
13: #define OPEN     0x3d     /* Dos function call to open a file     */
14: #define CLOSE    0x3e     /* Dos function call to close a file    */
15: #define DATETIME 0x57     /* " to get or set file's date & time */

16: #define DEFTIME  0x0      /* The default time returned by gtime when a file
17:                           * doesn't exist.
18:                           */

19: /*---------------------------------------------------------------------
20:  *      iswhite(c)      evaluates true if c is white space.
21:  *      skipwhite(s)    skips the character pointer s past any white space
22:  *      skipnonwhite(s) skips s past any non-white characters.
23:  */

24: #define iswhite(c)      ((c) == ' ' || (c) == '\t')
25: #define skipwhite(s)    while( iswhite(*s) )        ++s;
26: #define skipnonwhite(s) while( *s && !iswhite(*s) ) ++s;
```

```
27: /*---------------------------------------------------------------------------
28:  * The entire makefile is read into memory before it's processed. It's
29:  * stored in a binary tree composed of the following structures:
30:  * depends_on and do_this are argv-like arrays of pointers to character
31:  * pointers. The arrays are null terminated so no count is required.
32:  * The time field is a 32 bit long consisting of the date and time
33:  * fields retuned from a DOS 0x57 call. The date and time are
34:  * concatanated with the date in the most significant 16 bits and the
35:  * time in the least significant. This way they can be compared as
36:  * a single number.
37:  */

38: typedef struct _tn
39: {
40:         struct _tn      *lnode;       /* pointer to left sub-tree         */
41:         struct _tn      *rnode;       /* pointer to right sub-tree        */
42:         char            *being_made; /* name of file being made          */
43:         char            **depends_on;/* names of dependant files         */
44:         char            **do_this;    /* Actions to be done to make file */
45:         long            time;         /* time & date last modified        */
46: }
47: TNODE ;

48: /*---------------------------------------------------------------------------*/

49: static TNODE    *Root     = 0  ;        /* Root of file-name tree          */
50: static FILE     *Makefile      ;        /* Pointer to opened makefile      */
51: static int      Inputline = 1  ;        /* current input line number       */
52: static char     *First    = "" ;        /* Default file to make            */

53: extern char     *malloc();              /* From standard library           */
54: extern char     **getblock();           /* Declared in this module         */
55: extern char     *getline();             /* ditto                           */
56: extern TNODE    *find();                /* ditto                           */

57: /*---------------------------------------------------------------------------*/

58: char    *gmem( numbytes )
59: {
60:         /*      Get numbytes from malloc. Print an error message and
61:          *      abort if malloc fails, otherwise return a pointer to
62:          *      the memory.
63:          */

64:         extern  char    *calloc();
65:         char    *p;

66:         if( !( p = calloc(1,numbytes) ))
67:                 err("Out of memory");

68:         return p;
69: }

70: /*---------------------------------------------------------------------------*/

71: char    **stov(str, maxvect )
72: char    *str;
73: {
74:         /*      "Str" is a string of words seperated from each other by
75:          *      white space. Stov returns an argv-like array of pointers
76:          *      to character pointers, one to each word in the original
77:          *      string. The white-space in the original string is replaced
78:          *      with nulls. The array of pointers is null-terminated.
79:          *      "Maxvect" is the number of vectors in the returned
80:          *      array. The program is aborted if it can't get memory.
81:          */
82:
83:         char    **vect, **vp;

84:         vp = vect = (char **)  gmem( (maxvect + 1) * sizeof(str) );
```

C Chest Listing (Listing continued, text begins on page 96)

```
 85:         while( *str && --maxvect >= 0 )
 86:         {
 87:                 skipwhite(str);
 88:                 *vp++ = str ;
 89:                 skipnonwhite(str);
 90:                 if( *str )
 91:                         *str++ = 0;
 92:         }

 93:         *vp = 0;
 94:         return( vect );
 95: }


 96: /*---------------------------------------------------------------------------*/

 97: long    gtime( file )
 98: char    *file  ;
 99: {
100:         /*      Return the time and date for file.
101:          *
102:          *      The DOS time and date are concatanated to form one
103:          *      large number. Note that the high bit of this number
104:          *      will be set to 1 for all dates after 2043, which will
105:          *      cause the date comparisions done in make() to not work.
106:          *      THIS ROUTINE IS NOT PORTABLE (because it assumes a 32
107:          *      bit long).
108:          */

109:         static REGS     regs;
110:         short           handle = 0;     /* Place to remember file handle */
111:         long            time;

112:         gregs( &regs );
113:         regs.h.ah = OPEN ;              /* Open the file                 */
114:         regs.h.al = 0;                  /*      for read                 */
115:         regs.x.dx = (short) file ;

116:         if(  dos(&regs) & CARRY )
117:         {
118:                 /* File doesn't exist. Return a huge date & time        */

119:                 return DEFTIME;
120:         }
121:         else
122:         {
123:                 handle    = regs.x.ax;
124:                 regs.x.bx = handle;     /* Put file handle in BX        */
125:                 regs.h.ah = DATETIME;   /* Get or set date and time     */
126:                 regs.h.al = 0;          /* Get the date and time        */

127:                 if( dos( &regs ) & CARRY )

128:                         err("DOS returned error from date/time request");

129:                 time = ((long)( regs.x.dx ) << 16)  |
130:                                         ((long)( regs.x.cx ) & 0xffffL) ;

131:                 regs.h.ah = CLOSE  ;    /* Close the file handle        */
132:                 regs.x.bx = handle ;

133:                 if( dos( &regs ) & CARRY )
134:                         err("DOS returned error from file close request");

135:                 return time;
136:         }
137: }

138: /*---------------------------------------------------------------------------*/
```

```
139: TNODE   *makenode()
140: {
141:         /* Create a TNODE, filling it from the makefile
142:          * and return a pointer to it. Return NULL if there are no more
143:          * objects in the makefile.
144:          */

145:         char     *line, *lp;
146:         TNODE    *nodep;
147:         unsigned date, time;
148:         /*      First, skip past any blank lines or comment lines.
149:          *      Return NULL if we reach end of file.
150:          */

151:         do{
152:                 if( (line = getline(MAXLINE,Makefile)) == NULL )
153:                         return( NULL );

154:         } while ( *line == 0 || *line == COMMENT );


155:         /*      A this point we've gotten what should be the dependancy
156:          *      line. Position lp to point at the colon.
157:          */

158:         for( lp = line; *lp && *lp != ':' ; lp++ )
159:                 ;


160:         /*      If we find the colon postion, lp to point at the first
161:          *      non-white character following the colon.
162:          */

163:         if( *lp != ':' )

164:                 err( "missing ':'" );   /* This will abort the program */
165:         else
166:                 for( *lp++ = 0; iswhite(*lp) ; lp++ )
167:                         ;

168:         /*
169:          *      Allocate and initialize the TNODE;
170:          */

171:         nodep              = (TNODE *) gmem( sizeof(TNODE) );
172:         nodep->being_made  = line ;
173:         nodep->time        = gtime( line );
174:         nodep->depends_on  = stov( lp, MAXDEP );
175:         nodep->do_this     = getblock( Makefile );

176:         return( nodep );
177: }

178: /*-------------------------------------------------------------------------*/

179: dependancies()
180: {
181:         /* Manufacture the binary tree of objects to make. First
182:          * is a pointer to the first target file listed in the
183:          * makefile (ie. the one to make if one isn't explicitly
184:          * given on the command line. Root is the tree's root pointer.
185:          */

186:         TNODE   *node;

187:         if( node = makenode() )
188:         {
189:                 First = node->being_made ;

190:                 if( !tree(node, &Root) )
191:                         err("Can't insert first node into tree !!!\n");
```

```
192:                    while( node = makenode() )
193:                            if( !tree( node, &Root ) )
194:                                    free( node );
195:                    return 1;
196:            }

197:            return 0;
198: }


199: /*---------------------------------------------------------------------------*/

200: char    *getline( maxline, fp )
201: FILE    *fp;
202: {

203:            /*      Get a line from the stream pointed to by fp.
204:             *      "Maxline" is the maximum input line size (including the
205:             *      terminating null. A \ at the end of line is
206:             *      recognized as a line continuation, (the lines
207:             *      are concatanated). Buffer space is gotten from malloc.
208:             *      If a line is longer than maxline it is truncated (ie.
209:             *      all characters from the maxlineth until a \n or EOF is
210:             *      encountered are discarded.
211:             *
212:             *      Returns: NULL on a malloc failure or end of file.
213:             *               A pointer to the malloced buffer on success.
214:             */

215:            static   char   *buf ;
216:            register char   *bp  ;
217:            register int    c, lastc;

218:            /* Two buffers are used. Here, we are getting a worst-case buffer
219:             * that will hold the longest possible line. Later on we'll copy
220:             * the string into a buffer that's the correct size.
221:             */

222:            if( !(bp = buf = malloc(maxline)) )
223:                    return NULL;

224:            while(1)
225:            {
226:                    /* Get the line from fp. Terminate after maxline
227:                     * characters and ignore \n following a \.
228:                     */

229:                    Inputline++;            /* Update input line number     */

230:                    for( lastc=0; (c = fgetc(fp)) != EOF && c!='\n'; lastc = c)
231:                            if( --maxline > 0 )
232:                                    *bp++ = c;

233:                    if( !( c == '\n' && lastc == '\\' ) )
234:                            break;

235:                    else if( maxline > 0 )          /* erase the \ */
236:                                    --bp;
237:            }
238:            *bp = 0;


239:            if( (c == EOF && bp == buf)  ||  !(bp = malloc((bp-buf)+1))  )
240:            {
241:                    /*      If EOF was the first character on the line or
242:                     *      malloc fails when we try to get a buffer, quit.
243:                     */
```

```
244:                    free(buf);
245:                    return( NULL );
246:            }

247:            strcpy ( bp, buf );     /* Copy the worst-case buffer to the one  */
248:                                    /* that is the correct size and ...       */
249:            free   ( buf );         /* free the original, worst-case buffer,  */
250:            return ( bp  );         /* returning a pointer to the copy.       */
251: }

252: /*-------------------------------------------------------------------------*/

253: char    **getblock( fp )
254: FILE    *fp;
255: {
256:            /* Get a block from standard input. A block is a sequences of
257:             * lines terminated by a blank line. The block is returned as
258:             * an array of pointers to strings. At most MAXBLOCK lines can
259:             * be in a block. Leading white space is stripped.
260:             */

261:            char    *p, *lines[MAXBLOCK], **blockv = lines ;
262:            int     blockc = 0;

263:            do {
264:                    if( !(  p = getline(MAXLINE,Makefile)  ))
265:                            break;

266:                    skipwhite(p);

267:                    if( ++blockc <= MAXBLOCK )
268:                            *blockv++ = p;
269:                    else
270:                            err("action too long (max = %d lines)", MAXBLOCK);

271:            } while( *p );


272:            /*      Copy the blockv array into a safe place. Since the array
273:             *      returned by getblock is NULL terminated, we need to
274:             *      increment blockc first.
275:             */

276:            blockv = (char **) gmem( (blockc + 1) * sizeof(blockv[0]) );
277:            movmem( lines, blockv, blockc * sizeof(blockv[0]) );
278:            blockv[blockc] = NULL ;

279:            return blockv;
280: }

281: /*-------------------------------------------------------------------------*/


282: err( msg, param )
283: char    *msg;
284: {
285:            /*      Print the error message and exit the program.
286:             */

287:            fprintf(stderr,"Mk (%s line %d): ", MAKEFILE, Inputline );
288:            fprintf(stderr, msg, param );
289:            exit(1);
290: }

291: serr( msg, param )
292: char    *msg, *param;
293: {
294:            /*      same as err() except the parameter is a string pointer
295:             *      instead of an int.
296:             */
```

C Chest Listing (Listing continued, text begins on page 96)

```
297:          fprintf(stderr,"Mk (%s line %d): ", MAKEFILE, Inputline );
298:          fprintf(stderr, msg, param );
299:          exit(1);
300: }

301: /*---------------------------------------------------------------------------*/

302: make(what)
303: char *what;

304: {
305:          /*      Actually do the make. The dependancy tree is descended
306:           *      recursively and if required, the dependancies are
307:           *      adjusted. Return 1 if anything was done, 0 otherwise
308:           */

309:          TNODE       *snode;             /* Source file node pointer     */
310:          TNODE       *dnode;             /* Dependant file node pointer  */
311:          int         doaction = 0 ;      /* If true do the action        */
312:          static char *zero    = (char *)0;
313:          char        **linev  = &zero ;


314: #ifdef DEBUG
315:          static  int     recurlev = 0;   /* Recursion level              */

316:          printf("make (lev %d): making <%s>\n", recurlev, what );
317: #endif

318:          if( !(snode = find(what, Root)) )
319:                  serr("Don't know how to make <%s>\n", what );


320:          if( !*(linev = snode->depends_on))  /* If no dependancys        */
321:                  doaction++;                 /* always do the action.    */

322:          for( ; *linev ; linev++ )           /* Process each dependancy  */
323:          {
324: #ifdef DEBUG
325:                  recurlev++;
326: #endif
327:                  make( *linev );
328: #ifdef DEBUG
329:                  recurlev--;
330: #endif

331:                  if( !(dnode = find(*linev, Root)) )
332:                          serr("Don't know how to make <%s>\n", *linev );

333: #ifdef DEBUG
334:                  printf("make (lev %d): source    file ", recurlev);
335:                  ptime( what, snode->time );
336:                  printf("make (lev %d): dependant file ", recurlev);
337:                  ptime( *linev, dnode->time );
338: #endif
339:                  if( snode->time <= dnode->time )
340:                  {
341:                          /* If source node is older than (time is less than)
342:                           * dependant node, do something. If the times are
343:                           * equal, assume that neither file exists but that
344:                           * the action will create them, and do the action.
345:                           */
346: #ifdef DEBUG
347:                          printf("make (lev %d): %s older than %s\n",
348:                          recurlev, what, *linev );
349: #endif
350:                          doaction++;
351:                  }
```

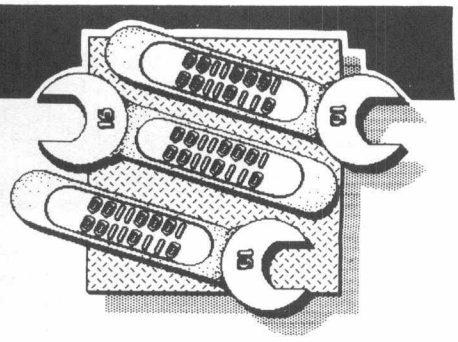
```
#ifdef DEBUG
                else
                    printf("make (lev %d): %s younger than %s\n",
                                                    recurlev, what, *linev );
#endif
        }
        if( doaction )
        {
#ifdef DEBUG
                printf("make (lev %d): doing action:\n",
                                                    recurlev, *linev, what);
#endif
                for( linev = snode->do_this; *linev; linev++ )
                {
                        printf("%s\n", *linev); /* Echo action to screen */
#ifndef DEBUG
                        if( system(*linev) )
                                serr("Can't process <%s>\n", *linev );
#endif

                        /*      Change the source file's time to reflect
                         *      any modification.
                         */

                        snode->time = gtime( snode->being_made );
                }
        }

#ifdef DEBUG
        printf("make (lev %d): exiting\n", recurlev );
#endif
        return doaction;
}


/*---------------------------------------------------------------------------
 *      Tree routines:
 */

TNODE   *find( key, root )
char    *key;
TNODE   *root;
{
        /* If key is in the tree pointed to by root, return a pointer
         * to it, else return 0.
         */
        register int    notequal ;
        register TNODE  *rval    ;

        if( !root )
                return 0;

        if( !(notequal = strcmp(root->being_made,key)) )
                return( root );

        return(  find( key, (notequal > 0) ? root->lnode : root->rnode) );
}

/*---------------------------------------------------------------------------*/

tree( node, rootp )
TNODE   *node, **rootp ;
{
        /* If node's key is in the tree pointed to by rootp, return 0
         * else put it into the tree and return 1.
         */

        register int    notequal ;
        register TNODE  *rval    ;
```

```
409:            if( *rootp == NULL )

410:            {
411:                    *rootp = node;
412:                    return 1;
413:            }

414:            if( !(notequal = strcmp( (*rootp)->being_made, node->being_made)))
415:                    return 0;

416:            return( tree( node, notequal > 0 ? &(*rootp)->lnode
417:                                             : &(*rootp)->rnode)    );
418: }

419: /*---------------------------------------------------------------------------*/

420: main( argc, argv )
421: char    **argv;
422: {
423:            /*      A stupid version of the unix make utility.
424:             *
425:             *
426:             */

427:            if( !(Makefile = fopen(MAKEFILE, "r")) )
428:                    err("can't open %s\n", MAKEFILE );

429:            if( !dependancies() )
430:                    err("Nothing to make");
431:            else
432:                    make( argc > 1 ? argv[1] : First );
433: }
434: #ifdef DEBUG
435: /*---------------------------------------------------------------------------
436:  *      Msc. Debugging routines
437:  */

438: ptime( file, t )
439: char    *file;
440: long    t;
441: {
442:            /* Print out the time and date field of a TNODE as
443:             *                  "mm-dd-yy hh:mm:ss"
444:             * File is the file name.
445:             */

446:            int date, time;

447:            date = (t >> 16) & 0xffffL ;
448:            time = t & 0xffffL ;

449:            printf("%s: ", file );
450:            printf("%02d-%02d-%02d, ", (date >> 5 ) & 0x0f, date & 0x1f,
451:                                                    80 + ((date >> 9) & 0x7f) );

452:            printf("%02d:%02d:%02d, ", (time >> 11) & 0x1f, (time >> 5) & 0x3f,
453:                                                    (time << 1) & 0x3e  );
454:            printf("\n");
455: }

456: /*---------------------------------------------------------------------------*/

457: pnode( node )
458: TNODE *node;
459: {
460:            /*      Print out the tree node pointed to by "node"
461:             */
```

```
462:          char **linev;
463:          printf("+-----------------------------\n");
464:          printf("|  node at 0x%x\n",                              node );
465:          printf("+-----------------------------\n");
466:          printf("| lnode  = 0x%x,  rnode = 0x%x\n",node->lnode,node->rnode);
467:          printf("| time   = 0x%lx =",              , node->time           );
468:          ptime( "", node->time );
469:          printf("| target = <%s>\n"                , node->being_made     );
470:          printf("| dependancys:\n"                                        );
471:          for( linev = node->depends_on; *linev; printf("|\t%s\n", *linev++))
472:                  ;

473:          printf("| actions:\n" );
474:          for( linev = node->do_this; *linev; printf("|\t%s\n", *linev++))
475:                  ;
476:          printf("+-----------------------------\n");

477: }

478: /*-------------------------------------------------------------------------*/

479: trav( root )
480: TNODE   *root;
481: {
482:          /*      Do an in-order traversal of the tree, printing the
483:           *      node's contents as you go.
484:           */

485:          if( root == NULL )
486:                  return;

487:          trav( root->lnode );
488:          pnode( root );
489:          trav( root->rnode );
490: }
491: #endif
```

End Listing

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

Bug Report: TEE Filter

Dr. Fred Sinal of Hampton, Virginia, writes: "There is a slight bug in your listing of TEE in the April 1985 '16-Bit Software Toolbox' column. Actually the bug is in DOS, but that's life. If the file does not end with a CR/LF, we get a 'disk is full' message. DOS is forcing CR/LF to the console and confusing the character count. This problem may be covered up by eliminating the pair of lines

```
cmp ax,nchar
jne tee7
```

before the label **tee4**. Of course this reopens the disk full problem, when redirecting the standard output. I could not find a truly satisfactory fix."

Some Musings about DRI

Most of you have probably seen the recent splashy full-page ads for Digital Research's GEM that are appearing in nearly every magazine except *Sports Illustrated*. These ads lead you to believe that you can get a 16-color equivalent of Macintosh MacPaint for the IBM PC by just picking up GEM and GEM-Paint for a few pence at your corner computer store.

Beware: The fine print in the ad says "appropriate graphics hardware required." The appropriate hardware turns out to be the IBM Enhanced Graphics Adaptor, which will set you back about $1200 for the required controller board, memory expansion, and higher resolution RGB monitor. If you use GEM on an "ordinary" IBM graphics adaptor, the icons, windows, and all the rest are in vibrant black and white.

Speaking of Digital Research, look before you leap into spending any money on the product that DRI is advertising (with tongue in cheek, I'm sure) as "Concurrent PCDOS." It's concurrent, I guess, but it certainly doesn't resemble current versions of PCDOS. "Concurrent PCDOS" does not support the hierarchical file structure or any of the Unix-like executive services that are present in MSDOS or PCDOS versions 2 and 3. In other words, "Concurrent PCDOS" supports only the functionality of MSDOS or PCDOS version 1, which was more or less a clone of CP/M.

Thus, almost any of the more powerful MSDOS applications you are likely to buy will not run properly under "Concurrent PCDOS." Furthermore, a recent review in *PC Tech Journal* demonstrated that a set of programs run concurrently under "Concurrent PCDOS" will actually take longer to complete than if you simply run them consecutively under normal MSDOS. You can draw your own conclusions, but Digital Research's hopes of recapturing its old dominance of the personal computer operating system market seem more like delusions to me.

More on Square Roots

Tom Prince, of Marblehead, Massachusetts, writes:

"I was interested in your discussion of square rooting in the May 1985 column. A similar algorithm in 8088 code appeared in *Dr. Dobb's* over a year ago, encouraging me to add a floating point square root to my collection of Z80 code. Since the Z80 has enough registers to perform 32-bit calculations, it can do the square root much faster than the 8080 or even 8088 can do a floating point divide. Imagine my surprise to see an integer square root that takes ten times as long as a divide, even when coded in assembler! Even a conversion to floating point to take the root ought to be faster.

"So I wrote up the following code for a well-known '32-bit' processor. It turns out to be slower than sqrt(float( )) but certainly doesn't take ten times as long as an integer divide. Of course it takes longer than the three integer divides that are included and would run faster if coded in assembler. Would it be too much to ask to provide high-level language versions of routines like these that work perfectly well, if a bit slower?"

Mr. Prince's code can be found in Listing One (page 122).

Mac Fan Strikes Back

Mr. S. Allen, of Ojai, California, writes:

"I am, and have been for quite a while, an ardent admirer of *DDJ*. I have been hard-pressed to find a magazine of such technical excellence aimed at the more advanced computer user.

"I do, however, have a GRIPE! I was quite excited when you began the 16-Bit Toolbox. I expected to find droves of information on the latest hardware innovations and the support software to go with them. You can imagine my chagrin when I discovered a thinly disguised 'The IBM PC is Wonderful' column. To call the IBM PC, with its 8088, a 16-bit machine is pushing things a little, and then to have the writer launch irrational tirades against a true 16-bit machine (the Mac) lit my Bulls__t High Level Light.

"Please, be honest. If you are going to have an IBM PC column, call it that. If you are going to have a 16-bit column, let's have it be about 16-bit machines. From reading the present 16-Bit Toolbox, I must assume that the writer is either in IBM's or Intel's pocket. Your magazine has a long-standing reputation for impartiality. I implore you, do not destroy it."

Well, I suspect that the flip-flop

controlling Mr. Allen's Bulls—t High Level Light is in need of replacement. Let me first assure *DDJ* loyal readers that I am not in either IBM or Intel's pocket. If I were, I'd ask them for a suitably monstrous amount of money, retire, and pass this column on to other younger and more energetic souls.

People who actually read this column know that it is heavily devoted to reports of bugs in Microsoft, IBM, or Intel products, so that programmers can work around them. These bug reports hardly constitute advertisements for the companies involved (except, I guess, for those who have the philosophy that any publicity is good publicity). I defy anyone to look back at the previous columns and anywhere find the message "The IBM PC is Wonderful."

As for our emphasis on the Intel 8086 family of microprocessors and the MSDOS operating system, these remain the dominant force in the personal computer world and seem likely to remain so for the forseeable future; consequently, they are of primary interest to those of us who write programs for a living. Whether the 8088 is really a 16-bit processor or not is a moot point; it runs the same machine code as the 8086, it acts like an 8086, and it is everywhere. Quibbling over the bus width is a waste of our time.

We certainly are interested in extending our coverage of other 16-bit processors, and as people who actually read this column will also know, we have given considerable coverage to the 68000 and Macintosh over the last few months and will continue to do so as long as our readers indicate an interest in it.

Regarding 68000 Assemblers

Mr. Michael Aichlmayr, of Tacoma, Washington, writes:

"In regards to Mr. Howell's remarks in your May 1985 column, I would like to add myself to the list of people who feel it is truly possible to build an efficient assembler for the 68000—one that would run in a reasonable amount of memory and execute in a productive amount of time.

"I don't feel that the 68(x)xx family is nearly as complicated as Mr. Howell implies. I think elegant simplicity is a more accurate statement. Any good programmer could make great use of generic addressing modes and instructions. I think I would hazard to go so far as to say that I believe it could be done in C and run in 64K on a 68(x)xx processor (the horrible overhead of the [Intel] 80(x)xx would make it a real challenge). I have a 6809 assembler written in C that runs easily in 64K of memory on every machine for which it is compiled.

"It's very sad to realize that the abundance and low cost of memory have given programmers an excuse to be gross and inefficient. I have seen code that could run in my HP 29C calculator take up more than 64K of memory and 'require' a 128K IBM PC or compatible to run. It makes me think that if a machine came standard with a megabyte of RAM, someone would come up with a desk calculator that would hog the machine.

"Mr. Howell remarks that he has a 6809 system that he likes better than his IBM compatible. I direct his attention to the TSC (Technical Systems Consultants) 68000 macro assembler. I have been able to coax the assembler to run in 32K on a 6809 system running under the FLEX operating system. I realize it does not produce relocatable object modules; however, that piece of code would be fairly small given the architecture.

"Having a 6809 machine (which I built), a Mac (which I tolerate), and a DG One IBM compatible (which I use), I think I can safely assume that I have some knowledge of hardware and software. I find it very aggravating that IBM chose the 80(x)xx architecture over that of the 68(x)xx, but I have never seen a processor more crippled than that in the Mac.

"I eagerly await the arrival of the Atari ST, which I have had the opportunity to play with for a while. If all appearances are correct, this is a good use of a 68000. The machine runs at 8 MHz, has 32K [of RAM devoted to] memory-mapped color graphics, and is said to come standard with 512K of RAM for $699.00! I'm not sure what the GEM OS is going to do to the thing since I have little regard for DRI, but it doesn't look bad so far."

More Mac Feedback

Kirk Kerekes, of Tandata Inc., in Tulsa, Oklahoma, submitted a thoughtful and interesting letter:

"As circumstances seem to have started a debate in your column *re* the Mac as a development environment, I felt it appropriate to provide some facts to fuel the debate and perhaps redirect it to more substantive issues.

"FACT: There is no assembler that I know of for the Mac that requires the use of two (or more!) Macs. This myth has been floating around for quite a while and was recently mentioned in your column. This myth doubtless arises from the fact that there are two Apple-issue debuggers that use a second computer as a control device to allow undisturbed screen displays on the system being debugged. As there are around half a dozen debuggers that do *not* require a second unit, and at least one of the two-machine debuggers will use *any* reasonable terminal as a remote controller, I see no reason to regard this situation as anything but a blessing for the developer. The debuggers are available as part of the Inside Mac supplement program and are distributed as a group.

"If you are doing development on the Mac and are a fan of debuggers, then the $100 price tag of the supplement could be justified for them alone. (I've gotten at least $100 worth of micro-floppies through my supplement subscription, just valued as media!)

"FACT: There are numerous complete, stable high-level languages for the Mac. A recent glance reveals at least three Forths, half a dozen Cs, a Pascal or two, Microsoft BASIC, and NEON, an object-oriented Forth-like language. I can state that at least one product of a professional nature, MegaMax C, is 'real,' well-supported, and quite clean. Assembler fans can obtain the MacAsm mentioned in your column or ask almost anybody for a copy of the prerelease versions of the Apple Editor/Assembler/Linker/Resource Compiler package. By the

time this might see print, the official release of the MDS+Inside Mac package should have occurred. This package is already on the certified developer's price list. The price is lower than any complete serious development environment available for MSDOS.

"FACT: The bloated size of most of the early Mac applications is the result of the use of Lisa Pascal as the development environment. There is absolutely nothing about the Mac that requires/generates bulky code. I have created useful Mac-interface programs with a load module size less than 1K using the MegaMax C compiler. An 'empty' C program (i.e., main( ) { }) yields a load module of about 300 bytes. Most of this appears to be information that any application stores for use by the Finder. I just tried the same experiment with Computer Innovations C86 under MSDOS and got a 5K EXE file result, due to C86's insistence on loading the standard I/O library whether or not it is invoked.

"POINT: Nobody at Apple denies that the Mac concept, from mouse on up, is derived from the Xerox PARC research. The point in the Mac is to take the PARC concepts and implement them in an affordable machine. Go price a Xerox Star and then price a Mac (first, *find* a Star!)." [As this column went to press, Xerox announced a new incarnation of the Star, the model 6085, that costs $4995 with 1 Mb of RAM, 10 Mb hard disk, a 15-inch display, and a mouse . . . . RD]

"OPINION: I develop professionally on MSDOS, CP/M, and Mac systems. I use the major development products in each environment for both C and assembly language. It is my opinion that there are better development environments available for the Mac than for MSDOS or CP/M. The Mac development products tend to be less expensive, too. There are also some real losers available for all these environments. I'm not going to name names; we all have our own favorite turkeys.

"OPINION: There are a number of programming professionals who feel threatened by the Mac. It is not my purpose to speculate why this should be the case. It *is* my purpose to assert that no professional need feel threatened by the Mac. With the tools currently available, it is a joy to develop on, and there are enough 'goodies' in the system toolbox to keep just about anybody happy. There is definitely a learning curve involved, enough of one to be a barrier to dabblers, but anyone familiar with good modern programming practices can produce useful programs in very short order. The Mac environment punishes sloppy programming practices and greatly rewards programmers who plan thoroughly before they code.

"Maligning the Mac is not going to make it go away. Being annoyed by Steve Jobs or marketing hype is not realistic. The only tool Apple has to battle IBM in the public mind is flamboyance. Steve Jobs' opinions may not be yours, but you know his name, *Playboy* readers know his name, and an amazingly large percentage of the general public knows his name. Now, what is the name of the CEO of IBM? Of *any* officer of IBM? See how it works?

"I propose that further discussion of development on the Mac actually center on development on the Mac . . . not on myths and spurious side issues! There is plenty to discuss, believe me!"

In reply, I only question whether the only tool Apple has to battle IBM is flamboyance. The tried and true tools of delivering quality products on time seem to have served some companies pretty well (such as Compaq, which has taken on IBM head-on and is doing better than Apple at it in many respects). The rest of Mr. Kerekes' points are well taken, and indeed we will try to stick to substantive discussion of the Mac in the future. By the way, the CEO of IBM is John Akers (last I heard); although this fact probably *isn't* a part of the general knowledge of the average *Playboy* reader, it is fairly common knowledge in the computing community.

Reading about the Mac

I have run across two recent articles about the Macintosh that are both helpful and balanced (unlike the rabid pro or con articles that are prevalent in the popular computer press).

The first is simply a detailed, objective, nonhysterical review of the Macintosh written by Larry Press of Small Systems Group; this review appeared in *Abacus*, Vol. 2, No. 2 (Winter 1985), page 51. The author is an experienced computer user with no particular fish to fry who is oriented toward the use of the Macintosh in educational contexts.

Abacus, by the way, is an intriguing publication that shows signs of developing into a unique and useful magazine. The quality of writing is high, the percentage of advertisements low, there is virtually no boosterism of "in" computers or software products, and the magazine is eclectic. The issue mentioned contained articles on human computers (how calculating prodigies perform arithmetic), a report on personal computers at Amos Tuck School, an exposition of "The Applicative Style of Programming," a review of the special *Scientific American* issue on software, and a report by chess master David Levy on a match between himself and CRAY BLITZ (winner of the 1983 World Computer Chess Championship). *Abacus* is published by Springer Verlag New York Inc., a source of scientific and academic texts.

The second notable article is "Apple Macintosh Software" by Dr. Ted Lewis of the Computer Science Department of Oregon State University, published in *IEEE Software*, March 1985, page 89. This is a combination review and tutorial of "the coding standards and methods forced upon application programmers who accept the challenge of writing applications for the Macintosh." It includes a simple example program with pull-down menus and an explanation of the structure and contents of the resulting "resource file." This article is the first reasonable explanation I have seen of this subject and is orders of magnitude more comprehensible than the Apple documentation.

Dr. Lewis concludes: "The ideas and techniques incorporated into the Macintosh software are admirable, but I am struck by the boldness of Apple Computer. Programming a Macintosh is not easy for conventional programmers, and this has contributed to the slow appearance of Macintosh software. Does Apple expect everyone to change? Do they think that a technical achievement like the Macintosh is inducement enough to persuade programmers to wade through 800 pages of *Inside Macintosh* merely to discover the merits of object-oriented programming and Desktop?

"The Macintosh software is unconventional, nonstandard, and probably horribly nonportable. The dialect of Pascal used is related to UCSD Pascal, but there is little hope of separating it from the Macintosh ROM routines. Yet, the software is nearly a work of art; all objects are organized in a logical and concise manner. The notion of a resource file is powerful and will most likely be emulated by programmers everywhere."

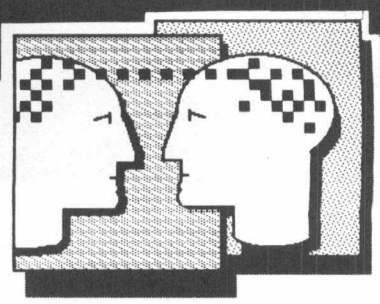
DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 196.**

```
function isqrt(i)
            integer*4 i,js #32 bits
            byte j1(4)
            equivalence(js,j1(1)) #with reversal, j1(4) is high byte of js
#you didn't take care of zero and negative arguments!
            js=i
            #count high order 0 bits 8 at a time
            for(k0=0;j1(4)==0;k0=k0+8)js=ishft(js,8) #shift left 8 bits
            for(;js>0;k0=k0+1)js=ishft(js,1) #count remaining 0 bits
            #get initial approximation within 50%
            isqrt=ishft(3,ishft(31-k0,-1)-1) #3*2**((29-k0)/2),not for i=1
            do k=1,3 #improve by Newton iterations
                  isqrt=ishft(i/isqrt+isqrt,-1) #(i/isqrt+isqrt)/2
return
end
```

Listing One
Tom Prince's square root algorithm

CP/M EXCHANGE



by Robert Blum

The CP/M Exchange RCP/M system is available for your use 24 hours a day, 7 days a week. Reach it by dialing (404) 449-6588.

Online . . . at Last

For more than two years I have been promising to bring up a RCP/M system in support of this column—finally, it's online. I hope you will participate in its use and share with me your thoughts on how I can improve it.

To reach it, call (404) 449-6588 any hour of the day or night, seven days a week. The modem is a Racal-Vadic 3452, which is capable of communicating at either 300 or 1200 baud. The other hardware components are an Ampro single-board Z80 microcomputer driving two 5¼-inch Teac floppy disk drives and a 20 Mb hard disk drive. This configuration should be more than adequate, at least for the present, to support the amount of activity that I anticipate.

The main purpose of the system is to provide improved access to the information printed in this column. The complete text of all the programs and updates printed here will be available for downloading. I hope this will stimulate even more reader participation in the column by providing a convenient method for submitting items of interest. I will also maintain a current list of all known CP/M bug reports and any fixes that might be available; this includes application notes and fixes published by DRI.

When your call is answered, hit the return key several times to indicate the baud rate you are using. Next you will be asked whether your terminal needs nulls for proper operation. Once beyond these two preparatory steps, you will enter into the message system. From this point on, I think the system is self-prompting enough to get you started without requiring a lengthy tutorial.

If you call to transfer program files, enter a "J" at the main menu prompt to jump to CP/M. Once under CP/M control, execute the MAP transient program to get a system map, which will guide you around the system.

Rather than spend a lot of time talking here about what you can expect when calling my RCP/M system, I suggest that you pick up your phone and call. Again, please leave a message outlining your impressions of the system and any improvements that you might want to suggest.

The Free Software Catalog and Directory

Public domain software is software that legally can be copied, used, and given to another user without payment of a royalty to its author.

Estimates project that over 40 Mb of program source code are contained on the 300+ volumes of CP/M public domain software. In this vast library can be found just about any type of utility program that you might need to help manage your CP/M system. A number of business application programs, mostly written in CBASIC, are present as well.

Until now, many would-be public domain software freaks have been defeated by the thought of searching through over 5000 program titles to find just the few desired. Fortunately, Robert A. Froehlich, a long-term member of SIG/M and an avid user of public domain software, foresaw this problem and has compiled a reference book detailing and indexing each program.

The Free Software Catalog and Directory is a 475-page 8½ x 11-inch book printed on newsprint stock. It focuses on the two largest and most readily available free software libraries, the CP/M Users Group (CPMUG) and the Special Interest for Microcomputers (SIG/M), while providing all the information necessary for personal computer owners to take advantage of this resource.

The introduction begins by setting aside any fears one might have about using public domain software. It concludes by talking a little about the main sources of free software. But most important is the elaborate tutorial—the best I have seen—for successfully completing a session with a remote bulletin board system (BBS). This section in itself makes the book a worthwhile investment for the novice and experienced user alike. The remainder of the introduction prepares the reader for dealing with source code files and explains how to go about making changes to them, should that be necessary.

The real meat of this book is contained in the next six sections. Every file in the entire 92-volume set of the CPMUG library and the first 162 volumes of the SIG/M library is described in detail. The first and second sections, the largest of the six, include complete information for each file as follows: filename, size in Kbytes, CRC checksum, date of entry, language if program source code, author's name, revisor's name, file title, keywords assigned to the file, and finally a textual description of the file.

Of course, the main purpose of this book is to provide ready access to information about the routines or programs of *interest to* you. The next four sections cross reference each file alphabetically by keyword, language type, author name, and filename. For example, if you want to find a program to compare two files for equality, your initial search might start in the keyword section under utilities. You find there two entries. If the one

for comparing binary files sounds like it would serve well, note the reference: SIG/M volume 115. You can then track down a complete description of the chosen file by referring to the second section of the book, which covers the SIG/M volumes.

As if all this weren't enough, the book concludes by listing several hundred user groups and bulletin board systems by name, location, and phone number.

This book has become the most used of any in my bookcase. It is a tremendous bargain at $9.95. The author has accomplished a phenomenal task in both compiling this much material and cross indexing it for ease of access. Whether you are an individual user of free software or the librarian of a user group, you will benefit from having this book.

If you're having trouble finding a source of free software in your local area, call the "CP/M Exchange" RCP/M system at (404) 449-6588 for a list of user groups where public domain software can be found.

8088: 8- or 16-bit Processor?

The following paragraphs, taken from the introduction to the Intel *iAPX 88 Book*, set the record straight on how the manufacturer intended the 8088 to be classified:

"This book describes the unique Intel 8088 microprocessor, the outstanding choice for 8-bit microcomputer applications requiring both high performance and low cost.

"The Intel 8088 is the most powerful 8-bit microprocessor available today, yet as easy to use as other 8-bit microprocessors designers have used for years."

Outside of the marketing rhetoric, these two paragraphs contain the phrase "8-bit" often enough to dispel any doubt that the 8088 was designed as an 8-bit processor.

Exiting Properly to CP/M

Two accepted methods for returning to CP/M from an application program are documented by DRI in the CP/M 2.2 reference manual. Although both are said to achieve the same result, in practice they react very differently and, depending on the situation, can cause a great deal of frustration.

Steve Russell of SLR Systems brought this incompatibility to my attention. In the process of testing one of his assemblers, he stumbled onto a difference between exiting to CP/M by jumping to memory location zero (BOOT) and by calling the BDOS with a System Reset function. The problem occurred when he tried executing several assemblies through a submit file in conjunction with XSUB. The first assembly finished successfully, but all the others failed because for some reason XSUB was no longer active after the first assembly.

Apparently, the documentation wasn't updated when XSUB was released. When loaded, XSUB's first action is to relocate itself immediately under the CCP and to modify the BOOT address so that it no longer points to the BIOS jump table but to the XSUB warm start routines. This modification is necessary so that XSUB can trap any further warm start requests to avoid having to reload memory with a fresh copy of the operating software. Thus, XSUB remains resident and active. But if the BDOS is called with the System Reset function, a fresh copy of the operating software is immediately loaded without checking whether XSUB or a similar program is active.

Because XSUB was designed for use within SUBMIT batch procedures, the result of a program returning to CP/M via the BDOS reset call is that suddenly XSUB no longer remains active: the BOOT vector stored at memory location 0 has been reset, and a fresh copy of the operating software is loaded. Any other programs in the same procedure that depend on XSUB to provide console input from the batch file will then fail to execute as planned.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 197**.

Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

#107 SEPTEMBER 1985 $2.95 (3.95 CANADA)

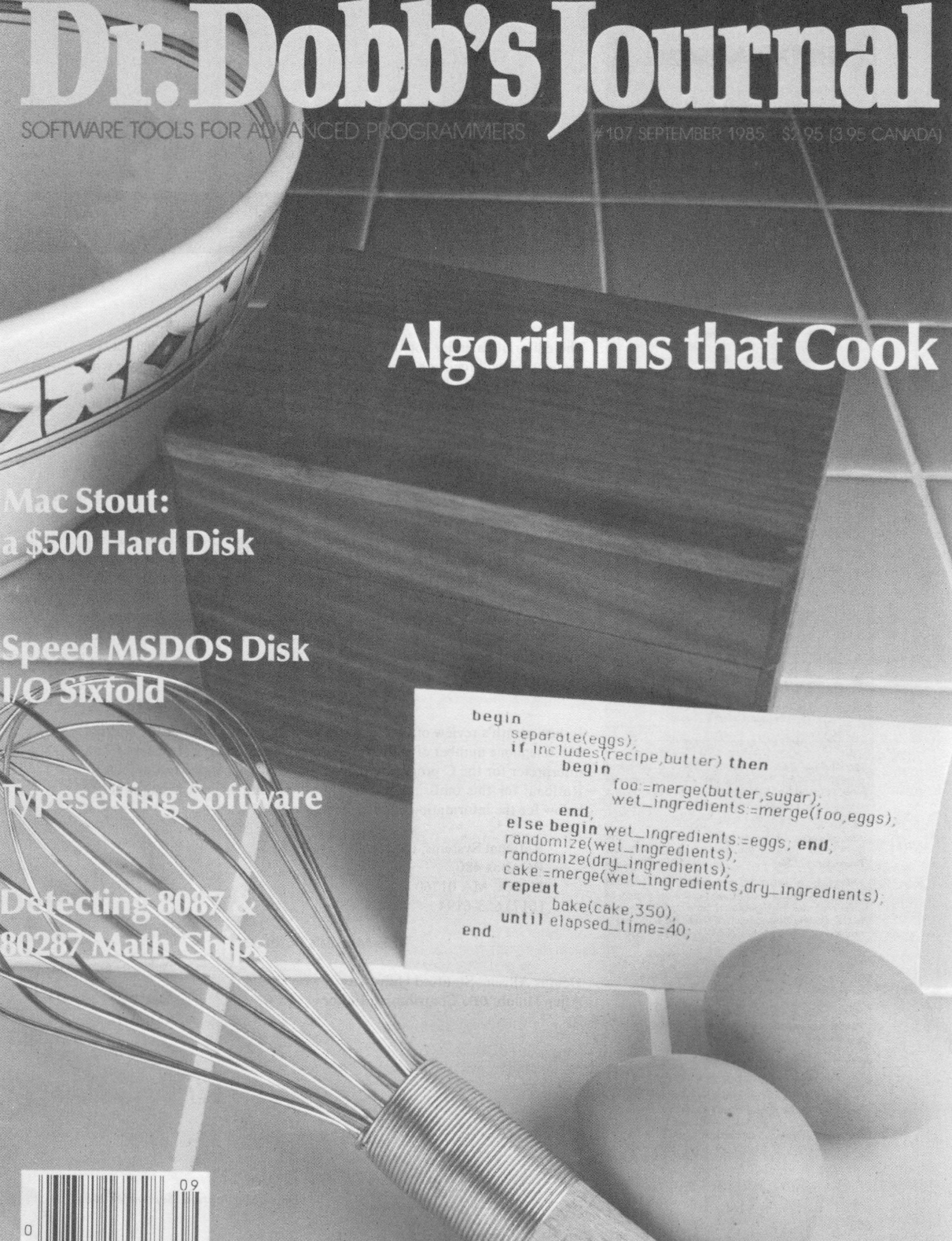
Algorithms that Cook

Mac Stout: a $500 Hard Disk

Speed MSDOS Disk I/O Sixfold

Typesetting Software

Detecting 8087 & 80287 Math Chips



Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Vince Leone*
Typesetter *Jean Aring*

Production
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover Artist *Tom Upton*

Circulation
Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Snee*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Kathleen Boyd*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*
Administrative Coordinator *Kobi Morgan*

Advertising
Advertising Director
Shawn Horst (415) 424-0600
Advertising Sales
Walter Andrzejewski (617) 567-8361
Lisa Boudreau (415) 424-0600
Beth Dudas (714) 643-9439
Michele Beaty (317) 875-0557
Advertising Systems Manager
Ron Copeland (415) 424-0600
Advertising Administrative Manager
Anna Kittleson (415) 424-0600

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $46 for airmail to Canada, $62 for airmail to other countries. Foreign subscriptions must be pre-paid in U.S. Dollars, drawn on a U.S. Bank. For subscription problems, call: outside of CA 800-321-3333; within CA 619-485-9623 or 566-6947.

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 686-1520 TELEX: 620430 (WUI)

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

September 1985
Volume 10, Issue 9

CONTENTS

Editorial Calendar

The editorial calendar for the first three months of 1986 has been set. In January we will focus on programming for the 68000, in February we will present a sampling from the structured languages Pascal, Ada, and Modula 2, and finally in March we will look ahead to what computing will be like in the future, with particular concentration on parallel processing. Article submissions and new product announcements should be in our hands no later than three months in advance of any issue.

Late Breaking News

In "16-Bit Software Toolbox" in June and again this month Ray Duncan mentioned patches distributed by MicroPro Technical Support that allow WordStar to use the HP LaserJet's boldface, italics and various fonts. As this issue was going to press Ray received a letter from MicroPro stating that the LaserJet patches were intended as a temporary fix and are no longer available. Updated printer enhancement disks that include LaserJet support have been sent to all MicroPro dealers.

Corrigendum

In last month's review of C Compilers we accidentally left out the address and telephone number of Rational Systems, manufacturer of Instant C, an interpreter for the C programming language. We apologize to the folks at Rational for this omission and include their address and phone number below for the information of our readers:

Rational Systems, Inc.
P.O. Box 480
Natick, MA 01760
(617) 653-6194

This Month's Referees

Dennis Allison, Stanford University
Allen Holub, *DDJ* Contributing Editor

Dr. Dobb's Journal

ARTICLES

Parallel Pattern Matching and Fgrep *by Ian Ashdown* **46** The Aho-Corasick algorithm for searching for multiple patterns in parallel is implemented in the Unix utility fgrep. **(Reader Ballot No. 195).**

Bose-Nelson Sort *by Joe Celko* **68** The attractions and limitations of the unstable sort. **(Reader Ballot No. 196).**

Two TEX Implementations for the IBM PC *by Richard Furuta and Pierre A. MacKay* **80** In honor of algorist Donald Knuth a review of two small TEXs. **(Reader Ballot No. 197).**

COLUMNS

Dr. Dobb's Clinic *by D. E. Cortesi* **10** A summary of papers on compiler design; the fastest way to request disk input in MSDOS. **(Reader Ballot No. 190).**

C Chest *by Allen Holub* **18** A simple arithmetic expression analyzer illustrates the fundamental operations of recursive descent compilers. **(Reader Ballot No. 192).**

Unix Exchange *by Axel Schreiner* **30** Communication between Unix and processes: the theory of signals; individual signals; how you control them. **(Reader Ballot No. 191).**

Software Designer *by Michael Swaine* **42** A survey of sort and search algorithms. **(Reader Service No. 194).**

Mac Toolbox *by John Bass* **94** John Bass inaugurates this new column with MacSCSI, a do-it-yourself hard disk interface for the Mac. **(Reader Service No. 198).**

CP/M Exchange *by Bob Blum* **110** Farewell to CP/M Plus; one user's impressions of the AMPRO Little Board. **(Reader Service No. 199).**

16-Bit Software Toolbox *by Ray Duncan* **114** The HP LaserJet; the IBM and Microsoft Macro Assemblers; Numeric coprocessors; Micro Solutions UNIFORM; Control-C under MSDOS. **(Reader Service No. 200).**

DEPARTMENTS

Editorial 6

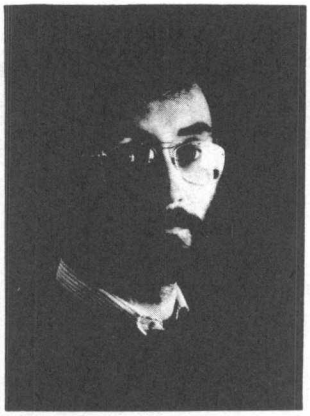
Letters 8

DDJ Classifieds 100

Of Interest 122

Advertiser Index 128

EDITORIAL



At *Micro Cornucopia*'s recent Semi-Official Gathering of hardware designers, I heard George Morrow express the computer vendor's dilemma: supplying solutions without knowing the problems. There is, Morrow explained, a very high wall. Computer hardware manufacturers dwell on one side, and in the dense jungle on the other side prowl strange questing beasts called customers. The manufacturers blindly, randomly throw machines over this wall. Occasionally one of them is rewarded with the sound of feeding somewhere beyond the wall, and they all rush to throw their machines at this latest feeding spot.

Publishing a magazine for advanced programmers is not quite such a random process: there are various means by which we can know our customers. We read what you write on the postage-paid response cards included in every issue. We do regular reader surveys, because interests can change quickly in so volatile a field as computer programming. And we have begun a spot survey that will give us quick information about our readers' interests.

Listening to your feedback has helped us in shaping an editorial calendar for 1986. On your advice, we will be looking more closely at advanced processors in 1986, while not abandoning machines of the 8080/6502 class. We asked for feedback on our hardware issue in July, and although you were not univocal, we think we got the message. We will consequently not run a hardware issue next year, but will run hardware articles whenever they seem appropriate and good.

The trick is to satisfy the diverse interests simultaneously. As this issue exemplifies, we are focusing more on algorithms, and in other ways attempting to present tools that are as portable, as broadly useful as possible. But to follow that path to the end would mean never publishing any assembly code, and that would cut out some of the best pieces that come into the office.

So we'll continue the juggling act. We'll publish articles focusing on particular topics, presenting specialized programming tools, sometimes with code written in assembly language when that's the best medium for it, but always with tips on ways to port these tools to other environments. We'll maintain an emphasis on the underlying algorithms. And we'll try to maintain balance in the magazine, making sure that we're supplying something for everyone.

But we are still interested in good assembly language programming; in fact, our first issue of 1986 will focus on programming for the Motorola 68000 processors. It's not too late to send us your 68K code, but don't wait: the deadline for submissions is October 1.

Of course, some 68000 machines try to pass as 8088s, like the Amiga, the Mac-killer that Commodore has thrown over the wall into the IBM area, just south of a mob of hostile dealers. My friend Norm down at the Fab Lab has been researching that commonly observed phenomenon of two people achieving the same breakthrough at the same time. Norm's preliminary findings indicate that this may be part of some larger resonance in technological decision making. For example, just as IBM dropped the 5¼-inch floppy disk, Commodore picked up that medium as the means of getting software onto the Amiga quickly—IBM PC software, that is. Commodore hopes people will use the Amiga as an IBM PC with good games until the native-mode and/or graphics interface-based software arrives. If the software doesn't arrive—ah, is that why they didn't put the company name on the machine?

Michael Swaine

LETTERS



Hardware vs. Software

Dear *DDJ*,

I just received the July issue of *Dr. Dobbs*. In answer to the question you pose in your editorial: No, do not include hardware articles. Other magazines, such as *Byte*, do a fine job of covering hardware. Few magazines do such a fine job of covering software.

If you expand your scope to include hardware, you will dilute your focus, and certainly reduce your value to me.

Guy Scharf
2163 Jardin Drive
Mountain View, CA 94040

Dear *DDJ*,

In answer to your editorial query about hardware articles, I'm in favor of having semi-annual hardware issues, with monographs and project papers available from you (at a nominal fee, of course). There are many of us that can't afford the big bucks for some of the hardware items, but have the skills to build them if the instructions are adequate.

Vladimir Ushakoff
4753 Preston-Fall City Rd., SE
Fall City, WA 98024-5705

Dear *DDJ*,

I have thoroughly enjoyed the hardware articles in recent issues and would like to see more of them. I had begun to think no one cared about the hardware hacker anymore. I can see your point, though, that *Dr. Dobb's* may not be the appropriate vehicle for such articles since it really is a software journal.

Why not start a separate journal, a hardware companion to *DDJ*? It could contain reviews of new chips with application examples and surveys of different classes of chips (display or disk controllers, for example) as well as construction articles. From the response you report to the recent hardware articles, I think you can see the enthusiasm there might be for such a project.

Try one issue dedicated entirely to hardware and see how it is received. You already have the best software journal. Why not have the best hardware one too?

David Nye
209 W. Lowe's Creek Rd.
Eau Claire, WI 54701

Dear *DDJ*,

This is in answer to Michael Swaine's question about coverage of hardware in *Dr. Dobb's Journal*. It seems to me that there is only one way to reply: there are already plenty of articles on hardware tinkering in other journals, and it is out of character for *DDJ* to use its limited space for such articles. An occasional, exceptional article is just tolerable. Reviews of innovative hardware, such as have appeared in *DDJ* through all its years of publication, are helpful, as even the ads are. Anything more than such as these is inappropriate and decreases the value of the journal to those of us who subscribe to and read it for what it claims to be: a journal of "Software Tools for Advanced Programmers." *Dr. Dobb's* is unique and uniquely valuable, and let's keep it that way.

In our department, we have a complete file of *DDJ*, some volumes of it (especially those with articles on Forth) getting dog-eared, and all of them well used. Of other journals, only a few back issues have been saved by our engineers, for articles on roll-your-own computers or components that they may build—someday. I would wager that this is pretty much the way it is with most *DDJ* readers.

Even in the days when many, if not most, personal computer users operated with homemade hardware from kits, *DDJ* addressed software rather than hardware needs. In these days of (almost) standardized and universally available hardware, the audience for useful software articles is larger and still growing.

Wayne C. Williams
Greenville, NC 27834

Unix Exchange

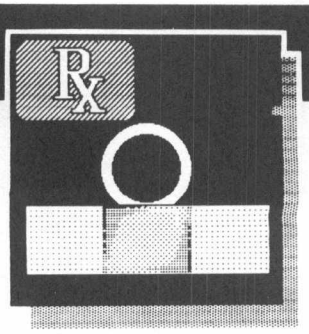
Dear *DDJ*,

I have a comment on Axel Shreiner's June 1985 Unix Exchange column on tricks with the C preprocessor. In the section on program argument standards, a set of argument parsing macros are presented after the author writes "it would be so simple to develop a standard as in" the macros. Why, I ask, is a standard being proposed that has already been implemented in a software tool that is readily available? The **getopt** command line option parser makes it simple to parse Unix command line options consistently, and unlike the proposed macros, is well accepted. Public domain versions of **getopt** have been posted to the USENET. In my opinion, the code written with **getopt** is more readable than the macros presented in the article. One reason for this might be that the macros do not behave anything like functions in the C language, and so might have unpredictable behavior.

Should you be interested, I would be happy to send a printed version of the **getopt** parser for your readers.

Gary Perlman
Wang Institute
Tyng Road
Tyngsboro, MA 01879 DDJ

DR. DOBB'S CLINIC



by D. E. Cortesi

Pascal Sources

We did some reading before writing last month's diatribe on Turbo Pascal. Hating to let good research go to waste and with the hope of finding one or two compiler implementors in the readership, we thought we'd list some of the better sources. All of these are from the British journal *Software: Practice and Experience* (*SP&E*). You should be able to find back issues in a university library.

What To Optimize

A lot of data has been published on the way people use language features, with profound implications for the design of a compiler's generated code. The first paper on this subject—and one of the most-cited computer science papers ever—is D. E. Knuth's "An Empirical Study of Fortran Programs" (*SP&E*, Vol. 1, No. 2, 2nd Quarter 1970). In this pioneer study of static and dynamic usage of language features, Knuth found what many others have since verified: the most-used features are the simplest ones, and the 80–20 rule is a good approximation for dynamic execution frequency in most programs.

A more recent work in this line is M. Shimasaki, *et al.*, "An Analysis of Pascal Programs in Compiler Writing" (*SP&E*, Vol. 10, No. 2, February 1980). This team studied the static and dynamic frequency with which Pascal's features were used, examining several Pascal compilers written in Pascal. In static usage, they found that 30–40% of all statements were assignments, 30–40% were procedure calls, and most of the rest were IFs. The proportion of procedure calls was much higher than Knuth and others had found in Fortran and PL/I programs; furthermore, about half of the procedures were named only once! This may not be typical of ordinary Pascal programs, as compilers are more likely to be planned and written in top-down modular style.

When they compiled the compilers to P-code and instrumented the P-machine, they found that 40% of all executed instructions were load-to-stack operations. On breaking that down, they discovered 9% of all operations were "load constant," 19% were "load global variable," while 11% were loads from some level other than global. Of the latter, 71% (7.8% of all operations) loaded a local of the active procedure (this presumably included loading parameters). The remainder—only 3% of all operations—referred to locals of some intermediate procedure.

In "A Contextual Analysis of Pascal Programs" (*SP&E*, Vol. 12, No. 2, February 1982), R. P. Cook and I. Lee reported a static analysis of 120,000 lines of Pascal code. They produced some striking numbers: for example, that 89% of all subprograms had fewer than five local variables or that 47% of all parameters passed were constants. 99% of all subprograms had five or fewer parameters, and 92% had no more than two. These are important findings if you are designing a compiler's subroutine calling sequence. In a machine with a few registers and a stack, should you try to pass parameters in registers? Definitely yes: if you have as few as *two* scratch registers, nine subprograms in ten will receive *all* their parameters in registers!

Cook and Lee substantiated Knuth's observation that the simplest features are the most used. Among many other numbers, they reported that 94% of all array references used but a single subscript, of which 19% were written as a constant and 63% as the name of a simple variable. A compiler that, when looking at a subscript, checks for "name, delimiter" and "constant, delimiter" before going into a full expression parse is a compiler that will save time, four tries in five.

As to stack-frame usage, they found references to intermediate-level names to comprise less than 2% of all references occurring in procedures and less than 3% of those in functions. This "would seem to indicate that expensive mechanisms to implement up-level frame addressing are largely unnecessary." It would also seem to validate the design choice made in C—to have only local and global name scopes.

We can't resist pointing out that this kind of survey doesn't require deep theoretical insight or great originality. The information is genuinely significant and useful, but generating it requires only persistence, curiosity, read access to a quantity of source code, and—for dynamic studies—write access to the source code of a compiler. These things aren't rare among *DDJ* readers; if you were to study, say, the usage of C constructs in public-domain code, *DDJ* would be pleased to publish the results. And you don't have to do a broad-spectrum census like the ones described here. A survey of a single feature—the use of operators in expressions, the number of locals or parameters, even the length of names—is worth doing, *especially since* one such survey is likely to generate tools that can be used in others.

How to Optimize

We did run across a couple of papers that were truly original and insightful. No compiler designer should overlook them.

The first is J. Welsh's "Economic Range Checks in Pascal" (*SP&E*, Vol. 8, No. 1, January 1978). This paper doesn't deserve the obscurity into which it seems to have fallen. Welsh describes how he made a Pascal compiler use all the information available to it to identify which expressions might violate a subrange or array bound at runtime and which expressions absolutely could not. Knowing that, the compiler could generate code for runtime range checks where they were needed and omit them where they were not. The result was "highly successful" at reducing the cost of runtime checks to levels that are tolerable in production code.

The implications of Welsh's methods go well beyond range checks on array subscripts and assignments to subrange variables. After all, binary integers are only a subrange of the natural numbers, so Welsh's technique applies equally well to any integers. For example, it would allow the compiler to insert code to check for integer overflow where it might happen and omit it where it can't. Conceivably, the compiler, having forseen potential—or inevitable!—integer overflow, could try to make this impossible by rearranging an expression or converting an intermediate result to real.

Better still, the technique permits the compiler to know exactly how much binary precision is required by many integer variables and many intermediate results. Therefore, the compiler could allocate some variables as bytes even when the programmer had declared them as plain integer; or allocate byte-sized temporary variables in some expressions; or do some integer operations (especially comparisons) using faster byte-mode instructions.

Best of all, a compiler using Welsh's technique could give the programmer marvelous feedback. It could issue warning messages like: "This expression gets a runtime check because variable K might exceed a value of 135." You would look at your code and say, "Oh, so it might, I better fix that," or you might say, "No, it can't, I better declare K as a subrange instead of integer so my intentions will be clearer."

Finally, anyone who cares about compiler design should read D. Hanson's "Simple Code Optimizations" (*SP&E*, Vol. 13, No.8, August 1983). This is a superb paper, at once practical and educational. From the abstract: "The simplicity of most programs suggests that straightforward optimizations would produce the greatest dividends. This paper describes three such optimizations suitable for one-pass compilers . . . none requires major changes to the size or structure of the compiler [but each] results in at least a 10% reduction in object code size and a corresponding reduction in execution time."

Getting Standard

If you want to write a verifiably standard Pascal compiler—or bring an old one up to snuff (hint, hint)—you could begin in worse ways than by getting in touch with the Software Consulting Services of Lehigh University (Ben Franklin Technology Center 125, Murray H. Goodman Campus, Lehigh University, Bethlehem, PA 18015). There you can obtain the latest version of the British Standards Institute's Pascal Validation Suite and Quality Control Package. The Suite contains 740 test programs. The Package consists of the Standard Pascal Model Implementation (a complete Pascal compiler in ISO standard Pascal), a program that audits any Pascal source for ISO validity (it has been applied to itself, of course), and a set of tools that let you automate your test runs under Unix.

The notice of this offer (*Sigplan Notices*, Vol. 20, No. 6, June 1985) neglects to mention price or media formats. But it exists, so you've got no excuses left, Phillipe . . . .

MSDOS Disk Speed

When we published your figures on throughput, we remarked on how unlikely some of the MSDOS numbers seemed. Users of IBM PCs and Zenith Z100s reported throughput rates, using 5-inch diskette drives, of 9 kilobytes per second and more. These were strikingly faster than the rates reported by any 8-bit system with similar disks and competitive with 8-bit systems using hard disks and M-drives.

How does MSDOS do it, we asked, and perhaps we gave the impression we doubted the numbers. (We never supposed the numbers were false; we did suspect that some unreported buffering or caching factor in MSDOS was producing apparent high speed while not doing the same amount of I/O as other systems.)

Well, we've measured the effect with our own stopwatch and it's real, alright. We still don't know how it's done, but we do know how your programs can tap into it for a major speed improvement.

Five Ways To Read

There are five distinct ways for a program to request disk input in MSDOS on an IBM PC or compatible. The modern, preferred way is to open a "handle" for the file with DOS function 3Dh then use function 3Fh to ask for input of one byte up to 64K.

Two ways are based on the CP/M-compatible functions that use a File Control Block (FCB). The program must first open an FCB with DOS function 0Fh and establish the Disk Transfer Area (buffer address) with function 1Ah. Then it can request that one record be transferred from the file to the buffer with function 14h. The size of a record is set in the FCB; it may be any number from one byte up to 64K. Alternatively, the program may use function 27h to request reading any number (from one to 64K) of records of the current size, so long as their aggregate length doesn't exceed 64K.

The fourth method of input is to use DOS interrupt (not function) 25h. This reads raw sectors unrelated to the file structure of a disk. In order to use it for file input, your program would have to interpret the FAT, disk directory, and subdirectories on its own.

The fifth way is to call on the IBM or compatible ROM BIOS through interrupt 13h. Like DOS interrupt 25h, this provides raw sector input unrelated to the file system. It's even more device oriented, however, in that you have to address data by side and cylinder, whereas the DOS interrupt

uses a scheme of "logical sector numbers."

We set out to see which of these ways was the fastest and, if possible, to figure out which of them produced those high throughput numbers and why. We ran a series of experiments on a DOS 2.1 system in an XT-ized PC. We confined our recorded measurements to the double-sided diskette drive; hard disks vary widely from brand to brand, and we couldn't find out any of this one's vital statistics such as number of tracks and cylinders or seek times.

First we formatted a new diskette. On it we created a file of 16 cylinders (144K) using the BASIC program in Listing One (page 17). That file, which commenced in cylinder 1, was the data source for all experiments. The object was to see how fast it could be read.

The DOS command Copy set the initial hurdle. We read the file with

```
COPY TESTFILE.001 NUL
```

and timed it: 9.8 seconds. That's quick; just how quick wasn't clear until we tried other programs.

Measure Molasses

Our first trial was written in BASIC (see Listing Two, page 17). BASIC makes it easy to calculate a program's elapsed time. Unfortunately, we had no difficulty timing this one with a hand-held watch; it took 122 seconds to read the file. Although there's no good reason why an I/O-bound job should run any faster when compiled, we figured it couldn't very well run slower. It didn't, but at 64 seconds it still wasn't what we'd call fast.

Maybe one of the other compiled languages would do better? We compiled the program of Listing Three (page 17) with Microsoft Pascal version 3.3. It ran in 61 seconds. Then we tried the program in Listing Four (page 17), compiled by Microsoft C version 3.0. Sixty-one seconds, ho hum. Here's how four language implementations do at reading a sequential text file; the numbers are multiples of the Copy command's time:

BASICA	12.5
BASCOM	6.5
Pascal 3.3	6.2
C 3.0	6.2

You can say this makes the languages look bad, or you can say it makes Copy look extremely good. But how does Copy do it?

Down to Brass Stacks

The remaining tests had to be done at the assembly-language level. Rather than writing and assembling programs for each of many tests, we used the Assemble command of Debug to write short programs. These could be parameterized by setting counts in their registers and by modifying constants with Enter. The entry of the handle-input test, for example, resembled Listing Five (page 16). Once that code was entered, the program could be set up for a certain input length by entering a new immediate value into one instruction. For example,

```
-e 10e,48
```

sets up to read two cylinders' worth of data on each DOS call.

We took a sequence of measurements, varying the number of bytes requested per DOS interrupt. For handle input, the results were:

CX=bytes	Seconds
200h	61.8
1200h	13.6
2400h	10.2
4800h	12.0
9000h	10.2

Reading a handle by sector-sized chunks (200h bytes) is clearly not a good idea (the similarity of that time to those turned in by the C and Pascal compilers is suggestive, however). Reading by tracks (1200h bytes) is five times faster, and reading by cylinders (2400h) is faster by a remarkable factor of six. But block sizes beyond 9K, one cylinder, don't give further improvement.

The new handle operations (there must be a better name for them) are supposed to replace the older CP/M-like ones. To do that, they had better be no slower, and our tests indicate they are not. First we tried reading single records with function 0fh while varying the size of the record:

Record	Seconds
80h	64.5
1200h	13.8
2400h	10.0
4800h	12.0
9000h	10.0

The results are practically identical to those for function 3Fh above—differences of 0.2 second aren't significant for hand-made timings. The time when reading 128-byte records is suggestively close to the BASCOM time.

When we held the record size constant at 200h and read different numbers of records with function 27h, we got identical times. The lesson is that no matter which DOS function you use, you should request at least 4,608 (nine times 512) bytes at a time. To ask for less is to multiply your diskette input time by a factor of 3 to 6.

Greased Lightning

An assembly program reading full cylinders can approach the speed of the Copy command. Is that the limit? We suspected as much but continued our experiments.

When we used the ROM BIOS interrupt 13h to read the same cylinders, the best time we could achieve was 13 seconds, 30% *longer* than the best time we got with DOS input requests! Now we know two things: the DOS BIOS doesn't use the ROM BIOS for its disk input, and DOS's BIOS does it *better*.

That made it mandatory to try out DOS interrupt 25h, presumably the foundation of the DOS file services. Here are the results:

CX=sectors	Seconds
1h	56.0
2h	31.0
4h	19.6
8h	12.5
9h	6.8
12h	6.8
24h	10.2
48h	8.5

Now, ponder those numbers a moment. The best times, as before, occur when the program asks for one track or one cylinder per call. Our program set up by forcing the disk to cylinder 0; during the timed portion,

it read cylinders 1–16, inclusive. That's 32 tracks read plus 16 one-track seeks. It takes 200 milliseconds for a 5-inch diskette drive to rotate once. There are just 34 periods of 200-milliseconds in 6.8 seconds. Therefore, DOS sequential input can reach and sustain an input rate of one track per disk rotation!

Hurrah for DOS; no faster input rate is possible. But we are left with some mysteries. What does DOS do differently from the ROM BIOS to get such performance out of the diskette adaptor? Why is 4-cylinder input slower than 2-cylinder; why is 3-cylinder input slower still? How does the Copy command interface to DOS? It consistently shaved a couple of tenths of a second off the best time we could get out of handle or FCB input.

To summarize these results formally: for $1 \leqq n \leqq 8$, if a program requests $n+1$, 512-byte sectors on each input operation, its input time will be roughly proportional to $1/n$. We can see the practical difficulties of giving every open file a 4,608-byte buffer, but still, isn't it sad how slow your compiled programs run compared to the Copy command?

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

Dr. Dobb's Clinic

(Text begins on page 10)

Listing One

```
100 'Create a 16-cylinder file on a d/s, 5-inch diskette
110 OPEN "TESTFILE.001" FOR OUTPUT AS #1
120 LET D$ = SPACE$(254) :' plus CR, LF = 256 bytes
130 FOR CYL = 1 TO 16
140     FOR SIDE = 0 TO 1
150         FOR SECTOR = 1 TO 9 :' DOS 2.x and above
160             FOR BYTES = 1 TO 512 STEP 256
170                 PRINT#1,D$
180             NEXT BYTES
190         NEXT SECTOR
200     NEXT SIDE
210 NEXT CYL
220 CLOSE #1
```

End Listing One

Listing Two

```
100 'Read a file and see how long it takes
110 ' -- function to convert TIME$ to seconds
120 DEF FNSEC(X$) = 3600 * VAL(LEFT$(X$,2)) +
60 * VAL(MID$(X$,4,2)) +
VAL(RIGHT$(X$,2))
130 OPEN "TESTFILE.001" FOR INPUT AS #1
140 STARTS$ = TIME$
150 WHILE NOT EOF(1)
160     LINE INPUT#1,D$
170 WEND
180 STOPS$ = TIME$
190 PRINT FNSEC(STOPS$)-FNSEC(STARTS$);" seconds."
```

End Listing Two

Listing Three

```
program readtest(output);
var
        fin : text;
        dat : string(255);
begin
        assign(fin,'TESTFILE.001');
        reset(fin);
        writeln('start timing',chr(7));
        while not eof(fin) do
                readln(fin,dat);
        writeln('stop timing',chr(7));
end.
```

End Listing Three

Listing Four

```
#include <stdio.h>
main()
{
        FILE *fin;
        char dat[256];
        if (NULL != (fin = fopen("TESTFILE.001","r")))
        {
                fputs("start timing \x7\n",stderr);
                while(NULL != fgets(dat,255,fin)) ;
                fputs("stop timing \x7\n",stderr);
        }
        else
                fputs("can't open file\n",stderr);
}
```

End Listing Four

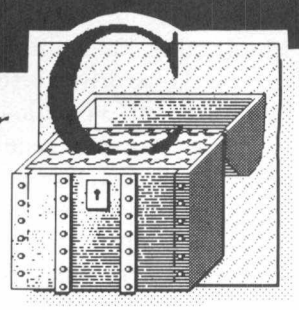
Listing Five

```
C>debug
-f 0 Lffff 0
-e 80 'A:TESTFILE.001',0
-a 100
0C44:0100 mov dx,80 ; filespec string
0C44:0103 mov ax,3d00 ; open it
0C44:0106 int 21
0C44:0108 mov bx,ax ; handle in BX
0C44:010A mov dx,200 ; data to DS:200
0C44:010D mov cx,200 ; block size
0C44:0110 mov ax,3f00 ; read CX bytes
0C44:0113 int 21
0C44:0115 cmp ax,cx ; read ok?
0C44:0117 jz 110 ; if so repeat
0C44:0119 nop ; breakpoint here
0C44:011A mov ax,3e00 ; close BX=hdl
0C44:011D int 21
0C44:011F jmp 100 ; set up for next
0C44:0121 ^C
```

End Listings

C CHEST

How Compilers Work—A Simple Desk Calculator



by Allen Holub

Over the years *DDJ* has published several recursive descent compilers, most notable among which are the various versions of Small-C. This month we're going to talk about how this sort of compiler works. In the interest of clarity we'll reduce the problem from recognizing a real computer language to analyzing a small arithmetic expression, a function that is part of a desk calculator program. The same techniques are applicable to both. Our expression analyzer will take as input an ASCII string representing an arithmetic expression. Only numbers, parentheses, and the operators **+ − /** and ***** are legal. The analyzer will return the result of the evaluation. For example, if you give it the string **(3+1)*2** the analyzer will return the number **8**. The routine is pretty stupid, but the purpose of this exercise is to understand compilers, not to analyze complicated expressions.

Every compiler has three functionally distinct parts. These parts are often combined, but it's best to look at them as separate functions. The first part of the compiler is a "token" recognizer. A token is some collection of ASCII characters from the input stream that are meaningful to the compiler when taken as a group. That is, a program can be seen as a collection of tokens, each of which is made up of one or more sequential ASCII characters. For example, the ASCII character ; is a token in C, similarly the keyword **while** is a token. The matter is complicated by operators like **+**, **++** and **+=**, all of which are single tokens. A token is, then, a sort of programming atom, an indivisible part of the language (you cannot, for example, say **wh ile**) and a token recognizer is a subroutine that, when called, will return the next token from the input stream. Usually tokens are represented internally as an enumerated type or as a set of integer values corresponding to **#define**s in a header file somewhere, and the token recognizer returns this integer value. (Small-C doesn't do this. Rather, various subroutines within the compiler retrieve the tokens from input one at a time as they are needed.)

The second part of the compiler, and the part that does most of the work, is the "parser." The verb to parse retains its meaning when applied to compilers: "to resolve (a sentence, etc.) into its component parts of speech and describe them grammatically."[1] Just replace the word "sentence" with "program." Computer languages may be described by means of a formal grammar (we'll look at these in a moment) and the parser breaks up a program into its component parts and interprets the parts in a larger, grammatical context. That is, a parser organizes the tokens returned from the token recognizer in such a way that the compiler can generate code conveniently.

A good (though not very practical) way to look at the process is as the creation of a "parse tree." For example, The expression **(a − b)*(c − d)** can be organized into the tree shown in Figure 1 (page 20).

The third part of the compiler, the code generator, traverses the parse tree in an orderly way, generating code according to certain rules. For example, if we do a "post order" traversal of the tree shown in Figure 1 (visit the left node, the middle node, the right node and then the root recursively) the tokens will be read in the following order:

(a b −) (c d −) *

Now, the expression may be evaluated by applying the following rules to each token in the tree as it is visited:

1) if the token is a parenthesis, do nothing;
2) if the token is a variable (**a**,**b**,**c** or **d**), push the variable onto a stack;
3) if the token is a minus (**−**), pop two items off the stack, subtract them and push the result;
4) if the token is an asterisk (*****), pop two items off the stack, multiply them together and push the result.

When you're done parsing (traversing the tree), the answer will be on the top of the stack. Owners of Hewlett Packard calculators will be familiar with the process. In a real compiler, the actual rule applied will be some function of the type of token found and the position of that token in the parse tree.

There are several flavors of parsers.[2] Most compilers use table driven parsers for several reasons. It's easier to automate compiler creation with table driven parsers; they're also more efficient. The Unix utility YACC (Yet Another Compiler Compiler), when given a formal description of a programming language, creates a set of tables that can be used by a generic, table driven parser. Similarly, LEX (LEXical analyzer) can output a C program that recognizes tokens.[3]

Unfortunately, most public domain compilers (and many commercial ones) don't use the more sophisticated table driven methods (Small-C is no exception). These compilers use a parsing method known as "recursive descent." Recursive descent parsers are easier to understand than their table driven cousins. However, they have disadvantages. They are inherently inefficient, using large amounts of stack space, and con-

struction of them cannot be automated. To change the way a recursive decent compiler works, you *have* to change the compiler itself. To change a table driven compiler you need only change the table. So, maintenance is a problem with recursive descent compilers.

Grammars: Representing Computer Languages

The best way to start writing a program is to reduce the problem to some sort of symbolic form. Pseudocode, flow charts, Warnier-Orr diagrams are all examples of this kind of symbolic reduction. Compilers are no exception to this process. When writing a compiler, you start by representing the programming language to be compiled in a formal, symbolic format called a "grammar." Any one programming language can be described by several grammars. The type of parser you're going to use will determine which of these is best for your application. The most useful notation used for grammars is the Backus-Naur Format (abbreviated BNF), which we'll see in a moment.

In order to create our expression analyzer, we need to start with a grammar. The first question to ask is: what exactly is an expression? You'll remember from high school that an expression is composed of factors. A factor by itself (i.e. a single number) is an expression, as are two factors separated by an operator. BNF representations of these two rules are:

```
<expression> ::= <factor>
<expression> ::=
<factor> <operator> <factor>
```

We can save some typing by using the vertical bar to represent **OR**:

```
<expression> ::= <factor>
¦<factor> <operator> <factor>
```

You'll note in these definitions that no element of the BNF definition of the expression is a real symbol, one that can actually be found in the input stream. That is, <factor> and <operator> both have to be defined further before they can be related to a real program. Symbols, such as <factor>, which need further definition, are called "non-terminal" symbols. Symbols that *can* be found in the input are called "terminal" symbols. The four terminal symbols that can be operators are **+ − /** and *. A BNF rule for <operator> is:

```
<operator> ::= + ¦ - ¦ * ¦ /
```

Defining a factor is a little harder. A factor can be a number but it can also be another expression (as in: **a+b−d**, **a+b** is one factor and **d** is the second factor). A BNF definition of factor is:

```
<factor> ::= <number>
           ¦<expression>
```

The only symbol yet to be defined is <number>. Since a number is an easy thing for the token recognizer to find, we'll cheat a little and just define <number> in English. Our entire grammar is shown in Figure 2 (at right).

Notice that every non-terminal used to the right of a **::=** is also defined to the left and that no terminal symbols are found to the left of a **::=**. Let's test the grammar by plugging in an example: **1+2**. **1** and **2** are both <number>s so we can replace them with the equivalent non-terminal symbols by using rule 4:

```
        1 + 2
<number> + <number>
```

According to rule 3, a single number is also a factor, so we can do another replacement:

```
<number> + <number>
  <factor> + <factor>
```

The **+** can be evaluated using rule 2:

```
    <factor> + <factor>
<factor> <operator> <factor>
```

And finally, by using rule 1, we can replace the above with a single <expression>:

```
<factor> <operator> <factor>
        <expression>
```

So, we can reduce the input tokens **1+2** to an <expression> using the rules of the grammar. Therefore, we conclude that **1+2** is a legal expression in our grammar.

What if there's an error in the expression? Let's try to parse **1+***. We can apply rules 3 and 6 to yield:

```
<number><operator><operator>
```

and then apply rule 4 to get:

```
<factor><operator><operator>
```

But there is no rule we can apply to reduce this any further. So, we conclude that **1+*** is not an expression as defined by our grammar.

Parsing with a Grammar

So a parser can be seen as a program that reduces a collection of input tokens to a single non-terminal. We have just "parsed" the expression **1+2**. To turn a parser into a real compiler, we need to make it do something active too, namely, generate code. So, we associate an action rule with each grammatical rule. Our grammar, slightly shuffled around and with action rules added is shown in Figure 3 (page 20). Every time we apply a grammatical rule, we'll also perform the action specified in the equivalent action rule. **1+2**, parsed with the grammar in Figure 3 is shown in Figure 4 (page 20). A somewhat more involved example is given in Figure 5 (at left). You're beginning to see how the process works. If the action rules had generated the code necessary to perform the operation, rather than actually doing the operation itself, we'd have a compiler.

As you've probably noticed, the grammar just defined isn't very useful. At very least we'd like to have parentheses and negative numbers. We'd also like to be able to negate an entire expression (i.e. **−(17*11)**). You'll also notice in the above examples that the expression is just parsed left to right, with all possible substitutions made as we parse. A more realistic grammar performs its substitutions in a somewhat more complex way, and the grammar has to reflect this complexity. In addition, a gram-

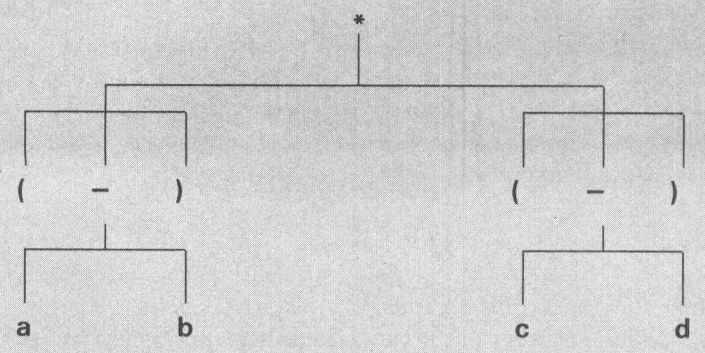


Figure 1
Parse Tree for the Expression (a – b) * (c – d)

```
1) <expression>    ::=   <factor>
                     |   <factor> <operator> <factor>
2) <operator>      ::=   + | - | * | /
3) <factor>        ::=   <number> | <expression>
4) <number>        ::=   A string of ASCII characters in the range '0' to '9'
```

Figure 2
A Simple Expression Recognition Grammar

Grammar:

```
1) <expression> ::= <factor>
2) <expression> ::= <factor> <operator> <factor>
3) <operator>   ::= + | - | * | /
4) <factor>     ::= <number>
5) <factor>     ::= <expression>
6) <number>     ::= Any string of ASCII characters in the range '0' to '9'.
```

Action rules:

1) Do nothing.
2) Pop two objects off the stack, apply the operator remembered in rule 3 and then push the result.
3) Remember the operator for rule 2.
4) Push the number onto the stack.
5) Do nothing.
6) Translate the ASCII string into a number.

Figure 3
Adding Actions to the Grammar

			rule:	action:
1	+	2		
<number>	+	2	6	Translate ASCII to int
<factor>	+	2	4	Push 1
<factor>	+	<number>	6	Translate ASCII to int
<factor>	+	<factor>	4	Push 2
<factor>	<operator>	<factor>	3	Remember the +
<expression>			2	Pop the 1 & 2, apply + and push the result.

Figure 4
Parsing 1 + 2 Using Grammar in Figure Three

mar has to be organized so that the parser can always tell what rule to apply based on the current input symbol and the current rule being processed. A better expression recognizing grammar is given in *Figure 6* (at left). We'll use this grammar in our actual program.[4]

A Recursive Descent Parser

The best way to see how a parser works is to look at one. Before discussing the parser proper, I want to talk a little about how the program (Listing, page 25) is organized. The actual subroutines in the parser are highly recursive. As such, they'll use up a lot of stack space as they work. Because of this stack usage, we want to pass as few parameters as possible to the subroutines (because all these parameters take up stack space). So, we make global those variables that would normally be passed to the subroutines as arguments. However, this practice, introduces new problems. In C, all non-static global variables are shared between all modules in a program. But the expression parser is probably going to be a library routine and we don't want it to interfere with the normal workings of the rest of a program. Moreover, we don't want the programmer to have to remember that certain globals are used by a particular library routine and can't be used anywhere else. So, we make the globals static. We'll also make static those subroutines that are only used internally. Now, however, we need some way to initialize the static globals from outside the parser module. We do this initialization with the "access routine" starting on line 84 of the Listing (the only externally accessible subroutine in the module). This access routine (called **parse( )**) does nothing but initialize our globals and then call **expr( )** to do the work.

Another organizational concern is the **main( )** routine on lines 34–79. The primary purpose of **main( )** is to test **parse( )**, thus the **#ifdef/#endif** on lines 32 and 81. **DEBUG** is not **#defined** when we compile for inclusion in a library. The **main( )** routine given is moderately useful in its own right. You can enter the expression (**17/(2*12)**) from the command line or you can just type **expr** and then

enter expressions as the program prompts you, sort of a rudimentary desk calculator.

Moving back to parsers, there are a few things to notice about the grammar in Figure 6. First, the left-most symbol following the ::= is always either a terminal or the same non-terminal for all rules. That is, all rules associated with <expr> have <factor> as their left-most symbol. The left-most symbol of all <factor> rules is either a terminal ((or −) or the non-terminal <constant>. The left-most symbol of a constant has to be an ASCII digit. This property of the grammar is required by the parser so that it can know what rule to apply in a given situation. For example, when evaluating an <expr>, the parser will always apply a rule associated with <factor> first.

A second property of the grammar is that the definitions for <expr> and <factor> are recursive. An <expr> is defined in terms of other <expr>s. The recursion in <factor> is two levels deep. A <factor> is defined in terms of an <expr>, which is in turn defined in terms of a <factor>. The recursion in the *grammar* suggests that we can also use recursion in a parser that implements the grammar.

So, given an appropriate grammar, we can translate that grammar directly into a parser. In the program given here, all non-terminal symbols in the grammar have an equivalent subroutine with the same name. The routine for <expr> starts on line 104, for <factor> on line 124 and for <constant> on line 155.

Looking again at the grammar in Figure 6, we see that the first thing done in all the <expr> rules (1–5). is to look for a <factor>. Similarly, the first thing the subroutine **expr( )** does is call the subroutine **factor( )** (on line 108). Looking back at the grammar, the next thing <expr> does is look for a terminal symbol (either a * / + − or a null string). The equivalent code is the switch on lines 110–117. The default case takes care of the null terminal (rule 1). The recursive evaluation of <expr> in rules 2–5 is also done in the switch. On line 119 **expr( )** returns the evaluated expression.

	Apply rule:	Action:
1 + 2 − 3	−	Start here
<number> + 2 − 3	6	Translate "1" to int
<factor> + 2 − 3	4	Push 1
<factor> <operator> 2 − 3	3	Remember +
<factor> <operator> <number> − 3	6	Translate "2" to int
<factor> <operator> <factor> − 3	4	Push 2
<expression> − 3	2	Pop two numbers, apply + and push result.
<factor> − 3	1	Do nothing.
<factor> <operator> 3	3	Remember −
<factor> <operator> <number>	6	Translate "3" to int
<factor> <operator> <factor>	4	Push 3
<expression>	2	Pop two numbers off the stack, subtract and push the result.

Figure 5
Parsing 1 + 2 − 3

<expr>	::=	<factor>	(1)
	¦	<factor> * <expr>	(2)
	¦	<factor> / <expr>	(3)
	¦	<factor> + <expr>	(4)
	¦	<factor> − <expr>	(5)
<factor>	::=	(<expr>)	(6)
	¦	− (<expr>)	(7)
	¦	<constant>	(8)
	¦	− <constant>	(9)
<constant>	::=	A string of ASCII characters in the range '0'–'9'.	

Figure 6
A More Realistic Expression Recognizing Grammar

Factor( ) is somewhat more complex. It first checks (on lines 128–132) for the leading minus sign required by rules 7 and 9. After stripping off the minus, rules 6 and 7 become identical, similarly rules 8 and 9 are identical once the minus is gone. So, **factor( )** now decides which rule to process by looking for a leading **(** (on line 134). If it doesn't find the parenthesis, rule 8 is processed (line 135) by calling the subroutine **constant( )**, otherwise rule 6 is processed by skipping past the parenthesis and then calling **expr( )** (lines 138–139). We can also do some error checking here by looking for a close parenthesis when **expr( )** returns (lines 143–147).

The final part of the parser is the routine **constant( )** on lines 155–169. This routine is essentially **atoi( )**, however it advances the string pointer past the end of a number and flags an error if a number isn't found.

You'll note that in this program (and in the Small-C Compiler) the three functional parts of the compiler are merged together. There is no explicit token recognizer, rather each routine is responsible for advancing the global string pointer (**Str**) past the token being processed. Similarly, the code generation part of the compiler is integrated into the parser. In our example, code generation is replaced by the various return statements. In a real compiler the routine **factor( )** would generate code to push a value onto a run-time stack rather than return a value. The switch on lines 110 to 117 would be replaced by something like:

```
switch( *Str )
{
case '+':
        Str++;
        expr( );
        codegen(1);
        break;
case '-':
        Str++;
        expr( );
```

```
        codegen(2);
        break;
    /* etc. */
    }
```

Since the code to push one number onto the stack is generated in **factor( )**, The first number will be pushed by the **factor( )** call on line 108. By the time **expr( )** returns, the code to push the second number will have been generated (by the **factor( )** call inside **expr( )**). The call to **codegen(1)** inside the switch generates the code needed to pop two numbers off the stack, add them together, and push the result. The **codegen(2)** call behaves similarly, but it subtracts rather than adds.

So, that's the bulk of the problem. Hopefully a better understanding of what's going on will help when you try to sort out the workings of compilers such as Small-C.

Footnotes

[1] *The Compact Edition of the Oxford English Dictionary* (Oxford: Oxford University Press, 1971) p. 2083.

[2] A good, short, description of table-driven parsing techniques can be found in: Dr. Henry A. Seymour, "An Introduction to Parsing," *Dr. Dobb's Journal*, 98 (December, 1984), pp 78–86. A more in-depth look at the subject, and at compiler design in general, can be found in: Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Reading, Addison-Wesley, 1979; P. M. Lewis, D. J. Rosenkrantz and R. E. Stearns, *Compiler Design Theory* (Reading: Addison-Wesley, 1976).

[3] Axel T. Schreiner and H. George Friedman, Jr., *Introduction to Compiler Construction with Unix*, (Englewood Cliffs: Prentice-Hall, 1985) is the best guide to YACC that I know of. It takes you, step by step, through the entire process of generating a C compiler using YACC and LEX. More terse descriptions of both programs are in: Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler," *Unix Programmer's Manual* Vol. 2 (New York: Holt, Rinehart and Winston, 1979) pp. 353–387 and "Lex—A Lexical Analyzer Generator," *ibid.*, pp. 388–400.

[4] A grammar for the C language is in Kernighan & Ritchie, *The C Programming Language* (Englewood Cliffs: Prentice-Hall) 1978 p. 214–219.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 192.**

C Chest Listing (Text begins on page 18)

```
 1: #include <stdio.h>
 2:
 3: /*  EXPR.C:  (C) Copyright 1985, Allen I. Holub.  All rights reserved
 4:  *
 5:  *     Evaluate an expression pointed to by str. Expressions evaluate
 6:  *     right to left unless parentheses are present. Valid operators
 7:  *     are * + - / for multiply add, subtract and divide. The expres-
 8:  *     sion must be formed from the following character set:
 9:  *     {  0123456789+-*()/  }. White space is not allowed.
10:  *
11:  *      <expr>      ::=    <factor>
12:  *                       | <factor> * <expr>
13:  *                       | <factor> / <expr>
14:  *                       | <factor> + <expr>
15:  *                       | <factor> - <expr>
16:  *
17:  *      <factor>    ::=     ( <expr> )
18:  *                       | -( <expr> )
19:  *                       |  <constant>
20:  *                       | -<constant>
21:  *
22:  *      <constant> ::=  A string of ASCII chars in the range '0'-'9'.
23:  *
24:  *-------------------------------------------------------------------
25:  *  Global variables:
26:  */
27:
28: static  char    *Str  ;  /* Current position in string being parsed  */
29: static  int     Error ;  /* # of errors found so far                 */
30:
31: /*-------------------------------------------------------------------*/
32: #ifdef DEBUG
33:
34: main(argc, argv)
35: char    **argv;
36: {
```

(Continued on next page)

C Chest Listing (Listing Continued, text begins on page 18)

```
37:         /*      Routine to exercise the expression parser. If an
38:          *      expression is given on the command line it is
39:          *      evaluated and the result is printed, otherwise
40:          *      expressions are fetched from stdin (one per line)
41:          *      and evaluated. The program will return -1 to the
42:          *      shell on a syntax error, 0 if it's in interactive
43:          *      mode, otherwise it returns the result of the
44:          *      evaluation.
45:          */
46:
47:         char buf[133], *bp = buf ;
48:         int err, rval;
49:
50:         if( argc > 2 )
51:         {
52:                 fprintf(stderr, "Usage:  expr [<expression>]");
53:                 exit( -1 );
54:         }
55:
56:         if( argc > 1 )
57:         {
58:                 rval = parse( argv[1], &err );
59:                 printf(err ? "*** ERROR ***" : "%d", rval );
60:                 exit( rval );
61:         }
62:
63:         printf("Enter expression or <CR> to exit program\n");
64:
65:         while( 1 )
66:         {
67:                 printf("? ");
68:
69:                 if( gets(buf) == NULL  ||  !*buf )
70:                         exit(0);
71:
72:                 rval = parse(buf, &err);
73:
74:                 if( err )
75:                         printf("*** ERROR ***\n");
76:                 else
77:                         printf("%s = %d\n", buf, rval);
78:         }
79: }
80:
81: #endif
82: /*------------------------------------------------------------------------*/
83:
84: int     parse( expression, err )
85: char    *expression;
86: int         *err;
87: {
88:         /* Return the value of "expression" or 0 if any errors were
89:          * found in the string. "*Err" is set to the number of errors.
90:          * "Parse" is the "access routine" for expr(). By using it you
91:          * need not know about any of the global vars used by expr().
92:          */
93:
94:         register int    rval;
95:
96:         Error = 0;
97:         Str   = expression;
98:         rval  = expr();
99:         return( (*err = Error) ?  0  :  rval );
100: }
101:
102: /*------------------------------------------------------------------------*/
```

(Continued on page 28)

C Chest Listing (Listing Continued, text begins on page 18)

```
103:
104: static int expr()
105: {
106:         int     lval;
107:
108:         lval = factor();
109:
110:         switch (*Str)
111:         {
112:         case '+':  Str++;   lval += expr();     break;
113:         case '-':  Str++;   lval -= expr();     break;
114:         case '*':  Str++;   lval *= expr();     break;
115:         case '/':  Str++;   lval /= expr();     break;
116:         default :                               break;
117:         }
118:
119:         return( lval );
120: }
121:
122: /*-------------------------------------------------------------------*/
123:
124: static int factor()
125: {
126:         int     rval = 0 , sign = 1 ;
127:
128:         if ( *Str == '-' )
129:         {
130:                 sign = -1 ;
131:                 Str++;
132:         }
133:
134:         if ( *Str != '(' )
135:                 rval = constant();
136:         else
137:         {
138:                 Str++;
139:                 rval = expr();
140:
141:                 if ( *Str == ')' )
142:                         Str++;
143:                 else
144:                 {
145:                         printf("Mis-matched parenthesis\n");
146:                         Error++  ;
147:                 }
148:         }
149:
150:         return (rval * sign);
151: }
152:
153: /*-------------------------------------------------------------------*/
154:
155: static int constant()
156: {
157:         int     rval = 0 ;
158:
159:         if( !isdigit( *Str ))
160:                 Error++;
161:
162:         while ( *Str && isdigit(*Str) )
163:         {
164:                 rval = (rval * 10) + (*Str - '0') ;
165:                 Str++;
166:         }
167:
168:         return( rval );
169: }
```

End Listing

by Axel Schreiner

This column is reprinted from the German quarterly *unix/mail* (Hanser Verlag, Munich, Germany). It is copyrighted © 1984 by Axel T. Schreiner, Ulm, West Germany. It may be reproduced as long as the copyright notice is included and reference is made to the original publication.

/lib/libc.a(signal.o)

Signal(2)[1] does not *generate* a signal but is the meager protection offered against the cruelty of **kill(2)**. This column demonstrates how to deal with signals. Because this topic presupposes a broad range of knowledge, we start with a theoretical overview.

Theory

Signal is the term for a number of events that a process may encounter in a more or less unforeseen fashion:

SIGHUP
may happen when the controlling terminal is switched off (i.e., once the interface loses the carrier)
SIGINT
can happen if the interrupt key is pressed at the controlling terminal, usually break or del
SIGQUIT
can happen if the quit key is pressed at the controlling terminal, usually ^ \
SIGKILL
is signal 9, which can be sent explicitly with the **kill** command or the **kill(2)** system call
SIGSYS
results from a system call with bad parameters—primarily if programs from a foreign operating system are run under Unix
SIGPIPE
is sent to the writer of a pipe, if the corresponding reader exits
SIGALRM
is received by a process once a time interval set up by **alarm(2)** has expired
SIGTERM
is signal 15, which is sent by **kill** as a default.

There are 16 signals, ranging from 1 (**SIGHUP**) to 16. They result from events at the controlling terminal, through errors in the process itself, or through actions of other processes. Signal 16 has no predefined meaning. The list above does not contain the names of the signals provoked by arithmetic errors (e.g., division by zero), by unknown machine instructions, or by access to nonexisting memory regions; these situations are somewhat machine dependent.

The system call **kill(pid, sig)** sends signal **sig** to the process **pid**. **Kill(2)** returns a 0 on success and −1 if, for example, a signal is sent to a nonexisting process. The super-user may send signals to arbitrary recipients; everybody else can send signals only to processes with the same user number. There is a process 0, but if a signal is sent to **pid 0**, it goes to all processes in the user's own process group. The super-user, by sending a signal to **pid − 1**, reaches all processes in the entire system with the exception of processes 0 and 1. This call is really only for the benefit of */etc/init* (i.e., for the process controlling the timesharing operation).

If a process does not take defensive measures, receiving a signal is deadly. For some signals, a memory dump is produced as a file core (assuming the process has appropriate privileges with respect to the working directory, the terminated program text, etc.). **SIGINT** and **SIGQUIT** differ precisely in that only **SIGQUIT** produces a memory dump. As a rule, you can terminate a process only by using the interrupt key. If the system is short of disk space, the super-user might occasionally execute

```
find / -name core -a -exec rm {} \;
```

to locate and remove all memory-dump cores.

A process can defend itself against all signals with the exception of **SIGKILL**.

```
#include <signal.h>
   . . .
   signal(sig, SIG_IGN);
```

SIG_IGN requests that the signal **sig** not be received by the process at all.

```
int f( );
   . . .
   signal(sig, f);
```

Receipt of the signal **sig** causes the function **f** to be called, which receives the signal number as an argument.

```
#include <signal.h>
   . . .
   signal(sig, SIG_DFL);
```

This *finally restores the* default setting (i.e., recognition that a signal is deadly and might produce a memory dump).

Signal states (i.e., **SIG_DFL**, **SIG_IGN**, or acceptance by a function) are preserved across **fork(2)**. **SIG_IGN** or **SIG_DFL** also remain in effect across **exec(2)**. A catch function set up with **signal(2)**, of course, cannot be retained across **exec(2)** (this system call eliminates program text and data in the running process); signals connected to a function are therefore implicitly returned to **SIG_DFL** during **exec(2)**.

Once a catch function has been set up using **signal(2)**, it is invoked only once for most signals. As soon as the process receives the signal (i.e., even before the catch function is actually called, the signal setting is returned to **SIG_DFL** for the next occurrence of the signal. That results in a race condition: if a signal is received twice in rapid succession, it is very likely to be deadly.

A catch function usually has the following form:

```
#include <signal.h>

static  trap(sig)
register int sig;
{
        signal(sig, SIG_IGN);
        . . .
        signal(sig, trap);
}

main( )
{
        . . .
        signal(SIGINT, trap);
        . . .
}
```

At the beginning of the function, the signal state is set to **SIG_IGN**. At the end, the signal is reconnected to the same function. In between there can be essentially arbitrary program text; however (on the PDP-11, for example), at this point the floating-point registers have not been saved.

A catch function will typically set a global variable that is inspected in the main program at a suitable point. As an alternative, we can jump back into the main program:

```
#include <setjmp.h>
#include <signal.h>

static jmp_buf reset;

static  trap( )
{
        . . .
        longjmp(reset, 1);
}

main( )
{
        . . .
        setjmp(reset);
        signal(SIGINT, trap);
        . . .
}
```

Setjmp(3) arranges that a subsequent call of **longjmp( )** has a target point: the program then will return a second time from the call to **setjmp( )**. The value returned is 0 the first time and the value of the second argument to **longjmp( )** thereafter. Once the program returns to the main routine, a major command loop can be restarted and so on. This is the customary method for error and interrupt handling in programs such as **ed**.

A process can die or enter a signal-catching function only if it is ready for execution. If a process is swapped, it must be brought back into main memory before it can die! This is especially true while executing code in a device driver; a process can be blocked by the driver so that it does not become ready to execute even when it receives a signal—such processes can usually be eliminated only by a system restart. If you write device drivers, you should arrange for suitable mechanisms (**ioctl(2)**) in the event a magnetic tape transport or a floppy disk drive goes on the blink.

Take definitions in this section with a grain of salt: Berkeley Unix has more signals than, say, Unix Version 7, and there is even an additional signal state (**SIG_HOLD**) which eliminates the race condition in Unix Version 7. In an attempt to be generally applicable, our discussion is limited to Unix Version 7.

Process Group and Controlling Terminal

Starting Unix primarily means getting */etc/init* to execute as process 1. When the timesharing service is started, this process reads the file */etc/ttys* and generates one process for each terminal, which is marked appropriately in this file. Each new process is the first one in a new process group, being the first in a chain of descendants to open a connection to a terminal device. The terminal itself is the controlling terminal of the process group.

The controlling terminal is inherited during **fork(2)** and **exec(2)** even if there is no further file connection to it! The only way to obtain a process without a controlling terminal is to generate it via process 1 (in the file */etc/rc*, from which the timesharing service is started) and never to connect such a process to a terminal.

/bin/sh(wait)

Signals caused at a terminal (i.e., **SIGHUP**, **SIGINT**, and **SIGQUIT**) are always sent to all processes in a process group. An interactive shell (i.e., one connected to a terminal for standard input and output) has to defend itself against **SIGINT** and **SIGQUIT**, otherwise a terminal session would be quickly over.

Every shell ignores **SIGQUIT**. An interactive shell catches **SIGINT**, but the catch function does nothing. As a consequence, the shell command **wait** can be terminated

using the interrupt key: **wait(2)** is one of the system calls terminated early by a signal.

Because **SIGINT** and **SIGQUIT** are ignored by default in a background process, one can use **wait $!** rather than the much less efficient **ps** to discover if the last background process **$!** is already done; either **wait** does not happen (i.e., the command returns immediately), or the user interrupts the wait state using **interrupt** and knows that the background process is still active.

Depending on the Bourne shell implementation, **wait** does not necessarily accept a parameter. Without a parameter, the command applies to all background processes together.

SIGTERM is also ignored by an interactive shell. This implies that you can use **kill 0** in a dire emergency to eliminate all of your own processes, because the shell itself, and thus the terminal session, is not eliminated by **SIGTERM**.

/bin/sh(logout)

The C shell has a special **logout** command. For the Bourne shell, this command can be constructed by recording the process number of the login shell using the following commands in **$HOME/.profile**:

```
SHELLID=$$; export SHELLID
```

Logout then is the following shell script (e.g., in a file */usr/bin/logout*):

```
kill -16 $SHELLID
```

With **trap "echo logout" 0** in **$HOME/.profile**, we can also report that a terminal session is about to be concluded as a response to ^**D**.

/bin/nohup

A background process will ignore **SIGINT** and **SIGQUIT** but not **SIGHUP**. This can cause background processes to die as soon as the terminal is turned off following the end of a terminal session. If your terminal interface and driver operate in this fashion, you should issue such background commands with the prefix **nohup**, which will insulate the command against a **SIGHUP** signal. Also, your successor at the terminal will be happier, because standard and diagnostic output of your command will be redirected to a file **nohup.out** in your directory and not clutter up the screen.

/bin/kill

The **kill** command, Berkeley style, is used to send a signal to one or more processes. Usually, **SIGTERM** is sent, but a signal number may be specified explicitly. As an example of the **kill(2)** system call, we show a version of **kill** where the signal number may be specified mnemonically. The following Bourne shell script could be used:

```
no=
case $1 in
-[Hh][Uu][Pp])
        no=-1 shift;;
        ...
-*)
        no=$1 shift;;
esac
kill $no $*
```

If we move */bin/kill*, for example, to */usr/bin/kill* and install this script as */bin/kill* . . . well, your system will recursively get a severe headache as soon as the next **kill** is issued. Your command search path **PATH**, which can be displayed using **echo $PATH**, normally consists of the active directory (i.e., of nothing) followed by */bin* and then */usr/bin*; **kill** in the last line of the shell script presumably will be found in */bin* (i.e., will be the shell script just issuing the command!). To avoid recursion, the last line must reference the moved original **kill** command.

By the way, a shell script always should fully qualify system commands or set **PATH** so that private commands cannot influence public shell scripts.

The C shell executes **kill** itself; the Bourne shell, however, does not. In an emergency, a user may have too many processes and no way to kill them from within the Bourne shell. The C program in Listing One (page 37) has the same capabilities as the shell script above. However, if the program is called as **exec ekill** . . . , it does not need a separate process table entry, and it will return with a new copy of the Bourne shell. Called without an argument, the program will display a list of signal names.

Observe that **exit( )** must be explicitly called at the end of **main( )**; in this program, we define **exit( )** so that either the current process calls up a Bourne shell using **exec(2)** or the process itself is terminated using **_exit( )**. By the way, **exit( )** cannot be defined local to the program (i.e., using static), otherwise library functions called by the program could still drag in the official version of **exit( )**.

We have used two functions that may be useful in other applications:

```
#include <stdio.h>

char *strsave(s)        /* save string dynamically */
register char *s;
{
        register char *save = calloc(strlen(s) + 1,
            sizeof(char));

        if (save)
                return strcpy(save, s);
        perror("strsave"), exit(1);
}
```

Although mentioned in the C book by Kernighan and Ritchie, **strsave( )** unfortunately has not made its way into the C library. This version assumes that **strcpy( )** delivers its first argument as a result—which is claimed by **string(3)** but unfortunately not by the library that **lint** on our system uses for checking library calls . . . .

The following function converts its **string(!)** argument to upper case:

```
#include <ctype.h>

char *stoupper(s)       /* string to UPPER CASE */
register char *s;
{
        register char *cp;

        for (cp = s; *cp; ++cp)
                if (islower(*cp))
                        *cp = toupper(*cp);
        return s;
}
```

Note that (at least in some implementations) **toupper( )** will deliver upper case even if the argument is not a lower-case letter! Actually, this is quite reasonable. If we already know the character to be a lower-case letter, we need not call **islower( )**; this will save only a few microseconds if the call is not made implicitly by **toupper( )**.

/etc/init

Controlling Multi-user Service

Process number 1, **init**, in some implementations of Unix Version 7 reacts to a signal and reads the file */etc/ttys* again. A terminal can be included in multi-user operations if it is mentioned in this file on a line starting with 1. If the line starts with 0, the terminal is excluded.

If **init** accepts a suitable signal, we can control multi-user service dynamically by changing */etc/ttys* appropriately and then issuing the signal. This happens in the program in Listing Two (page 38), installed as **attach** and **detach** in Perkin Elmer's Edition VII; **attach** arranges for the terminals mentioned as arguments to participate in multi-user operations and **detach** drops the terminals and eliminates the associated processes.

Use these commands with some caution; if signals are sent to **init** in rapid succession, process 1 might be eliminated and with it essentially the entire system! The program therefore must first modify the file for all arguments and then send a single signal. Because reading the file is the more expensive operation, we have elected to compare each line of the file with all arguments to the command.

The terminal name appears at the end of a line in the file. We are willing to select a terminal based on the last few letters of its name. Of course, the argument **e** will then cause console *as well as* **ttye** to be attached or detached . . . .

The program itself is quite simple: we copy */etc/ttys* into a temporary file and change the relevant lines. For efficiency, we compare each line with all arguments before writing it to the output file. Because **attach** is less destructive than **detach**, the program really must be called as **detach** for a terminal to be disconnected.

If at least one line was changed, we replace */etc/ttys* by the copy and inform **init**. Normally only the super-user can change */etc* (!), therefore only the super user can create the copy. Thus renaming should be possible:

```
#include <stdio.h>

rename(new, old)            /* rename a file */
char *new, *old;
{
        unlink(new);
        if (link(old, new) == -1)
        {
                fputs(old, stderr);
                fputs(", ", stderr);
                perror(new), exit(1);
        }
        if (unlink(old) == -1)
                perror(old), exit(1);
}
```

Blocking Signals

While */etc/ttys* is being replaced, we would prefer to be uninterrupted because we might otherwise find our system without a list of user terminals during the next bootstrap. Not even the super-user can prevent **SIGKILL**, but all other signals can be blocked with the following **disable( )** function; a subsequent call to **enable( )** will later restore the original situation:

```
#include <signal.h>

static int (*sigs[NSIG-1])( );


disable( )
{
        register int i;

        for (i = 1; i < NSIG; ++ i)
                if (i != SIGKILL)
                        sigs[i-1] = signal(i, SIG_IGN);
}


enable( )
{
        register int i;

        for (i = 1; i < NSIG; ++ i)
                if (i != SIGKILL)
                        signal(i, sigs[i-1]);
}
```

/lib/libc(abort.0)

At least according to chapter 3 of the manual, **abort( )** executes the PDP-11 **iot** instruction and thus normally will terminate the process with a core dump.

Beginning with Unix Version 7, one can send signals to oneself. A portable way to terminate a process with a core dump is the following:

```
#include <signal.h>
```

```
abort( )
{
        signal(SIGQUIT, SIG_DFL);
        kill(getpid( ), SIGQUIT);
}
```

/bin/stty

Depending on context, you may need to perform various cleanup tasks at various points in a program: somewhere there exists a temporary file, elsewhere a terminal echo may have been turned off, elsewhere yet a file may have the wrong protection keys, and in well-founded cases a second or third process might be on the prowl. It is relatively easy to deal with such cases, one at a time, as shown in Listing Three (page 39).

Here **echo( )** takes care of the necessary cleanup operations. If the routine was called through a signal, the same signal is subsequently sent back to the process itself.

In the general case, we should be able to stack the repairing functions by using the following functions; these mostly combine the ideas of **disable( )** and **abort( )**:

int catchall(f) int (*f)( );
declares, similar to **signal(2)**, **f** as a reaction to all signals except **SIGKILL**.

int catch(f) int (*f)( );
does the same, but only for those signals that are not ignored at present.

int recatch(i) int i;
calls the aforementioned function **f( )**, with the number of the signal effecting the call as an argument—at the beginning of this function, the signal itself has been set to **SIG_DFL**, and **recatch(i)** arranges for the signal i to be set just as it was set with the last call of **catchall( )** or **catch( )**.

int uncatch( )
arranges for all signals the setting that was in effect before the last call to **catchall( )** or **catch( )**.

The functions return 0 on success and −1 on failure.

We can use these functions to stack *signal* reactions. If we want to execute only the latest reaction, we conclude a trap function using **recatch( )**. If all reactions to the signal should be executed, we specify **uncatch( )** at the end of each trap function and send ourselves the signal again. At the beginning of a trap function, the signal should be set to **SIG_IGN** in either case.

The functions themselves are not very complicated. Essentially, they dynamically manage a stack on which the old settings of the signals are stored, as obtained from **signal(2)**. See Listing Four (page 40).

catch( ) contains the typical statements that should always be used if **SIGINT** or **SIGQUIT** should be caught. The first call to **signal( )** reports whether the signal is currently being ignored; in this case, the setting of the signal has not yet been changed. Only if the signal is not being ignored is it connected to the desired new reaction. These statements prevent a background process (for which these signals are ignored by default) from being subject to these signals again.

/usr/games

What does the program in Listing Five (page 40) do? Due to a complete absence of **goto** statements, this must be a so-called structured program. According to an often published opinion, however, such programs should be essentially immediately intelligible. Hint: Even if you do everything else as a super-user, don't do so here. Otherwise your system will self-destruct, and this columnist shall disavow any knowledge of this program.

Notes

[1] **Signal(2)** here denotes the function **signal( )**, explained in chapter 2 in the *Unix Programmer's Manual*.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 191.**

Unix Exchange (Text begins on page 30)

Listing One

```
#include <stdio.h>
#include <signal.h>
#include <ctype.h>

#define eq(a,b) (strcmp((a), (b)) == 0)

static char *signals[ ] = {
    "HUP", "INT", "QUIT", "ILL", "TRAP",
    "IOT", "EMT", "FPE", "KILL", "BUS",
    "SEGV", "SYS", "PIPE", "ALRM", "TERM",
    0 };
static char eflag;    /* argv[0][0] */

exit(code)            /* terminate or new shell */
int code;
{
    if (eflag == 'e'
        && isatty(fileno(stdin))
        && isatty(fileno(stdout)))
            execl("/bin/sh", "sh", "-i", 0);
    _exit(code);
}

static int signum(name)          /* number of a signal */
char *name;
{
    register char **sig = signals,
            *ucase = stoupper(strsave(name));

    while (*sig)
            if (eq(ucase, *sig++))
            {
                cfree(ucase);
                return sig - signals;
            }
    return -1;
}
```

(Continued on next page)

Unix Exchange

Listing One

(Listing Continued, text begins on page 30)

```
main(argc, argv)
 int argc;
 register char **argv;
{
    register int sig = SIGTERM;

    eflag = **argv;
    if (*++argv && **argv == '-')
    {
            if (isdigit(*++*argv))
                sig = atoi(*argv);
            else
                sig = signum(*argv);
            if (sig <= 0 || sig >= NSIG)
            {
                fputs(*argv, stderr);
                fputs(": is not a signal\n", stderr);
                exit(1);
            }
            ++argv;
    }
    if (*argv)
            do
                if (isdigit(**argv)
                    && kill(atoi(*argv), sig) == -1)
                        perror(*argv);
            while (*++argv);
    else
            for (argv = signals; *argv; ++ argv)
                fprintf(stderr, "%d\t%s\n",
                        argv - signals + 1, *argv);
    exit(0);
}
```

End Listing One

Listing Two

```
#include <stdio.h>

#define LEN     20                          /* max. line length */
#define TTYS    "/etc/ttys"                 /* terminal table */
#define TMP     "/etc/ttys.tmpXXXXXX"       /* temporary copy */
#define TELL    kill(1, 2) == -1 && (perror("init"), exit(1))

#define eq(a,b) (strcmp((a), (b)) == 0)

main(argc, argv)
int argc;
char **argv;
{
    register char **argp;
    char buf[LEN], *last,
        *tmp = mktemp(TMP),
        set = **argv == 'd'? '0': '1';
    int change = 0;

    if (! freopen(TTYS, "r", stdin))
        perror(TTYS), exit(1);
    if (! freopen(tmp, "w", stdout))
        perror(tmp), exit(1);
    while (fgets(buf, sizeof buf, stdin))
    {
        last = buf + strlen(buf) - 1;
        if (*last != '\n')
        {
            fputs(buf, stderr);
            fputs(": too long\n", stderr);
            exit(1);
        }
        *last = '\0';
        for (argp = argv+1; *argp; ++ argp)
            if (**argp
                && eq(last - strlen(*argp), *argp))
                    break;
        if (*argp && buf[0] != set)
        {
            ++ change;
            buf[0] = set;
        }
        puts(buf);
    }
    if (change)
    {
        fflush(stdout);
        rename(TTYS, tmp);
        TELL;
    }
    else
        unlink(tmp);
}
```

End Listing Two

Listing Three

```
#include <signal.h>
#include <sgtty.h>
#include <stdio.h>

static struct sgttyb sgttyb;

echo(i)
register int i;
{
    stty(fileno(stdout), &sgttyb);
    if (i)
        kill(getpid( ), i);
}

main( )
{
    int flags;

    gtty(fileno(stdout), &sgttyb);
    flags = sgttyb.sg_flags, sgttyb.sg_flags &= ~ECHO;
    stty(fileno(stdout), &sgttyb);
    sgttyb.sg_flags = flags;

    . . .
    signal(SIGINT, echo);
    . . .
    echo(0);
    signal(SIGINT, SIG_DFL);
    . . .
}
```

End Listing Three

(Listing Four begins on next page)

 (Listing Continued, text begins on page 30)

Listing Four

```
#include <signal.h>

static struct signals {
    int (*s_sigs[NSIG-1])( );
    struct signals *s_prev;
    } *top;

static struct signals *push( )
{
    register struct signals *p = (struct signals *)
        calloc(1, sizeof(struct signals));

    if (p)
        p→s_prev = top, top = p;
    return p;
}

int catchall(f)
register int (*f)( );
{
    register int i;
    register struct signals *p = push( );

    if (p)
    {
        for (i = 1; i < NSIG; ++ i)
                if (i != SIGKILL)
                p→s_sigs[i-1] = signal(i, f);
        return 0;
    }
    return -1;
}

int catch(f)
register int (*f)( );
{
    register int i;
    register struct signals * p = push( );

    if (p)
    {
        for (i = 1; i < NSIG; ++ i)
            if (i != SIGKILL
                && (p→s_sigs[i-1] = signal(i, SIG_IGN))
                    != SIG_IGN)
                signal(i, f);
        return 0;
    }
    return -1;
}

int uncatch( )
{
    register int i;
    register struct signals *p = top;

    if (p)
    {
        for (i = 1; i < NSIG; ++ i)
            if (i != SIGKILL)
                signal(i, p→s_sigs[i-1]);
        top = top→s_prev;
        cfree(p);
        return 0;
    }
    return -1;
}

int recatch(i)
register int i;
{
    if (top)
    {
        signal(i, top→s_sigs[i-1]);
        return 0;
    }
    return -1;
}
```

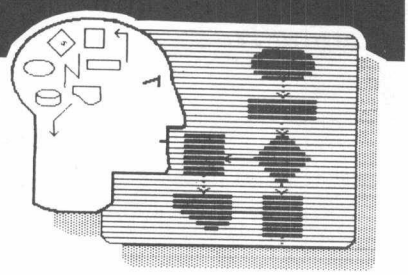
End Listing Four

Listing Five

```
#include <signal.h>

catch( ) {}

main( )
{
    int vater, ich, sohn;

    signal(16, catch);
    for (vater = 0; ich = getpid( ); vater = ich)
        switch (sohn = fork( )) {
        default:
            pause( );
        case -1:
            signal(SIGALRM, catch);
            alarm(2);
            getchar( );
            alarm(0);
            if (vater)
            {
                signal(16, catch);
                kill(vater, 16);
                pause( );
            }
            kill(sohn, 16);
            wait(0);
            exit(0);
        case 0:
            ;
        }
}
```

End Listings

THE SOFTWARE DESIGNER



by Michael Swaine

All code is built on algorithms, and all algorithms have to be communicated in code, even if only in pseudocode. So how does this month's *DDJ* more than any other month's warrant description as an algorithms issue? Well, we're presenting two articles that emphasize perennial algorithmic topics in programming: sorting and searching. Each of these articles takes a somewhat uncommon approach, and presents a program that is algorithmically distinctive. We hope they're more than distinctive, of course: we want to present algorithms that are of practical use to advanced programmers, so we've encouraged the authors to suggest applications in which their approaches might represent the algorithms of choice.

Sorting and searching algorithms are probably as thoroughly analyzed as any programming techniques. What follows here is not a definitive survey of sorting and searching techniques, but a summary of the common algorithms with the generally accepted judgements on their practical uses. None of this is new material, but it may set the stage for this month's sorting and searching algorithms.

Speed is the first performance measure we think of applying to a sort routine. But what speed? Worst case, best case, or "typical" case? And what's typical? It's known that any algorithm that sorts by comparing items must, for some sequence of **n** items, use **O(n log n)** comparisons; that is, the lower bound on the worst case is **order-(n log n)** time; but some algorithms have much worse worst-case times than this and still perform acceptably in most real-life situations. And speed isn't everything; it isn't even particularly important in some applications.

Sometimes a simple sort algorithm is the best. If you're sorting fewer than 100 items, if the sort routine will only be used once or twice and discarded, or if the data to be sorted are likely to be already nearly sorted, you probably would do just as well to implement a simple selection sort. The logic of this algorithm is lucid: find the smallest element, put it at the top of the list, and iterate down the list.

```
algorithm Selectionsort;
begin
for i:=1 to n do
   begin
   min:=i;
   for j:=i+1 to n do
      if a[j]<a[min] then min:=j;
   swap(a[i],a[min]);
   end;
end;
```

Simple algorithms are simple to debug and make it easier to convince yourself of their correctness. According to Robert Sedgewick, this simple selection sort algorithm is the technique of choice if you are sorting files with large records and small keys and actually have to perform the rearrangement, rather than just juggling indices. And it may be the technique of choice if you have to write a sort cold, have it running as quickly as possible, and then discard it. But this is an **O(n^2)** algorithm; many situations call for something more efficient.

Shellsort is a more efficient algorithm, and, although it's well-known, it is also insufficiently analyzed. Its overall efficiency has yet to be characterized. It may not be as efficient as, say, Quicksort, but it can be perfectly adequate for many applications. Its efficiency depends on the selection of a set of values that control the granularity of its early passes through the file. In the degenerate case with granularity **1** (**k=1**), the core of the algorithm, presented here, moves gradually down the list, keeping the upper portion of the list, above the item under consideration, sorted. That's the core, and it is a sort algorithm unto itself, but the full Shellsort does more. By beginning with larger values for **k** and working down to **k=1**, Shellsort in effect removes some of the large-scale disorder from the file, giving the **k=1** pass better data to work with. That turns out to be a good thing, since the core algorithm, hence the **k=1** pass, is especially sensitive to order; its worst-case performance is quadratic, its best, for a completely ordered file, linear.

Shellsort is probably the simplest algorithm that represents a common technique in sort algorithms: shifting strategies as the data becomes more organized. In the case of Shellsort the shift is accomplished by adjusting one parameter, but in some variations on Quicksort quite different algorithms are employed at different stages in the sorting.

```
algorithm Shellcore;
begin
for i:=2 to n do
   begin
   j:=i;
   while a[j-k]>a[i] do
      begin
      a[j]:=a[j-k];
      j:=j-k;
      end;
   a[j]:=a[i];
   end;
end;
```

Quicksort was invented by C. A. R. Hoare in 1960. It's been widely used,

deeply analyzed, and much tweaked since then. It has a worst-case **O(n^2)** performance, but has **O(n log n)** expected time under the assumption that all permutations of the to-be-sorted elements are equally likely. Quicksort is fast and relatively memory efficient, and is often used in library routines.

Quicksort is defined here as a recursive procedure. Its arguments, **left** and **right**, define the endpoints of the sort. **Middle** is a point selected by procedure **Partition**. **Partition** is any procedure that puts the middle-th element (or identical elements) in its (their) proper place and ensures that all elements to the left of it (them) are less than it (them), and all elements to its (their) right are greater. **M** is the number of identical elements that **Partition** puts in their proper place.

```
algorithm Quicksort(left,right);
begin
if right>left then
   begin
   Partition(left,middle,right);
   Quicksort(left, middle -1);
   Quicksort(middle+m,right);
   end;
end;
```

Various methods have been employed to improve Quicksort's performance, such as R. S. Scowen's Quickersort and M. H. van Emden's Qsort (algorithms 271 and 402 in the *Collected Algorithms of the ACM*, respectively). Unwinding the recursion is one technique that can increase the speed. Another method involves abandoning the algorithm for something like Shellcore for very small subfiles. A third technique helps to avoid worst-case times: it involves using some sort of randomizing technique in **Partition** to select the middle point.

Heapsort is an efficient sort algorithm that uses a data structure that has applications in queuing. One of its virtues is that it has **O(n log n)** as a worst-case time. The algorithm works by constructing a semiordered tree structure called a heap, sawing off the root (which will be the largest element remaining), rebuilding the resulting forest into another heap, and iterating. Procedure **Heapprop** rearranges items to ensure that the heap property applies. **Makeheap** constructs the initial heap.

```
algorithm Heapsort;
begin
Makeheap;
for i:=n to 2 step -1 do
   begin
   swap(a[1],a[i]);
   Heapprop(i-1);
   end;
end;
```

Bubblesort is a curious phenomenon. I learned it early in my programming experience. You probably did, too. Does anyone know why? It is neither very simple nor very efficient, and its detailed operation is less clear than that of other sorts, although its operation is not hard to describe. Essentially, it consists of a number of passes through the file, swapping adjacent elements as necessary.

```
algorithm Bubblesort;
begin
repeat
   t:=a[1];
   for j:=2 to n do
      if a[j-1]>a[j] then
         begin
            t:=a[j-1];
            swap(a[j],a[j-1]);
         end;
until t=a[1];
end;
```

How you sort depends on what you have to sort, including how it is already arranged as well as other aspects of the data. Another criterion you might apply to a sort algorithm is stability. If you intend to sort on more than one key and not have the sorting on one key destroy the sorting on the other, you need a stable sort. There are instances in which this is not necessary, and there is much power in knowing what you can get along without. In this issue (page 68) we present a sort algorithm that has some speed advantage when stability is not a consideration.

Knowing that the data is not randomly arranged can also have a marked effect on the expected performance of the algorithms. One of the most common sorting situations involves adding a few items to an already sorted file; here the more sophisticated algorithms can't compete with such simple algorithms as Shellcore. Finally, in sorting files too large for in-core methods, tape or disk access speed limitations can override all other considerations in searching for a fast algorithm. The basic algorithm for external sorts is: sort (relatively) small chunks of the file in memory, then merge these sorted chunks. Much of the analysis of external sort algorithms has to do with minimizing trips to the external well. In a virtual memory system, Sedgewick argues, it is reasonable for certain in-core algorithms (like Quicksort) to ignore the problem of external reaches entirely, since the sort's minimization criteria coincide with those of the designer of the virtual memory system.

Searching algorithms encompass a large chunk of computer science. Parsing, for example, is built on pattern matching, which is searching. Efficient implementation of a database hinges on good search algorithms. Move-search routines in chess programs implement some very complex strategies for finding things. Search techniques depend intimately on the data structure being searched, and many search algorithms are fundamentally more complex than the sort algorithms presented here. Search algorithms are harder to specify in a few lines.

The simplest search algorithm is a sequential scan of the file, and it has an advantage in addition to its simplicity: it does not upset the organization of the file. Generally, though, speed is a consideration, and an algorithm like a binary search is warranted. Binary search splits the (sorted) file more or less in half at each step, restricting its subsequent search to the appropriate half of the remaining file. This cuts search time from a linear function of the number of items in the file to a logarithmic function. Binary search can proceed implicitly on a linearly-organized file or can involve explicit construction of a binary tree. Other search methods involve

(Continued on page 67)

Parallel Pattern Matching and Fgrep

by Ian Ashdown

The file-search utility Fgrep is not as flexible as Grep, but its parallel pattern-matching algorithms allow for one-pass processing.

Preparing an index to a large technical book; finding all references to one person in several years' worth of minutes of meetings; searching for all occurrences of a specific sequence of events in the data obtained from a scientific experiment: all of these problems and many more are examples of searching for patterns in a set of data. The question is: What is the most efficient search algorithm to use?

For single patterns, such as searching for the phrase "binary tree" in a text file, two algorithms are of particular interest: the Boyer-Moore algorithm[1] and the Knuth-Morris-Pratt algorithm.[2] Both are fast and reasonably simple to implement. However, neither is appropriate when more than one pattern at a time must be searched for.

An algorithm that is appropriate for multiple patterns appeared in a paper published in the June 1975 issue of *Communications of the ACM*.[3] Entitled "Efficient String Matching: An Aid to Bibliographic Search" and written by Alfred Aho and Margaret Corasick of Bell Laboratories, the paper presents "a simple and efficient algorithm to locate all occurrences of any of a finite number of keywords in a string of text." Without modification to the algorithm, "keyword" can be taken to mean any pattern and "a string of text" to mean any sequence of symbols, be it ASCII text, digitized data obtained from a video camera, or whatever.

The algorithm has some interesting features. For instance, most pattern-matching schemes must employ some form of backtracking, or rescanning of the string, when an attempted match to a pattern fails or multiple patterns are to be matched. The Aho-Corasick algorithm differs in that it searches for all of the patterns in parallel. By doing so, it can process the string in one pass.

The algorithm also recognizes patterns that overlap in the string. For example, if the patterns are "he," "she," and "hers," the algorithm will correctly identify matches to all three in the string "ushers."

As for speed of execution, it is independent of the number of patterns to be matched! This surprising feature is a direct result of searching for the patterns in parallel.

Curiously, this algorithm has all but disappeared from the literature of computer science. Of the hundreds of textbooks and articles written since that time that discuss pattern matching, only a few refer to the paper, and none that I am aware of present the Aho-Corasick algorithm itself.

On the other hand, if you work with Unix you may have used a utility based on the algorithm: Fgrep. This useful tool searches for and displays all occurrences of a set of keywords and phrases in one or more text files. Although it does not accept wildcard characters in its patterns like Unix's more flexible grep utility, fgrep does illustrate the speed of the Aho-Corasick algorithm. Typically, fgrep is five to ten times faster than grep in searching files for fixed string patterns.

Ian Ashdown, byHeart Software, 1089 W. 21st St., North Vancouver, British Columbia V7P 2C6 Canada

Given the algorithm's general usefulness, I thought it appropriate to reintroduce Aho and Corasick's work in this article. At the same time, it seemed a shame that fgrep has been restricted to Unix. Therefore, the source code in C for a full implementation of fgrep[4] has been included (see the listing, page 54). This serves not only to demonstrate the algorithm but also to bring the idea of a public domain Unix one small step closer to reality.

Inside the Machine

The Aho-Corasick algorithm consists of two phases: building a finite state automaton (FSA) then running a string of data through it, with each consecutive symbol considered a separate input. Before we analyze these phases, a quick summary of what finite state automata are and how they work is in order. (For a more detailed discussion, see Aho and Ullman's book on compiler design.[5])

In essence, an FSA is a conceptual machine that can be in any one of a finite number of "states." It also has a set of "state transition" rules and a set of "inputs," which together with the set of states define what inputs cause the machine to change from one state to another. Finally, because a machine serves no purpose without producing some output, an FSA has one or more "terminal" states. Output is produced only when the FSA enters such states.

A physical example of an FSA could be a light switch. This device has two states, on and off. Its inputs consist of someone pushing the switch up or down. On is a terminal state, where the associated lamp produces visible radiation. The state transition rules can be presented in a simple truth table:

	Off	On
Up	On	–
Down	–	Off

Alternatively, the rules could be shown as a state transition diagram (Figure 1, page 47) where no state transition on a particular input (as shown in the truth table) is

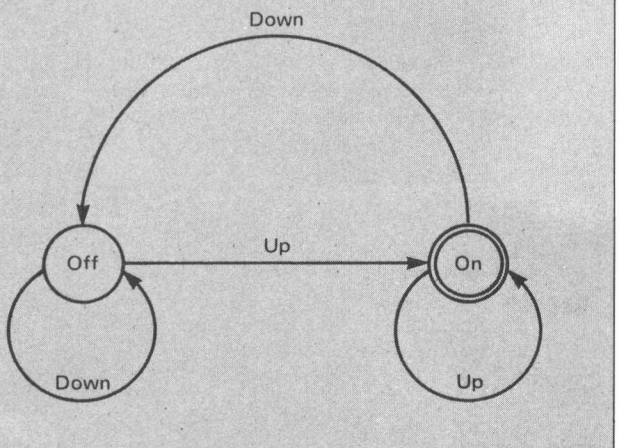


Figure 1

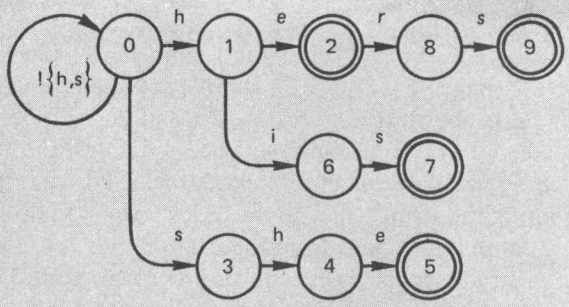


Figure 2a
Goto function

Current State:	1	2	3	4	5	6	7	8	9
Failure State:	0	0	0	1	2	0	3	0	3

Figure 2b
Failure function

Current State:	Output:
0	--
1	--
2	{he}
3	--
4	--
5	{she, he}
6	--
7	{his}
8	--
9	{hers}

Figure 2c
Output function

Current State:	Input Symbol:	Next State:	Current State:	Input Symbol:	Next State:
0	h	1	4	e	5
	s	3		i	6
	.	0		h	1
1	e	2		s	3
	i	6		.	0
	h	1	6	s	7
	s	3		h	1
	.	0		.	0
2,5	r	8	8	s	9
	h	1		h	1
	s	3		.	0
	.	0			
3,7,9	h	4			
	s	3			
	.	0			

where "." represents any symbol not included in the list of symbols for the indicated state(s).

Figure 2d
Move function

equivalent to a transition from a state to itself (as shown in the diagram).

Getting ahead of ourselves for a moment, let's look at Figure 2 (page 47). Figure 2a shows part of an FSA (the *other parts* shown in Figures 2b, 2c, and 2d, will be explained shortly). By having sequences of states, the FSA can recognize patterns in an input stream. For example, presenting the string "she" to the FSA shown in Figure 2a would cause it to change from State 0 to States 3, 4, and 5 as each input symbol of the string is processed. State 5 is a terminal state that indicates that the pattern "she" was recognized.

Two types of FSA can be built, one nondeterministic (Algorithm 1, page 48) and the other deterministic (Algorithm 2, page 48). The nondeterministic one (NFSA) uses functions called go_to, failure and output, while the deterministic FSA (DFSA) uses move and output. The difference between the two is that the NFSA can make one or more state transitions per input symbol, but the DFSA makes only one. With fewer transitions to make, a DFSA can process its input in less time than an equivalent NFSA.

The go_to function accepts as input parameters the current state of the NFSA and the current input symbol, and it returns either a state number (the go_to state transition), if the input symbol matches that expected by the patterns being matched, or else a unique symbol called FAIL. The only exception to this is State 0: go_to(0,X) returns a state number for all input symbols X. If symbol X does not match the beginning symbol of any of the patterns, the state number returned is 0.

The failure function is called whenever the go_to function returns FAIL. Accepting the current state as its input parameter, it always returns a state number, the failure state transition.

For the DFSA, the move function accepts the current state number and input symbol and always returns the next state number. There are no failure state transitions in deterministic FSAs.

Both the NFSA and the DFSA use the output function. As defined by Aho and Corasick, this function prints the patterns as they are matched by the FSA. However, the definition of this function can be much more general. In fact, the FSA can be implemented in such a way that it executes any arbitrary set of procedures whenever it recognizes a pattern.

FSAs used by the Aho-Corasick algorithm can be either nondeterministic or deterministic. Although a DFSA executes more quickly than its equivalent NFSA, it also requires more memory to encode its state transition tables. Which type you choose for the algorithm depends upon the application and the constraints of execution speed and memory usage.

As an example, assume a set of patterns to be matched, "he," "she," "his," and "hers," with ASCII characters as the set of input symbols. (These have been purloined from Aho and Corasick.) The FSAs needed to recognize these patterns are shown in Figure 2. Let's look at the NFSA first. Taking as input the string "ushers," the machine is run using Algorithm 1. Starting in State 0, the first symbol from the string is "u." Because only symbols "h" and "s" lead from State 0 to other states, go_to(0,u) returns 0, and the NFSA remains in State 0.

The next symbol from the string is "s." As shown in Figure 2a, the NFSA makes a go_to state transition to State 3. The next symbol ("h") causes a go_to transition to State 4, and the next ("e") to State 5. An output is defined for State 5 (Figure 2c), and the NFSA, having recognized two pattern matches, prints "she,he."

The next character is "r." No go_to transitions are defined for State 5, so go_to(5,r) returns FAIL. This causes failure(5) to return 2 (from Figure 2b), making the NFSA perform a failure transition to State 2. From here, Algorithm 1 executes go_to(2,r), which causes the NFSA to enter State 8.

Finally, the last input symbol, "s," leads the NFSA to enter terminal State 9, and the output defined for this state causes "hers" to be printed. In all, Algorithm 1 recognized the patterns "he," "she," and "hers" in the string "ushers." The state transitions made can be summarized as:

Input Symbol:	u	s	h	e		r	s	
Current State:	0	0	3	4	5	2	8	9

```
Input:   A string of symbols and a pattern-matching machine
         consisting of functions go_to, failure and output.
Output:  Matched patterns.

begin
  state = 0
  while there are more symbols in the string do
    begin
      a = next symbol in the string
      while go_to(state,a) = = FAIL do
        state = failure(state)
      state = go_to(state,a)
      if output(state) != NULL then
        print output(state)
    end
end
```

Algorithm 1
Nondeterministic Pattern-Matching Machine

```
Input:   A string of symbols and a pattern-matching machine
         consisting of functions move and output.
Output:  Matched patterns.

begin
  state = 0
  while there are more symbols in the string do
    begin
      a = next symbol in the string
      state = move(state,a)
      if output(state) != NULL then
        print output(state)
    end
end
```

Algorithm 2
Deterministic Pattern-Matching Machine

Now let's look at the DFSA. Using the same input string, "ushers," this machine makes move state transitions in accordance with Algorithm 2 and Figure 2d. Its state transitions can be summarized as:

Input Symbol:	u s h e r s
Current State:	0 0 3 4 5 8 9

The only difference is that the failure transition to State 2 was skipped. By avoiding any failure transitions, Algorithm 2 can recognize a pattern in fewer state transitions and hence less time than Algorithm 1.

Software Construction

Having seen how FSAs work, we will now look at how they are built, starting with the computation of the go_to function by Algorithm 3 (page 49).

The go_to function is initially defined to return FAIL for every input symbol and every state. Each pattern is run through procedure "enter," which tries to match the pattern, symbol by symbol to the existing partially constructed go_to function. When a failure occurs, a new state is created and added to the go_to function, using the current symbol of the pattern as the input for the state transition. Thereafter, each consecutive symbol of the pattern causes a new state to be created and added.

When the last symbol of each pattern has been processed by procedure "enter," its associated state is made a terminal state. As shown in Algorithm 3, this means that the output for the state is defined as the current pattern, which Algorithm 1 will print when the NFSA is run. However, it is easy to see that you can rewrite the statement "output(state) = pattern" in Algorithm 3 to assign whatever procedures to output(state) you want.

Finally, after all of the patterns have been processed, go_to(0,X) is redefined to return 0 for all symbols X that still return FAIL.

When Algorithm 3 completes, it has computed the go_to function of the FSA and partially computed the output function. If you simulate Algorithm 3 by hand with "he," "she," "his," and "hers" as its patterns, you will see that the output for State 5 will be "she," not "she,he." It remains for Algorithm 4 (page 51) to complete the output function as it computes the failure function.

Let's define the depth of a state as the number of go_to state transitions that must be made from State 0 to reach it. For example, the depth of State 4 in Figure 2a is 2, while that of State 9 is 4. The failure state transition for all states of depth 1 is State 0. The algorithm used to compute the failure transitions for all states of depth greater than 1 can be expressed in its simplest form as:

```
for each depth D do
  for each state S of depth D-1 do
    for each input symbol A do
      if (T = go_to(S,A)) != FAIL then
        begin
          R = failure(S)
          while go_to(R,A) = = FAIL do
            R = failure(R)
          failure(T) = go_to(R,A)
        end
```

To demonstrate this using our example, let's compute one failure transition for a state of depth 2. The states of depth 1 are 1 and 3. The only input symbols for which the go_to function does not return FAIL are "e" and "i" for State 1 and "h" for State 3. Taking State 3 for "S" and symbol "h" for "A" above, we have "T" being 4, "R" being "failure(3)", which is 0, go_to(0,h) returning 1, and finally "failure(4)" being set to go_to(0,h). The failure transition for State 4 is thus State 1.

Algorithm 4 expands on the above algorithm in two ways. First, it uses a queue (a first-in, first-out list) to maintain the order of states by depth to be processed. Second, it accepts as input the output function and combines the outputs of the terminal states where appropriate. In our example, the output "he" of State 2 would be combined with the output "she" of State 5 to produce the final and correct "she,he" output for State 5.

The "move" function is computed from the go_to and failure functions by means of Algorithm 5 (page 51). Essentially, all this algorithm does is precompute all possible sequences of failure state transitions. It is similar to Algo-

```
Input:   Array of patterns {y[1],y[2] . . . . y[k]} where each
         pattern is a string (one-dimensional array) of symbols.
         (Function go_to(s,a) is defined to return FAIL for all
         states "s" and all input symbols "a" until defined
         otherwise by procedure enter.)
Output:  Function go_to and partially computed function output.

begin
  newstate = 0
  for i = 1 to i = k do
    enter(y[i])
  for each symbol a do
    if go_to(0,a) = = FAIL then
      go_to(0,a) = 0
end

procedure enter(pattern)          /* "pattern" is an array of */
begin                             /* "m" symbols */
  state = 0
  j = 1
  while go_to(state,pattern[j]) != FAIL do
    begin
      state = go_to(state,pattern[j])
      j = j + 1
    end
  for p = j to p = m do
    begin
      newstate = newstate + 1
      output(newstate) = NULL
      go_to(state,pattern[p]) = newstate
      state = newstate
    end
  output(state) = pattern
end
```

Algorithm 3
Computation of go_to Transitions

rithm 4. Because there is considerable redundancy between them, they can be merged to compute the failure and move functions concurrently, as is shown in Algorithm 6 (page 53).

Details, Details

So far, the functions go_to, failure, move, and output have been shown as something that you assign outputs to for specified inputs. This assumes a much higher-level programming language than most of today's offerings. Implementing these algorithms in C, Pascal, BASIC, or Fortran requires some extra code and work.

Aho and Corasick programmed their original version in Fortran. This is evident from their paper, where they suggest that the failure and output functions be implemented as one-dimensional arrays accessed by state numbers. With a language such as C that supports complex data structures, the code for the functions can be more elegantly written by replacing the state numbers and arrays with dynamically allocated structures. Each structure can have as members pointers to linked lists of go_to and move transitions, a pointer to the appropriate failure transition, and a pointer to a character string for the output function.

Aho and Corasick also note that the go_to and move functions could be implemented as two-dimensional arrays, with the number of states forming one dimension and the set of input symbols forming the other. However, this would require enormous amounts of memory for an ASCII character set and several hundred states. Their recommendation was that the go_to and move transitions for each state be implemented as linked linear lists or binary trees.

Because the FSA will usually spend most of its time in State 0 for large input symbol sets, it is advantageous to have the go_to or move functions execute as quickly as possible for this state. Therefore, as Aho and Corasick suggest, a one-dimensional array can be assigned to State 0. (Note that because no failure transitions are defined for State 0, its go_to and move transitions are one and the same.) The array is accessed directly, using the input symbol as the index, with the corresponding array entry being the state transition for that symbol.

An improvement not covered by Aho and Corasick can be made when you realize that several states often will have the same move transitions (see Figure 2d as an example). In such cases, the state structures can be assigned a common pointer to one linked list of move transitions.

Some applications may call for the algorithm to be coded with predefined patterns in read-only memory Here it is usually desirable to maximize the speed of execution while minimizing the code size. Aho and Ullman discuss a method of encoding the state transition tables that combines the compactness of linked lists with the speed of di-

```
Input:    Functions go_to and output from Algorithm 3.
Output:   Functions failure and output.

begin
  initialize queue
  for each symbol a do
    if (r = go_to(0,a)) != 0 then
      begin
        add r to tail of queue
        failure(r) = 0
      end
  while queue is not empty do
    begin
      s = head of queue
      remove head of queue
      for each symbol a do
        if (t = go_to(s,a)) != FAIL then
          begin
            add t to tail of queue
            r = failure(s)
            while go_to(r,a) == FAIL do
              r = failure(r)
            failure(t) = go_to(r,a)
            output(t) = output(t) + output(failure(t))
          end
  end
end
```

Algorithm 4
Computation of Failure Transitions

```
Input:    Function go_to from Algorithm 3 and function failure
          from Algorithm 4.
Output:   Function move.

begin
  initialize queue
  for each symbol a do
    begin
      move(0,a) = go_to(0,a)
      if (r = go_to(0,a)) != 0 then
        add r to tail of queue
    end
  while queue is not empty do
    begin
      s = head of queue
      remove head of queue
      for each symbol a do
        if (t = go_to(s,a)) != FAIL then
          begin
            add t to tail of queue
            move(s,a) = t
          end
        else
          move(s,a) = move(failure(s),a)
  end
end
```

Algorithm 5
Computation of Move Transitions

rect array access. Their method (which Unix's yacc compiler-compiler utility uses to encode its parsing tables) is *unfortunately* too involved to discuss here. Interested readers are referred to Aho and Ullman's book for details.

Putting It All To Work

Implementing the Aho-Corasick algorithm is fairly straightforward, although, as you can see, the code required to flesh it out to a full emulation of Unix's fgrep is somewhat involved. If you want to use the algorithm in other programs, simply remove the fgrep shell and add whatever interface routines you require.

As an information retrieval utility that searches arbitrary text files, fgrep is useful but by no means complete. One helpful extension would be the ability to specify Boolean operators (AND, OR, and XOR) for combinations of patterns.

Also, rather than simply print matched patterns for its output, the FSA can execute whatever procedures you choose to associate with the patterns as they are matched. From this you could create a macro processor, where the FSA accepts an input string, finds matches to predefined patterns, substitutes the corresponding macro expansions, and emits the result as an output string.

For those with RAM memory to spare (a megabyte or so), consider how fast a spelling checker that makes no disk accesses beyond initially loading itself could be made to run . . . fifty thousand or more words in RAM and your text file is checked almost as fast as it can be read.

If you come up with an original and fascinating use for the Aho-Corasick algorithm, or if you derive new utilities from fgrep, by all means let me know about it. Better yet, donate your code to a public domain software group. That alone would repay me for developing the code and writing this article.

```
Input:    Functions go_to and output from Algorithm 3.
Output:   Function move.
begin
  initialize queue
  for each symbol a do
    begin
      move(0,a) = go_to(0,a)
      if (r = go_to(0,a)) != 0 then
        begin
          add r to tail of queue
          failure(r) = 0
        end
    end
  while queue is not empty do
    begin
      s = head of queue
      remove head of queue
      for each symbol a do
        if (t = go_to(s,a)) != FAIL then
          begin
            add t to tail of queue
            r = failure(s)
            while go_to(r,a) = = FAIL do
              r = failure(r)
            failure(t) = go_to(r,a)
            output(t) = output(t) + output(failure(t))
            move(s,a) = t
          end
        else
          move(s,a) = move(failure(s),a)
  end
end
```

Algorithm 6
Merged Computation of Failure and Move Transitions

References

[1] Boyer, R. S., and J. S. Moore, "A Fast String Searching Algorithm," *CACM* 20:10 (October 1977), pp. 762 – 772.

[2] Knuth, D. E., J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Comp., 6:2* (June 1977), pp. 323–350.

[3] Aho, A. V., and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *CACM* 18:6 (June 1975), pp. 333–340.

[4] Bell Telephone Laboratories, *Unix Programmer's Manual*, Volume 1, Holt, Rinehart and Winston, 1983, pp. 70–71.

[5] Aho, A. V., and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, 1977, pp. 73–124.

[6] *See also* Aoe, J., Y. Yamamoto, and R. Shimada, "A Method for Improving String Pattern Matching Machines," *IEEE Transactions On Software Engineering, SE-10:1 (January 1984), pp. 116*–120.

DDJ

(Listing begins on next page)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**

Fgrep Listing (Text begins on page 46)

```
/* FGREP.C - Search File(s) For Fixed Pattern(s)
 *
 * Version 1.03       February 11th, 1985
 *
 * Modifications:
 *
 *   V1.00 (84/12/01)   - beta test release
 *   V1.01 (85/01/01)   - added -P option
 *                      - improved command line validation
 *   V1.02 (85/01/06)   - modified "run_fsa()" and "bd_move()"
 *   V1.03 (85/02/11)   - added -S option
 *
 * Copyright 1985:     Ian Ashdown
 *                     byHeart Software
 *                     1089 West 21st Street
 *                     North Vancouver, B.C. V7P 2C6
 *                     Canada
 *
 * This program may be copied for personal, non-commercial use
 * only, provided that the above copyright notice is included in
 * all copies of the source code. Copying for any other use
 * without previously obtaining the written permission of the
 * author is prohibited.
 *
 * Machine readable versions of this program may be purchased for
 * $35.00 (U.S.) from byHeart Software. Supported disk formats
 * are CP/M 8" SSSD and PC-DOS (v2.x) 5-1/4" DSDD.
 *
 * Notes:
 *
 * The algorithm used in this program constructs a deterministic
 * finite state automaton (FSA) for pattern matching from the sub-
 * strings, then uses the FSA to process the text string in one
 * pass. The time taken to construct the FSA is proportional to
 * the sum of the lengths of the the substrings. The number of
 * state transitions made by the FSA in processing the text
 * string is independent of the number of substrings.
 *
 * Algorithm Source:
 *
 * "Efficient String Matching: An Aid to Bibliographic Search"
 * Alfred V. Aho & Margaret J. Corasick
 * Communications of the ACM
 * pp. 333 - 340, Vol. 18 No. 6 (June '75)
 *
 * USAGE: fgrep [-vclnhyefxps] [strings] <files>
 *
 * where:
 *
 *      -v      All lines but those matching are printed.
 *      -c      Only a count of the matching lines is printed.
 *      -l      The names of the files with matching lines are
 *              listed (once), separated by newlines.
 *      -n      Each line is preceded by its line number in the
 *              file.
 *      -h      Do not print filename headers with output lines.
 *      -y      All characters in the file are mapped to upper
 *              case before matching. (This is the default if the
 *              string is given in the command line under CP/M,
 *              as CP/M maps everything on the command line to
 *              upper case. Use the -f option if you need both
 *              lower and upper case.) Not a true UNIX "fgrep"
 *              option (normally available under "grep" only),
 *              but too useful to leave out.
 *      -e      <string>. Same as a string argument, but useful
 *              when the string begins with a '-'.
 *      -f      <file>. The strings (separated by newlines) are
 *              taken from a file. If several strings are listed
 *              in the file, then a match is flagged if any of
 *              the strings are matched. If -f is given, any
 *              following argument on the command line is taken
 *              to be a filename.
 *      -x      Only lines matched in their entirety are printed.
 *      -p      Each matched line is preceded by the matching
 *              substring(s). Not a UNIX "fgrep" option, but too
 *              useful to leave out.
 *      -s      No output is produced, only status. Used when
 *              when "fgrep" is run as a process that returns a
 *              status value to its parent process. Under CP/M, a
```

(Continued on page 56)

```
 *              non-zero value returned by "exit()" may terminate
 *              a submit file that initiated the program,
 *              although this is implementation-dependent.
 *
 * DIAGNOSTICS:
 *
 * Exit status is 0 if any matches are found, 1 if none, 2 for
 * error condition.
 *
 * BUGS:
 *
 * The following UNIX-specific option is not supported:
 *
 *      -b      Each line is preceded by the block number in
 *              which it was found
 *
 * Lines are limited to 256 characters.
 *
 */

/*** Definitions ***/

#define TRUE              -1
#define FALSE              0
#define MAX_LINE         257  /* Maximum number of characters */
                              /* per line plus NULL delimiter */
#define CP/M                  /* Comment out for compilation */
                              /* under UNIX */

#define CMD_ERR    0    /* Error codes */
#define OPT_ERR    1
#define INP_ERR    2
#define STR_ERR    3
#define MEM_ERR    4

/*** Typedefs ***/

typedef int BOOL;       /* Boolean flag */

/*** Data Structures ***/

/* Queue element */

typedef struct queue
{
  struct state_el *st_ptr;
  struct queue *next_el;
} QUEUE;

/* Transition element */

typedef struct transition
{
  char lchar;                   /* Transition character */
  struct state_el *nextst_ptr;  /* Transition state pointer */
  struct transition *next_el;
} TRANSITION;

/* FSA state element */

typedef struct state_el
{
  TRANSITION *go_ls;    /* Pointer to head of "go" list */
  TRANSITION *mv_ls;    /* Pointer to head of "move" list */
  struct state_el *fail_state;  /* "failure" transition state */
  char *out_str;        /* Terminal state message (if any) */
} FSA;

/*** Global Variables and Structures ***/

/* Dummy "failure" state */


FSA FAIL_STATE;

/* Define a separate data structure for State 0 of the FSA to
 * speed processing of the input while the FSA is in that state.
 * Since the Aho-Corasick algorithm only defines "go" transitions
 * for this state (one for each valid input character) and no
 * "failure" transitions or output messages, only an array of
```

```
 * "go" transition state numbers is needed. The array is accessed
 * directly, using the input character as the index.
 */

FSA *MZ[128];   /* State 0 of FSA */

BOOL vflag = FALSE,     /* Command-line option flags */
     cflag = FALSE,
     lflag = FALSE,
     nflag = FALSE,
     hflag = FALSE,
     yflag = FALSE,
     eflag = FALSE,
     fflag = FALSE,
     xflag = FALSE,
     pflag = FALSE,
     sflag = FALSE;

/*** Include Files ***/

#include <stdio.h>
#include <ctype.h>

/*** Main Body of Program ***/

int main(argc,argv)
int argc;
char **argv;
{
  char *temp;
  BOOL match_flag = FALSE,
       proc_file();
  void bd_go(),
       bd_move(),
       error();

  /* Check for minimum number of command line arguments */

  if(argc < 2)
    error(CMD_ERR,NULL);

  /* Parse the command line for user-selected options */


  while(--argc && (*++argv)[0] == '-' && eflag == FALSE)
    for(temp = argv[0]+1; *temp != '\0'; temp++)
      switch(toupper(*temp))
      {
        case 'V':
          vflag = TRUE;
          break;
        case 'C':
          cflag = TRUE;
          break;
        case 'L':
          lflag = TRUE;
          break;
        case 'N':
          nflag = TRUE;
          break;
        case 'H':
          hflag = TRUE;
          break;
        case 'Y':
          yflag = TRUE;
          break;
        case 'E':
          eflag = TRUE;
          break;
        case 'F':
          fflag = TRUE;
          break;
        case 'X':
          xflag = TRUE;
          break;
        case 'P':
          pflag = TRUE;
          break;
        case 'S':
          sflag = TRUE;
          break;
        default:
          error(OPT_ERR,NULL);
```

(Continued on next page)

```
    }

/* "pflag" can only be TRUE if the following flags are FALSE */

if(vflag == TRUE || cflag == TRUE || lflag == TRUE ||
    xflag == TRUE || sflag == TRUE)
  pflag = FALSE;

/* Check for string (or string file) argument */

  if(!argc)
    error(CMD_ERR,NULL);

  /* Build the "go" transitions. */

  bd_go(*argv++);
  argc--;

  /* Build the "failure" and "move" transitions */

  bd_move();

  /* Process each of the input files if not "stdin". */

  if(argc < 2)
    hflag = TRUE;
  if(!argc)
  {
    if(proc_file(NULL,FALSE) == TRUE && match_flag == FALSE)
      match_flag = TRUE;
  }
  else
    while(argc--)
      if(proc_file(*argv++,TRUE) == TRUE && match_flag == FALSE)
        match_flag = TRUE;

  /* Return status to the parent process. Status is zero if any
   * matches are found, 1 if none. */

  if(match_flag == TRUE)
    exit(0);
  else
    exit(1);
}

/*** Functions and Procedures ***/

/* PROC_FILE() - Run the FSA on the input file "in_file". Returns
 *               TRUE if a match was found, FALSE otherwise.
 */

BOOL proc_file(in_file,prt_flag)
char *in_file;
BOOL prt_flag;
{
  char buffer[MAX_LINE],        /* Input string buffer */
       *nl,
       *index(),
       *stoupper(),
       *fgets();

  long line_cnt = 0L,   /* Line counter */
       mtch_cnt = 0L;   /* Matched line counter */
  BOOL mtch_flag,       /* Matched line flag */
       run_fsa();
  FILE *in_fd,
       *fopen();
  void error();

  if(in_file != NULL)   /* A file was specified as the input */
  {
    if(!(in_fd = fopen(in_file,"r")))
      error(INP_ERR,in_file);
  }
  else
    in_fd = stdin;
```

(Continued on page 60)

Fgrep Listing (Listing Continued, text begins on page 46)

```
  /* Read in a line at a time for processing */

  while(fgets(buffer,MAX_LINE,in_fd))
  {
    if(nl = index(buffer,'\n'))  /* Remove newline */
      *nl = '\0';
#ifdef CP/M
    if(fflag == FALSE || yflag == TRUE)
      stoupper(buffer);
#else
    if(yflag == TRUE)
      stoupper(buffer);
#endif
    line_cnt++;          /* Increment the line counter */
    if((mtch_flag = run_fsa(buffer)) == TRUE)
      mtch_cnt++;        /* Increment matched line counter */
    if(cflag == FALSE && lflag == FALSE && sflag == FALSE &&
        ((mtch_flag == TRUE && vflag == FALSE) ||
        (mtch_flag == FALSE && vflag == TRUE)))
    {
      if(hflag == FALSE && prt_flag == TRUE)
        printf("%s: ",in_file);
      if(nflag == TRUE)
        printf("%05ld: ",line_cnt);
      puts(buffer);
    }
  }
  if(lflag == TRUE && mtch_cnt > 0)
    printf("%s\n",in_file);
  else if(cflag == TRUE && sflag == FALSE)
    printf("%ld\n",mtch_cnt);
  if(in_file != NULL)
    fclose(in_fd);
  if(mtch_cnt)           /* Match found */

    return TRUE;
  else                   /* No match found */
    return FALSE;
}

/* RUN_FSA() - Run the finite state automaton with string "str"
 *             as input. Return TRUE if match, FALSE otherwise.
 */

BOOL run_fsa(str)
register char *str;
{
  register FSA *st_ptr;
  char *message = NULL;
  BOOL msg_flag = FALSE;
  FSA *go(),
      *move();

  st_ptr = NULL;         /* Initialize FSA */
  if(xflag == FALSE)
  {
    /* Process the next input character in the string */

    while(*str)
    {
      st_ptr = move(st_ptr,*str);

      /* Print terminal state message and update FSA */

      if(st_ptr == 0 && message)
      {
        printf("--> %s\n",message);
        message = NULL;
        st_ptr = move(st_ptr,*str);
      }
      str++;
      if(st_ptr)
        if(st_ptr->out_str)      /* Terminal state? */
          if(pflag == TRUE)
          {
            /* Save terminal state message */

            message = st_ptr->out_str;
            msg_flag = TRUE;
```

```
        }
        else
          return TRUE;
  }
  if(message)  /* Print any remaining terminal state message */
    printf("--> %s\n",message);

    return msg_flag;
  }
  else  /* Match exact lines only */
  {
    while(*str)
    {
      st_ptr = go(st_ptr,*str++);
      if(!st_ptr || st_ptr == &FAIL_STATE)
        return FALSE;   /* Line not matched */
    }
    return TRUE;        /* Exact line matched */
  }
}

/* GO() - Process "litchar" and return a pointer to the FSA's
 *        corresponding "go" transition state. If the character
 *        is not in the FSA state's "go" transition list, then
 *        return a pointer to FAIL_STATE.
 */

FSA *go(st_ptr,litchar)
FSA *st_ptr;
register char litchar;
{
  register TRANSITION *current;

  /* If State 0, then access separate State 0 data structure of
   * the FSA. Note that there are no failure states defined for
   * any input to FSA State 0.
   */

  if(!st_ptr)
    return MZ[litchar];
  else
  {
    /* Point to the head of the linked list of "go" transitions
     * associated with the state.
     */

    current = st_ptr->go_ls;

    /* Transverse the list looking for a match to the input
     * character.
     */

    while(current)
    {
      if(current->lchar == litchar)
        break;
      current = current->next_el;

    }

    /* Null value for "current" indicates end of list was reache
     * without having found match to input character.
     */

    return current ? current->nextst_ptr : &FAIL_STATE;
  }
}

/* MOVE() - Process "litchar" and return a pointer to the FSA's
 *          corresponding "move" transition state.
 */

FSA *move(st_ptr,litchar)
FSA *st_ptr;
register char litchar;
{
  register TRANSITION *current;

  /* If State 0, then access separate State 0 data structure of
   * the FSA.
   */
```

(Continued on next page)

Fgrep Listing (Listing Continued, text begins on page 46)

```
if(!st_ptr)
  return MZ[litchar];
else
{
  /* Point to the head of the linked list of "move" transitions
   * associated with the state.
   */

  current = st_ptr->mv_ls;

  /* Transverse the list looking for a match to the input
   * character.
   */

  while(current)
  {
    if(current->lchar == litchar)
      break;
    current = current->next_el;
  }

  /* Null value for "current" indicates end of list was reached
   * without having found match to input character. The
   * returned pointer is then to State 0.
   */

    return current ? current->nextst_ptr : NULL;
  }
}

/* BD_GO() - Build the "go" transitions for each state from the
 *           command-line arguments.
 */

void bd_go(str)
char *str;
{
  register char litchar;
  char *nl,
       buffer[MAX_LINE],        /* Input string buffer */
       *stoupper(),
       *fgets(),
       *index();
  FILE *str_fd,
       *fopen();
  void error(),
       enter();

  /* Initialize FSA State 0 "go" transition array so that every
   * invocation of "go()" with "state" = 0 initially returns a
   * pointer to FAIL_STATE.
   */

  for(litchar = 1; litchar <= 127; litchar++)
    MZ[litchar] = &FAIL_STATE;

  /* If the -f option was selected, get the newline-separated
   * strings from the file "str" one at a time and enter them
   * into the FSA. Otherwise, enter the string "str" into the
   * FSA.
   */

  if(fflag == TRUE)
  {
    if(!(str_fd = fopen(str,"r")))
      error(STR_ERR,str);

    while(fgets(buffer,MAX_LINE,str_fd))
    {
      if(nl = index(buffer,'\n'))       /* Remove the newline */
        *nl = '\0';
      if(yflag == TRUE)
        stoupper(buffer);
      enter(buffer);
    }
    fclose(str_fd);
```

```
    }
    else
    {
      if(yflag == TRUE)
        stoupper(buffer);
      enter(str);
    }

    /* For every input character that does not lead to a defined
     * "go" transition from FSA State 0, set the corresponding
     * element in the State 0 "go" transition array to indicate
     * a "go" transition to State 0.
     */

    for(litchar = 1; litchar <= 127; litchar++)
      if(MZ[litchar] == &FAIL_STATE)
        MZ[litchar] = NULL;
}

/* ENTER() - Enter a string into the FSA by running each
 *           character in turn through the current partially-
 *           built FSA. When a failure occurs, add the remainder
 *           of the string to the FSA as one new state per
 *           character. (Note that '\0' can never be a valid
 *           character - C uses it to terminate a string.)
 */

void enter(str)
char *str;
{
  FSA *s,
      *go(),
      *create();
  TRANSITION *current,
             *insert();
  char *strsave();
  register char *temp;
  register FSA *st_ptr = NULL;  /* Start in FSA State 0 */
  register FSA *nextst_ptr;
  void error();

  /* Run each character in turn through partially-built FSA until
   * a failure occurs.
   */

  temp = str;
  while((s = go(st_ptr,*temp)) != &FAIL_STATE)
  {
    temp++;
    st_ptr = s;

  }

  /* Process the remainder of the string */

  while(*temp)
  {
    /* If a new state, then create a new state and insert
     * transition character and "go" transition in current
     * state. (Note special case of FSA State 0.)
     */

    if(!st_ptr)
      nextst_ptr = MZ[*temp++] = create();
    else if(!(current = st_ptr->go_ls))
    {
      nextst_ptr = create();
      st_ptr->go_ls = insert(nextst_ptr,*temp++);
    }
    else
    {
      /* ... or it was the character that the FSA returned a
       * "failure" for. Find the tail of the current state's list
       * of "go" transitions, create a new state and append it
       * to the current state's "go" list.
       */

      while(current->next_el)
        current = current->next_el;
      nextst_ptr = create();
      current->next_el = insert(nextst_ptr,*temp++);
    }
    st_ptr = nextst_ptr;
  }
```

(Continued on next page)

Fgrep Listing (Listing Continued, text begins on page 46)

```
  /* Make string terminal state's output message */

  st_ptr->out_str = strsave(str);
}

/* INSERT() - Create a new "go" transition and return a pointer
 *            to it.
 */

TRANSITION *insert(st_ptr,litchar)
FSA *st_ptr;
char litchar;
{
  TRANSITION *current;
  char *malloc();
  void error();

  if(!(current = (TRANSITION *)malloc(sizeof(TRANSITION))))
    error(MEM_ERR,NULL);
  current->lchar = litchar;
  current->nextst_ptr = st_ptr;
  current->next_el = NULL;
  return current;
}

/* CREATE() - Create an FSA state and return a pointer to it. */

FSA *create()
{
  FSA *st_ptr;
  char *malloc();
  void error();

  if(!(st_ptr = (FSA *)malloc(sizeof(FSA))))
    error(MEM_ERR,NULL);
  st_ptr->go_ls = st_ptr->mv_ls = NULL;
  st_ptr->fail_state = NULL;
  st_ptr->out_str = NULL;
  return st_ptr;
}

/* BD_MOVE() - Build the "failure" and "move" transitions for
 *             each state from the "go" transitions.
 */

void bd_move()
{
  register char litchar;
  register FSA *r,      /* Temporary FSA state pointers */
               *s,
               *t;
  FSA *go(),
      *move();
  TRANSITION *current,
             *insert();
  QUEUE *first,         /* Pointer to head of queue */
        *last;          /* Pointer to tail of queue */
  void add_queue(),
       delete_queue();

  last = first = NULL;  /* Initialize the queue of FSA states */

  /* For each input character with a "go" transition out of FSA
   * State 0, add a pointer to the "go" state to the queue. Note
   * that this will also serve as the "move" transition list for
   * State 0.

   */

  for(litchar = 1; litchar <= 127; litchar++)
    if(s = go(NULL,litchar))
      add_queue(&first,&last,s);

  /* While there are still state pointers in the queue, do ... */

  while(first)
```

(Continued on page 66)

Fgrep Listing (Listing Continued, text begins on page 46)

```
{
  /* Remove State "r" pointer from the head of the queue. */

  r = first->st_ptr;
  delete_queue(&first);

  /* Skip (terminal) state with no "go" transitions" */

  if(!r->go_ls)
    continue;

  /* Make "move" transition list for terminal state same as its
   * "go" transition list.
   */

  if(r->out_str)
    r->mv_ls = r->go_ls;

  /* For every input to State "r" that has a "go" transition to
   * State "s", do ...
   */

  for(litchar = 1; litchar <= 127; litchar++)
  {
    if((s = go(r,litchar)) != &FAIL_STATE)
    {
      /* If a defined "go" transition exists for State "r" on
       * on input "litchar", add a pointer to State "s" to the
       * end of the queue.
       */

      add_queue(&first,&last,s);

      /* Calculate the "failure" transition of State "s" using
       * the following algorithm.
       */

      t = r->fail_state;
      while(go(t,litchar) == &FAIL_STATE)
        t = t->fail_state;
      s->fail_state = go(t,litchar);
    }
    else
    {
      /* ... otherwise set the pointer to State "s" to a
       * pointer to the precalculated "move" transition of
       * State "r"'s failure state on input "litchar".
       */

      s = move(r->fail_state,litchar);
    }

    /* Add State "s" as the "move" transition for State "r" on
     * input "litchar" only if it is not State 0 and "r" is not
     * a terminal state.
     */

    if(s && !r->out_str)
      if(!r->mv_ls)   /* First instance of the list? */
        current = r->mv_ls = insert(s,litchar);
      else            /* No, just another one ... */
        current = current->next_el = insert(s,litchar);
  }
 }
}

/* ADD_QUEUE() - Add an instance to the tail of a queue */

void add_queue(head_ptr,tail_ptr,st_ptr)
QUEUE **head_ptr,
      **tail_ptr;
FSA *st_ptr;
{
  QUEUE *pq;
  char *malloc();
  void error();

  /* Allocate the necessary memory and set the variables. */

  if(!(pq = (QUEUE *)malloc(sizeof(QUEUE))))
    error(MEM_ERR,NULL);
```

```
  pq->st_ptr = st_ptr;
  pq->next_el = NULL;

  if(!*head_ptr)        /* First instance of the queue? */
    *tail_ptr = *head_ptr = pq;
  else                  /* No, just another one ... */
    *tail_ptr = (*tail_ptr)->next_el = pq;
}

/* DELETE_QUEUE() - Delete an instance from the head of queue */

void delete_queue(head_ptr)
QUEUE **head_ptr;
{
  *head_ptr = (*head_ptr)->next_el;
}

/* STRSAVE() - Save a string somewhere in memory */

char *strsave(str)
char *str;
{
  int strlen();
  char *p,
       *malloc();
  void error();

  if(p = malloc(strlen(str) + 1))
    strcpy(p,str);
  else
    error(MEM_ERR,NULL);
  return p;
}

/* STOUPPER() - Map entire string pointed to by "str" to upper
 *              case.
 */

char *stoupper(str)
register char *str;
{
  register char *temp;

  temp = str;
  while(*temp)
    *temp++ = toupper(*temp);
  return str;
}

/* ERROR() - Error reporting. Returns an exit status of 2 to the
 *           parent process.
 */

void error(n,str)
int n;
char *str;
{
  fprintf(stderr,"\007\n*** ERROR - ");
  switch(n)
  {
    case CMD_ERR:
      fprintf(stderr,"Illegal command line");
      break;
    case OPT_ERR:
      fprintf(stderr,"Illegal command line option");
      break;
    case INP_ERR:
      fprintf(stderr,"Can't open input file %s",str);
      break;
    case STR_ERR:
      fprintf(stderr,"Can't open string file %s",str);
      break;
    case MEM_ERR:
      fprintf(stderr,"Out of memory");
      break;
    default:
      break;
  }
  fprintf(stderr," ***\n\nUsage: fgrep [-vclnhyefxps]");
  fprintf(stderr," [strings] <files>\n");
  exit(2);
}

/*** End of FGREP.C ***/
```

End Listing

(Continued from page 44)

using B-trees and hash functions, two subjects we expect to expand on in future issues.

This issue of *DDJ* focuses on useful algorithms, and is intended to serve as a prelude to a new emphasis on algorithmic analysis of the program listings in *DDJ*. The reason for such an emphasis is simple: to increase the portability and usability of the tools we publish. Processor-specific magazines can get away with publishing processor-specific code. We don't want, and we think you don't want *DDJ* to become a processor-specific magazine, so we're trying to broaden the applicability of our offerings by providing the keys to allow you to extract what you can use from the listings. Those keys include, we think, well-documented high level code and clear discussion of the underlying algorithms and useful techniques embodied in the code.

And what about someday-useful algorithms? The algorithmic cutting edge? The Fgrep algorithm in this issue uses limited parallelism to gain speed. Next month we'll look at the Novix Forth chip and what it gains from a little parallelism. But deep parallelism of the sort realizable in multiprocessor machines of the future will require different algorithms, different models of the computer. What lies beyond Von Neumann architecture? We hope to investigate parallelism in upcoming issues of *DDJ*.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 194.**

Bose-Nelson Sort

by Joe Celko

Sorting algorithms can be divided into two types: those that preserve the original sequence of the records (also called stable sorts) and those that don't. For example, if a stable alphabetic sort is applied twice to an address file, first on city names then on states, the result will be a list ordered by cities within states. The same file sorted with a non-stable sort would lose the city ordering on the second sort. Although the ability to use stable sorts in a series makes them look more attractive than the non-stable sorts, the non-stable sorts are faster.

All sorting algorithms consist of an exchange operation that swaps two items in the file or array and puts them into sorted order. The number of exchanges done by a sort determines how fast the sort runs—and this number varies with the existing order in the input file.

This gives us a worst-case number of exchanges, a best-case number of exchanges, and an expected number of exchanges. For example, the Quicksort algorithm[1] has a best-case time for a file that is already sorted (most algorithms do quite well in this case) and it has a worst case for a file in reverse order. Its expected performance is **(n log2(n))** exchanges, where **(n)** is the number of records. This makes Quicksort one of the best sorting algorithms known to us. But do we know just how well a sort can perform?

In theory, the fewest number of exchanges required to sort a file of **(n)** items is the ceiling of **log2(n!)** exchanges. For example, for a file of five items, we have **log2(5!) = log2(120) = 6.9068**, which rounds up to seven exchanges. The next question is whether such algorithms perform equally well for different numbers of records.

The answer is mixed: optimal algorithms exist for some values and not for others. No single algorithm seems to be minimal for all values. Many of the minimal algorithms are highly specialized. For example, Dr. H. B. Demuth discovered an algorithm for sorting five items—and only five—in seven exchanges.[2]

Can an algorithm be generalized for any fixed number of records and still come pretty close to being optimal? This question leads us to the P-operator discovered by R. C. Bose and R. J. Nelson.[3] This is a recursively defined function that generates exchange pairs for a fixed number of items.

The bad news is that the algorithm takes a long time to generate these exchange pairs; you really should not use it directly to sort a file. The good news is that you need generate the pairs only once, and you have a very good sort for a given number of items. An application of this might be sorting a poker or bridge hand in a

No sort algorithm is optimal under all conditions. This one is useful when stability is not a consideration or when parallel processing is possible.

Joe Celko, 8833 Sunset Blvd #304, Los Angeles, CA 90069.

card-playing program.

The *P*-operator can be defined as ordered tuples of integers with a set of transformation rules that are applied left to right[4] The rules appear in the Figure (at right) and can be used to write a recursive program. The one given in Listing One (page 70) is credited to Dennis Allison. Another approach is to use an explicit stack and push quintuples, as shown in Listing Two (page 76). This second program will run faster due to the recursive procedure calls and lack of **printf** functions in C. Readers with accounts on CompuServe can obtain the source code and a short piece of documentation for Listing Two by accessing [70665, 1307].

What is happening in this confus*ing set of rules* is that the original array is divided into two ordered sequences, which are merged into one ordered sequence. The two subsequences, in turn, are ordered by recursive calls to the same procedure. For five terms, the Bose-Nelson P-operator program will generate the following C program:

```
{
swap(1, 2);
swap(4, 5);
swap(3, 5);
swap(3, 4);
swap(1, 4);
swap(1, 3);
swap(2, 5);
swap(2, 4);
swap(2, 3);
}
```

You can check this program by taking five playing cards and dealing them face down in a row. Turn over cards 1 and 2; if they are out of order, swap them and turn them face down again. Repeat this procedure until you have made all of the swaps in the program then turn over all of the cards. They should be in sorted order now.

It is worth noting that this program has nine steps and that we already know seven is the optimal number. So, contrary to earlier beliefs,[5] the Bose-Nelson does not produce optimal sorts. (It is left as an exercise for the reader to see how well the Bose-Nelson sort performs compared

```
P(1, x, y) => Swap(x, y);

P(2, i, 1) => Null
P(2, i, x) => P(2,i,a) P(2,(i+a),(x–a)) P(3,i,a,(i+a), (x–a))

P(3, i, 0, j, y) => Null
P(3, i, x, j, 0) => Null
      (These tuples should never appear, but they are given for completeness.)
P(3, i, 1, j, 1) => P(1, i, j)
P(3, i, 1, j, 2) => P(1, i, (j+1)) P(1, i, j)
P(3, i, 2, j, 1) => P(1, i, j) P(1, (i+1), j)
P(3, i, x, j, y) => P(3, i, a, j, b) P(3, (i+a), (x–a), (j+b), (y–b))
   P(3, (i+a), (x–a), j, b)

where  a =  [x/2]
       b =  [(y+1)/2] if (x is even)
            [y/2] if (x is odd)
```

Figure

Transformation Rules for Ordered Tuples

with the optimal values.) However, the swap pairs do not depend on the previous results, which makes it easy to use them for hard-wired sorting machines. It also makes it possible to use them in parallel processor computers: in the above five-item sort, the calls to swap(1, 2) and swap(4, 5) can be done at the same time.

References

[1] Hoare, C. A. R., "Algorithm 64: Quicksort," *Communications of the ACM*, 4 (1961) p. 321.

[2] Demuth, H. B., PhD thesis, Stanford University, 1956, pp. 41–43.

[3] Bose and Nelson, "A Sorting Problem," *JACM*, Vol. 9 (1962) pp. 282–296.

[4] [RIC 1972] Rich, Robert P., Internal Sorting Methods Illustrated with PL/I Programs, Prentice Hall, 1972.

[5] Knuth, Donald, *Sorting and Searching*, Addison-Wesley, 1973.

DDJ

Reader Ballot

Vote for your favorite feature/article.
Circle Reader Service **No. 196.**

Bose-Nelson Sort

(Text begins on page 68)

Listing One

```
* Recursive version of Bose-Nelson sort */
#include "STDIO.H"
#include "atou.c" /* ASCII to unsigned integer */

int count = 0;

main (argc, argv)
int argc;
char argv[];
{ int n;
  if (argc < 2)
     { printf ("USAGE: bose n >outfile\n");
       exit();
     }
  else
     { n = (atou(*(ap = *++argv) == '-'? ap+1: ap);
       bosesort (1, n);
       printf ("There were %d swaps\n", count);
     }
}

bosesort(i, j)
int i, j;
{  int m;
   if (n > 1)
      { m = 1 + (j-1+1)/2 - 1;
        bosesort (i, m);
        bosesort ((m+1), j);
        bosemerge (i, m, (m+1), j);
      }
}

bosemerge (i1, i2, j1, j2)
int i1, i2, j1, j2;
{ if ((i2 == i1) && (j2 == j1))
     { printf ("swap(%d, %d);\n", i1, j1);
       count += 1;
     }
  else if ((i2 == (i1 + 1)) && (j2 == j1))
     { printf ("swap(%d, %d);\n", i1, j1);
       printf ("swap(%d, %d);\n", i2, j1);
       count += 2;
```

(Continued on page 72)

Bose-Nelson Sort (Listing Continued, text begins on page 68)

Listing One

```
    }
 else if ((i2 == il) && (j2 == (jl+1))
    { printf ("swap(%d, %d);\n", il, j2);
      printf ("swap(%d, %d);\n", il, jl);
      count += 2;
    }
 else
    { i_mid = il + (i2-il+1)/2 -1;
      if (even(i2 - il+1) && (i2-il !=  j2-jl))
         j_mid = jl + (j2-jl+1)/2;
      else
         j_mid = jl + (j2-jl+1)/2 - 1;

      bosemerge (il, i_mid, jl, j_mid);
      bosemerge (i_mid+1, i2, j_mid+1, j2);
      bosemerge (i_mid+1, i2, jl, j_mid);

    }
}

even (x) int x; { return (1 - (1 & x));}
```

End Listing One

```
 /* non-recursive version of Bose-Nelson sort */
 #include "STDIO.H"
 #include "atou.c" /* ASCII to unsigned integer */
 #define MAXSTACK 2500  /* pick a high number */

 int count = 0;
 int top = 0;
 int stack[MAXSTACK];

 main (argc, argv)
 int argc;
 char argv[];
 { int n;
   if (argc < 2)
      { printf ("USAGE: bose n >outfile\n");
        exit();
      }
   else
      { n = (atou(*(ap = *++argv) == '-'? ap+1: ap);
        bosesort (n);
        printf ("There were %d swaps\n", count);
      }
 }


 bosesort(n)
 int n;
 { int a, b, i, j, x, y;
   printf ("{\n");
   push5 (2,1,n,0,0);
   while (top >0)
    switch (stack[top])
    {
```

Bose-Nelson Sort

Listing Two

```
case 0: { printf("}\n");
          top = 0;
        }
        break;

case 1: { printf("swap(%d,%d);\n", stack[top-1], stack[top-2]);
          top -= 3;
          count++;
        }
        break;

case 2: { i = stack[top-1];
          x = stack[top-2];
   /* these two variables are defined for easy reading */
          top -= 3;
          if ( x != 1)
             { a = (x/2);
               push3(3, i, a, (i+a), (x-a));
               push3(2, (i+a), (x-a));
               push3(2, i, a):
             }
        }
        break;


case 3: { i = stack[top-1];
          x = stack[top-2];
          j = stack[top-3];
          x = stack[top-4];
   /* these four variables are defined for easy reading */
          a = (x/2);
          if (even(x))
             b = (y+1)/2;
          else
             b = (y/2);
          top -= 5;
          if ((x == 1) && (y == 1))
             push3 (1, i, j);
          else if ((x == 1) && (y == 2))
                  { push3(1, i, j);
                    push3(1, i, (j+1));
                  }
          else if ((x == 2) && (y == 1))
                  { push3(1, (i+1), j);
                    push3(1, i, j);
                  }
          else    { push5(3, (i+a), (x-a), j, b);
                    push5(3, (i+a), (x-a), (j+b), (y-b));
                    push5(3, i, a, j, b);
                  }
        }
        break;
```

```
    default:
            { printf ("FATAL: Error in stack\n");
              exit();
            }
            break;
  } /* end of while loop */
}/* end of main */

even (x)
int x;
{ return (1 - (1 & x));}

push3(a, b, c)
int a, b, c;
{ stack[++top] = c;
  stack[++top] = b;
  stack[++top] = a;
  if (top > MAXSTACK)
     { printf ("FATAL: stack overflow at %d\n", top);


       exit();
     }
}


push5(a, b, c, d, e)
int a, b, c;
{ stack[++top] = e;
  stack[++top] = d;
  stack[++top] = c;
  stack[++top] = b;
  stack[++top] = a;
  if (top > MAXSTACK)
     { printf ("FATAL: stack overflow at %d\n", top);
       exit();
     }
}
```

End Listings

Two TEX Implementations for the IBM PC

Richard Furuta and
Pierre A. MacKay

"An experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TEX for formatting, and a language like Pascal for programming." —Donald Knuth, *The WEB System of Structured Documentation.*

TEX, as many readers will know, is Donald Knuth's system for typesetting technical text. The name is based on the Greek root in the word 'technical'; it is in no way associated with the state of Texas, and only remotely associated even with the word 'text'. The shibboleth of the true convert is the pronunciation of the final letter, which should sound rather like the 'ch' at the end of the name of J. S. Bach. The official derivation is from *tékhnē*='art' or 'craft'.

A system intended for the creation of beautiful books.

Knuth is a Professor of Computer Science at Stanford University and is best known for his ongoing series of books collectively entitled *The Art of Computer Programming*, perhaps the most important central reference source in Computer Science. TEX grew out of the need for a typesetting program that could be used to prepare new volumes in the series—in particular, the need for a program that would take care of the delicate formatting needed for mathematics typesetting. In the process of working out this specific problem, Knuth supplied just about every need that could be imagined for text typesetting. There are certain things TEX does not do; it has no built in graphics capabilities, and it does not provide for arbitrary rotations of characters and lines, but it should be noted that there are very few digital phototypesetters which could make use of such capacities even if they were offered. There are good general mechanisms within TEX which may someday be used to supply some of these special refinements if the output devices for them become readily available. For now, TEX offers precisely what Don Knuth claims for it in the preface to the *TEXbook*, "a system intended for the creation of beautiful books." TEX will be used, and is being used for everything from two-line business letters to railway schedules, but the heart of the program is book-production, and it can only be truly understood in that light.

An earlier version of this program, now distinguished as TEX78, was developed for the Stanford University DEC-10 system, using the SAIL programming language. Subsequently, TEX78 was translated into Pascal, and ported to a variety of mainframes. Based on this experience, the program was completely redesigned and reimplemented, and it is this newest version of the program that is today called TEX.

One of the commonest criticisms of TEX78 was its sheer size and con-

Richard Furuta and Pierre A. MacKay, Univ. of Washington, Seattle WA.

sequent inaccessibility. Most users of TEX78 would not readily have believed that the current, more powerful version of the program could possibly be made available in the small systems world. But here it is in two different versions for the PC-XT and its look-alikes. As we shall demonstrate below, it takes a large XT to run TEX, and it takes a good deal of disk space to store it, but both programs are genuine TEX, and have been tested and validated.

Throughout this review, we will refer to two TEX-related publications: *The TEXbook* and *TUGboat*. *The TEXbook* is the manual that describes TEX. Written by Donald Knuth and published by Addison-Wesley, it costs $14.95 in bookstores. The *TUGboat* is the newsletter of the TEX Users Group (also known as "TUG"). No matter which version of TEX you purchase, we believe that you will want to become a member of the TEX Users' Group. The initial yearly membership cost is $20.00, and this includes a subscription to the *TUGboat*. TUG is the best way to keep up-to-date with the latest TEX developments and the *TUGboat* is an excellent place for you to share your own TEX discoveries. Write to the TEX Users' Group, at P.O. Box 9506, Providence, Rhode Island 02940 for more information.

The distribution of MicroTEX and PCTEX marks a break from the pattern established under the sponsorship of the TEX Users Group for larger systems. The broad aim for larger systems through the network of TUG site coordinators has been to place full source code in the hands of the systems programmer at each location, so that the site will have full control over compilation and validation. This approach has depended on the assumption that the site will have a great deal of directly or indirectly addressable programming memory, and a robust and fully functional Pascal compiler. (In the past two years, experience has led to the formation of "David Fuchs' Law of Pascal Compilers," that TEX will smoke out a bug in every compiler created earlier than TEX itself.)

Even on a mainframe system, a TEX compilation can take hours (some have looked as if they were going to run for days), and the compilation time on a PC must be quite alarming. These two versions of TEX for the PC give the user the benefit of the special efforts of two long established members of the TEX community. Lance Carnes, the author of PCTEX, is the TUG site coordinator for "small" TEX. David Fuchs, the author of MicroTEX, is second only to Donald Knuth in his long association with TEX. Both versions produce an identical output file from identical input files (more later about the output format, called DVI), and both have gone successfully through the "torture test" validation prescribed by Donald Knuth for true implementations of TEX. The implementation methods, however, are quite different. MicroTEX was developed through a translation from Pascal to C, while PCTEX has followed the usual route of direct Pascal compilation. (Microsoft's Pascal compiler came through rather handsomely in this effort, with only one bug.) There are various arcana complicating the above histories. David Fuchs wrote a post-optimizer for the C compiler to improve performance, and Lance Carnes had to work up a collection of segmented pointer types for Pascal along with the address calculation routines to go with them.

Each version of TEX is delivered in a large box. MicroTEX arrives with a copy of *The TEXbook* (also available separately in bookstores), a 92 page installation and introduction manual and 8 diskettes. The diskettes contain TEX, fonts, and DVI-EPS, a program that prints TEX's output files on the IBM graphics printer and its near relatives. We tested MicroTEX on an XT. We understand that the current version also runs on an AT.

PCTEX's box includes a binder containing a 26 page installation manual, a 128 page introduction to PCTEX, including a description of their own macro package, intended for beginners, a summary of the AMS-TEX macro package, and a reprint of the manual for the LATEX macro package. A program to print output on IBM and Epson graphics printers and *The TEXbook* can be purchased separately. PCTEX distributes TEX, its associated font tables, and the macro packages on 5 diskettes. If PC-DOT, the program to print the output, is purchased, an additional 8 diskettes contain the program along with its fonts. Two versions of TEX are provided—one for a PC with 512K-bytes of memory and another for one with 640K-bytes. MicroTEX automatically reconfigures itself to use as much memory as is available, from 512K-bytes to 640K-bytes. We successfully installed and tested PCTEX on both an XT and an AT.

The system requirements for these two versions of TEX are similar. Each requires an IBM PC with a hard disk, a floppy disk drive, and at least 512K-bytes of RAM. You will need about 5 megabytes of hard-disk storage although this amount can be trimmed down once the system is installed.

Taking the MicroTEX distribution first, it gets very high marks for ease of installation. At first accidentally, but then deliberately, I made every non-destructive error I could think of, and MicroTEX was forgiving of all of them. There is one slight warning I would give, however. After TEX is loaded, you are instructed to copy in the compressed version of the fonts, and then to get the process of decompression started and walk away for the 50 or so minutes it takes. Not quite. Immediately after the process starts, you have to supply the disk containing the system's COMMAND.COM file in drive A. If you have walked away too early, you will come back to find that nothing at all has happened in the past hour.

Other than that, everything is smooth. Within two hours, TEX is running, and you can try out the sample files on the directory, one of which is called SAMPLE.TEX. MicroTEX offers four levels of print quality on the Epson printer, and the observed times for each are given

below. (SAMPLE.DVI is a reasonable test, since it loads a fair number of fonts.) At this point, if you are running the minimum 512K-byte configuration on your XT, you may run into trouble, however. Print your SAMPLE.LOG file, and DOS will load the printer driver into memory. Now try to run TEX, and you will be told, "Not enough memory for TeX." The MicroTEX manual warns the user that it may be necessary to leave out some of the resident drivers, but with DOS 3.0 in 512K-bytes, it is necessary to leave them all out. One might almost imagine that the program was distributed as an incitement to purchase that last 128K-bytes of memory.

PCTEX also installs smoothly, although we didn't deliberately try to make errors this time. The installation manual provides a step by step description, and installing TEX takes about half an hour. Installing PC-DOT and its associated fonts takes another hour or so—as with the MicroTEX installation, most of this time is spent waiting for fonts to be uncompressed. Unlike the MicroTEX installation, you have to attend to PCTEX during the installation process, swapping floppies as requested.

The installation of PCTEX involves a few more steps than the installation of MicroTEX, but you gain certain advantages as a result. To understand these it is necessary to consider how TEX is put together to make a working typesetting system. A pristine version of TEX is a very minimal typesetting program indeed. It provides the necessary primitive operations for character positioning at the lowest possible level, a sort of "assembler code" for typesetting. No reasonable person would even consider formatting a page at this level. The other side of the program is an extremely powerful macro processor. (The macro processor was an afterthought and an adjunct to the old TEX78, but it is at the very heart of the present program.) Only after a large number of macros has been processed and evaluated do you have a fully operational typesetting system. All currently available versions of TEX come with a macro file known as PLAIN.TEX, which forms the basis for any further refinements. PLAIN.TEX provides a great variety of convenient handles for common typesetting conventions, preloads a number of fonts, and arranges for the input of the hyphenation dictionary that is one of TEX's most remarkable features. (The stories of incompetent hyphenation algorithms in computer-assisted typesetting systems make up quite a tale of horror.) Getting all this stuff converted to tokens and premasticated into an efficient form for TEX to use takes time and the large-machine distributions of TEX actually provide for two separate compilations of the program, one known as INITEX to do the massaging and cough up a predigested format file, and VIRTEX, a second, smaller version with a weaker digestive system, which can read in the predigested PLAIN.FMT (notice the change in suffix) at a relatively high speed. In MicroTEX this has all been done for you, and the preloaded TEX is instantly ready to run. In PCTEX you get to do it yourself.

The executable file provided with PCTEX is a VIRTEX with a special hook provided to call in the capabilities of INITEX as an overlay. The initial run of TEX is done with a special flag on the command line, and is devoted entirely to masticating and regurgitating the PLAIN.TEX file. (These alimentary metaphors are extremely pervasive in the documentation of TEX.) From then on, although you will be told that the TEX you are running has no format preloaded, the program will read in the PLAIN.FMT file automatically, without being prompted, and at more or less the optimum speed for a disk to memory transfer. As a result, PCTEX takes about 20 seconds longer to go into action than MicroTEX, but it gains in many important ways. One of these is immediately evident from listing the files in the TEXINPUT directory. As we noted above, PCTEX comes with both $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX for advanced mathematics formatting and with LATEX, which is a powerful document formatting package using a syntax that allows you to describe a document by its logical structure instead of by its appearance. Both of these macro files are large—LATEX is *immense*, requiring 640K-bytes to run—and you would soon weary of waiting for the macro-interpreter to do its thing with them if you had to read them in at the start of every invocation of TEX. The LATEX or the $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX format file, however, can be read into place in almost the same time as the smaller PLAIN.FMT.

Another, even more subtle advantage of having an INITEX available has to do with the hyphenation table. Hyphenation rules differ considerably from language to language, and a user with a large amount of French or German text to set will more or less be forced to create a new FMT file to get satisfactory results. The creation of a new hyphenation table is non-trivial, and neither version of TEX for the PC provides the program that does it, but assuming that PATGEN has been run on some larger system, it still remains that the only practical way to get at the new hyphenation patterns is through the use of INITEX. Incidentally, this question of hyphenations is one of the only genuine limitations in the use of TEX for continuous text. There is not, at present, any way of switching from one hyphenation pattern to another in the course of a run. You may hyphenate efficiently in English, or in French, but not in both together.

Once either MicroTEX or PCTEX has begun reading the user input file there is very little difference between them. The TEX TRIP.TEX "torture-test" guarantees that we can assume that they are functionally identical and that they correctly recognize and process TEX. It is theoretically possible that a TEX might pass the torture-test and later produce a DVI file that was subtly different from that produced by all other TEXs, but it has not happened yet, and if it does, the test will probably be revised to eliminate the error. The normal benchmark is the set-

ting of the TEXbook, and in announcements in a recent edition of *TUGboat*, the XT speed for this effort was given as 26.4 seconds a page for MicroTEX and "about 25 seconds a page" for PCTEX. In normal use, that is a very close race indeed. (PCTEX's running time on the AT is about six seconds a page.)

A true TEX-hacker will notice another difference between the two programs. MicroTEX is based on the latest version of TEX (we now have Version 1.4), while PCTEX is based on a somewhat older version (1.0, in our case). Usually, this would not be significant since the change from version to version has been quite small. Unfortunately, however, Version 1.3 incorporated major changes to TEX's memory management routines that greatly reduced the possibility of running out of memory while TEX-ing a document. We are assured that PCTEX will be brought up to date soon.

The version of LATEX distributed with PCTEX is also outdated, which may cause some difficulties if you try to move those files to another computer since LATEX is still being developed and is undergoing rapid change. We transferred over the latest version of LATEX and were able to run it on PCTEX after shortening one font name that was too long. If you are running a newer version of LATEX on another computer, we expect that you will want to update PCTEX's copy before building the FMT file.

We understand that $\mathcal{A}_{\mathcal{M}}\mathcal{S}$-TEX is now being developed using PCTEX, so we believe that the version distributed here is completely up to date.

Throughout this review, we have mentioned TEX's device-independent output file. Now, the time has come to turn back to general issues and to consider the implications of this.

In order to appreciate the nature of the DVI file, it is helpful to know something about the actual genesis of TEX. TEX is not a 'word-processor' in any ordinary sense of that mildly repellent term. The target machine for the original program was a genuine digital phototypesetter, using silver halide typesetting film to record characters painted on a CRT. That particular machine, in fact, has the highest resolution known in the industry—almost four times that of its nearest rival. But TEX does not stop even there. It produces a file for an imaginary typesetting device with a resolution of $1/2^{16}$ printer's points (there are 72.27 points to the inch), and leaves it to a second program to interpret that file for real-world devices. This output file is known as the 'device-independent' (DVI) file and is guaranteed to be exactly the same for any given source file when produced by a properly installed and tested TEX.

This notion of device independence goes far deeper than that in most other systems. It allows such an absolute confidence in the quality of results that the DVI file has seriously been proposed as a standard for document interchange in Europe. The suggestion is that on-line document retrieval from libraries be sent in DVI format for selective printing at the requesting site. But this is not to claim that TEX and the DVI file are the preferred system for all applications. The principal virtues of TEX carry with them certain constraints.

Perhaps the least generally understood are the constraints inherent in writing for a high-resolution phototypesetting system. At the present time, and with the presently available fonts, TEX optimizes for the best output device available. On an Alphatype, Autologic, Compugraphic or any of the other high-resolution typesetters for which DVI interpreters have been written, the interword spacing and inter-letter spacing are evaluated with round-off errors too small for the human eye to detect. At the most popular laser-printer and electrostatic resolutions (200, 240, and 300 dots to the inch) round-off errors are usually inoffensive, but they are nearly always discernable. It is simply impossible to do justice to the delicacy of high-resolution typesetting at 300 dots to the inch. (When we say that a device's resolution is, for example, 200 dots to the inch, we mean that the individual dots making up an image can be positioned to a resolution of $1/200th$ of an inch.)

This is not to say that the results are bad. Several thousand happy users of TEX have never seen their output at any resolution better than 200 dots to the inch, but others have wondered why the program cannot be better tuned to laser-printer resolutions and to the newer carefully designed laser-printer fonts such as Bigelow and Holmes's Lucida. The answer is that it can be so tuned, through the use of the 'TEX Font Metric' TFM file, a file which provides TEX with all the vital statistics about font characters, such as height, depth, width, letter-spacing adjustments (kerning), etc. If the manufacturer or distributor is willing to provide the necessary information, a TFM file can be made up to match any font currently available. It is not even a very laborious job. But a TFM file is not a font. It will not produce characters on any output device at all.

In some senses this can be an advantage. Suppose you wish to produce a book in the Baskerville font, and you have found a typesetter manufacturer who will provide you with sufficient information about its Baskerville fonts to produce the needed TFM file. For your final, high quality output, you have the problem solved; TEX will read the TFM file and make all its calculations based on the character widths, etc., of that font. But you do not want to do the proof copies on a slow, expensive photographic processor, and there is no 300-dot/inch Baskerville on your laser printer, far less on your graphics impact printer. If you are willing to shut your eyes to the bad letter spacing, and are concerned primarily with correcting typos and misspellings, you can make some font that you do have on your low to medium resolution printer masquerade as Baskerville. The laser printer output will look dreadful, but the cost of proof correction will be kept within reason. Conversely if the laser printer output is what you

really want, there is no reason to use an ersatz high-resolution font at all. Create a TFM file that genuinely matches one of the fonts specially designed for use at 300 dots/inch and your laser-printer output will be *very* acceptable indeed.

It is even possible to create a TFM file and an associated format file for a daisy wheel printer. To run TEX in this mode is to give up at least nine-tenths of its power as a typesetting system, but it can be done. The important thing is to recognize that letter-spacing and interword spacing are governed by the values supplied in the TFM file, and once the choice has been made, the DVI format will record that choice with breathtaking accuracy. No matter what the resolution is of your target machine, the DVI file will represent something far better than the best that that machine is capable of.

So while there are no technical reasons why TEX output cannot be tuned to particular output devices, such tuning is almost never done in practice, and in fact this tuning is usually considered to be an undesirable thing to do. The reason for this is that tuning introduces device and resolution dependencies into the DVI file and these dependencies will negate the more general concept of device independence provided by TEX. For not only are the positionings of the characters of a page described in a device-independent manner, but the characters themselves are also defined in a device-independent fashion.

The fonts used by TEX are produced from descriptions—programs, if you will—written in a language called METAFONT. A METAFONT program defines a font's characters by specifying the movement of a pen over a Cartesian coordinate system. As the name of the system implies, a "meta-font" is defined in this fashion—in other words, a single METAFONT description can be used to produce a *family* of related fonts (for example, separate fonts in roman and bold face) by varying the parameters that control values such as the pen shape and size. The important result in the context of this review, is that the description also is resolution-independent. In other words, the description will produce font bitmaps for use at different resolutions when the appropriate parameters are altered.

At present, only one generally available family of fonts, the Computer Modern family, has been defined through METAFONT. Variations within the family provide serifed roman, bold, italic, and slanted faces, a typewriter face, and a set of sans-serifed faces, each in a range of point sizes. TEX use is tightly coupled to the Computer Modern font family because of the flexibility afforded by the METAFONT description.

Since the Computer Modern family of fonts can be generated at almost any resolution needed by a device, and since the DVI file format represents the output with a precision unattainable by any device, the TEX concept of "device independence" can be restated as providing the *same* output on *every* output device, constrained only by the limitations of the device. There are two aspects to this formulation of device independence. One is that of *source-level independence*. In other words, every implementation of TEX should produce an identical DVI file from a TEX source file, if that source file uses the standard TEX environment—the Computer Modern fonts and the facilities defined in *The TEXbook*. The second aspect is that of *output-level independence*. A DVI file produced by any implementation of TEX should be able to be printed on any suitable printer, if the standard TEX environment has been used. DVI files can be, and frequently are, transferred from one model of computer to another in order to take advantage of the printers attached to the particular computers. As a practical aside, the most frequently encountered problems in developing the mechanisms that allow the transferring of a DVI file from one computer environment to another are related to the differing computer architectures and to the file transfer itself. The DVI file is defined as a stream of 8-bit bytes. Finding a file transfer mechanism that allows transmission of all eight bits of the byte sometimes involves a time consuming search. Differences in the ways in which these bytes are packed into the computer's words also may cause difficulties that require careful attention to overcome.

As the standard TEX environment is so important to maintaining device independence, you should consider the capabilities and cost of the printer that you will need carefully before deciding to purchase TEX. To get the full benefit of TEX, the output device you choose must be powerful enough to simulate TEX's ideal device and further, your device must allow you enough flexibility to allow you to use the TEX fonts.

What this means is that you should not plan to use a mechanized typewriter with TEX. Technically, you certainly could build the TFM font width tables to describe the typewriter's characters, but practically, TEX documents produced in this way will be incompatible with everyone else's and you will be unable to use most of TEX's more powerful features. What you will want to have is either a printer that allows you to do full page bit graphics or one that allows you to define your own characters and place them arbitrarily on the page (or, of course, one that allows you to do both).

Very few printers can print a DVI file directly. Generally, the DVI file is converted into a form compatible with the printer by an application program called, in TEX lingo, the *Output Driver*. As a practical matter, before you buy a printer for TEX use, you should make sure that the appropriate Output Driver software already exists for your computer. Writing such a conversion program is not an extraordinarily hard programming project, but getting the output to look "right" involves surprising subtleties. It is often more convenient to purchase the conversion software so you can get directly to the matter of creating beautiful documents.

Printers appropriate for TEX use can be placed into three different categories, based on the resolution

at which fonts can be defined and at which elements of the image can be placed.

Low-resolution printers are usually impact printers. The lowest resolution at which recognizable output can be produced is about 75 dots/inch. Some printers of this type can print at resolutions up to 200 dots/inch. At the lowest resolutions, the output from these printers is often unreadable, particularly in mathematical equations and in footnotes where the type sizes tend to become quite small. At the highest resolutions, the printers can take quite long to print each page—up to five minutes per page in some cases. Both PCTEX and MicroTEX make available software to print TEX output on a number of these printers (both products support the IBM Graphics printer and the Epson RX and FX printers).

At the other end of the range are *high-resolution printers.* These print at resolutions greater than 1000 dots/inch and are generally phototypesetters—devices producing output on photographic paper that must be developed before the image can be inspected. They are frequently quite slow (several minutes per page of output) and quite expensive to purchase and to operate. They are, however, the only acceptable way to produce output suitable for publication in high-quality books. Fortunately, a number of businesses own phototypesetters and are equipped to print TEX `DVI` files from IBM PC disks. The advertisements in the TEX Users Group Newsletter, the *TUGboat*, are the best way to locate these businesses.

In the middle is an increasingly important group of devices: the *medium-resolution printers.* Most of these printers are *laser printers*, and the 300 dot/inch resolution is becoming somewhat of a standard. While it is not possible for devices of this resolution to produce true book publication-quality output (the advertising claims of manufacturers notwithstanding), it is quite realistic to expect to be able to use these printers to produce output for less demanding applications such as newsletters, memos, and proofs of material that will ultimately be sent out for typesetting on a high-resolution printer. Some of the laser printers are not suitable for TEX use, as will be discussed, but those that have successfully printed TEX output range in price from about $4000 upward and print at speeds from about five pages per minute upward.

A laser printer is built from a *marking engine* and a *controller.* The marking engine is what puts the marks onto the paper. The controller drives the marking engine and provides the hardware and software interfaces that are visible to the person programming an application to print on the laser printer.

When comparing different laser printers, it is important to distinguish between the functions of marking engine and those of the controller. For example, it is usually correct to assume that maintenance statistics for one laser printer will also hold for a different printer if the two use the same marking engine. It is incorrect, however, to assume that information on the graphical capabilities of one manufacturer's laser printer will be valid for another's, even if the two printers both use the same marking engine.

As illustration, one commonly used marking engine is the Canon LBP-CX. This marking engine, used by many different laser printer manufacturers, is the heart of the Imagen 8/300, the QMS 800, the HP LaserJet, and the Apple LaserWriter. What distinguishes these laser printers from one another are the interfaces implemented by their controllers. The HP LaserJet, for example, is essentially an upgrade for a mechanized typewriter-like device. It does not allow fonts to be loaded over its hardware interface line and full-resolution bitmaps are limited to about one quarter page. We do not know of software that allows printing of TEX's output format on the HP LaserJet and suspect that such software will be extremely difficult, if not impossible, to write because of the printer's limitations. On the other hand, the Apple LaserWriter implements an interface language called PostScript. PostScript is a general purpose programming language, Forth-like in appearance, and is one of the most sophisticated printing languages available (PostScript is based on an earlier Xerox-defined printing language, which is named InterPress). The LaserJet and the LaserWriter use the same marking engine but are radically different in the sophistication of the functions that they provide.

Indeed, to a programmer providing TEX support, the interface language defined by the laser printer's controller is usually more important than which marking engine is being used. The interface language is generally the same or similar for a company's laser printers, even though those products may use significantly differing marking engines.

The companies that make PCTEX and MicroTEX have announced that they will be supporting TEX output from the IBM PC for the Apple LaserWriter, the Imagen laser printers, and the QMS laser printers. Another relatively low cost laser printer, the DEC LN-03, has been supported, although not yet on the IBM PC. A full list of supported devices is printed in each edition of the *TUGboat.*

An additional difference between the two implementations becomes apparent when comparing the two Epson drivers. DVI-EPS, included with the MicroTEX package, pro-

vides four levels of printing on the Epson. We printed a fairly dense page of TEX output at each of these levels (see Figure, page 88) and observed that the time to print the page ranged from about $22\frac{1}{2}$ minutes at the best quality (level 1) to only $3\frac{1}{2}$ minutes at the lowest quality (level 4). Intermediate timings were 12 minutes for level 2 and 10 minutes for level 3. In contrast, PC-DOT, available as an option with PCTEX, only provides two printing levels. The draft mode produced output similar to DVI-EPS's level 2, and took $12\frac{1}{2}$ minutes for our sample page. Final mode in PCTEX is comparable to DVI-EPS' level 1, and took $23\frac{1}{2}$ minutes to print. In all cases, print time seemed to depend more on the density of the page than on other factors.

23 minutes is a substantial time to wait for a page of output to be printed. Consequently, we believe that DVI-EPS is significantly preferable to PC-DOT. In particular, DVI-EPS' lowest quality level 4 printing at under four minutes for the page will get quite heavy use during document development and proofing. What is desperately needed are mechanisms that allow the DVI file to be previewed on the PC's screen, allowing the slow printing pass to be bypassed altogether.

In summary, despite the essential similarity of output from the two implementations of TEX for the PC, there are numerous differences that allow for a meaningful comparison. MicroTEX comes as a complete package, including TEX, the format file, the DVI-EPS printer driver and its numerous fonts at several different pixel-densities, and the authoritative documentation of *The TEXbook* itself. The version of TEX is absolutely up to date, and the driver offers a wide range of output quality. All of this together comes to $495.00. You can run the program in 512K-bytes of memory, but only at the price of some inconvenience (no resident drivers in your DOS).

PCTEX offers various options. The basic price of $279.00 gives you TEX, separately compiled for two different sizes of memory, and the AMS-TEX, and LATEX macro files, together with the mechanism for loading them efficiently. At the moment PCTEX is still at version 1.0 by contrast with MicroTEX at version 1.4. The PC-DOT driver program at $100.00 is not so broadly functional as the DVI-EPS program, If you want *The TEXbook*, you must order it separately for $15, but you do get the user documentation for LATEX and AMS-TEX.

In our judgment, the ideal package would be PCTEX for the TEX together with DVI-EPS to drive the printer. Only those with larger pockets might consider this, since it requires buying both PCTEX and MicroTEX. Otherwise, especially for those with technical documentation in mind, we strongly favor the PCTEX offering. We believe that the inclusion of INITEX capabilities in PCTEX is so important that it overrides the inconveniences caused by the lag in versions and by the less functional printer driver.

(Addison-Wesley is distributing a complete INITEX *in the August update of MicroTEX.–Ed.)*

[This review was formatted with MicroTEX on an IBM AT with 512K in $2\frac{1}{2}$ minutes. The fully-composed output was uploaded to a Stanford mainframe and sent to an Autologic APS-micro-5 typesetter, with the results seen here.]

PCTeX, Version 1.0
Company: Personal TEX, Inc.
20 Sunnyside, Suite H
Mill Valley, CA 94941
(415) 388-8853
Price: $279.00; PCDOT $100.00

MicroTeX, Version 1.4A1
Company: Addison-Wesley
Publishing Company, Inc.
Reading, MA 01867
(617) 944-3700
Price: $495.00

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 197.**

SCSI Your Mac

by John Bass

I got tired of waiting on those sluggish Mac floppy drives, tired of waiting for an affordable hard disk for the Mac. I got so tired, finally, that I went to work. My efforts paid off, and I was pleased enough with the result to think that others might appreciate my MacSCSI® interface, too.

One of the most obvious advantages to doing your own hard disk interface for the Mac is cost. I think MacSCSI is the hard-disk equivalent of *DDJ*'s January 1985 "Fatten Your Mac" 512K upgrade: the hacker's hard disk upgrade at budget prices, if you're willing to shop around for disks and SASI/SCSI controllers. Here in Silicon Valley we see hard disks at the computer swaps for between $100.00 and $500.00, often either refurbished or new discontinued models that were surplus. Likewise, SASI/SCSI controllers are seen at between $50 and $150. Thus, with the MacSCSI interface and some bargain hunting you can put a hard disk on your Mac for less than $500. New production drives and controllers are available between $900 and $3000, depending on the size and performance.

Another advantage is the experience you'll gain, both in writing your own bells and whistles into the supporting software, and in simply cracking open your Mac and getting to know its innards better. And in this regard, the MacSCSI interface is unlike the 512K upgrade: you don't have to take a soldering iron to a delicate, expensive logic board to install MacSCSI. You don't have to modify the Mac logic board at all.

The parts for this venture are few. The MacSCSI host adapter consists of four parts: an NCR5380 single chip SCSI interface, two 74LS10's to provide the address decodes, and a 50 pin header for the SASI/SCSI cable. The rest of a complete setup is a SASI/SCSI controller, hard disk drive, power supply, and cables for power and data. You can make your own case from a bookcase speaker for a few dollars, you can build a nice custom cabinet, or you can buy an off-the-shelf hard disk enclosure complete with power supply and cables.

Figure 1 (page 95) tells the story: it contains the schematic for the MacSCSI interface. The host adapter is memory mapped into the Mac's ROM address space above its last valid address. To accomplish this, I used selection equations for the NCR5380 of

$$NCRSEL = \overline{ROMSEL} * A20 * \overline{A11}$$
$$NCRRD = NCRSEL * A4$$
$$NCRWRITE = \overline{UDS}$$

By using address lines to enable the read and write enables on the NCR5380, I avoided needing to run any wires to chips or to cut away at the main circuit board; the only interface point to the Mac is its ROM socket. This scheme does, though, require that the software read an NCR5380 register at one address and write it at another—quite a reasonable tradeoff for not having to do any cuts or adds to the main logic board, I think.

While it's not clear whether Apple would honor a warranty or Apple Care contract on a Mac upgraded this way, if done carefully the upgrade will not physically alter or harm the Mac. If you have problems with your Mac (other than the MacSCSI upgrade), most helpful dealers should be willing to service your Mac if your boards are not altered or damaged. You may even find some helpful dealers and independent repair centers that will install the upgrade for you at their normal shop rates (about an hour of their time).

If you're up for a project, the interface can be assembled using off-the-shelf parts and point-to-point wiring. If you do it this way, you'll need to make a chip carrier from perf board with holes on 0.1 inch centers. You can salvage the socket pins for the chip carrier by cutting apart two Augat low profile 28 pin sockets with machined pins. Then attach the perf board to a wooden support block and drill out to 0.055-inch in the pin pattern for the Mac's ROMs. Keeping the wooden block attached, press the pins into the perf board using a vise or press (be careful not to crush the pins). The wooden block provides a die to protect the socket pins during the insertion process. You can then insert the other parts into the perf board and wire them using point-to-point soldering on the back side. Finally, and carefully, remove the ROMs from the board, insert them into the chip carrier, and plug the chip carrier with the ROMs back into the ROM sockets.

I'm skipping any discussion on how to take your Mac apart and reassemble it; for that, see the installation section on page 96.

Now for the hard part. I've come up with several options for getting the SCSI ribbon cable out of the case, none of them elegant. One is to cut a hole in the back of the case for an exit. Another is a panel-mounted connector installed in the case (my preferred solution). Or you can run a ribbon cable out between the bottom case seam in front, although you'll have to widen the seam slightly to do this. A fourth solution is not to bring the cable out at all, of course. General Computer has already demonstrated the feasibility of installing a hard disk inside the Mac case. I haven't

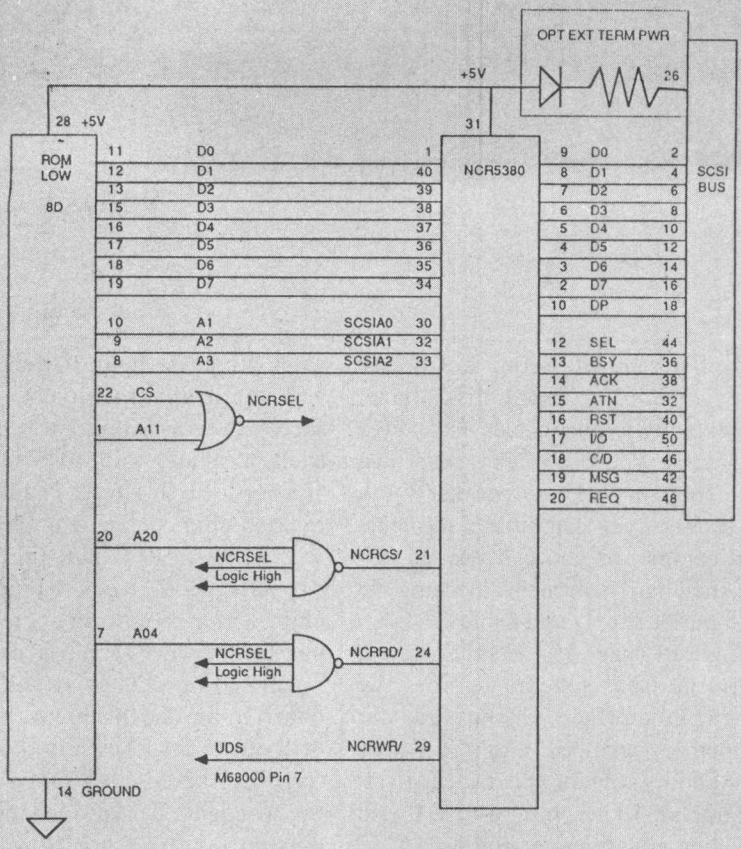


Figure 1

Mac SCSI host adaptor by DMS Design—Schematic Logic Drawing

Mac SCSI Selects

READ	WRITE	
RESET	INITIATOR START	0xE
DATA	TARGET START	0xC
STATUS	DMA START	0xA
SCSI STATUS	SELECT	0x8
TARGET CMD	TARGET CMD	0x6
MODE	MODE	0x4
INITIATOR CMD	INITIATOR CMD	0x2
SCSI DATA	SCSI DATA	0x0
0x50001X Odd Bytes	0x50002X Even Bytes	

Macintosh Memory Map

Bank	Address	Overlay	Overlay/
0	00xxxxx	Rom	Ram
1			
2	400xxxx	Rom	Rom
	50000xx	Mac SCSI	Mac SCSI
3	60xxxxx	Ram	
4	9FFFFx	SCC Read	SCC Read
5	BFFFFx	SCC Write	SCC Write
6	DFxxFF	IWM	IWM
7		VIA	VIA

Figure 2

Device Memory Maps

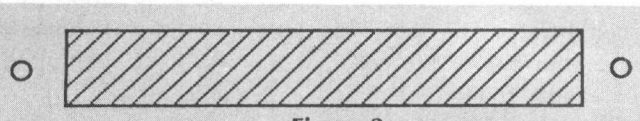


Figure 3

Template for Cut Out

Installation Outline for External Cabling

Materials Required

- Xcelite XTD-15 screwdriver with a 6-inch extension
- wooden ruler
- two cotton towels
- hand drill, ¼- and ⅛-inch drill bits
- 10-inch Mill Bastard flat file
- IC extractor or a small screwdriver to remove the 28 pin ROMs
- AMP connector 1-499970-0; AMP Ground Plane 102793-4
- paper template (see Figure 3, page 95)

Opening the Case

Disconnect the Mac from all external cables, power, mouse, keyboard etc. Place the Mac facedown on a towel to prevent marring the plastic case. Remove the five case screws with the Xcelite XTD-15 Torx screwdriver; two are located under the handle, one underneath the battery case plate, and two at the bottom of the case. Use the long edge of a wooden ruler to pry the case apart. Applying pressure to the power and I/O connectors, while lifting on the case, should release it from the faceplate. Set the foil EMI sheild aside until reassembly.

Cutting the connector slot

Using the paper template, mark the slot to be cut into the back of the Macintosh. Remove most of the plastic from the slot with a handdrill, and square off the edges with the flat file. With the slot cut, recenter the template, mark and drill the ⅛-inch holes for the 50 pin feed-thru header. Bolt the header to the inside of the case, with pin 1 on the bottom.

Removing the Motherboard ROMs

Disconnect the diskdrive and analog board cables from the motherboard; remove the motherboard from the chassis and place it on the second towel. The ROMs are in the IC sockets marked ROM HI and ROM LO. Using a chip extractor or small screwdriver, carefully remove the chip marked ROM HI. Place the ROM in the socket marked ROM HI on the Mac SCSI board, and repeat the procedure for ROM LO. Double check the orientation of the ROMs by noting that the notch on each chip points away from the 50 pin header.

Installing the Mac SCSI board

Orient the Mac SCSI board, with the 50 pin header toward the I/O connectors on the motherboard, and simply plug it into the vacated ROM sockets. Reinsert the motherboard into the chassis, taking care not to damage the 50 pin header. Reconnect the diskdrive and analog board cables, followed by the large Mac SCSI interface cable.

Closing the Mac back up

Set the Macintosh facedown and position the EMI sheild over the I/O connectors. Replace the back, making sure each side is completely seated, and secure the five holding screws.

tried this yet.

The software for MacSCSI is so far pretty simple. It consists of a routine to format the drive and a disk driver. Details of the operation of these two routines will vary a little depending upon the controller manufacturer and drive type. The Listing in this article (page 98) is for a XEBEC S1410 controller with rev D ROMs and a Seagate ST506 5Mb drive, both of which are common on the used market in Silicon Valley. As a starting point I used the RAMdisk supplied with the Aztec C Compiler release. The strategy was to use the basic RAMdisk driver as a local cache and use a simple LRU algorithm for replacement. Using the Aztec C example for the explorer desk accessory, we recoded the driver into C, for a slight loss in performance. On a 128k Mac the cache size should be set between 1 and 10. On a 512k Mac you should use some number between 30 and 300, optimally.

Other development environments and C compilers may be used as well with minor changes in the coding and installation procedures. For Aztec C the documentation on the RAMdisk and explorer desk accessory provide all the magic incantations necessary to convert the source files into an installed driver.

Questions, anyone? How large a disk, you ask? It depends in part on the software; you can reasonably handle up to about five Mb of storage on the Mac without partitioning; at ten Mb the number of files gets unwieldy. Then you'll just need more sophisticated software. Do you have to have a Fat Mac to support MacSCSI? No, but while this hard disk upgrade can work with many applications on a 128k Mac, a 512k Mac is highly recommended.

What if you don't want to take the time to do it yourself? For the lazy, the MacSCSI board is available from Fastime, P.O. Box 12508, San Luis Obispo, CA 93406 for $150.00 assembled and tested, along with machine-readable copies of the sources listed here. You can also get a complete kit, including drives from 10Mb to 110Mb, tape backup, and extended software drivers with multiple soft partitions to support the larger drives. Drivers for disk sharing and the Appletalk file server are planned or under development. Write for more information.

John L. Bass
DMS Design
P.O. Box 1456
Cupertino, CA 95014

DDJ

(Listings begin on page 98)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 198.**

```
all: clean FormatSCSI ChkSCSI MountSCSI MacSCSI BootSCSI
        echo done

clean:
        rm FormatSCSI MountSCSI MacSCSI BootSCSI ChkSCSI
        rm TestSCSI MSCSI
        rm *.bak

print:
        cat makefile > .bout
        cat newpage > .bout
        cat sl41Ø.c > .bout
        cat newpage > .bout
        cat np_fmt.c > .bout
        cat newpage > .bout
        cat np_dvr.c > .bout
        cat newpage > .bout
        cat newpage > .bout

FormatSCSI:
        cc np_fmt.c
        cc sl41Ø.c
        ln -mo FormatSCSI np_fmt.o sl41Ø.o sys:lib/mixcroot.o -lc
        cprsrc DRVR 3Ø sys:system FormatSCSI
        rm np_fmt.o sl41Ø.o

ChkSCSI:
        cc np_chk.c
        cc sl41Ø.c
        ln -mo ChkSCSI np_chk.o sl41Ø.o sys:lib/mixcroot.o -lc
        cprsrc DRVR 3Ø sys:system ChkSCSI
        rm np_chk.o sl41Ø.o

MacSCSI:
        cc -bu np_dvr.c
        cc -bu sl41Ø.c
        ln -d -n MacSCSI -I 28 -R 4Ø np_dvr.o sl41Ø.o -lc -o MacSCSI
        cp -f MSCSI TestSCSI
        cprsrc DRVR 28 MacSCSI TestSCSI
        rm np_dvr.o sl41Ø.o

MountSCSI:
        cc mountscsi.c
        ln -mo MountSCSI mountscsi.o sys:lib/mixcroot.o -lc
        cprsrc DRVR 3Ø sys:system MountSCSI
        cp -f MountSCSI MSCSI
        cprsrc DRVR 28 MacSCSI MountSCSI
        rm mountscsi.o

BootSCSI:
        cc bootscsi.c
        ln -mo BootSCSI bootscsi.o sys:lib/mixcroot.o -lc
        cprsrc DRVR 3Ø sys:system BootSCSI
        rm bootscsi.o

/*
 * sl41Ø.c version 1.Ø, July 2Ø, 1985
 *
 * MacSCSI I/O routine for XEBEC S141Ø with ST5Ø6 drive and similar
 * SASI/SCSI controllers that are pre-SCSI standard. Other controller
```

```
 * drive combinations will require changes. This version was developed
 * on the Mac under Aztec C.
 *
 * Copyright 1985 by John L. Bass, DMS Design
 * PO Box 1456, Cupertino, CA 95014
 * Right to use, copy, and modify this code is granted for
 * personal non-comercial use, provided that this copyright
 * disclosure remains on ALL copies. Any other use, reproduction,
 * or distribution requires the written consent of the author.
 *
 * Sources are available on diskette from Fastime, PO Box 12508
 * San Luis Obispo, Ca 93406 -- (805) 546-9141. Write for ordering
 * information on this and other Mac products.
 */

/*
 * SASI/SCSI Command and Sense Blocks
 */
struct scsicmd {
        char    sc_cmd;                 /* command code */
        char    sc_adrH;                /* High Byte of address */
        char    sc_adrM;                /* Middle byte of address */
        char    sc_adrL;                /* Low byte of address */
        char    sc_arg;                 /* count or interleave value */
        char    sc_vendor;              /* vendor byte -- seek algorithm */
};

struct scsisense {
        char    ss_code;                /* status code */
        char    ss_adrH;                /* address when valid */
        char    ss_adrM;
        char    ss_adrL;
};

/*
 * NCR5380 registers on the MacSCSI host adapter found at
 * location 0x500000
 */
struct NCR5380 {
        char wr_data,   filla;          /* force scsi bus data */
```

(Continued on page 101)

```
        char wr_icmd,   fillb;          /* initiator command */
        char wr_mode,   fillc;          /* ncr5380 mode */
        char wr_tcmd,   filld;          /* target command */
        char wr_sele,   fille;          /* select enable */
        char wr_send,   fillf;          /* start send operation */
        char wr_trec,   fillg;          /* start target recieve */
        char wr_irec,   fillh;          /* start initiator recieve */
        char fill00,    rd_data;        /* current scsi bus data */
        char fill01,    rd_icmd;        /* initiator command status */
        char fill02,    rd_mode;        /* ncr 5380 mode */
        char fill03,    rd_tcmd;        /* target command status */
        char fill04,    rd_bstat;       /* current bus status */
        char fill05,    rd_stat;        /* chip bus and status info */
        char fill06,    rd_input;       /* input data */
        char fill07,    rd_reset;       /* reset strobe */
};

/*
 * Phase defines        TCMD                BSTAT
 */
 #define        P_DOUT  0x00            /* 0x60 data out */
 #define        P_DIN   0x01            /* 0x64 data in */
 #define        P_CMD   0x02            /* 0x68 command */
 #define        P_STAT  0x03            /* 0x6C status */
 #define        P_MOUT  0x06            /* 0x70 Message Out */
 #define        P_MIN   0x07            /* 0x74 Message In */

/*
 * Global for drive size for use by format routine
 */
long    ScsiDrvSize = 4L * 17L * 153L;

/*
 * ScsiReset -- assume this is the only host adapter in system and do
 * a hard reset of the buss and controllers.
 */

ScsiReset() {
        long cntr;
        register zero = 0;
        register struct NCR5380 *ncr = 0x500000;

        ncr->wr_data = zero;
        ncr->wr_mode = zero;
        ncr->wr_tcmd = zero;
        ncr->wr_icmd = 0x80;
        for(cntr=0x40000;cntr>0;cntr--);
        ncr->wr_icmd = zero;
}

/*
 * ScsiCmd - Select the target controller 0, build and transfer
 * the command block.
 */
ScsiCmd(opcode,lun,blk,len,ctl) {
        struct scsicmd cmd;
        long cntr;
        register zero = 0;
        register struct NCR5380 *ncr = 0x500000;
        register char *ptr;
```

(Continued on next page)

Listing One

```
        ncr->wr_tcmd = zero;                            /* select controller */
        ncr->wr_data = 1;
        ncr->wr_icmd = 0x05;
        for(cntr=0x40000;cntr>0 && (ncr->rd_bstat & 0x40) == zero;cntr--);
        ncr->wr_icmd = zero;
        cmd.sc_cmd  = opcode;           /* build scsi command block */
        cmd.sc_adrH = lun<<5;
        cmd.sc_adrM = blk>>8;
        cmd.sc_adrL = blk;
        cmd.sc_arg  = len;
        cmd.sc_vendor = ctl | 1;        /* force XEBEC ST506 Halfstep */
        ScsiOut(6,&cmd,P_CMD);          /* send cmd to ctlr */
}

/*
 * ScsiOut/ScsiIn - transfer bytes on data bus with req/ack handshake
 * In the interest of speed we ignore edge following, the controller
 * will respond within a microsecond or so during a particular phase.
 * The rest of the loop is unfolded and optimized for the Aztec C
 * compiler to generate 9 memory references per byte. Best case would
 * be 7 memory references.
 */
ScsiOut(len,ptr,phase)
register char *ptr;
{
        register high = 0x11;
        register long low = 0x01;
        register char *i,*d;
        register zero = 0;
        register struct NCR5380 *ncr = 0x500000;

        d = &ncr->wr_data;
        i = &ncr->wr_icmd;
        ncr->wr_tcmd = phase;
        ncr->wr_icmd = 0x01;
        do {
                while((ncr->rd_bstat & 0x20) == zero);    /* sync with req */
                if((ncr->rd_stat & 0x08) == zero) break;  /* if done */
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                while((ncr->rd_bstat & 0x20) == zero);    /* sync with req */
                if((ncr->rd_stat & 0x08) == zero) break;  /* if a cmdblk */
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
        } while ((len -= 8) > 0);
        ncr->wr_icmd = zero;
        return(0);
}

ScsiIn(len,ptr,phase)
register char *ptr;
{
        register high = 0x10;
        register long one = 0x01;
        register char *i,*d;
        register zero = 0;
```

(Continued on page 104)

Listing One

```
        register struct NCR5380 *ncr = 0x500000;

        d = &ncr->rd_data;
        i = &ncr->wr_icmd;
        ncr->wr_tcmd = phase;
        do {
                while((ncr->rd_bstat & 0x20) == zero);
                if((ncr->rd_stat & 0x08) == zero) break;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
        } while ((len -= 8) > 0);
        return(0);
}

/*
 * ScsiStat - get the last two bytes to finish a command sequence
 */
ScsiStat() {
        register zero = 0;
        register struct NCR5380 *ncr = 0x500000;
        register char *ptr;
        short stat;

        ptr = &stat;
        ncr->wr_tcmd = P_STAT;
        while((ncr->rd_bstat & 0x20) == 0);
        *ptr++ = ncr->rd_data;
        ncr->wr_icmd = 0x10;
        ncr->wr_icmd = zero;
        ncr->wr_tcmd = P_MIN;
        while((ncr->rd_bstat & 0x20) == 0);
        *ptr++ = ncr->rd_data;
        ncr->wr_icmd = 0x10;
        ncr->wr_icmd = zero;
        ncr->wr_tcmd = zero;
        return(stat);
}

/*
 * ScsiFmt -- Do a default drive format (ST506)
 */
ScsiFmt() {
        ScsiCmd(0x04,0,0,12,0);         /* issue format command */
        return(ScsiStat());                     /* return status */
}

/*
 * ScsiRead and ScsiWrite -- do the I/O for a specified sector
 * and the related data.
 */
ScsiRead(sector,data)
char *data;
{
```

```
        ScsiCmd(0x08,0,sector,1,0);     /* issue read command */
        ScsiIn(512,data,P_DIN);         /* transfer in data */
        return(ScsiStat());                     /* return status */
}
ScsiWrite(sector,data)
char *data;
{
        ScsiCmd(0x0A,0,sector,1,0);     /* issue write command */
        ScsiOut(512,data,P_DOUT);       /* transfer out data */
        return(ScsiStat());                     /* return status */
}

/*
 * MacSCSI non-partitioned format routine.
 * This version was developed on the Mac under Aztec C.
 *
 * Copyright 1985 by John L. Bass, DMS Design
 * PO Box 1456, Cupertino, CA 95014
 * Right to use, copy, and modify this code is granted for
 * personal non-comercial use, provided that this copyright
 * disclosure remains on ALL copies. Any other use, reproduction,
 * or distribution requires the written consent of the author.
 *
 * Sources are available on diskette from Fastime, PO Box 12508
 * San Luis Obispo, Ca 93406 -- (805) 546-9141. Write for ordering
 * information on this and other Mac products.
 */

struct Volume {
                short   drSigWord;      /* should be $D2D7 */
                long    drCrDate;       /* date and time of initialization */
                long    drLsBkUp;       /* date and time of last backup */
                short   drAtrb;         /* volume attributes */
                short   drNumFls;       /* # of files in file directory */
                short   drDirSt;        /* first logical block of file dir */
                short   drBlLen;        /* # of logical blocks in file dir */
                short   drNmAlBlks;     /* # of allocation blocks on volume */
                long    drAlBlkSiz;     /* size of allocation blocks */
                long    drClpSiz;       /* # of bytes to allocate */
                short   drAlBlSt;       /* logical block number of first
                                                        allocation block */
                long    drNxtFNum;      /* next unused file number */
                short   drFreeBks;      /* # of unused allocation blocks */
                char    drVN;           /* length of volume name */
                char    drFill[512-37]; /* volume name & start of alloc. map */
};
struct  Volume  v;

char    zeros[512];                             /* block worth of nulls */

long    ScsiDrvSize;                    /* number of 512 byte sectors */

main() {
                char *p;
                int     i;

#ifdef hack
        printf("Scsi reset\n");
        ScsiReset();
        printf("Drive being formatted\n");
        if(ScsiFmt()) {
                printf("Format Failed\n");
                exit(1);
        }
```

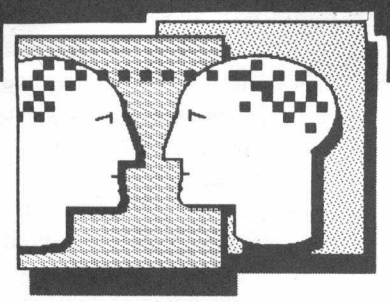
Mac Toolbox Listing (Listing Continued, text begins on page 94)

```
#endif
        v.drSigWord = 0xd2d7;
        v.drCrDate  = v.drLsBkUp = 0;
        v.drAtrb    = 0;
        v.drNumFls  = 0;
        v.drNxtFNum = 1;
        v.drDirSt   = 4;
        printf("Directory Start:      %6.6d\n",v.drDirSt);
        v.drBlLen   = 28;
        printf("Directory BlkLen:     %6.6d\n",v.drBlLen);

        v.drAlBlSt = v.drDirSt + v.drBlLen;
        printf("First Allocation Blk: %6.6d\n",v.drAlBlSt);

        v.drAlBlkSiz = 512L * (ScsiDrvSize/640L + 1L);
        v.drClpSiz  = v.drAlBlkSiz;
        printf("Allocation BlockSize: %6.6ld\n",v.drAlBlkSiz);

        v.drNmAlBlks = (ScsiDrvSize-v.drAlBlSt)/(v.drAlBlkSiz>>9);
        v.drFreeBks = v.drNmAlBlks;
        printf("Allocation Blocks:    %6.6d\n",v.drNmAlBlks);

        for(v.drVN = 0,p="MacSCSI";*p;p++) v.drFill[v.drVN++] = *p;

        printf("Initializing Drive\n");
        ScsiWrite(0,zeros);
        ScsiWrite(1,zeros);

        if(ScsiWrite(2,&v)) {
                printf("Write of config block failed\n");
        }
        for(i=3;i<ScsiDrvSize;i++) if(ScsiWrite(i,zeros)) {
                printf("Write failed at block %d\n",i);
        }
        printf("Checking Drive\n");
        for(i=0;i<ScsiDrvSize;i++) if(ScsiRead(i,zeros)) {
                printf("Read failed at block %d\n",i);
        }
        printf("Format completed ok\n");
}
main()
{
        printf("Exit code %d\n",OpenDriver("\P.MacSCSI"));
}

main()
{
        *(int *)0x210 = 5;              /* boot drive to MacSCSI */
}
```

End Listing

Due to a lack of space in this issue the listing for the Mac SCSI disk driver (np_dvr.c) will be published in the October issue.

CP/M EXCHANGE



by Robert Blum

The CP/M *Exchange* RCP/M *system is available for your use 24 hours a day, 7 days a week. Reach it by dialing (404) 449-6588.*

Goodbye Old Friend

Effective July 1, 1985 Digital Research is changing the manner in which they support their products. Some products, the more fortunate ones, will continue to receive support in varying levels, while other, older and more mature products will no longer be supported.

Product support has been broken into four levels. The highest level of support a product can have is level "A", also called priority support. It was not explained exactly what this means, but I assume that any support calls made for a level "A" product will be given priority over other calls.

Level "B" support is active support. You may encounter some delays in receiving support for level "B" products. Priority is given to questions submitted on CompuServe; next phone calls will be answered; and finally letters will be answered.

Level "C" support is limited to questions submitted on CompuServe and phone calls. Level "D" support is for mature products that are not actively supported.

What does all this mean? To the CP/M Plus user it means that you won't be able to call DRI any longer expecting to get help, because CP/M Plus has been assigned support level "D". Effectively, this appears to eliminate any further hope of buying an unconfigured copy of CP/M Plus. I doubt that dealers will continue to sell a product that isn't supported by the manufacturer. A few hardware manufacturers are still packaging it with their machines; hopefully they will also be able to support it.

Fortunately for the one million or so of us using CP/M 2.2, it was assigned to level "C" support. Some time still remains for the once most popular operating system for microcomputers. When support is finally dropped on CP/M 2.2 I doubt that any great impact will be felt. There are hundreds of reference books available for it and a good supply of heavily experienced programmers knowledgeable on the system.

AMPRO Little Board

In the May 1985 issue of *DDJ* Richard Conn reviewed the AMPRO Little Board and Bookshelf Computers. Since the electronic version of this column is running 24 hours a day on an AMPRO Little Board, I have decided to offer some impressions of this computer that I have formed as a user.

Measuring a mere 5¾-inches × 7¾-inches, the AMPRO Little Board is a complete 64K Z-80 computer on a single board. Its size, in fact, is the same as a mini-floppy disk drive; the mounting tabs on the board are spaced to match the mounting holes on any standard drive. The input/output facilities include two asynchronous serial RS-232 ports, a Centronics compatible parallel interface, and a single chip floppy disk controller capable of handling up to four 5¼-inch drives. To round out the package is a well-implemented CP/M 2.2 enhanced by ZCPR3 and a large set of utility programs.

All that remains to form a complete system is to supply a meager 5 watts of power, a CRT terminal attached to one of the serial ports, and mini-floppy disk drives of your choice. Finally, a cabinet would probably be desirable to provide protection for the delicate components.

If your need for external storage exceeds that provided by the floppy disk drives, a SASI/SCSI peer buss daughter board is available as an option. This board, along with an updated BIOS and several utility programs, will allow a hard disk to be attached to the Little Board. When used, this option can account for up to 20 Mb of additional storage.

I don't pretend to be an electrical engineer and I don't have any inside information from the people who designed the Little Board. It's apparent to me, however, that the principles of reliability and cost effectiveness were held above all others when this computer was designed.

The component density, ICs per square inch of board space, is far less than what I am accustomed to. Even though this is not documented, I believe that forced air cooling of the Little Board is not necessary if adequate air circulation and normal room temperature is maintained. Better yet, if a cabinet with forced air cooling is used, component life should not be adversely affected by temperature extremes. And, even when the board is attached directly to the bottom of a floppy disk drive, heat should not be a problem.

Serial Interface

The Z80 Dual Asynchronous Receiver/Transmitter (DART) is used to implement the two serial ports. This device is the little brother to the Z80 SIO. They are practically identical in all ways including software programming. So similar, in fact, that they can be interchanged in the same socket. The difference that exists between the two is the DART's inability to handle synchronous transmission protocols such as BI-SYNC and SDLC.

Operation and setup of the DART is handled completely through software.

Programming is accomplished by loading the desired transmission parameters into one or more of the five write registers contained on the chip. For instance, the common transmission characteristics of word length, number of stop bits, and baud rate are all completely programmable through software by loading the proper values into write registers four and five. The more advanced features, enabling interrupts and setting interrupt vectors, are handled in like manner.

Both serial channels can be programmed for a full range of baud rates. Channel A can be used up to a maximum of 38.4K baud, which to my knowledge is fast enough to drive any commercially available CRT at its maximum rate. Channel B is limited to 9600 baud, however. Nevertheless, I don't think the lower baud rate will present any practical limitations since most auxiliary devices, modems and printers, are unable to receive data faster than at 9600 baud.

The otherwise complete serial section is limited, however, by the number of signal lines that are brought out to the mating connector. In addition to the input and output data lines and two ground lines, only two control lines, one input and one output, are made available. In any other than the most simple application, a single flow control line may prove to be inadequate.

Parallel Interface

The Centronics compatible parallel interface is implemented with a single 8 input D-type latch. Here again, the number of control lines is limited. Only the peripheral device busy line is brought in for flow control. The other control lines, paper empty etc., are conspicuously missing.

Floppy Disk Controller

The floppy disk controller used on the Little Board is the Western Digital 1770. It is similar to its predecessor, the WD179X, but also contains a digital data separator and write precompensation circuity. The most noticeable change, however, is in its size; the WD1770 is completely contained in a single 28 pin package. Only a few years ago several inches of tightly packed board space were needed to implement a floppy disk control circuit. The 1770 represents a dramatic reduction in chip count and overall board complexity that should result in improved reliability.

If there is a drawback to the Little Board disk controller it is its inability to control 8-inch drives. At first I thought this would be a big problem. Then I experienced the over 800K capacity of a 96TPI format 5¼-inch disk.

Booting Up

Starting up the Little Board for the first time is a snap. After attaching the floppy disk drives with a standard 34 pin ribbon cable, all you need to do is to attach a CRT terminal set to 9600 baud to serial channel A. There are no other special requirements. Simply insert the CP/M disk into disk drive A and hit reset. The standard monitor ROM will load the operating system from disk and begin its operation. Nothing could be easier.

Once the system is up and running, you may want to change the default serial port parameters or activate an option that automatically loads a program at cold start. A reconfiguration program is provided to enable you to make these changes without going through the hassle of editing and reassembling the BIOS portion of CP/M.

Another option of the Little Board that I find most useful is the ability to reconfigure dynamically one of the four floppys to that of another computer. For example, often I need to read disks formatted on a Morrow system. To do this I simply assign drive E: to one of my physical drives and set the disk parameters to simulate the Morrow system. Any further access to drive E: now causes CP/M to respond as if it were actually running on the other computer.

Two associated utility programs are provided to use the drive E: option. One will allow you to set automatically drive E: to any of about a dozen different disk types simply by selecting it from the menu. The other prompts you for the various disk parameters used by CP/M.

A host of other utility programs are provided with the Little Board. I have found all of them to be well implemented and very useful.

User Impression

I have been using the Little Board daily for several months. And over the last three weeks it has been the main processor for my 24 hour per day Bulletin Board System. During that time the Little Board has not been down a single moment.

The only fault that I find significant enough to mention is the failure to bring all the serial and parallel control lines out to their connectors, although, in all fairness, I haven't needed them. As I mentioned earlier, I have attached a modem and a serial printer to channel B without problem.

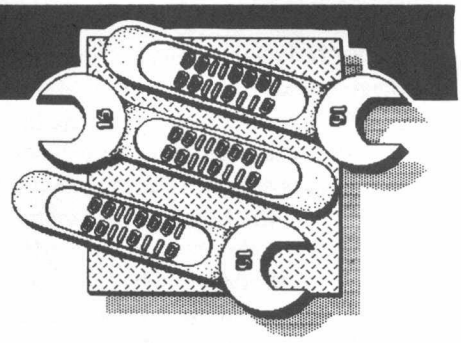
Wrapping Up

I haven't been able to finish everything that I wanted to say about the Little Board; I'm working on adding a real time clock and disk data interrupts to the BIOS. That material will have to wait until next month.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 199.**

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

The programs published in this month's column are available for downloading from the Laboratory Microsystems RBBS at (213)306-3530 (300 or 1200 baud).

Laserjet Redux

After the June 1985 16-Bit Software Toolbox appeared in print, I received some "hot and bothered" phone calls from friends at Hewlett-Packard about the tone of my comments on the Laserjet Printer. They knew very well that I consider the Laserjet to be a fantastic printer and the greatest thing since sliced bread, but they thought that readers might get an altogether different impression by reading the column. Upon reviewing the June column, I'm afraid I must agree with them.

First, the recommendation to switch to Spellbinder was strictly that of the dealer and is not an official position of Hewlett-Packard. Furthermore, Hewlett-Packard distributes a technical bulletin to its dealers with information on converting IBM PC WordStar for use with the Laserjet Printer (though the conversion doesn't offer as many capabilities as the specially patched version I got from MicroPro Technical Support; but see page 4), and the HP-150 and HP-110 versions of WordStar support the Laserjet directly. Finally, the slow speed (2–3 pages/minute) is the result of the strategy used by IBM PC WordStar for microjustification; the Laserjet is capable of 8 pages/minute when driven by other word processors.

As a gesture of atonement to Laserjet fans, this month's Listing One (page 117) is a file-printing program in Lattice C that prints two pages across in "Landscape" mode along with filenames, a time and date stamp, and page numbers.

We should note in passing that the new version of Microsoft Word offers full support for the Laserjet and for the IBM Enhanced Graphics Adaptor; between the three, it should be possible to turn out some excellent-looking documents.

Microsoft Assembler

In the June 1985 column, I also inadvertently introduced some confusion about the IBM Macro Assembler Version 2. Although *my* IBM Product Center didn't know about it, there actually was a reduced price update offer from IBM for owners of the Macro Assembler Version 1.0; it involved sending $75 and the cover sheet from your original manual to IBM, and in return they would ship you the new assembler and documentation. This update offer expired at the end of June 1985.

I have since learned that the reference section on the assembler mnemonics, which I complained had disappeared from the manual, was expanded into another volume called the *Macro Assembler Programmer's Reference*. This second manual is supposed to be delivered to you along with the operations manual and the disk, but none of the stores I've been to are displaying it with the Macro Assembler—so if you want it, you'll have to know to ask for it.

The respective version numbers used by IBM and Microsoft are also causing a lot of confusion. The IBM Macro Assembler Version 2.0 appears to be essentially the same as the Microsoft Macro Assembler Version 1.25, released over a year ago. The SALUT utility is apparently IBM's sole addition to the package. There never was a Microsoft Macro Assembler Version 2.0; Microsoft jumped directly to Macro Assembler Version 3 with its latest release in order to make the version number match the current release of MSDOS/PCDOS.

David Rabbers writes: "Microsoft MASM Version 3 is greatly worth the purchase. It not only includes a better assembler (implements the rest of the 80286 op-codes), which they claim has fewer bugs (at least some old ones are fixed and I haven't found the new ones yet), but you get two new utilities that alone are worth the $75 upgrade fee. First is a version of the Unix utility MAKE. I have never seen MAKE before, but this one seems simplistic though adequate for anything I can envision. The second utility is SYMDEB, a very good symbolic debugger. It is an enhancement to DEBUG with a natural and compatible (well, pretty close) extension to the DEBUG command syntax . . . ." SYMDEB can handle symbol tables generated by the Microsoft high-level language compilers.

Readers have reported perplexing problems when trying to convert to the IBM Macro Assembler 2.0. A number of programs that assembled and ran properly with IBM Macro Assembler 1.0 won't assemble without errors—or if they do assemble without errors, they won't run. An example of this is the statement:

```
name1 EQU (THIS BYTE)-
    (OFFSET name2)
```

This does not give an error message in itself (it assembles, strangely enough, as an equals sign followed by absolutely nothing), but it causes a cascade of phase errors throughout the rest of the program. Everything works fine when the statement is changed to:

```
name1 EQU $-name2
```

Steve Itzkowitz reported another odd experience working with the program DSKWATCH, which is available on many IBM PC bulletin boards. It appears to assemble identically with both IBM Macro Assemblers 2.0 and 1.0. However, with Microsoft Macro Assembler 1.27 and 3.0, it will not assemble cleanly, and when the errors are corrected by a trial and error process, the resulting program does not run. Other readers are invited to send in their own experiences and comments.

Detecting Intel Numeric Coprocessors

When writing 8086 applications that perform floating-point arithmetic, you should test for the presence of an Intel 8087 and exploit it if it is available. The accepted strategy for this test has been to initialize the processor, and then to write the control word into a previously zeroed memory location and examine the result:

```
fninit
mov    variable,0
fnstcw variable
...
cmp    variable,03ffh
jeq    np_present
jmp    np_absent
```

If an 8087 is present, the control word will be stored into the memory variable as 03FFH; if it is not present, the contents of the variable will not change. It is important not to use any wait instructions during this test: if the 8087 is not present, the wait will cause the 8086 simply to hang (this is why the **fninit** and **fnstcw** variations of the Intel mnemonics are used above, instead of **finit** and **fstcw**).

It turns out that programs using this strategy will fail to detect the presence of an 80287 in a PC/AT or other 80286-based system. Although the 80287 is software compatible with the 8087 in every other way, it initializes one of the bits in the lower byte of its control word opposite to the 8087. So the compare instruction in the above code sequence should be changed to

```
cmp byte ptr variable+1,3
```

Outstanding Software Value

I have been using a program called UNIFORM by Micro Solutions on my Kaypro IV for two years to read and write disks in the various incompatible CP/M 5¼-inch disk formats and have been very happy with it. It has saved me untold dollars in downloading fees. Now Micro Solutions has outdone itself with a release of UNIFORM for the IBM PC that ought to serve as a standard of quality, documentation, and ease of use for other software vendors.

Basically, UNIFORM for the PC consists of an installable system driver (UNIFORM.SYS) and a separate control and disk initialization program (UNIFORM.EXE). When the driver is linked into your operating system by including its name in the CONFIG.SYS file on the boot disk, it causes your second floppy disk drive (B:) to double as a logical drive (C:). When the drive is addressed as B:, it appears to be a normal MSDOS disk drive. But when it is addressed as C:, it almost magically makes disks in CP/M formats appear as though they were MSDOS disks!

You can do directories on CP/M disks, copy files to and from your other MSDOS format drives, erase or rename files, and in general completely ignore the fact that the disk in physical drive B: is formatted and structured in a way that is totally alien to the MSDOS operating system. In my opinion, this is a real software coup.

The separate control program is menu driven and allows you to set up the logical drive C: for some 80 different soft-sectored 5¼-inch CP/M formats, including the Epson QX-10, the Kaypro, and the DEC VT-180. In addition, if your system has a special controller for 80-track 5¼-inch drives or 8-inch drives, UNIFORM can handle that too, which gives you an even wider selection of disk formats to choose from. UNIFORM can also initialize disks in all of these different formats. This is a boon to small software houses—it allows them to centralize their distribution files on a large fixed disk hooked to an MSDOS machine, without dropping support for their CP/M customers. UNIFORM for the IBM PC costs $69.95 (it's easily worth ten times that) and is available from Micro Solutions at 125 South Fourth Street, DeKalb, Illinois 60115. Phone (815)756-3411.

Breaking with MSDOS

The MSDOS Control-Break handler, whose address is stored in the INT 23H vector, is a continual thorn in the side of application programmers. Whenever MSDOS detects a Control-C in a character stream (especially an unintentional or deliberate Control-C entered at the keyboard by the operator) during an I/O operation, it transfers directly to this handler. The default action of this handler is simply to blow the unsuspecting application out of the water . . . leaving temporary files on the disk, critical data files unclosed, critical interrupt vectors unrestored, etc. This, of course, is antithetical to the concept of a robust application, which should *always* be able to accomplish a graceful exit.

The programmer can take two approaches to the Control-C problem. The first is to disable and/or avoid Control-C detection by a variety of

methods, such as using only those I/O function calls that are not Control-Break sensitive; disabling Control-C detection during miscellaneous MSDOS operations with the MSDOS function call 33H; and putting all of the character drivers into "raw I/O" or "uncooked" mode.

A more straightforward way to beat the Control-C rap is simply to replace the MSDOS INT 23H handler with your own. On the IBM PC, you can also take over the INT 1BH vector, which is called by the ROM BIOS keyboard driver whenever a Control-Break is detected. This is easy enough for applications written in assembly language but a little more tricky for those using high-level languages. Note that the INT 23H handler is a standard MSDOS feature, and code that captures this interrupt is portable across all MSDOS systems. INT 1BH, however, is a function of the IBM ROM BIOS and will not be valid on other machines unless they are a very close IBM compatible.

Listing Two (page 119) is the source code for assembly language functions that can be linked with Lattice C applications to take control of the Control-Break interrupt handlers. This code should be readily portable to other C compilers and to assembly language applications. The function "capture" is called with the address of an integer variable within the C program; it saves the address of the variable, points the INT 1BH and 23H vectors to its own interrupt handler, and returns. When a Control-C or Control-Break is detected, the interrupt handler sets the integer variable within the C program to "True" and returns—the program can then poll this variable at its leisure. The function "release" simply restores the INT 1BH and INT 23H vectors to their original values, thereby disabling the application's interrupt handler. Listing Three (page 120) is a short C program that illustrates the use of these functions.

It is a good idea to use the MSDOS Get Interrupt and Set Interrupt function calls to inspect and modify the contents of the interrupt vectors. This will keep your application compatible with multitasking environments such

16-Bit (Text begins on page 114)

Listing One

```
/**
 **
 **      LJ.C -- A printing utility for the HP LaserJet
 **
 **      This program prints a series of files on the LaserJet
 **      printer.   The files are printed in a "landscape" font at
 **      17 characters to the inch.    To take advantage of this
 **      density, two "pages" of information from the file are
 **      printed on each piece of paper (left and right halves).
 **
 **      Usage is:       LJ  file1 file2 file3 ...
 **
 **      Where file# is a valid MS-DOS filename, included on the
 **      command line.  This program is compatible with Lattice C
 **      on the IBM PC and the HP Touchscreen computers.
 **
 **      Joe Barnhart    original version          May 5, 1985
 **      Ray Duncan      date and time stamping     May 22, 1985
 **      Joe Barnhart    revised date stamping      June 6, 1985
 **
 **/

#include <h\stdio.h>

#define MAXLINE 56              /* maximum lines per page */
#define PAGE    '\f'            /* for compilers without '\f' */
#define TAB     8               /* width of one tab stop */

typedef struct {
            int ax, bx, cx, dx, si, di;
        } REGSET;

main(argc, argv)
    int   argc;
    char  *argv[];
{
    int   filenum;
    FILE  *fp, *prn, *fopen();

    if( ( prn = fopen( "PRN:", "w" ) ) == NULL )
        printf( "Error opening printer as file.\n" );
    else {
        /* initialize the LaserJet for landscape printing */
        fprintf( prn, "\033E\033&l1O\033(s17H\033&l8d6E" );
        for( filenum = 1; filenum < argc; filenum++ ) {
            fp = fopen( argv[filenum] ,"r" );
            if( fp == NULL )
                printf( "File %s doesn't exist.\n", argv[filenum] );
            else {
                printf( "Now printing %s\n", argv[filenum] );
                printfile( fp, prn, argv[filenum] );
                fclose( fp );
            }
        }

        fprintf( prn, "\015\033E" );            /* clear LaserJet */
    }
}

printfile(fp,prn,filename)
    FILE  *fp,*prn;
    char  *filename;
{
    int pagenum = 1;

    while( !feof( fp ) ) {
        fprintf( prn, "\033&a0r85m5L\015" );    /* set left half */
        printpage( fp, prn );                   /* print page */
        if( !feof( fp ) ) {                     /* if more .. */
            fprintf( prn, "\033&a0r171m91L" );  /* set right half */
            printpage( fp, prn );               /* print another */
        }
        stamp( prn, filename, pagenum++ );      /* title */
        fputc( PAGE, prn );                     /* kick paper */
    }
}

printpage(fp,prn)
    FILE  *fp,*prn;
{
    char  c;
    int   line,col;
```

(Continued on next page)

as TopView and MS Windows. The interrupt vector table belongs to the operating system, not to the application, and it's not nice to modify memory directly that doesn't belong to you. If the fabled "Big DOS" ever comes along that runs in 80286-protected mode, your application will just get aborted if you try to twiddle the vector table without using the proper operating system services provided for that purpose.

Although in this little example program the INT 23H vector is restored by the "release" function, MSDOS will restore this vector for you automatically when your program exits. However, because INT 1BH is an IBM PC-specific interrupt handler and so is not known to MSDOS, it is *absolutely mandatory*, should your program modify this vector, that it restore the vector properly before exiting. Otherwise, the vector will be left pointing to some random area in the next program that runs—and the next time you press Control-Break, a system crash is the best you can hope for.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 200**.

16-Bit (Listing Continued, text begins on page 114)

Listing One

```
    line = col = 0;
    while( line < MAXLINE )
        switch( c = fgetc(fp) ) {
            case '\n':                          /* newline found */
                col = 0;                        /* zero column */
                line++;                         /* adv line cnt */
                fputc('\n',prn);
                break;
            case '\t':                          /* TAB found */
                do
                    fputc('\040',prn);
                while ( (++col % TAB) != 0 );
                break;
            case PAGE:                          /* Page break or */
            case '\377':                        /* EOF found */
                line = MAXLINE;                 /* force terminate */
                break;
            default:                            /* no special case */
                fputc(c,prn);                   /* print character */
                col++;
                break;
        }
}

stamp( prn, filename, pagenum )
    FILE  *prn;
    char  *filename;
    int   pagenum;
{
    char  datestr[10], timestr[10];

    fprintf( prn, "\033&a51171M" );             /* widen margins */
    fprintf( prn, "\015\033&a58R" );            /* move to row 58 */
    fprintf( prn, "File: %-133s", filename );
    fprintf( prn, "Page %-3d", pagenum);
    timestamp( timestr );
    datestamp( datestr );
    fprintf( prn, "   %s   %s", datestr, timestr);
}


datestamp( datestr )
    char  *datestr;
{
    REGSET regs;
    int  month, day, year;

    regs.ax = 0x2a00;
    int86( 0x21, &regs, &regs );
    month = ( regs.dx >> 8 ) & 255;
    day   = regs.dx & 255;
    year  = regs.cx - 1900;
    sprintf( datestr, "%02d/%02d/%02d", month, day, year);
}

timestamp( timestr )
char    *timestr;
{
    REGSET  regs;
    int  hours, mins;

    regs.ax = 0x2c00;
    int86( 0x21, &regs, &regs );
```

```
    hours = ( regs.cx >> 8 ) & 255;
    mins  = regs.cx & 255;
    sprintf( timestr, "%02d:%02d", hours, mins );
}
```

End Listing One

Listing Two

```
BREAK.ASM Control-C Handler for Lattice C


        title   Control-Break handler for Lattice C programs
        name    break
        include dos.mac

;
; Control-Break Interrupt Handler for Lattice C programs
; running on IBM PCs (and ROM BIOS compatibles)
;
; Ray Duncan, May 1985
;
; This module allows C programs running on the IBM PC
; to retain control when the user enters a Control-Break
; or Control-C.  This is accomplished by taking over the
; Int 23H (MS-DOS Control-Break) and Int 1BH (IBM PC
; ROM BIOS Keyboard Driver Control-Break) interrupt
; vectors.  The interrupt handler sets an internal
; flag (which must be declared STATIC INT) to TRUE within
; the C program; the C program can poll or ignore this
; flag as it wishes.
;
; The module follows the Lattice C parameter passing
; conventions, and also relies on the Lattice file DOS.MAC
; for the definition of certain constants and macros.
;
; The Int 23H Control-Break handler is a function of MS-DOS
; and is present on all MS-DOS machines, however, the Int 1BH
; handler is a function of the IBM PC ROM BIOS and will not
; necessarily be present on other machines.
;
        if      lprog
args    equ     6               ;offset of arguments, Large models
        else
args    equ     4               ;offset of arguments, Small models
        endif

cr      equ     0dh             ;ASCII carriage return
lf      equ     0ah             ;ASCII line feed

        pseg

        public  capture,release ;function names for C

;
; The function CAPTURE is called by the C program to
; take over the MS-DOS and keyboard driver Control-
; Break interrupts (1BH and 23H).  It is passed the
; address of a flag within the C program which is set
; to TRUE whenever a Control-Break or Control-C
; is detected.  The function is used in the form:
;
;               static int flag;
;               capture(&flag)
;

capture proc    near            ;take over Control-Break
        push    bp              ;interrupt vectors
        mov     bp,sp
        push    ds
        mov     ax,word ptr [bp+args]
        mov     cs:flag,ax      ;save address of integer
        mov     cs:flag+2,ds    ;flag variable in C program
                                ;pick up original vector contents
        mov     ax,3523h        ;for interrupt 23H (MS-DOS
        int     21h             ;Control-Break handler)
        mov     cs:int23,bx
        mov     cs:int23+2,es
        mov     ax,351bh        ;and interrupt 1BH
        int     21h             ;(IBM PC ROM BIOS keyboard driver
        mov     cs:int1b,bx     ;Control-Break interrupt handler)
        mov     cs:int1b+2,es
        push    cs              ;set address of new handler
        pop     ds
        mov     dx,offset ctrlbrk
        mov     ax,2523H        ;for interrupt 23H
        int     21h
```

(Continued on next page)

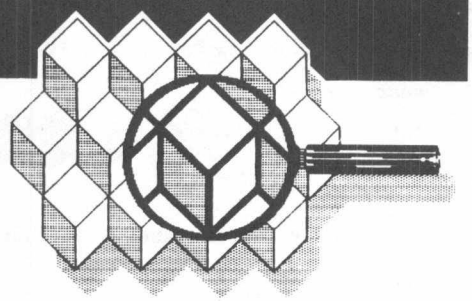
Listing Two

```
        mov     ax,0251bH       ;and interrupt 1BH
        int     21h
        pop     ds              ;restore registers and
        pop     bp              ;return to C program
        ret
capture endp


;
; The function RELEASE is called by the C program to
; return the MS-DOS and keyboard driver Control-Break
; interrupt vectors to their original state.  Int 23h is
; also automatically restored by MS-DOS upon the termination
; of a process, however, calling RELEASE allows the C
; program to restore the default action of a Control-C
; without terminating.  The function is used in the form:
;
;               release()
;
release proc    near            ;restore Control-Break interrupt
                                ;vectors to their original state
        push    bp
        mov     bp,sp
        push    ds
        mov     dx,cs:intlb     ;set interrupt 1BH
        mov     ds,cs:intlb+2   ;(MS-DOS Control-Break
        mov     ax,251bh        ;interrupt handler)
        int     21h
        mov     dx,cs:int23     ;set interrupt 23H
        mov     ds,cs:int23+2   ;(IBM PC ROM BIOS keyboard driver
        mov     ax,2523h        ;Control-Break interrupt handler)
        int     21h
        pop     ds              ;restore registers and
        pop     bp              ;return to C program
        ret
release endp


;
; This is the actual interrupt handler which is called by
; the ROM BIOS keyboard driver or by MS-DOS when a Control-C
; or Control-Break is detected.  Since the interrupt handler
; may be called asynchronously by the keyboard driver, it
; is severely restricted in what it may do without crashing
; the system (e.g. no calls on DOS allowed).  In this
; version, it simply sets a flag within the C program to
; TRUE to indicate that a Control-C or Control-Break has
; been detected; the address of this flag was passed
; by the C program during the call to the CAPTURE function.
;
ctrlbrk proc    far             ;Control-Break interrupt handler
        push    bx              ;save affected registers
        push    ds
        mov     bx,cs:flag      ;set flag within C program
        mov     ds,cs:flag+2    ;to "True"
        mov     word ptr ds:[bx],-1
        pop     ds              ;restore registers and exit
        pop     bx
        iret
ctrlbrk endp

flag    dw      0,0             ;long address of C program's
                                ;Control-Break detected flag
int23   dw      0,0             ;original contents of MS-DOS
                                ;Control-Break Interrupt 23H
                                ;vector
intlb   dw      0,0             ;original contents of ROM BIOS
                                ;keyboard driver Control-Break
                                ;Interrupt 1BH vector

        endps

        end

TRYBREAK.C Demo of Control-C Handler

/*

        Demonstrate Control-Break interrupt handler for Lattice C
        Ray Duncan, May 1985

*/
```

```
#include <h\stdio.h>

main(argc, argv)
    int   argc;
    char  *argv[];

{   int hit = 0;
    int c = 0;
    static int flag = 0;

    capture(&flag);
    puts("\nTRY has CAPTURED interrupt vectors\n");

    while ( (c&127) != 'Q')
    {   hit = kbhit();
        if (flag != 0)
                { puts("\nControl-Break detected\n");
                  flag=0;
                }
        if (hit != 0)
         {  c=getch();
            putch(c);
         }
    }
   release();
   puts("\n\nTRY has RELEASEd interrupt vectors\n");

}
```

End Listings

OF INTEREST



by Alex Ragen

IBM PC

Computers like the Compaq used to be called "portables," but that misnomer soon gave way to the more accurate "transportable" and more recently to "lugable," which is really the only way to describe those 30 pound monsters. There are now available, however, several machines that are truly portable.

Morrow introduced its Pivot II at Comdex in Atlanta. The original Pivot had a 16-line display, but the new model has a full 25-line display. There are three versions, priced between $1995 and $3795 depending on various optional features.

The big problem with portables of this type is the screen. Data General has changed screens twice on its impressive (but expensive) DG/1 in response to complaints about readability. The Pivot II uses a technology Morrow calls LumiCon, which combines the high quality of electroluminescent displays with the low cost and power requirements of liquid crystal displays (LCDs). Also a proprietary nonglare material makes the screen less susceptible to glare than the standard CRT display, Morrow claims.

The CPU is the 80C88, a CMOS version of the 8088, and the minimum memory configuration is 256K. The computer's dimensions are 13 × 6 × 9.5 inches, and it weighs only 13 pounds with two 5¼-inch drives. MS-DOS 2.11 and NewWord (a word processing program strikingly similar to WordStar) are included. The power pack and internal rechargeable Nicad battery provide more than three hours of continuous operation. There is also a 32K ROM, which includes a number of utilities like a calculator, diary, phone directory, and autodialer. For additional information, contact Morrow at 600 McCormick, San Leandro, California 94577 (415) 430-1970. **Reader Service Number 101.**

Kaypro too has introduced an 11-pound portable, the Kaypro 2000, whose price also begins at $1995. It features full IBM PC compatibility, 256K of RAM, a 25-line screen, a single 3½-inch 720K disk drive (in the minimum configuration), and WordStar and CalcStar bundled in. Its built-in battery pack provides for four hours of continuous operation. The adjustable LCD behaves like an IBM color card with a monochrome adapter. With the addition of an optional base unit (which contains a 5¼-inch drive), the machine can accept standard IBM expansion cards. Kaypro has also announced the 286i model A, a smaller version of its IBM PC/AT compatible 286i, priced at $2995 and including 512K of RAM and GW-BASIC. For further information, contact Kaypro at P.O. Box N, Del Mar, California 92014 (619) 481-4300. **Reader Service Number 103.**

Zenith's new 14.5-pound Z-171 portable PC offers full IBM PC compatibility, 256K of RAM (expandable to 640K), a 25-line flat panel backlit LCD, and two disk drives. The price is $2699. The company has also introduced the Z-200 Advanced PC, a PC/AT compatible model with some extra features, which comes in several configurations priced between $3999 for the single floppy version and $5599 for the 20 MB hard disk version. Other new products include the Z-138, a 24-pound "transportable" with a 7-inch amber display and detachable keyboard, which starts at $2099, and two additions to the Z-150 PC family: the Z-148 and Z-158. Contact Zenith Data Systems at 1000 Milwaukee Avenue, Glenview, Illinois 60025 (312) 391-8949. **Reader Service Number 105.**

You don't have to trade in your aging IBM PC/XT for an AT. Seattle Telecom & Data has announced the PC 286-MTM, a 16-bit iAPX286-based accelerator board with 2.1 MB of bank-switched (paged) memory for the XT. It can support nine consoles and is supported by Xenix, Pick, and Concurrent CP/M. The unit can also support a RAM disk driver, allowing the user to access 1.5 Mb of memory in RAM disk. The price is $1995. Contact the company at 2637 151st Place NE, Redmond, Washington 98052 (206) 883-8440. **Reader Service Number 107.**

Pfaster 286, a 80286-based add-on board that upgrades IBM PCs and PC/XTs to process data faster than an IBM PC/AT without losing the functionality of the native 8088, has been introduced by Phoenix Computer Products. The board is fully PC compatible, runs DOS version 2.0 and higher, and uses the native 8088 as a coprocessor to manage input/output. The price is $2395, with the 80287 available for an additional $350. Contact the vendor at 1420 Providence Highway, Suite 115, Norwood, Massachusetts 02062 (800)344-7200 (in Massachusetts (617) 762-5030). **Reader Service Number 109.**

A dedicated array processor, which plugs into a single expansion slot on the IBM PC, PC/XT, and PC/AT and performs Fast Fourier Transforms 100 to 10,000 times faster than software-computed FFTs, has been introduced by the Ariel Corporation. The price is $1850 (quantity one). Two or more units can be installed in one host and will work in parallel, further increasing throughput. Contact the vendor at 600 West 116th Street, Suite 84, New York, New York 10027 (212) 662-7324. **Reader Service Number 111.**

Hallock Systems, a supplier of coprocessors for Z80 systems, has introduced the Pro68, a Motorola 68000-based coprocessor board for the IBM PC. The board can be configured with between 256K and 3072K of RAM. The CPU runs at 10 MHz with no wait states, and the company says it runs eight times faster than the IBM PC and three times faster than the IBM PC/AT. Both CPM68K and OS9/68000 (a Unix-like operating system) are available, along with an editor, assembler, linker, debugger, and full K&R standard C compiler. Third party software includes all the popular high-level languages. For further information, contact the vendor at 267 North Main Street, Herkimer, New York 13350 (315) 866-7125. **Reader Service Number 113.**

Micro/Sys has introduced the BUS/BRIDGE family, which consists of 14 low-cost development packages that use the IBM PC as the host environment and provide for six target environments, including MULTIBUS (8085 and 8086), VMEbus (68000), STD BUS (Z80 and 8088), and MULTIBUS II (80286). Three software systems are supported: the target processor's assembly language, some of the IBM PC's popular compilers, and a special customized assembler and compiler. The 14 development packages consist of various combinations of the target hardware environments and software systems. Prices run from $1295 to $5595. Contact the vendor at 1011 Grand Central Avenue, Glendale, California 91201. **Reader Service Number 115.**

As everyone knows, APL on the IBM PC requires an 8087 or 80287, which (though they have come down in price somewhat in the last year) are still not exactly cheap. Also, some machines, like the PCjr, don't support the 8087, so until now APL has simply not been available for them. Enter the 8087 Eliminator, a software package from Fort's Software that functionally replaces the math coprocessor but costs only $49 ($75 for the PC/AT version). Contact the vendor at P.O. Box 396, Manhattan, Kansas 66502 (913) 537-2897. **Reader Service Number 117.**

If the PC/AT's 20 Mb disk capacity leaves you unimpressed, perhaps what you need is Reference Technology's CLASIX DataDrive Series 500, a laser disk drive with a capacity of 550 Mb, priced at $1535. The bad news is that every one of those bytes is read only. Just the thing for the Britannica or the unabridged Oxford English Dictionary but not for everyone, despite the very reasonable price. Contact the vendor at 1832 North 55th Street, Boulder, Colorado 80301 (303) 449-4157. **Reader Service Number 121.**

If 550 Mb is more than you need, Optotech has a 400 Mb WORM (Write Once Read Many times) optical disk cartridge available. Contact the vendor at 770 Wooten Road, Suite 109, Colorado Springs, Colorado 80915 (303) 570-7500. **Reader Service Number 123.**

What's a $45 billion company doing in the personal computer market? Whatever it wants to, according to some. What is that company hoping to achieve with TopView? Maybe all those royalty dollars flowing out to Bellevue every quarter are distressing a few souls in Armonk and Boca Raton, or maybe something more sinister is afoot. Who knows? But software developers are starting to bring out products with TopView in mind—although at this point TopView is mostly unfulfilled promise. Lattice has introduced the Lattice TopView Toolbasket for programmers writing applications to take advantage of TopView's multitasking window environment. There are 70 C functions to control window, cursor, and pointer operations, as well as printer control, cut and paste, and debugging functions. Memory requirements are 256K, but 512K are recommended. The price is $250 and the source code is available for $250 more. For further information, contact Lattice at P.O. Box 3072, Glen Ellyn, Ilinois 60138 (312) 858-7950. **Reader Service Number 125.**

Matrix Software Technology has introduced Synergy, a 12K operating system with Macintosh-like desktop and windows, which runs under either DOS or TopView and allows up to six programs to run concurrently in windows without modification. Synergy can also run alone, and because it requires only 12K of RAM, it will run on any PC and outperform TopView, according to the vendor, who can be contacted at 50 Milk Street, Boston, Massachusetts 02109. **Reader Service Number 127.**

Taking another approach, Quarterdeck Office Systems has introduced DESQVIEW, a new version of its floppy-based DESQ memory-resident multitasking software integrator. Priced at $99.95, it supports both keyboard and mouse. Contact the vendor at 1918 Main Street, Santa Monica, California 90405 (213) 392-9851. **Reader Service Number 129.**

The New York Amateur Computer Club has published a catalog of its PC/Blue Library: over 560 public domain programs for the IBM PC and compatibles. The catalog is priced at $5 ($7 for overseas airmail) including postage. The diskettes are $7 postpaid (North America) and $10 overseas air. Membership dues are $15 per year. Contact the club at P.O. Box 106, Church Street Station, New York, New York 10008.

Flagstaff Engineering is now distributing the DISKETTE CONNECTION and the WORD CONNECTION. The DISKETTE CONNECTION is an 8-inch stand-alone drive system with controller that allows a mini or mainframe to read from or write to a PC unit. The WORD CONNECTION is a set of software programs that provide the ability to read and write text documents from most word processing systems using an IBM PC equipped with the DISKETTE CONNECTION. All of the WORD CONNECTION programs transfer documents to a PCDOS file using IBM's Revisable Form Text standards for document architecture (DCA). For more information write Flagstaff Engineering, Box 1970 Flagstaff, AZ 86002, (602) 774-5188. **Reader Service Number 135.**

Copy-protected software is a royal pain in the ASCII, but with the help of TranSec Systems' UNlock, you can banish the pain forever, according to the vendor. The program "removes copy protection" and provides standard nonprotected DOS copies, which enable the user conveniently to

run the software from a hard disk or RAM disk. The original disk need no longer be present in drive A when the program is run. The price is $49.95. For further information, contact the vendor at 701 East Plantation Circle, Plantation, Florida 33324 (305) 474-7548. **Reader Service Number 137.**

KDS Corporation has announced its Knowledge Delivery System, an artificial intelligence program that produces expert systems of up to 16,000 production rules and 256,000 facts from up to 4,096 case histories. The fully menu-driven system communicates with the user in English. The cost is $495–$795, depending on options. Contact KDS at 934 Hunter Road, Wilmette, Illinois 60091 (312) 251-2621. **Reader Service Number 139.**

CP/M

For some time, *DDJ* readers have been writing to express concern about Digital Research's apparently waning level of support for CP/M 80. Colonial Data Systems is now making available CP/M 2.2 and CP/M Plus in the same unconfigured packages that DRI supplied directly in the past. The prices are $75 for CP/M 2.2 and $275 for CP/M Plus. Contact the vendor at 80 Picket District Road, New Milford, Connecticut 06776 (203) 355-3178. **Reader Service Number 163.**

A hand-sized computer from Britain with CP/M compatibility and up to 256K of RAM has been announced by AMS Numerics. The LCD screen can display two or four lines of 16 characters, and the keyboard is available with either 20 or 42 keys. Programs and data can be downloaded to the unit via its built-in RS232 port at rates up to 9600 baud. A parallel port allows interfacing with other electronic units, and there is also a separate bar code reader port. The unit weighs 29 ounces and operates in temperatures from 0–40°C. Power is supplied by a rechargeable, removeable battery pack, which can power the unit for up to three months at a time. Contact the vendor at Wallstreams Lane, Worsthorne Village, near Burnley in Lancashire, BB10 3PP, England. **Reader Service Number 165.**

Spectravideo has introduced an 11-pound, battery rechargeable, CP/M 2.2-based lap-sized portable computer with a 25-line display, a built-in 3.5-inch drive (360K formatted), and six pieces of bundled software (WordStar, ReportStar, CalcStar, MailMerge, DataStar, and Scheduler Plus) for a list price of under $1000. The product is scheduled for delivery in September 1985. The company also will offer three "transportable" CP/M computers and two MSDOS desktop machines. Contact the vendor at 3300 Seldon Court #10, Fremont, California 94539 (415) 490-4300. **Reader Service Number 167.**

The Private Line is a software-based DES system that can process any CP/M file at the rate of 10K per minute with a 4 MHz cpu. It will also overwrite a file with binary zeros and purge it from the directory, as well as display a file in hex format. Contact the vendor, Everett Enterprises, at P.O. Box 193, Bath, NC 27808 (919) 923-5621. **Reader Service Number 169.**

Note

In the July "Of Interest" column we stated that the EL286-88 Processor Converter from Edsun Labs allows you to replace directly an 8088 microprocessor with an 80286. A more accurate description would be to say that the EL286-88 is a VLSI component that greatly simplifies the hardware design of an 80286 upgrade board by converting 8-bit 8088 bus signals to those of the 16-bit 80286 while accomodating asynchronous clock rates.

DDJ

Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

#108 OCTOBER 1985 $2.95 ($3.95 CANADA)

The Future of Forth



Forth on a Chip

Forth on a Mac: MacFORTH & Neon

nroff Hyphenation on a Micro

MSDOS Quirks

Hobbles on Handles

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Vince Leone*
Typesetter *Jean Aring*

Production
Production Manager *Bob Wynne*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover Artist *Paul Ambrose*

Circulation
Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Kaminski*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Stephanie Barber*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*
Accountant *Marilyn Brown*
Adm. Coordinator *Kobi Morgan*

Advertising
Advertising Director
Shawn Horst *(415) 424-0600*
Systems Manager *Ron Copeland*
Administrative Manager *Anna Kittleson*
Advertising Sales
Walter Andrzejewski *(617) 567-8361*
Lisa Boudreau *(415) 424-0600*
Beth Dudas *(714) 643-9439*
Michele Beaty *(317) 875-0557*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $46 for airmail to Canada, $62 for airmail to other countries. Foreign subscriptions must be pre-paid in U.S. Dollars, drawn on a U.S. Bank. For subscription problems, call: outside of CA 800-321-3333; within CA 619-485-9623 or 566-6947.

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 686-1520 TELEX: 620430 (WUI)

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

ABC Audited Paid Circulation

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

October 1985
Volume 10, Issue 10

CONTENTS

In This Issue

Once again we present our annual Forth issue. Perhaps the most significant development in Forth over the past year has been the direct implementation of Forth engines in silicon logic. One such silicon Forth engine is the Novix NC4000 chip. In our feature article Leo Brodie describes some of the revolutionary features of this chip and the benefits that it offers to programmers.

This month we also provide a solution to the problems of developing tight, rapidly executing code for PROM-based applications written in Forth. H. Robinson, P.D. Morse and S.A. Bowhill of the Central Illinois Chapter of the Forth Interest Group have designed a post-compiler that selects from compiled, threaded Forth code the segments necessary for the execution of the PROM-based application and redirects to the RAM of the target machine any code that must be resident there.

Our review section this month also focuses on Forth, in particular, on Forth and the Macintosh. First of all, Bruce Horn examines Neon, an object-oriented programming environment that is "a cross between Smalltalk and Forth." Then, Alan Clute takes a look at MacFORTH, a development package designed to ease the writing of programs in the Mac environment.

Our columnists have added their usual monthly goodies. Dave Cortesi serves up some further tidbits from his study of MSDOS, showing how to handle ambiguous pathnames with DOS functions 4Eh and 4Fh, how to define MSDOS diskette formats and how to control Control-Z.

Allen Holub had hoped to code a version of the original TEX hyphenation algorithm in C for last month's issue, in which we reviewed two implementations of TEX for the IBM PC. Limitations on his time prevented him from realizing his intentions before deadline, but he did manage to finish off the project for this month's C Chest. So, if you want to find out how to hyphenate supercalifragilisticexpialidocious . . .

Bob Blum seems to be gleefully drowning in the deluge of new CP/M products that developers are sending to him. This month he finishes his discussion of the AMPRO Little Board, reviews SLRMAC, the 8080 assembler from SLR Systems, and offers some enticing glimpses of Date Stamper from Plu*Perfect Systems, DSD80, the debugger from Soft Advances and the latest single board computer from Micro Mint.

Finally, we continue this month in Mac Toolbox the program listings for the MacSCSI hard-disk interface presented by John Bass last month. Look for more on the MacSCSI next month.

Dr. Dobb's Journal

ARTICLES

A Threaded-Code Microprocessor Bursts Forth 42
by Leo Brodie
A discussion of the hardware features of the Novix NC4000 Forth chip and of the implications for software performance. **Reader Ballot No. 192.**

Design of a Forth Target Compiler 52
by H. Robinson, P.D. Morse and S.A. Bowhill
Members of CICFIG unveil a recompiler that generates target code for PROM applications from a host Forth system. **Reader Ballot No. 193.**

COLUMNS

Dr. Dobb's Clinic 10
by D.E. Cortesi
MSDOS and PCDOS programming notes: memory access across "chapter" boundaries; Control-Z; disk formats; ambiguous pathnames in applications programs. **Reader Ballot No. 190.**

C Chest 20
by Allen Holub
A bug in Lattice C Version 2.15; fixes to ls, the directory utility; the TEX hyphenation algorithm. **Reader Ballot No. 191.**

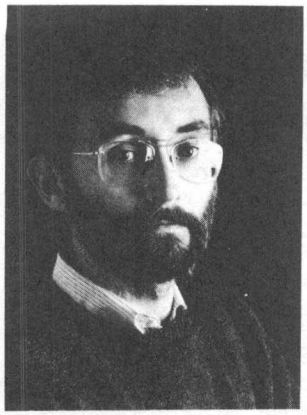
Mac Toolbox 106
by John Bass
A continuation of the listings for the MacSCSI hard-disk interface. **Reader Ballot No. 194.**

CP/M Exchange 114
by Bob Blum
The AMPRO Little Board continued; SLR Systems' SLRMAC; Plu*Perfect Systems' Date Stamper; Soft Advances' DSD80; a new single board computer from Micro Mint. **Reader Ballot No. 195.**

DEPARTMENTS

Editorial 6
Letters 8
Review 96 Kriya's Neon, Creative Solutions' MacFORTH
DDJ Classifieds 125
Advertiser Index 128

EDITORIAL



Charles Moore first created Forth to control astronomical instruments; lately his attention has been focused on the design and implementation of a microprocessor that uses Forth as its machine language. The Novix chip represents only one recent effort to put Forth on silicon. Rockwell fit a good chunk of Forth into its R65F11 chip, along with the 6502 instruction set. The British Metaforth MF16LP single-board computer, which is implemented with custom bipolar circuits, uses Forth as its machine language. Bipolar technology is also incorporated in the H4TH/X Forth engines from Hartronix, which provide a real-time system with up to 4,000 primitives in firmware.

None of these products (with the exception, perhaps, of the H4TH/20) is of much practical significance to the Forth programmer developing commercial software on and for personal computers. They are more likely to interest programmers working in areas similar to those worked on by the first Forth programmer: device control, process control, robotics. Forth on a chip can only help speed developments in these fields.

There are grounds for wondering whether the programmer who uses Forth to develop personal computer applications is in a dwindling minority. The virtues of Forth in such applications are similar to those of C. Are Forth's additional virtues the kind that will secure it a permanent place in personal computer software development? Perhaps they are, but doubts in that area may have been one of the reasons why the language Neon was developed as an extension of Forth. Even so, extending Forth to the point where it becomes a different language is not a new idea.

Forth seems in excellent health, but hardware Forths and extended Forths lead one to wonder in just what area Forth will see its greatest use in the next year, and even whether the language will soon cease to be regarded as a *high*-level language. One wonders: Where is Forth bound?

Where *DDJ* is bound is, we hope, evident from its ten-year trajectory: on a path of providing more software tools for advanced programmers. We can be more specific. Here is our 1986 editorial calendar.

January: 68000 Programming. Article deadline: past.
February: Pascal, Modula-2, Ada. Article deadline: now.
March: The Future of Programming: algorithms for parallel processing. Article deadline: 12/1/85.
April: Artificial Intelligence. Article deadline: 1/1/86.
May: At the Human Interface: designing usable software. Article deadline: 2/1/86.
June: Communications. Article deadline: 3/1/86.
July: Forth. Article deadline: 4/1/86.
August: C. Article deadline: 5/1/86.
September: Algorithms. Article deadline: 6/1/86.
October: 80286/80386 Programming. Article deadline: 7/1/86.
November: Graphics. Article deadline: 8/1/86.
December: Operating Systems. Article deadline: 9/1/86.

Plus, of course, our regular columns on C, Unix, 16-bit software, CP/M and Macintosh software development tools.

Michael Swaine

Michael Swaine



Hardware

Dear *DDJ*,

I vote for more and continuing hardware coverage, construction and otherwise. Every magazine except *BYTE* seems to be doing less and less with it. Yet, over time, Steve Ciarcia's hardware features are consistently among the most popular *BYTE* publishes.

The print spooler [in the July, 1985 issue of *DDJ*] can be simplified considerably if 6264 or similar 8K × 8 static RAM is used instead of 4164s. Seven 6264s would give 56K of RAM, leaving the top 8K of the 64K address space for ROM and the 6821 PIA. A 74LS138 ⅜ decoder could break the address space into 8K chunks and supply the enable signals for individual devices. Additional decoding would be required only in the top 8K, containing EPROM and the PIA. That could be done with a single NOR gate. Two 74LS541 buffer packages could drive A0–A12, R/W, and the enable (phase 2) clock. This buffering is desirable due to the loading imposed by the additional 3-state outputs that use of 7 6264s would involve. Software could be much simpler because of elimination of the refresh routines. System clocks would also be simplified. Total cost of construction wouldn't be much more than the spooler presented.

The S-100 real-time clock article was interesting. However, there were a few errors and omissions that deserve comment.

The use of the MM58167A's RDY line to induce WAIT states depends on rather critical timing relationships among various events. These timing considerations can be greatly reduced if an interface adapter such as the 8255 is used to control the clock chip. The only critical timing relationships that exist then are those involving the 8255's interface with the bus. Since it is much faster than the 58167 and its timing requirements more closely matched to system timing, fewer problems occur.

The POWER DOWN input needs to go low 1 microsecond before the power off only if the standby interrupt is connected to something. If this particular feature of the MM58167A clock chip is not being used at any time, the POWER DOWN input can be tied directly to +5 volt power. The 2N2222 and 2N2907 transistors and associated components can then be eliminated. In many cases the power circuit for pin 24 of the clock chip need be no more complex than two diodes, a resistor, a capacitor and a battery; however, if the clock is on a board that tends to be noisy, a more complex zener diode and transistor switch can be used there. An example of this can be seen in Steve Ciarcia's May 1982 *BYTE* article.

While the fastest fixed-interval interrupt frequency specified for the 58167 by National Semiconductor is 10 Hz, a 100 Hz interrupt can be obtained by selecting the counter-latch match (sometime called "alarm clock") interrupt address, then writing "don't care" states into the seconds/tens-of-seconds latches. A good deal of control can be obtained by writing valid comparison values into some of the other latches, and "don't care" into others. A "4" in the day-of-the-week latch and "don't care" everywhere else would get you 100 Hz interrupts all day every Wednesday. A "12" in the hours/tens-of-hours latches and "don't care" elsewhere gets you 100 Hz from 12:00:00 through 12:59:59 every day. Add the "4" in day-of-the-week and you get the 12:00:00 through 12:59:59 100 Hz interrupts every Wednesday. Other combinations of valid and "don't care" states get other patterns. For most people's purposes, this capability is, at best, somewhat interesting, but otherwise useless. For automatic data logging, however, it could be quite useful.

The GO command increments the counter if the seconds count is greater than 39, not 40, as stated on page 64.

It is frequently easier and more foolproof to use the seconds/tens-of-seconds counter reset address to start the clock at a known time. I think maybe the GO command was either dreamed up because there was an address that wasn't otherwise used, or else it happened by accident. A goof in a mask, perhaps? A bug that became a feature?

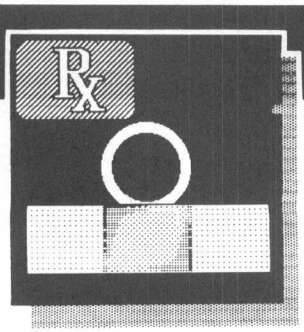
On page 62 it is stated: "Each January . . . the clock IC does not automatically change to a new year." While true, it is also misleading: the 58167 doesn't have years/tens-of-years counters (or latches). Any changing of the year gets done somewhere other than the 58167 clock circuit.

Someone who wanted to make a clock board that does have years counters could use an OKI MSM5832, National Semiconductor MM58274, or Motorola MCM146818 or MCM146819 or any other clock chip that tracks years.

Sincerely,
Frank Kuechman
8113 NE 25th Avenue
Vancouver, WA 98665

DDJ

DR. DOBB'S CLINIC



by D.E. Cortesi

Here's a batch of small items on MSDOS and PCDOS. The first is based on a note from Nick Hammond of Torrens, Australia. There, in a magazine called *Your Computer* and in a column not unlike this one, appeared the plaint of a puzzled user. Sometimes, he said, his everyday programs would misbehave strangely. DEBUG would bomb or fail to write a good file; FORMAT would loop or report track zero bad when it wasn't; and so forth. The only common factors were that the failures occurred when there were resident programs in the system and that they could be fixed by booting without the resident programs. Sometimes they could be fixed merely by changing the order in which resident programs were loaded.

Dismal Memory Access

Hammond wrote to explain this puzzle. When IBM designed the PC, he tells us, they fitted a Direct Memory Access (DMA) chip from an 8-bit family. It can address only 64K. To accommodate the 1Mb address space of the 8088, they tacked on an external "page register" to establish which of 16, 64K "pages" the DMA chip would use. Because an 8088 "paragraph" is 16 bytes and a "page" is usually 256 bytes, we might do better to call these 64K DMA blocks, "chapters."

What we must never call them, Hammond says, is "segments." Unlike the 65K segments the 8088 can address, which may begin at any paragraph boundary, there are only 16 DMA chapters, and they fall exactly on the 64K boundaries of storage.

As a result, the DMA chip is physically incapable of transferring a block of data that crosses a chapter boundary in a single operation. "You have to transfer as far as the boundary, change the page register, then transfer the rest," says Hammond. "This would be fine if the firmware in the BIOS ROM handled the housekeeping, but it doesn't."

For example, if you use BIOS interrupt 13h to access a floppy disk and your data area crosses a chapter boundary—if any byte in your buffer except the first has an address that is divisible by 65,536—the BIOS will return an error. Programs that don't specifically check for this condition will fail intermittently, depending on their load address, and that depends on what resident programs were loaded ahead of time.

It's our impression that I/O done through MSDOS does not suffer this constraint—just one more reason to avoid ROM calls.

Control That Zee!

MSDOS function 40h writes up to CX bytes to the file named by the "handle" in BX and returns in AX the number of bytes written. Ordinarily, all the bytes will be written and AX will equal CX afterward.

Usually the reason DOS will report writing fewer than CX bytes is that the output disk is full and there is no place to put them. That's a disastrous error from which a program has few courses of recovery.

We recently discovered another cause—one that isn't documented so far as we are aware. If the output data include a logical end of file byte (Control-Z), and if BX names the standard output handle (0001), and if standard output has *not* been redirected to a file, DOS will stop writing at the Control-Z and return the count of bytes that preceded it.

This behavior makes sense in a way; it makes it easy to copy a disk file to the screen without displaying the superfluous garbage that may follow logical end of file. On that account, we'd expect DOS to behave the same whenever the target file was implemented by a character device driver, whether it was named by a standard handle number or not. We didn't test to see if it was that consistent, or whether it only applied the rule to the handles for the standard output, print, and auxiliary files, or whether it was just a glitch on handle 0001, or whether it varied with the device driver.

The more consistent DOS is, the worse the problems are. How is a program supposed to distinguish a full disk from an incidental Control-Z? The only way we can think of is to use the ioctl function to get the attributes of the file's device driver, and see if it's a block device or a character one. Wonderful.

One program that doesn't check is the MS Pascal 3.0 runtime library. If a Pascal program copies from standard input to standard output and if input is an edited keyboard line including Control-Z and output is the screen, it will abort with a disk-full error.

Medium Smart

Further gleanings from our study of MSDOS can be found in the Table (page 12), a compilation of the numbers that define MSDOS diskette formats. We pieced it together from IBM and Microsoft manuals.

The second line of the table is headed "format flag-byte." That's what we call the first byte of the FAT, the same byte that Microsoft and the illiterati of Boca Raton persist in calling the "media descriptor," thereby driving us straight up the wall. The word "media" is *plural*, dammit, and is in no way a synonym for "disk." The recording *medium* used with dis-

kettes is an iron or chromium oxide in a plastic binder, but the named byte doesn't describe it.

Whatever you call it, that byte is a poor guide to disk layout. The same flags can be found on 5- and 8-inch disks of very different formats, and the flag F9h appears on both 80-track diskettes and fixed disks. There is a better indicator: DOS functions 1Ch and 36h both return in register DX the total number of allocation units on a disk. Read across the third-from-last row of Table 1; you'll see that each format has a unique number of allocation units.

Hear No Evil . . .

. . . is the policy of Microsoft regarding MSDOS. We found a bug in DOS and wrote it up in these words: "DOS function 43h, AL=00 (return file attributes of path *DS:DX) returns impossible values and no error in the carry flag when given the path 'C:\.' Under DOS 3.0, despite absence of a carry error signal, function 59h returns partly-correct, partly-contradictory information, suggesting an error was recognized." We attached debug output to prove these points and mailed it all to Seattle.

It was a waste of time. Microsoft is not interested in trouble reports from civilians; they only talk to companies. Or so said Doug Boa of Microsoft in an earnest reply. "Microsoft does not support either of our operating systems (MSDOS and Xenix) directly," he wrote. "We do not sell them to the public, but only to the equipment manufacturers . . . [who] are responsible for all support functions." Therefore, Boa said, "even though you are reporting a problem with a function call general to all MSDOS implementations, you still need to direct the report to the equipment manufacturer."

We like to think we can read a tone of honest regret in those lines. If we wrote an OS, it would just *kill* us to have to turn back trouble reports because we'd know that, first, the people who report them are more likely to give up than to submit them a second time to their OEMs, and second, the OEM is all too likely to ignore the reports, lose them, reject them out of ignorance (or habit), or just sit on them for a few months before passing them on.

So let's give a moment's sympathetic thought to the Microsoft developers, sitting around wistfully hoping that somebody will write them a letter they'll be allowed to read. But if you find a bug in MSDOS or Xenix, save your time and postage. We might add that the sysops of the Microsoft forum on CompuServe will also reject any technical queries or trouble reports on MSDOS or Xenix.

Narrow Paths

MSDOS permits a program to control files entirely in terms of compact strings of characters called pathnames. Having to convert the name of a file from a compact, punctuated string like C:DROSS.DAT to the fixed fields of a file control block and back again was a major irritation of file work in CP/M and early MSDOS. With pathnames, programs talk to DOS in the same terms with which users talk to programs. That should make things easier.

And it does, so long as you work only with single files with explicit names. Get what should be a pathname from the user and terminate the input string with a null byte; point to the string and tell DOS, "Open it." DOS takes care of case conversion and of parsing the string into its many parts: drive letter, directory names, filename, extension. If the path describes anything that DOS can accept as a file, your program gets back a numeric "handle" and is off to the races. If it doesn't, DOS returns either error 2, "file not found," or error 3, "path not found." The first means that the disk letter (if any) and directory names (if any) were valid but the filename and extension couldn't be found in the (last-named or default) directory. Error 3 means that something went wrong before the filename and extension could be checked. We should be delighted that DOS can handle all this complexity for us, because it would be a real punctuation (a real #$?!%$*&!) to have to program it.

It's as easy to erase, create, or rename any single file from its pathname, or set its attributes or timestamp, as it is to open one.

Winding Paths

Complications arise when you want to handle ambiguous pathnames (ones in which wild cards are used to represent multiple files), especially if you want your program to be as flexible as DIR or COPY.

Let's create an example. Suppose that on drive B you have a diskette with a small hierarchy of files:

```
top.dat
mydir
   inner1.dat
   inner2.dat
```

	5 1/4-inch					8-inch	
Nominal capacity	160K	180K	320K	360K	1.2M	250K	1.2M
Format flag-byte	FEh	FCh	FFh	FDh	F9h	FDh	FEh
Entries in root dir	64	64	112	112	224	68	192
Bytes per sector	512	512	512	512	512	128	1024
Sectors per cylinder	8	9	16	18	30	26	16
Cylinders per disk	40	40	40	40	80	77	77
Total sectors	320	360	640	720	2400	2002	1232
Reserved sectors							
for bootstrap	1	1	1	1	1	4	1
for FAT copies	2	4	2	4	14	12	4
for root dir	4	4	7	7	14	17	6
total reserved:	7	9	10	10	29	33	11
Available sectors	313	351	630	708	2371	1969	1221
Sectors per unit	1	1	2	2	1	4	1
Number of units	313	351	315	354	2371	492	1221
Bytes per unit	512	512	1024	1024	512	512	1024
Data capacity (Kb)	156.5	175.5	315.0	354.0	1185.5	246.5	1221.0

Table

Format data for a number of MSDOS diskette layouts. The "number of units," returned in DX by DOS function 1Ch, can be used to distinguish between layouts.

And suppose that at some time you moved to drive B and made MYDIR the current directory:

```
A>b:
B>cd mydir
B>a:
```

That, by the way, is a point that's poorly explained by the DOS manuals we've seen. They don't make it clear that *each* drive has its own current directory. A reference to a file by drive, name and extension, as in the example

```
A>type b:inner1.dat
```

means "look up the file named INNER1.DAT in the current directory of drive B." It doesn't mean the *root* directory of drive B; that would be

```
A>type b:\inner1.dat
```

Nor does it necessarily mean in the directory last named to CD; that might have been a new current directory for another drive.

Now consider all the ways that you can refer to the files on drive B using DIR. You can ask about all files by not supplying any operand,

```
A>b:
B>dir
```

which will list all files in the current directory. You can do the same, when another drive's the default, by naming the drive.

```
A>dir b:
```

You can ask for all the files in the *root* directory of the disk by specifying "root" with a backslash:

```
A>dir b:\
```

By convention of DIR and COPY such a reference to a directory is implicitly a reference to all the files in that directory. It saves your having to type *.*, as in

```
A>dir b:\*.*
```

or the equivalent double-dot backreference,

```
A>dir b:..\*.*
```

You may refer to files in any directory by naming the directories along the path to the files,

```
A>dir b:..\mydir\..\top.dat
```

including redundant ones, as in that example.

Primrose Paths

There are a lot of useful programs with specifications that start off like this: "for every file that matches the first operand, the program will . . ." For consistency, such programs ought to interpret their command operands just as generally as DIR does. Anything less will just confuse the users: they'll come back complaining "DIR finds this file, why doesn't your program?"

The key to handling ambiguous pathnames lies in the DOS functions 4Eh and 4Fh, search for first and next match to path, respectively. Point to a pathname string that may or may not be ambiguous and call DOS; if there's (another) file that matches the pathname, DOS drops information about it into the current data transfer area. One returned item is the filename and extension, formatted as a string.

Unfortunately, functions 4E/4F aren't as general as DIR is. They do not assume *.* after a directory or drive letter, for instance.

Even when they are successful, they leave you with a riddle: how can you *open* the file you've discovered? You have the pathname you used for the search, a user-entered string like b:\mydir*.dat, for example. You have the filename and extension returned by DOS, INNER2.DAT, for instance. Now you'd like to do something to the file you've found—open it or erase it or something—but you don't have enough information. You can't just point to the filename string and request an open. The file may reside in a disk or directory other than the current default, and DOS won't find it. The necessary qualifiers are in the search string, but you can't get at them without parsing the string. Parsing pathnames was the job that we so gladly turned over to DOS.

As you already suspect, we wouldn't be going on about this if we didn't have a solution. We found that we could use DOS function 4300h (return attributes of file using pathname) to tell the type of path a user has provided. With fair reliability, the user's path specification can be processed into two versions: a search path for use with functions 4E/4F, and a lead-in path to be prefixed to the filename string that those functions return.

The logic of it is explained at tedious length in the comments of Listing One (page 16), the source of a C function that does the job. This function, **fixpath()**, would be called once to initialize a path provided by the user. If it returns a zero, there are no files to be found down that path. Otherwise it has made two new paths, a search path and a prefix path. If the input path was, say,

```
b:..\
```

("all the files in the parent of the default directory of drive B"), **fixpath()** will return a search path of

```
b:..\*.*
```

and a prefix, or lead-in, path of

```
b:..\
```

However, if the input string was

```
b:\mydir
```

it would return a search path of

```
b:\mydir\*.*
```

and a lead-in path of

```
b:\mydir\
```

The search path is for input to functions 4E/4F. If function 4E returns an error, the original path was ungrammatical somehow. Otherwise, the search path will return the same filenames that DIR would print if it were given the same input path you gave to **fixpath()**. When you append one of these returned filename strings to the prefix path set up by **fixpath()**, you have a complete pathname to give to DOS functions 3Dh (open), 41h (erase), 43h (get/set attributes) or 56h (rename).

DDJ

Dr. Dobb's Clinic Listing (Text begins on page 10)

```
/* ============================================================
   fixpath() processes a DOS pathname for two different uses.
The input path, *ip, is presumably a command operand like the
first operand of DIR.  One output, the search path *sp, is
the input possibly augmented with "*.*" or "\*.*" so that it
will work reliably with DOS functions 4E/4F.  The other output
is a lead-in path, *lip, which can be prefixed to the simple
filename.ext returned by functions 4E/4F to make a complete
path for opening or erasing a file.
   The function returns 1 if it is successful, but 0 if the
input path can't be processed and should not be used.
   Some input paths can be processed here yet be invalid or
useless.  The search path produced from such an input will
cause an error return from function 4E (search first match).
============================================================ */
int fixpath(ip,sp,lip)
register char
        *ip, /* the input path */
        *sp, /* the search path */
        *lip; /* the lead-in or prefix path */
{
char *cp; /* work pointer for scanning paths */
/* ============================================================
   Pick off 4 special cases:
        (1) the null string, which we take to mean "*.*"
        (2) the simple "d:" reference to a drive, which we
            also take to mean "d:*.*"
        (3) the root-dir reference "d:\" which is mishandled
            by function 4300 of both DOS 2.1 and 3.0.
        (4) any reference that ends in "\"
In all cases, the input path is ok as a lead-in, but it needs
the global qualifier *.* added for searching.
============================================================ */
        if ( (strlen(ip)==0)            /* null string */
           ||(strcmp(ip+1,":")==0)      /* d: only */
           ||(ip[strlen(ip)-1]=='\\') ) /* ends in bkslsh */
        {
                strcpy(lip,ip);         /* input is ok prefix */
                strcpy(sp,ip);
                strcat(sp,"*.*");       /* add *.* for search */
                return(1);
        }
/* ============================================================
   Ok, we have a non-null string not ending in \ and not a lone
drive-letter.  Four possibilities remain:
        (1) an explicit file reference,  b:\mydir\mydat
        (2) an explicit directory reference, \mydir
        (3) an ambiguous file reference, *.* or b:\mydir\*.dat
        (4) an unknown name, a:noway or b:\mydir\nonesuch.dat
We can separate this with fair reliability using DOS function
4300h, get attributes from path.

   The DOS operations used here are based on the MS-C 3.0
library functions, but all C compilers have similar library
routines and header files.
============================================================ */
        regs.x.ax = 0x4300;     /* AH=43h, AL=00h */
        (char *)regs.x.dx = ip; /* DS:DX -> input path */
        regs.x.cx = 0;          /* clear CX */
        intdos(&regs,&regs);    /* call DOS */

        if ( (regs.x.cflag) && (regs.x.ax==0x0002) )
        {
        /* ------------------------------------------------------
           The path is valid, in that all directories and drives
        named in it are valid, but the final name is unknown. No
        files will be found in a search, so quit now.
        ------------------------------------------------------ */
                return(0);
        }

        if ( ((0!=regs.x.cflag)&&(regs.x.ax==0x0003))
           ||((0==regs.x.cflag)&&(0==(regs.x.cx & 0x0010))) )
        {
```

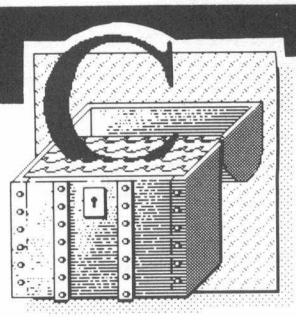
(Continued on page 18)

 (Listing continued, text begins on page 10)

```
        /* ======================================================
           Error 3, path not found, could mean total junk or a
        misspelt directory name, but most likely it just means
        the path ends in an ambiguous name.  If there's an error
        the initial search (function 4Eh) will fail.
           With no error and reg cx NOT showing the directory
        attribute flags 0010, we have a normal, unambiguous file
        pathname like "b:\mydir\mydat" or just "myprog.com".
           In either case the search path is the whole input
        path.  The lead-in is that shorn of whatever follows the
        rightmost \ or :, whichever comes last -- or just a null
        string if there is no \ or :.
        ====================================================== */
                strcpy(sp,ip); /* working copy of ip in sp */
                cp = sp+strlen(sp)-1;
                for(; cp > sp; --cp)
                        if (('\\'==*cp)||(':'==*cp)) break;
                if (cp>sp) ++cp; /* retain colon/slash */
                *cp='\0'; /* may make a null string */
                strcpy(lip,sp);
                strcpy(sp,ip); /* whole input for search */
                return(1);
        }

        if ((0==regs.x.cflag)&&(regs.x.cx & 0x0010))
        {
        /* ======================================================
           No error and the directory attribute returned in cx
        shows an unambiguous path that happens to end in the
        name of a directory, for instance "..\..\bin" or
        "b:\mydir"  Applying the rules of DIR or COPY, we
        have to treat these as global refs to all files named
        in the directory.  The search path is the input with
        "\*.*" tacked onto it.  The lead-in path is just the
        input plus a backslash.
        ====================================================== */
                strcpy(sp,ip);
                strcpy(lip,ip);
                strcat(sp,"\\*.*");
                strcat(lip,"\\");
                return(1);
        }
        else
        { /* unexpected events make me nervous, give up */
                return(0);
        }
}
```

End Listing

C CHEST



by Allen Holub

This month we're going to look at an implementation of Knuth's hyphenation algorithm. This code is yet another part of my version of the Unix text formatter, nroff.

Bugs and Fixes

Before getting started, here's a little miscellany. First, while bringing up a version of **printf( )** the other day, I found a bug in Version 2.15 of Lattice C (though the developer would probably argue that it's a feature).

When writing a subroutine with a varying number of arguments of varying types (like **printf**), arguments are fetched from the stack frame by casting a "generic" pointer to point at an individual object on the stack. After fetching the argument, the pointer is advanced past it. In other words, we want to cast a character pointer into a long pointer, fetch a long object off the stack, and then advance the character pointer by the size of a long. The first thing I tried was:

```
char *p ;
long x ;

x = *(long *)p++ ;
```

but this won't work correctly. Here, the problem is order of precedence. *, **(long *)**, and **++** are all at the same precedence level, but they associate right to left. When fully parenthesized, the associativity gives us:

```
x = *((long *)(p++))
```

Because the **++** binds more tightly to **p** than the cast, the compiler will assume that **p** is a character pointer (rather than a long pointer) for the sake of the increment, though it will fetch the long object correctly. In other words, the expression evaluates to a long, but 1 (rather than 4) is added to the pointer. I tried to force the precedence to behave by doing:

```
x = *( (long *)p )++ ;
```

but now Lattice gives me an error 25 (lvalue required) and won't compile it. By itself, **((long *)p)++** doesn't compile either. Because this statement indeed has an lvalue (the **x**), I was rather nonplused. I think that the compiler was confused by the cast, thinking that I was trying to do something such as **x++ = y** or:

```
int foo;
(long)foo = 5L;
```

which won't work because this would force **foo** to overflow (only an integer's worth of space is reserved for **foo**, and we're trying to squeeze a long object into it). However, **((long *)p)++** should work because we're just modifying the way that the pointer arithmetic works, not overflowing the target. A more clever error detection algorithm would take care of this exception. The following, though inelegant, does work:

```
char *p;

x = *(long *)p ;
p += sizeof( long );
```

While we're on the subject of bugs, several people have written about what we thought was a bug in **ls**, the directory utility printed a couple months back. The problem here is underlining. On a Compaq, directory names comes out at half intensity instead of underlined. The problem here is the hardware (and I contend that this is a bug, not a feature). The Compaq emulates an IBM color card running a monochrome monitor. The color card doesn't support underlining (I think that it should) but prints text in a different color rather than underlining it. On a monochrome display, this second color comes out as reduced intensity. All the IBM cards, color or otherwise, should support the attributes specified in the ANSI.SYS documentation. Anyway, the Paradise Modular Graphics card has the same problem as the Compaq. On the other hand, the Hercules Graphics card does underlining correctly.

There's a second bug in **ls** that's a little more serious. **Ls**, as it stands, runs fine under DOS Version 3.0. However, when running under Version 2.x, it couldn't find the root directory (\) when it was explicitly requested by an **ls ** command. A version-independent fix for the problem is given in Listing One (page 26).

Hyphenation

Moving on, the capability to hyphenate words is desirable in any text formatter or word processor. A lot of needless white space between words can be eliminated by hyphenation. Narrow columns look better, and we can get more words per page. Looking at the output of several commercially available word processors, I suspect that most of them derive their algorithms from the one given by Knuth in *T_EX and Metafont*, and I've used his algorithm here.[1]

Hyphenation is a harder problem than you would think at first. To use Knuth's examples, why is *record* hyphenated *rec-ord* when used as a noun and *re-cord* when as a verb? Consequently, it's impractical to attempt a truly universal algorithm—there are just too many exceptions. Knuth's algorithm is intentionally conservative—if it isn't absolutely sure how to split two syllables, it doesn't split them.

Algorithm Description

Knuth attacks the problem by breaking it into several parts, the first of which is exception removal. He keeps a dictionary of words that are known exceptions to his other rules and then sees if a word is an exception before processing it further. Strictly speaking, we could use a huge dictionary and dispense with the rest of the algorithm, but this would be too cumbersome to be practical. Knuth includes a dictionary of about 300 words in his book, some of which are rather useless. (How many times do you use "coneflower" in an average document?) There just wasn't room to print the dictionary look-up part of the hyphenator this month. It's a straightforward program—it reads a sorted list of exceptions and then does a simple binary search in a table.

The second part of the algorithm does suffix removal. The following suffixes are recognized:

> *-able, -ary, -cal, -cate, -cial, -cious, -cient, -dent, -ful, -ize, -late, -less, -ly, -ment, -ness, -nary, -ogy, -rapher, -raphy, -scious, -scope, -scopic, -sion, -sphere, -tal, -tial, -tion, -tional, -tive,* and *-ture.*

Some of these are qualified

> *-able* must be preceded by *e, h, i, k, l, o, u, v, w, x, y, nt,* or *rt*; *-ary* must be preceded by *ion* or *en*; *-cate* and *-late* must be preceded by a vowel (*a, e, i, o, u,* or *y*); *-nary* may not be preceded by *e* or *io*; *-cious* may not be preceded by *s*; and *-ize* must be preceded by *l.*

Another complication is *-ing.* Knuth says if *-ing* is preceded by fewer than four letters, don't insert a hyphen. If *-ing* is preceded by two identical consonants (other than *f, l, s,* or *z*) put the hyphen between the consonants, and if it's preceded by any letter but *l*, break as *-ing.* If the letter before *-ling* is *b, c, d, f, g, k, p, t,* or *z*, break before this letter (but do not break a *ck* combination before *-ling*); otherwise break *-ing.*

If an *-able, -ary, -ful, -ize, -less, -ly, -ment,* or *-ness* is recognized, then the algorithm is applied again to the residual word (the word less the suffix we just found). Similarly, if we fail to find a suffix and the word ends in *s, e,* or *ed*, we try again with the *s, e,* or *ed* removed.

Having removed suffixes, we now try to remove potential prefixes, working on the word without all the suffixes just recognized. The following prefixes will be recognized:

> *be-, com-, con-, dis-, equi-, equiv-, ex-, hand-, horse-, hy-per-, im-, in-, in-ter, in-tro-, lex-i-, mac-ro-, math-e-, max-i-, min-i-, mul-ti-, non-, out-, over-, pseu-do-, quad-, semi-, some-, sub-, su-per-, there-, trans-, tri-, un-der,* and *un-.*

The intermediate hyphens aren't inserted unless the entire prefix is found (e.g., hy-per-bole vs. hyp-notic). There are also exceptions here:

> *Be-* must be followed by *c, h, s,* or *w*; *dis-* may not be followed by *h* or *y*; *equi-* may not be followed by *v*; *trans-* must be followed by *a, f, g, l,* or *m*; *tri-* must be followed by *a, f,* or *u*; *un-* may not be followed by *der* or *i.*

As with suffixes, the algorithm is applied repeatedly after *dis-, im-, in-, non-, over-,* or *un-* is recognized.

At this juncture, the word has been partitioned into three parts: prefix, root, and suffix. We now apply several consonant-pair rules to the root portion of the word. Here, vowels are *a, e, i, o, u,* and *y*, and consonants are all other single letters. Moreover, the pairs *ch, gh, ph, sh,* and *th* are treated as single consonants. If a vowel-con-

Prefixes:

0	Failure, return original pointer
1-68	Get another character and branch to next state as indicated in table.
69	Get a character, hyphenate ***p**, branch to next state using ***p** as the current character.
70	hyphenate ***(p–1)** and branch as above.
71-80	don't exist.
81	hyphenate ***(p–1)** and return **(p–1)**.
82	hyphenate ***p** and ***(p–1)**, return **p**.
83	hyphenate ***p** and ***(p–2)**, return **p**.
84	hyphenate ***p** and ***(p–3)**, return **p**.
85	hyphenate ***(--p)**, don't return.
86	hyphenate ***p**, dont't return.
87	hyphenate ***p** and return **p**.

Suffixes:

0	Failure, if the word ends in *s, e* or *ed* strip it off and try again, else return original string pointer.
1-57	Get another character and branch to next state as indicated in table.
58-79	Don't exist.
80	**p += 1**, hyphenate ***(p+1)**.
81	**p += 2**, hyphenate ***(p+1)**.
82	**p += 3**, hyphenate ***(p+1)**.
83	hyphenate ***(p+1)**.
84	Failure, return original pointer.
85	Doesn't exist.
86	process an -ing.
87	hyphenate ***(p+1)** and return **p**.
88	hyphenate ***(p+2)** and return **(p+1)**.
89	hyphenate ***(p+5)** and ***(p+1)**, return **p**.

In both tables, unspecified transitions go to state 0. Unmarked transitions always go to the indicated state, regardless of input character. **P** points at the character being processed at present.

Figure 1
States and Associated Actions

sonant-consonant (VCC) pattern is found and both consonants are the same and neither consonant is an *l* or *s*, the break will be between the consonants. In the case of a *vowel-ll* or *vowel-ss* pattern, there will be a break only if the following letter isn't a vowel and the word doesn't end VCC*er* or VCC*ers*. If the pattern is *vowel-ck*, the break will be after the *ck*. If the pattern is *vowel-qu*, the break will be before the *qu*.

If a VCCV combination is found, the break will be between the consonants unless they are one of the following combinations:

bl, *br*, *cl*, *cr*, *chl*, *chr*, *dg*, *dr*, *fl*, *fr*, *ght*, *gl*, *gr*, *kn*, *lk*, *lq*, *nch*, *nk*, *nx*, *phr*, *pl*, *pr*, *rk*, *sp*, *sq*, *tch*, *tr*, *thr*, *wh*, *wl*, *wn*, or *wr*.

Finally, don't break between consonants if the word ends with VCC*er*, VCC*ers*, VCC*age*, VCC*ages*, VCC*est* and the two consonants are either *ft*, *ld*, *mp*, *nd*, *ng*, *ns*, *nt*, *rg*, *rm*, *rn*, *rt*, or *st*.

Once the word is hyphenated, Knuth goes back and fixes obvious mistakes. He takes back very short syllables and syllables ending with a silent *e*. I decided to use the exception dictionary to do this rather than adding more code to the algorithm proper. I should say that Knuth is not his usual lucid self when describing this part of the algorithm. A major factor in my not implementing the "short end" retention is that I can't, for the life of me, figure out what he was saying. If anybody can decipher this part of the algorithm description, please enlighten me.

Implementation

The code this month (Listing Two, page 26) supports the suffix removal, prefix removal, and consonant-pairs part of the algorithm. The Lattice compiler produces a 9063-byte object module (which seems a little bloated to me, though the code could be optimized a bit more if space is a problem). Hyphens are marked by setting the high bit of the character that will follow the hyphen (the first letter in the syllable). Only unhyphenated words composed of lower case letters will be processed. Among other things, this lets us avoid the problems associated with proper nouns, which shouldn't be hyphenated.

All global variables and all subroutines that don't need to be accessed from another module are declared static. The one externally accessible routine is the driver, **hyphen**( ) (on line 484), which calls the various other processing routines.

The suffix/prefix removal routines use table-driven state machines for pattern recognition. As usual, there is a trade-off between space and execution time. The fastest state machines use a large two-dimensional array in which one axis is the current state and the other the current input character. The table itself holds the next state. As most of the states have only one legal transition, this method uses an unnecessarily large amount of memory. At the other extreme, we can keep a list associated with each state that contains all legal input characters and the state to go to when that character is encountered. This method usually takes less space, but we have to search the list every time we process a character so execution time isn't great when the lists are too long.

The tables used here are a compromise between these two methods. Each table is an array of pointers to character arrays. The tables are indexed by current state. If there's only one legal transition out of a given state, the first byte of the associated array is 1, the next byte is the one legal input character, and the third byte is the state to go to when that character is found. All other input characters will cause a transition to state 0. The second type of array is used when there is more than one legal transition out of a state. In this case, the first byte of the array is 0 and the next 26 bytes are an array of next states indexed by current input character. The state tables begin on line 605 of the listing.

The state diagrams for these tables are shown in Figure 2 (page 24) and Figure 3 (page 25). In both machines, state 0 is a failure state. States with numbers greater than or equal to 80 are success states. These are the only states in which any sort of action is performed. The actions themselves are spelled out in Figure 1 (page 22). Note that the suffix recognition code goes through the word from back to front. Consequently, the state table for suffixes is a little weird looking (all the suffixes are backwards).

The table-processing routines start on lines 110 (suffix) and 221 (prefix). They are essentially the same, though different tables are used to do the processing. The subroutine **next**( ), when given the current state and input character, will return the next state. All possible action states are taken care of in the switches on lines 155 and 235.

Consonant-pair recognition is done by brute force rather than with a machine. (I tried to do a state machine, but it got too complicated). One of the places where the code size could be reduced is here. **Nextch**( ) (on line 318) gets the next input character and advances the string pointer past the character. If a consonant pair (*th*, *sh*, *ph*, *ch*, or *gh*) is found, both characters are skipped and an integer value (**#defined** on line 31 f.) is returned. The actual consonant-pair processing routine starts on line 393.

That's hyphenation. The algorithm works correctly with most of the words I've tried it on. My version successfully hyphenates su-per-califragi-lis-ticex-pialido-cious, just like Knuth's does. (Note that the algorithm doesn't insert every possible hyphen, but it doesn't put in any incorrect ones either.)

If anyone doesn't feel like typing in this month's program, machine readable copies of all source code from C Chest are now available on 5¼-inch IBM PC compatible disks from Software Engineering Consultants, P.O. Box 5679, Berkeley, CA 94705. The cost is $25 per disk (one disk is one month's column). In addition, my version of grep, the Unix generalized regular expression parser presented in *Doctor Dobb's Journal* #96, is still available for $35 from the same address.

Notes

[1] Donald E. Knuth, *TEX and Metafont, New Directions in Typesetting* (Digital Press), pp.180–186. DDJ

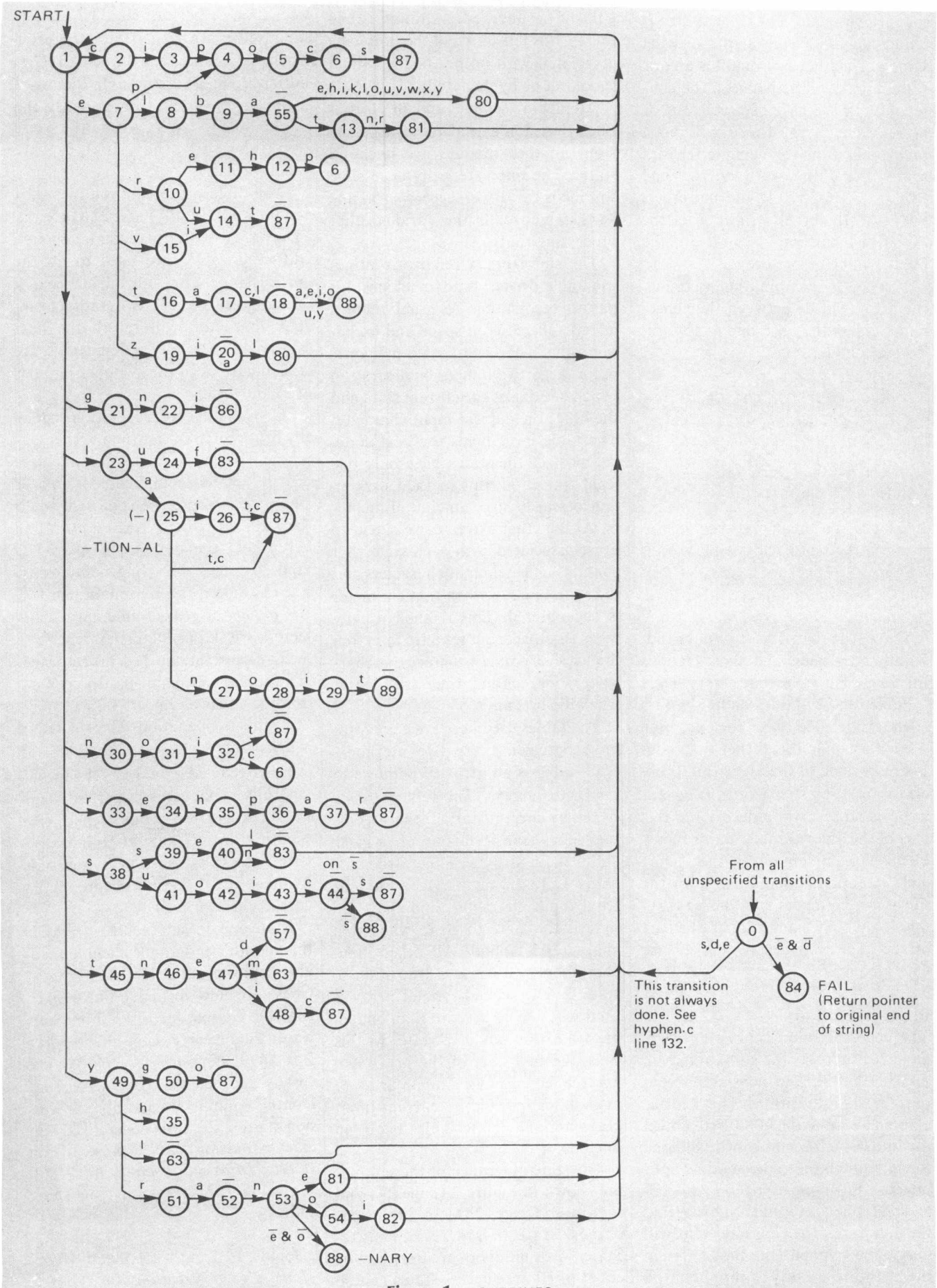


Figure 1 SUFFIXES

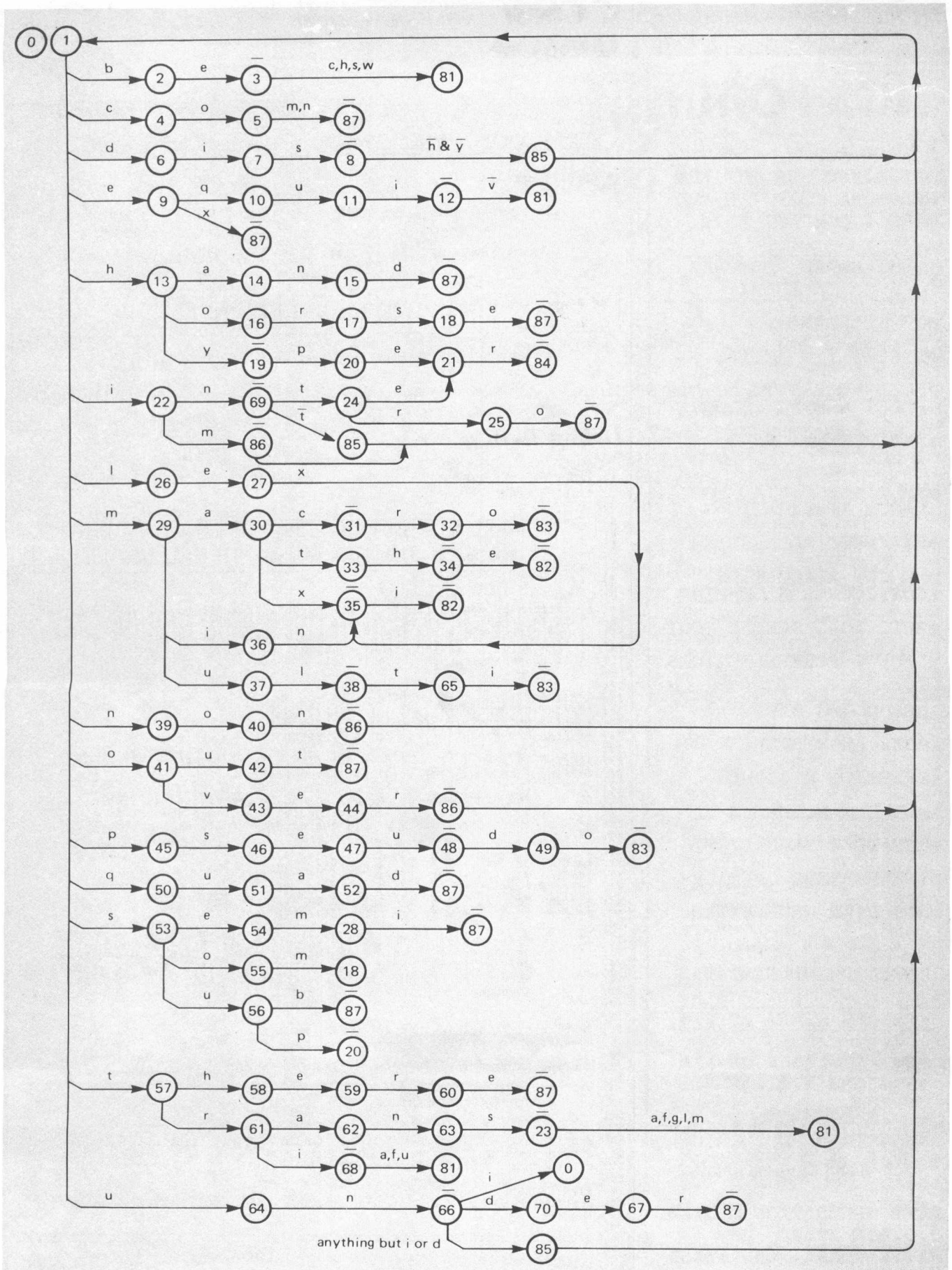


Figure 2 PREFIXES

C Chest (Text begins on page 20)

Listing One

```
+-------------------------------------------------------------------------+
|       Changes to ls.c to fix the root directory problem.                |
|                                                                         |
|       1) Add the subroutine isrootdir():                                |
+-------------------------------------------------------------------------+

isrootdir( name )
char *name;
{
        if( *name && name[1] == ':' )
                name += 2;

        return( (*name == '\\' || *name == '/') && !name[1] );
}

+-------------------------------------------------------------------------+
| 2) Modify the first if statement of the subroutine fixup_name()         |
|             (line 184 of the original listing) to read:                 |
+-------------------------------------------------------------------------+

        if( !isrootdir(name)  &&  !find_first(name, ALL, regs) )
```

End Listing One

Listing Two

```
 1: #include <stdio.h>
 2: #define DEBUG
 3:
 4: /*-------------------------------------------------------------------------*
 5:  *    HYPHEN.C - An implementation of the Knuth hyphenation algorithm     *
 6:  *                                                                         *
 7:  *       Copyright (c) 1985, Allen I. Holub.  All rights reserved.         *
 8:  *-------------------------------------------------------------------------*
 9:  */
10:
11: /* Various psuedo-subroutines. HYPHEN defines a bit to set when a hyphen
12:  * is inserted. HYPHENATE sets the bit, UNHYPHENATE clears it,
13:  * HAS_HYPHEN tests for it. The ER macro checks for an "er" at the end
14:  * of a word, it's used by the consonant pair checking routine.
15:  * Is consonant returns true if c is a consonant.
16:  */
17:
18: #define HYPHEN          0x80
19: #define HYPHENATE(c)    ( (c) |=  HYPHEN )
20: #define UNHYPHENATE(c)  ( (c) &= ~HYPHEN )
21: #define HAS_HYPHEN(c)   ( (c) &   HYPHEN )
22:
23: #define ER(p,end)       (*p == 'e' && *(p+1) == 'r' &&  (p+1) == end)
24: #define isconsonant(c)  ((c) && !isvowel(c))
25:
26: /*      The dipthongs ch, gh, ph, sh and th are treated as single
27:  *      consonants. The subroutine nextch() will map these two
28:  *      characters into a single character as follows:
29:  */
30:
31: #define CH ('z' + 1 )   /* {     0x7b  \173    */
32: #define GH ('z' + 2 )   /* |     0x7c  \174    */
33: #define PH ('z' + 3 )   /* }     0x7d  \175    */
34: #define SH ('z' + 4 )   /* ~     0x7e  \176    */
35: #define TH ('z' + 5 )   /* DEL   0x7f  \177    */
36:
37: /*      S(x) is used to initialize the state tables. It turned out to
38:  *      be easier to use a define than a typedef. ATYPE is used store
39:  *      strings in something other than chars (useful if we're keeping
40:  *      attributes around). I haven't tried compiling with ATYPE set to
41:  *      anything except char so be careful if you do.
42:  */
43:
44: #define S(x)  static char x[]
45: typedef char  ATYPE;
46:
47: #define LATTICE 1          /* Compile for Lattice C compiler     */
48:
49: /*-------------------------------------------------------------------------
50:  *      GLOBAL VARIABLES:
51:  */
52:
53: static char **States;    /* Points to table for current state machine  */
54: extern char *Suffixes[]; /* Suffix state table (at end of this module) */
55: extern char *Prefixes[]; /* Prefix state table          "              */
56:
57: #ifdef DEBUG
58: static int  Debug = 0;  /* True if debug diagnostics are to be printed */
59: #endif
60:
61: /*-------------------------------------------------------------------------*/
62:
63: #ifdef LATTICE
64:
```

(Continued on page 28)

Listing Two

```
65: extern  char        *stpchr();
66: #define index(s,c)  stpchr(s,c) /* For some reason the unix index()
67:                                  * function is called stpchr() by
68:                                  * Lattice.
69:                                  */
70: #define register    static      /* This will give more efficient code
71:                                  * since Lattice doesn't support any
72:                                  * register variables. DON'T EXPLICITLY
73:                                  * INITIALIZE AUTOMATIC VARIABLES IF
74:                                  * THIS #DEFINE IS ACTIVE.
75:                                  */
76:
77: #else   /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
78:
79: char    *index(s,c)
80: char    *s;
81: {
82:         /*      Return a pointer to the left-most occurance of the
83:          *      character c in the string s or NULL if the character
84:          *      isn't found.
85:          */
86:
87:         for(; *s ; s++ )
88:                 if( *s == c )
89:                         return s;
90:         return NULL;
91: }
92:
93: #endif
94:
95: /*---------------------------------------------------------------------*/
96:
97: isvowel(c)
98: ATYPE   c ;
99: {
100:        /* We check for a vowel in the order in which it is likely to
101:         * appear in English (ETAOIN SHRDLU). Return true if c is a vowel.
102:         */
103:
104:        c &= 0x7f;
105:        return( c=='e' || c=='a' || c=='o' || c=='i' || c=='u' || c=='y');
106: }
107:
108: /*---------------------------------------------------------------------*/
109:
110: static  ATYPE   *suffix( beg, end )
111: ATYPE   *beg, *end ;
112: {
113:         /*      Split off any suffixes, using the "Suffixes" state table
114:          *      (defined below) to recognize them.
115:          */
116:
117:         register char   *p;
118:         register int    state, c, c2, times ;
119:
120:         state   = 1;
121:         times   = 0;
122:         States  = Suffixes;
123:
124:         for( p = end; p >= beg ;)
125:         {
126:                 state = next( state, c = *p-- & 0x7f  );
127:
128:                 if( !('a' <= c && c <= 'z') ) /* Words containing non-
129:                         return end;           /* lower-case characters *.
130:                                               /* won't be hyphenated.  */
131:
132:                 /* Test for state 0 with an if instead of in the switch.
133:                  * Lattice C will make the switch very inefficient if the
134:                  * range of case values is too high. State 0 is wierd;
135:                  * we use it to test for trailing e or ed on failure.
136:                  * The first time through we set p to end so we can check
137:                  * the end of the word. The third time through we abort.
138:                  * This means that we'll retry if the word ends -s -e -d
139:                  * -ed -de -dd -ee. The extra endings shouldn't matter.
140:                  */
141:
142:                 if( state == 0 )
143:                 {
144:                         if( times == 0 )
145:                                 p = end ;     /* Strip trailing e or d */
146:
147:                         else if( times == 1 ) /* Strip trailing ed, de */
148:                                 p = end-1 ;   /* ee, or dd             */
149:                         else
150:                                 return end;
151:
152:                         times++;
153:                 }
154:
155:                 switch( state )
156:                 {
```

(Continued on page 30)

C Chest (Listing continued, text begins on page 20)

Listing Two

```
157:                    case 86:
158:                            /*
159:                             * Process -ing. This could have been done with
160:                             * more states but it seemed easier to do it here.
161:                             */
162:
163:                            c =    *p   & 0x7f ;
164:                            c2 = *(p-1) & 0x7f ;
165:
166:                            if( p - beg < 3 )
167:                            {
168:                                    return end;     /* FAIL */
169:                            }
170:                            else if(     ( c == c2 ) && ( !isvowel(c) )
171:                                      && ( c != 'f') && ( c != 's'    )
172:                                      && ( c != 'l') && ( c != 'z'    ) )
173:                            {
174:                                    --p;
175:                            }
176:                            else if( c == 'l'  &&  index("bcdfghkptz", c2)  )
177:                            {
178:                                 p -= (c2 == 'k' && (*(p-2) & 0x7f) == 'c')
179:                                                                  ? 1 : 2 ;
180:                            }
181:
182:                            /* fall through to state 87 */
183:
184:                    case 87:
185:                            HYPHENATE( *(p+1) );
186:                            return p;
187:
188:                    case 88:
189:                            HYPHENATE( *(p+2) );
190:                            return( p + 1 );
191:
192:                    case 89:
193:                            HYPHENATE( *(p+5) );
194:                            HYPHENATE( *(p+1) );
195:                            return p;
196:
197:                    case 82: p++;   /* p += 3 */
198:                    case 81: p++;   /* p += 2 */
199:                    case 80: p++;   /* P += 1 */
200:                    case 83:
201:                             end = p ;
202:                             HYPHENATE( *(p + 1) );
203:                             state = 1;
204:                             break;
205:
206:                    case 84:                        /* FAILURE STATE */
207:                            return end;
208:                    }
209:            }
210:
211: #ifdef DEBUG
212:            if( Debug )
213:                    printf("\n");
214: #endif
215:
216:            return end;
217: }
218:
219: /*------------------------------------------------------------------------*/
220:
221: static  ATYPE   *prefix( beg, end )
222: ATYPE   *beg, *end;
223: {
224:            /*      Split off any prefixes, using the "Prefixes" state table
225:             */
226:
227:            register ATYPE  *p;
228:            register int    state, c ;
229:
230:            state  = 1;
231:            States = Prefixes;
232:
233:            for( p = beg ; p < end ;)
234:            {
235:                    switch( state = next(state, c = *p++ & 0x7f) )
236:                    {
237:                    case 82: HYPHENATE( *p ); HYPHENATE( *(p-1) ) ; return p;
238:                    case 83: HYPHENATE( *p ); HYPHENATE( *(p-2) ) ; return
239:                    case 84: HYPHENATE( *p ); HYPHENATE( *(p-3) ) ; return p
240:
241:                    case 81: --p;
242:                    case 87: HYPHENATE( *p );
243:                             return p;
244:
245:                    case 85: --p;
246:                    case 86: beg   = p;
247:                             state = 1;
248:                    case 69: HYPHENATE( *p );
```

```
249:                    break;
250:
251:            case 70: HYPHENATE( *(p-1) );
252:                    break;
253:
254:            /*      The following states don't actually exist.
255:             *      Putting them in the table will cause Lattice to
256:             *      compile the switch in a more efficient manner.
257:             *      (see p 4-33 of the 1985 vintage manual).
258:             */
259:
260:            case 71: case 72: case 73: case 74: case 75:
261:            case 76: case 77: case 78: case 79:
262:                    break;
263:            }
264:    }
265:
266: #ifdef DEBUG
267:    if( Debug )
268:            printf("\n");
269: #endif
270:
271:    return beg;     /* Failure */
272: }
273:
274: /*---------------------------------------------------------------------
275:  *      This table is used to find exceptions to the VCCV rule. It is
276:  *      indexed by the first of the two consonents & contains pointers
277:  *      to strings, any character of which will form an exception if
278:  *      it's the second consonant.
279:  */
280:
281: static char     *vccv_except[] =
282: {
283:    /* a  */        ""      ,
284:    /* b  */        "lr"    ,
285:    /* c  */        "lr"    ,
286:    /* d  */        "gr"    ,
287:    /* e  */        ""      ,
288:    /* f  */        "lr"    ,
289:    /* g  */        "lr"    ,
290:    /* h  */        ""      ,
291:    /* i  */        ""      ,
292:    /* j  */        ""      ,
293:    /* k  */        "n"     ,
294:    /* l  */        "kq"    ,
295:    /* m  */        ""      ,
296:    /* n  */        "{kx"   , /* CH, k, x */
297:    /* o  */        ""      ,
298:    /* p  */        "lr"    ,
299:    /* q  */        ""      ,
300:    /* r  */        "k"     ,
301:    /* s  */        "pq"    ,
302:    /* t  */        "{r"    , /* CH, r    */
303:    /* u  */        ""      ,
304:    /* v  */        ""      ,
305:    /* w  */        "hlnr"  ,
306:    /* x  */        ""      ,
307:    /* y  */        ""      ,
308:    /* z  */        ""      ,
309:    /* CH */        "lr"    ,
310:    /* GH */        "t"     ,
311:    /* PH */        "r"     ,
312:    /* SH */        ""      ,
313:    /* TH */        "r"
314: };
315:
316: /*---------------------------------------------------------------------*/
317:
318: static  ATYPE   nextch( pp, endp )
319: ATYPE   **pp, *endp;
320: {
321:    /*  Advance *pp (one for a vowel or consonant, 2 for a dipthong)
322:     *  and return the character we've just skipped. Dipthongs are
323:     *  mapped to the single characters: TH, SH, PH, CH or GH.
324:     *  0 is returned once *pp reaches or passes endp.
325:     */
326:
327:    register ATYPE  rval, *p;
328:
329:    if( (p = *pp)  > endp )
330:            return (ATYPE) 0;
331:
332:    rval = *p++ & 0x7f ;
333:
334:    if( ( *p & 0x7f )  ==  'h' )
335:    {
336:            switch( rval )
337:            {
338:            case 't':       rval = TH;      p++;    break;
339:            case 's':       rval = SH;      p++;    break;
340:            case 'p':       rval = PH;      p++;    break;
341:            case 'c':       rval = CH;      p++;    break;
342:            case 'g':       rval = GH;      p++;    break;
343:            }
```

(Continued on next page)

C Chest (Listing continued, text begins on page 20)

Listing Two

```
344:         }
345:
346: #ifdef DEBUG
347:         if( Debug )
348:                 printf( "nextch(), got <%c>\n", rval );
349: #endif
350:
351:         *pp = p;
352:         return rval;
353: }
354:
355:
356: /*------------------------------------------------------------------------*/
357:
358: static  iswierd( x, y, p )
359: ATYPE   *p;
360: {
361:         /* Return true if the string pointed to by p ends in:
362:          *      XYer XYers XYage XYages XYest
363:          *
364:          * where XY is one of the pairs:
365:          *      ft ld mp nd ng ns nt rg rm rn rt st
366:          */
367:
368:         register int    c1, c2, c3;
369:
370:         c1 = *p++ ;
371:         c2 = *p++ ;
372:         c3 = *p   ;
373:
374:         return(
375:                 (
376:                         ( c1 == 'e' && c2 == 'r'                )
377:                     ||  ( c1 == 'a' && c2 == 'g' &&  c3 == 'e' )
378:                     ||  ( c1 == 'e' && c2 == 's' &&  c3 == 't' )
379:                 )
380:                   &&
381:                 (       ( x == 'f'  && y == 't' )
382:                     ||  ( x == 'l'  && y == 'd' )
383:                     ||  ( x == 'm'  && y == 'p' )
384:                     ||  ( x == 's'  && y == 't' )
385:                     ||  ( x == 'n'  && index( "dgst", y) )
386:                     ||  ( x == 'r'  && index( "gmnt", y) )
387:                 )
388:         );
389: }
390:
391: /*------------------------------------------------------------------------*/
392:
393: static  consonants( beg, end )
394: ATYPE   *beg, *end;
395: {
396:         /*      Hyphenate consonant pairs.
397:          *      Look for a VCC pattern: use the state machine:
398:          *
399:          *                    V          C          C
400:          * Start ----> 1 -----> 2 -----> 3 -----> 4
401:          *           ^   |C   ^   | V       | V       | (no fetch )
402:          *           |   |    |   v         |         |
403:          *           |<--+    +------<------+         |
404:          *           |                               |
405:          *           +-------------------------------+
406:          *
407:          *    The machine is implemente explicitly (with whiles and ifs
408:          *    ifs and such) rather than with a table driven engine.
409:          *    A new character is fetched in states 1, 2, & 3 (but not 4).
410:          *    When you exit reach state 4 c1 and c2 will hold the two
411:          *    consonants, cp will point to the beginning of the second
412:          *    consonant, beg will point just past the second consonant.
413:          */
414:         register ATYPE  c1, c2, *cp;
415:         register char   *p;
416:
417:         while( 1 )
418:         {
419:                 statel:
420:
421:                 do {
422:                         c2 = nextch( &beg, end );
423:                 } while( isconsonant(c2) );
424:
425:                 do {
426:                         for( c1 = c2; isvowel(c1) ; )
427:                         {
428:                                 cp = beg;
429:                                 c1 = nextch( &beg, end );
430:                         }
431:
432:
433:                         if(c1 == 'q' && *beg == 'u')    /*          Vqu */
434:                         {
435:                                 HYPHENATE( *cp );       /*         V-qu */
```

(Continued on page 34)

Listing Two

```
436:                                    nextch( &beg, end );     /*  skip the u */
437:                                    goto statel;
438:                            }
439:
440:                            cp = beg ;
441:                            c2 = nextch(&beg, end) ;
442:
443:                    } while( isvowel(c2) ) ;
444:
445:                    if( !cl || !c2 )
446:                            break;
447:
448:                    /*      At this point we have found a Vowel-Cons-Cons
449:                     *      sequence. Cl and c2 will hold the left and right
450:                     *      consonant respectively. Beg will point at the
451:                     *      character following the second consonant. Cp
452:                     *      will point at the second consant.
453:                     */
454:
455:                    if( cl == 'c' &&  c2 == 'k' )           /*        Vck  */
456:                    {
457:                            if( *beg )
458:                                    HYPHENATE( *beg );      /*        Vck- *
459:                    }
460:                    else if( cl == c2 )
461:                    {
462:                            if( (cl != 'l'  &&  cl != 's')  ||
463:                                    (isvowel(*beg) && !ER(beg,end)) )
464:                                    HYPHENATE( *cp );
465:                    }
466:
467:                    else if( isvowel( *beg ) )              /*        VCCV */
468:                    {
469:                            if( ! iswierd(cl, c2, beg) )
470:                            {
471:                                    if( !((p = vccv_except[ (int)cl - 'a' ])
472:                                                    && index(p, c2) ))
473:                                            HYPHENATE( *cp );
474:                            }
475:                    }
476:            }
477:
478:            if( HAS_HYPHEN(*end) )
479:                    UNHYPHENATE( *end );
480: }
481:
482: /*------------------------------------------------------------------------*/
483:
484: hyphen( beg, end )
485: ATYPE   *beg, *end;
486: {
487:            /*      Hyphenate the word deliniated by beg and end: First
488:             *      strip suffixes, then strip a trailing s e or ed,
489:             *      then strip prefixes. Only words with more than four
490:             *      letters and which consist of lower case letters only
491:             *      (no NULLs) will be hyphenated. 0 is returned if no
492:             *      attempt was made to hyphenate the word, one otherwise.
493:             *      If the word has already been hyphenated, 1 is returned
494:             *      but the word is not modified.
495:             */
496:
497:            register ATYPE  *prefixp, *suffixp;
498:
499:            if( end-beg <= 4 )
500:                    return 0;
501:
502:            for( prefixp = beg; prefixp <= end ; prefixp++ )
503:            {
504:                    if( HAS_HYPHEN(*prefixp) )
505:                            return 1;
506:
507:                    if( !islower(*prefixp & 0x7f) )
508:                            return 0;
509:            }
510:
511:
512:            suffixp = suffix(beg,end);      /* Hyphenate and remove any    */
513:                                            /*       suffixes
514:            if( suffixp == end )
515:            {
516:                    /*      If root has a trailing s e or ed remove it
517:                     *      before applying any other rules
518:                     */
519:
520:                    if(*end == 's' || *end =='e')
521:                            suffixp = end - 1 ;
522:
523:                    else if( *(end-1) == 'e'  &&  *end == 'd' )
524:                            suffixp = end - 2 ;
525:            }
526:
527:            prefixp = prefix( beg, suffixp);  /* Hyphenate and remove any */
528:                                              /* prefixes                 */
529: #ifdef DEBUG
530:            /*-----------------------------------------------------*/
```

```
531:         /*  Print the word in the form:  prefixes/root/suffixes */
532:         /*                                                       */
533:         bprint( beg, prefixp-1  );      /* Print prefixes       */
534:         putchar('/');
535:         bprint( prefixp, suffixp );     /* Print middle         */
536:         putchar('/');
537:         bprint( suffixp+1, end   );     /* Print suffixes       */
538:         /*-------------------------------------------------------*/
539: #endif
540:
541:         if( (suffixp - prefixp)  >=  3 )
542:         {
543:                 /* Apply the consonant pairs rules only if the word
544:                  * has at least 4 letters in it (after prefixes
545:                  * and suffixes have been removed
546:                  */
547:
548:                 consonants( prefixp, suffixp );
549:         }
550:
551:         return 1;
552: }
553:
554: /*-------------------------------------------------------------------------*/
555:
556: static  next( cur_state, cur_char )
557: ATYPE   cur_char  ;
558: int     cur_state ;
559: {
560:         /*      Given the current state and the current input character
561:          *      return the next state. The global variable States must
562:          *      point at the correct state table (Suffixes or Prefixes).
563:          */
564:
565:         char    *p = States[ cur_state ] ;
566:
567: #ifndef DEBUG
568:
569:         if( *p )
570:                 return (int)( (cur_char == p[1]) ? p[2] : 0 );
571:         else
572:                 return (int)( p[(cur_char - 'a') + 1] );
573: #else
574:
575:         int     rval;
576:         if( Debug )
577:                 printf("%s: Current state = %d, Input char = <%c>, ",
578:                                 States == Prefixes ? "PREFIX" : "SUFFIX",
579:                                 cur_state, cur_char );
580:
581:         if( *p )
582:                 rval = (int)( (cur_char == p[1]) ? p[2] : 0 );
583:         else
584:                 rval = (int)( p[(cur_char - 'a') + 1] );
585:
586:         if( Debug )
587:                 printf("Next state = %d\n", rval );
588:
589:         return( rval );
590: #endif
591: }
592:
593: /*-------------------------------------------------------------------------
594:  * State machine tables:
595:  *
596:  * There are two types of states. If the first entry is 0 then the state
597:  * is an array holding the next state, indexed by the current input
598:  * character. If the first entry is 1 then all transitions but one are to
599:  * state 0, the next character is the one exception and the transition
600:  * state follows.
601:  *
602:  * -----------------------------------------------------------------------
603:  */
604:
605: S(s0 )={ 0, 84,84,84,1,1,84,84,84,84,84,84,84,84,84,84,84,84,1,84,
606:                                                   84,84,84,84,84,84 };
607:
608: S(s1 )={ 0, 0,0,2,0,7,0,21,0,0,0,0,23,0,30,0,0,0,33,38,45,0,0,0,0,49,0};
609: S(s2 )={ 1, 'i', 3  };
610: S(s3 )={ 1, 'p', 4  };
611: S(s4 )={ 1, 'o', 5  };
612: S(s5 )={ 1, 'c', 6  };
613: S(s6 )={ 1, 's', 87 };
614: S(s7 )={ 0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,10,0,16,0,15,0,0,0,19 };
615: S(s8 )={ 1, 'b', 9  };
616: S(s9 )={ 1, 'a', 55 };
617: S(s10)={ 0, 0,0,0,0,11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,0,0,0,0,0 };
618: S(s11)={ 1, 'h', 12 };
619: S(s12)={ 1, 'p', 6  };
620: S(s13)={ 0, 0,0,0,0,0,0,0,0,0,0,0,0,0,81,0,0,0,81,0,0,0,0,0,0,0,0 };
621: S(s14)={ 1, 't', 87 };
622: S(s15)={ 1, 'i', 14 };
623: S(s16)={ 1, 'a', 17 };
624: S(s17)={ 0, 0,0,18,0,0,0,0,0,0,0,0,18,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
625: S(s18)={ 0, 88,0,0,0,88,0,0,0,88,0,0,0,0,0,88,0,0,0,0,0,88,0,0,0,88,0};
626: S(s19)={ 1, 'i', 20 };
627: S(s20)={ 1, 'l', 80 };
628: S(s21)={ 1, 'n', 22 };
629: S(s22)={ 1, 'i', 86 };
630: S(s23)={ 0, 25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,24,0,0,0,0,0 };
```

(Continued on next page)

 (Listing continued, text begins on page 20)

Listing Two

```
631: S(s24)={ 1, 'f', 83 };
632: S(s25)={ 0, 0,0,87,0,0,0,0,0,26,0,0,0,0,27,0,0,0,0,0,87,0,0,0,0,0,0 };
633: S(s26)={ 0, 0,0,87,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,87,0,0,0,0,0,0 };
634: S(s27)={ 1, 'o', 28 };
635: S(s28)={ 1, 'i', 29 };
636: S(s29)={ 1, 't', 89 };
637: S(s30)={ 1, 'o', 31 };
638: S(s31)={ 1, 'i', 32 };
639: S(s32)={ 0, 0,0,6,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,87,0,0,0,0,0,0 };
640: S(s33)={ 1, 'e', 34 };
641: S(s34)={ 1, 'h', 35 };
642: S(s35)={ 1, 'p', 36 };
643: S(s36)={ 1, 'a', 37 };
644: S(s37)={ 1, 'r', 87 };
645: S(s38)={ 0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,39,0,41,0,0,0,0,0 };
646: S(s39)={ 1, 'e', 40 };
647: S(s40)={ 0, 0,0,0,0,0,0,0,0,0,0,0,0,83,0,83,0,0,0,0,0,0,0,0,0,0,0 };
648: S(s41)={ 1, 'o', 42 };
649: S(s42)={ 1, 'i', 43 };
650: S(s43)={ 1, 'c', 44 };
651: S(s44)={ 0, 88,88,88,88,88,88,88,88,88,88,88,88,88,88,88,88,88,88,87,88,
652:                                                                 88,88,88,88,88,88 };
653: S(s45)={ 1, 'n', 46 };
654: S(s46)={ 1, 'e', 47 };
655: S(s47)={ 0, 0,0,0,87,0,0,0,0,48,0,0,0,83,0,0,0,0,0,0,0,0,0,0,0,0,0 };
656: S(s48)={ 1, 'c', 87 };
657: S(s49)={ 0, 0,0,0,0,0,0,50,35,0,0,0,83,0,0,0,0,0,51,0,0,0,0,0,0,0,0 };
658: S(s50)={ 1, 'o', 87 };
659: S(s51)={ 1, 'a', 52 };
660: S(s52)={ 1, 'n', 53 };
661: S(s53)={ 0, 88,88,88,88,81,88,88,88,88,88,88,88,88,54,88,88,88,88,88,
662:                                                                 88,88,88,88,88,88 };
663: S(s54)={ 1, 'i', 82 };
664: S(s55)={ 0, 0,0,0,0,80,0,0,80,80,0,80,80,0,0,80,0,0,0,0,13,80,80,80,80,
665:                                                                       80,0};
666:
667: static char     *Suffixes[] =
668: {
669:         s0,  s1,  s2,  s3,  s4,  s5,  s6,  s7,  s8,  s9,
670:         s10, s11, s12, s13, s14, s15, s16, s17, s18, s19,
671:         s20, s21, s22, s23, s24, s25, s26, s27, s28, s29,
672:         s30, s31, s32, s33, s34, s35, s36, s37, s38, s39,
673:         s40, s41, s42, s43, s44, s45, s46, s47, s48, s49,
674:         s50, s51, s52, s53, s54, s55
675: };
676:
677: /*-------------------------------------------------------------------------*/
678:
679: S(p0 )= {1, 0, 0 };
680: S(p1 )= {0, 0,2,4,6,9,0,0,13,22,0,0,26,29,39,41,45,50,0,53,57,64,
681:                                                                 0,0,0,0,0 };
682: S(p2 )= {1, 'e', 3 };
683: S(p3 )= {0, 0,0,81,0,0,0,0,81,0,0,0,0,0,0,0,0,0,81,0,0,0,81,0,0,0 };
684: S(p4 )= {1, 'o', 5 };
685: S(p5 )= {0, 0,0,0,0,0,0,0,0,0,0,0,87,87,0,0,0,0,0,0,0,0,0,0,0 };
686: S(p6 )= {1, 'i', 7 };
687: S(p7 )= {1, 's', 8 };
688: S(p8 )= {0, 85,85,85,85,85,85,85, 0,85,85,85,85,85,85,85,85,85,85,85,
689:                                                                 85,85,85,85, 0,85 };
690: S(p9 )= {0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,10,0,0,0,0,0,0,87,0,0 };
691: S(p10)= {1, 'u', 11 };
692: S(p11)= {1, 'i', 12 };
693: S(p12)= {0, 81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,
694:                                                                 81, 0,81,81,81,81 };
695: S(p13)= {0, 14,0,0,0,0,0,0,0,0,0,0,0,0,16,0,0,0,0,0,0,0,0,0,0,19,0 };
696: S(p14)= {1, 'n', 15 };
697: S(p15)= {1, 'd', 87 };
698: S(p16)= {1, 'r', 17 };
699: S(p17)= {1, 's', 18 };
700: S(p18)= {1, 'e', 87 };
701: S(p19)= {1, 'p', 20 };
702: S(p20)= {1, 'e', 21 };
703: S(p21)= {1, 'r', 84 };
704: S(p22)= {0, 0,0,0,0,0,0,0,0,0,0,0,0,86,69,0,0,0,0,0,0,0,0,0,0,0,0 };
705: S(p23)= {0, 81,0,0,0,0,0,81,81,0,0,0,0,0,81,81,81,0,0,0,0,0,0,0,0,0,0,0,0,0 };
706: S(p24)= {0, 0,0,0,0,21,0,0,0,0,0,0,0,0,0,0,0,0,25,0,0,0,0,0,0,0,0 };
707: S(p25)= {1, 'o', 87 };
708: S(p26)= {1, 'e', 27 };
709: S(p27)= {1, 'x', 35 };
710: S(p28)= {1, 'i', 87 };
711: S(p29)= {0, 30,0,0,0,0,0,0,0,36,0,0,0,0,0,0,0,0,0,0,0,37,0,0,0,0,0 };
712: S(p30)= {0, 0,0,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,33,0,0,0,35,0,0 };
713: S(p31)= {1, 'r', 32 };
714: S(p32)= {1, 'o', 83 };
715: S(p33)= {1, 'h', 34 };
716: S(p34)= {1, 'e', 82 };
717: S(p35)= {1, 'i', 82 };
718: S(p36)= {1, 'n', 35 };
719: S(p37)= {1, 'l', 38 };
720: S(p38)= {1, 't', 65 };
721: S(p39)= {1, 'o', 40 };
722: S(p40)= {1, 'n', 86 };
```

(Continued on page 38)

C Chest (Listing continued, text begins on page 20)

Listing Two

```
723: S(p41)= {0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,42,43,0,0,0,0 };
724: S(p42)= {1, 't', 87 };
725: S(p43)= {1, 'e', 44 };
726: S(p44)= {1, 'r', 86 };
727: S(p45)= {1, 's', 46 };
728: S(p46)= {1, 'e', 47 };
729: S(p47)= {1, 'u', 48 };
730: S(p48)= {1, 'd', 49 };
731: S(p49)= {1, 'o', 83 };
732: S(p50)= {1, 'u', 51 };
733: S(p51)= {1, 'a', 52 };
734: S(p52)= {1, 'd', 87 };
735: S(p53)= {0, 0,0,0,0,54,0,0,0,0,0,0,0,0,0,0,55,0,0,0,0,0,0,56,0,0,0,0,0 };
736: S(p54)= {1, 'm', 28 };
737: S(p55)= {1, 'm', 18 };
738: S(p56)= {0, 0,87,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,0,0,0,0,0,0,0,0,0,0,0 };
739: S(p57)= {0, 0,0,0,0,0,0,0,0,58,0,0,0,0,0,0,0,0,0,0,61,0,0,0,0,0,0,0,0 };
740: S(p58)= {1, 'e', 59 };
741: S(p59)= {1, 'r', 60 };
742: S(p60)= {1, 'e', 87 };
743: S(p61)= {0, 62,0,0,0,0,0,0,0,68,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
744: S(p62)= {1, 'n', 63 };
745: S(p63)= {1, 's', 23 };
746: S(p64)= {1, 'n', 66 };
747: S(p65)= {1, 'i', 83 };
748: S(p66)= {0, 85,85,85,70,85,85,85,85, 0,85,85,85,85,85,85,85,85,85,
749:                                                 85,85,85,85,85,85,85,85,85 };
750: S(p67)= {1, 'r', 87 };
751: S(p68)= {0, 81,0,0,0,0,0,81,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,81,0,0,0,0,0 };
752: S(p69)= {0, 85,85,85,85,85,85,85,85,85,85,85,85,85,85,85,85,85,85,85,85,85,
753:                                                 85,24,85,85,85,85,85,85 };
754: S(p70)= {1, 'e', 67 };
755:
756:
757: static char     *Prefixes[] =
758: {
759:         p0,  p1,  p2,  p3,  p4,  p5,  p6,  p7,  p8,  p9,
760:         p10, p11, p12, p13, p14, p15, p16, p17, p18, p19,
761:         p20, p21, p22, p23, p24, p25, p26, p27, p28, p29,
762:         p30, p31, p32, p33, p34, p35, p36, p37, p38, p39,
763:         p40, p41, p42, p43, p44, p45, p46, p47, p48, p49,
764:         p50, p51, p52, p53, p54, p55, p56, p57, p58, p59,
765:         p60, p61, p62, p63, p64, p65, p66, p67, p68, p69,
766:         p70
767: };
768:
769: /*=====================================================================
770:  *
771:  *            MSC DEBUGGING ROUTINES (INCLUDING A MAIN MODULE):
772:  */
773:
774: #ifdef DEBUG
775: bprint( b , e )
776: ATYPE   *b, *e;
777: {
778:         /*      Print all characters between b and e inclusive   */
779:
780:         while( b <= e && *b )
781:                 putchar( *b++ & 0x7f );
782: }
783:
784: /*--------------------------------------------------------------------*/
785:
786: char    *sgets( prompt, buf, maxch )
787: char    *prompt, *buf;
788: {
789:         /* Print prompt to stdout, then fill buf fm stdin. Get at most
790:          * maxch characters. Return pointer to string terminator (ie. the
791:          * null at the end of the string) or 0 on end of file.
792:          */
793:
794:         register int    c;
795:         register char   *start;
796:
797:         start = buf;
798:         printf( prompt );
799:
800:         while( (c = getchar()) != EOF && c != '\n' && --maxch > 0 )
801:                 *buf++ = c;
802:
803:         *buf = 0;
804:
805:         return (c == EOF && start == buf) ? (char *)0 : buf ;
806: }
807:
808: /*--------------------------------------------------------------------*/
809:
810: phyphen( p )
811: ATYPE *p;
812: {
813:         /*      Print a word with all the hyphens visible
814:          */
815:
816:         printf("<");
```

```
817:
818:         for( ; *p ; putchar(*p++ & 0x7f) )
819:                 if( HAS_HYPHEN(*p) )
820:                         putchar('-');
821:
822:         printf(">\n");
823: }
824:
825: /*------------------------------------------------------------------------*/
826:
827: main(argc, argv)
828: int     argc;
829: char    **argv;
830: {
831:         char    buf[132], *p;
832:
833:         if( argc > 1 )
834:                 Debug = 1;
835:
836:         while( p = sgets( "word: ", buf, 132 ) )
837:         {
838:                 if( !*buf )
839:                         printf("\n");
840:
841:                 else if( p-- > buf )
842:                 {
843:                         hyphen( buf, p );
844:                         putchar('\t');
845:                         phyphen( buf );
846:                 }
847:         }
848: }
849:
850: #endif
```

End Listing Two

Listing Three

```
+-------------------------------------------------------------------------+
|    Diagnostic output from hyphen.c when compiled with the DEBUG #defined:  |
+-------------------------------------------------------------------------+

word: be/cxxxe/able      <be-cxxxe-able>
word: be/hxxxh/able      <be-hxxxh-able>
word: be/sxxxi/able      <be-sxxxi-able>
word: be/wxxxk/able      <be-wxxxk-able>
word: com/xxxl/able      <com-xxxl-able>
word: con/xxxo/able      <con-xxxo-able>
word: dis/xxxu/able      <dis-xxxu-able>
word: equi/xxv/able      <equi-xxv-able>
word: ex/xxxxw/able      <ex-xxxxw-able>
word: hand/xxxxxx/able   <hand-xxxxxx-able>
word: horse/xxxxy/able   <horse-xxxxy-able>
word: hyper/xxxnt/able   <hy-per-xxxnt-able>
word: im/xxxxxxrt/able   <im-xxxxxxrt-able>
word: in/xxxxxxion/ary   <in-xxxxxxion-ary>
word: inter/xxxxen/ary   <in-ter-xxxxen-ary>
word: intro/xxxxx/nary   <in-tro-xxxxx-nary>
word: lexi/xxxxxxx/cal   <lex-i-xxxxxxx-cal>
word: macro/xxxxa/cate   <mac-ro-xxxxa-cate>
word: mathe/xxxxe/cate   <math-e-xxxxe-cate>
word: maxi/xxxxxi/cate   <max-i-xxxxxi-cate>
word: mini/xxxxxo/cate   <min-i-xxxxxo-cate>
word: multi/xxxxu/cate   <mul-ti-xxxxu-cate>
word: non/xxxxxxy/cate   <non-xxxxxxy-cate>
word: out/xxxxxxx/cial   <out-xxxxxxx-cial>
word: over/xxxxx/cious   <over-xxxxx-cious>
word: pseudo/xx/scious   <pseu-do-xx-scious>
word: quad/xxxxx/cient   <quad-xxxxx-cient>
word: semi/xxxxxx/dent   <semi-xxxxxx-dent>
word: some/xxxxxxx/ful   <some-xxxxxxx-ful>
word: sub/xxxxxxxl/ize   <sub-xxxxxxxl-ize>
word: super/xxxxa/late   <su-per-xxxxa-late>
word: there/xxxxe/late   <there-xxxxe-late>
word: trans/axxxi/late   <trans-ax-xxi-late>
word: trans/fxxxo/late   <trans-fxxxo-late>
word: trans/gxxxu/late   <trans-gxxxu-late>
word: trans/lxxxy/late   <trans-lxxxy-late>
word: trans/mxxxx/less   <trans-mxxxx-less>
word: tri/axxxxxxxx/ly   <tri-ax-xxxxxxx-ly>
word: tri/fxxxxxxxx/ly   <tri-fxxxxxxxx-ly>
word: tri/uxxxxxxxx/ly   <tri-ux-xxxxxxx-ly>
word: under/xxxxx/ment   <un-der-xxxxx-ment>
word: un/xxxxxxxx/ness   <un-xxxxxxxx-ness>
word: /xxxxx/ogy         <xxxxx-ogy>
word: /xx/rapher         <xx-rapher>
word: /xxx/raphy         <xxx-raphy>
word: /xxx/scope         <xxx-scope>
word: /xx/scopic         <xx-scopic>
word: /xxx/scion         <xxx-scion>
word: /xx/sphere         <xx-sphere>
word: /xxxxx/tal         <xxxxx-tal>
word: /xxxx/tial         <xxxx-tial>
word: /xxxx/tion         <xxxx-tion>
word: /xx/tional         <xx-tion-al>
word: /xxxx/tive         <xxxx-tive>
word: /xxxx/ture         <xxxx-ture>
word: /xxxxxl/ing        <xxxxxl-ing>
word: /xxxxl/ing         <xxxxl-ing>
word: /xxxg/gling        <xxxg-gling>
word: /xxxa/bling        <xxxa-bling>
```

(Continued on next page)

C Chest (Listing continued, text begins on page 20)

Listing Three

```
word: /xxxck/ling          <xxxck-ling>
word: /xxxg/kling          <xxxg-kling>
word: /duck/ing            <duck-ing>
word: /fit/ting            <fit-ting>
word: /hiss/ing            <hiss-ing>
word: /swell/ing           <swell-ing>
word: /fic/tion            <fic-tion>
word: /fic/tional          <fic-tion-al>
word: /fic/tionalize       <fic-tion-al-ize>
word: /fic/tionalized      <fic-tion-al-ized>
word: /fic/tionalizes      <fic-tion-al-izes>
word: /uuuccuuu/           <uuuc-cuuu>
word: /uuuquuuu/           <uuu-quuuu>
word: /uuuckuuu/           <uuuck-uuu>
word: /uuuccuuu/           <uuuc-cuuu>
word: /uuuchchuu/          <uuuch-chuu>
word: /uuughghuu/          <uuugh-ghuu>
word: /uuuphphuu/          <uuuph-phuu>
word: /uuushshuu/          <uuush-shuu>
word: /uuuththuu/          <uuuth-thuu>
word: /uuuxxxxxx/          <uuux-xxxxx>
word: /uuullaxxxx/         <uuul-lax-xxx>
word: /uuussexxxx/         <uuus-sex-xxx>
word: /uuufteee/e          <uuuf-teeee>
word: /uuuxxer/            <uuux-xer>
word: /uuuxxer/s           <uuux-xers>
word: /uuuxxag/e           <uuux-xage>
word: /uuuxxage/s          <uuux-xages>
word: /uuuxxest/           <uuux-xest>
word:
word:   <Test repetetive suffix and prefix removal:>
word:
word: disiminnonoverun/en/aryfulizelesslymentableness
                <dis-im-in-non-over-un-en-ary-ful-ize-less-ly-ment-able-ness>
word:
word: super/califragilisticexpialido/cious
                <su-per-califragilis-ticex-pialido-cious>
word:
word:   <All words following this point test exceptions to>
word:   <the hyphenation rules and shouldn't be hyphenated.>
word:
word: /uuussx/             <uuussx>
word: /uuullx/             <uuullx>
word: /uuuller/            <uuuller>
word: /uuuller/s           <uuullers>
word: /uuubluu/            <uuubluu>
word: /uuubruu/            <uuubruu>
word: /uuucluu/            <uuucluu>
word: /uuucruu/            <uuucruu>
word: /uuuchluu/           <uuuchluu>
word: /uuuchruu/           <uuuchruu>
word: /uuudguu/            <uuudguu>
word: /uuudruu/            <uuudruu>
word: /uuufluu/            <uuufluu>
word: /uuufruu/            <uuufruu>
word: /uuughtuu/           <uuughtuu>
word: /uuugluu/            <uuugluu>
word: /uuugruu/            <uuugruu>
word: /uuuknuu/            <uuuknuu>
word: /uuulkuu/            <uuulkuu>
word: /uuulquu/            <uuulquu>
word: /uuunchuu/           <uuunchuu>
word: /uuunkuu/            <uuunkuu>
word: /uuunxuu/            <uuunxuu>
word: /uuuphruu/           <uuuphruu>
word: /uuupluu/            <uuupluu>
word: /uuupruu/            <uuupruu>
word: /uuurkuu/            <uuurkuu>
word: /uuuspuu/            <uuuspuu>
word: /uuusquu/            <uuusquu>
word: /uuutchuu/           <uuutchuu>
word: /uuutruu/            <uuutruu>
word: /uuuthruu/           <uuuthruu>
word: /uuuwhuu/            <uuuwhuu>
word: /uuuwluu/            <uuuwluu>
word: /uuuwnuu/            <uuuwnuu>
word: /uuuwruu/            <uuuwruu>
word: /uuufter/            <uuufter>
word: /uuufter/s           <uuufters>
word: /uuuftag/e           <uuuftage>
word: /uuuftage/s          <uuuftages>
word: /uuuftest/           <uuuftest>
word: /uuulder/            <uuulder>
word: /uuumper/            <uuumper>
word: /uuunder/            <uuunder>
word: /uuunger/            <uuunger>
word: /uuunser/            <uuunser>
word: /uuunter/            <uuunter>
word: /uuurger/            <uuurger>
word: /uuurner/            <uuurner>
word: /uuurter/            <uuurter>
word: /uuuster/            <uuuster>
```

End Listings

A Threaded-Code Microprocessor Bursts Forth

Subroutines Without Performance Anxiety

by Leo Brodie

Subroutine calls in one clock cycle are one advantage as Forth goes to silicon.

The Novix NC4000 processor has arrived, and it threatens to shake most of our beliefs about microprocessor design. What implications does this new technology have for the art of programming? This article will explain the techniques used by Forth and by the NC4000 microprocessor, and explore the impact of this technology on software design.

Subroutines: The More the Better

Most programmers recognize the importance of dividing a large, complex program into a group of smaller, manageable pieces. The use of smaller "modules" is advantageous for the following reasons:

- smaller pieces of code are easier to think about, and therefore easier to design and to write;
- smaller pieces of code are easier to debug; the likelihood of introducing a bug is reduced, and, if one does appear, it will be easier to find;
- different parts of a program may be assigned to different programmers;
- subroutines can be used to hide information about things that might change in subsequent revisions; if a change must be made to a program, only the subroutine has to be changed.

Unfortunately, most conventional high-level languages (HLLs) are not optimized for the invocation of subroutines. One problem is the loss of time caused by saving registers before a jump to a subroutine and restoring them afterwards. This has been estimated as 40% of program execution time. Even more time-consuming is the invisible but significant code needed to pass arguments to and from the subroutine. At the machine level as well, processors are not generally optimized for invoking subroutine calls.

For these reasons, although the use of many, small subroutines is an ideal, nevertheless, in practice, programmers tend to writer fewer, larger subroutines. When efficiency is at stake, it's hard to appreciate the elegance of subroutines.

There is a high-level programming language optimized for subroutines: Forth. And now, a single-chip microprocessor is available that applies Forth's optimization techniques directly to a silicon architecture. The result is a machine that can execute high-level subroutine calls and returns in a single clock cycle!

Threaded Code

Let's begin by examining what the inherent architecture of Forth is, regardless of which processor it runs on. Forth is implemented using a technique called "threaded code." Suppose we define a word in Forth called PROCESS, consisting of three operations.

```
: PROCESS      STEP1  STEP2  STEP3  ;
```

In this definition, the name of the word (subroutine) is PROCESS. It consists of three lower-level subroutines. The

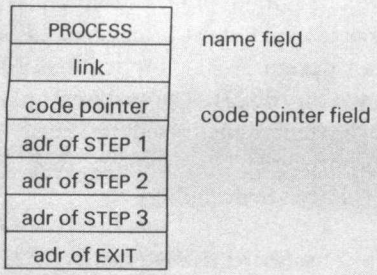


Figure 1
A compiled definition in the dictionary of an emulated Forth system

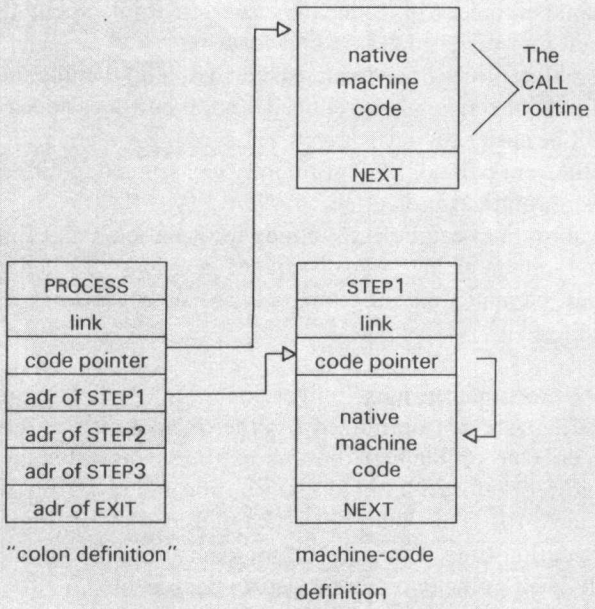


Figure 2
The code pointer of an assembler definition (e.g., STEP1) points to its own definition. The code pointer of a high-level definition (e.g., PROCESS) points to a prologue routine called CALL.

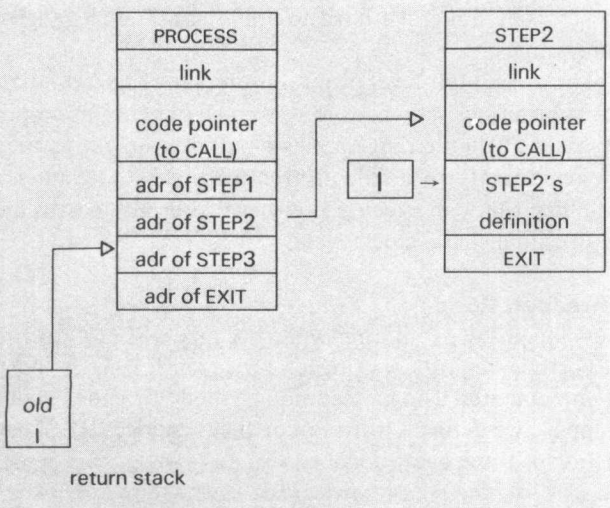


Figure 3
When STEP2's CALL routine is invoked, it saves I (the interpreter pointer) on the return stack, puts the address of the body of STEP2's definition in I, then invokes NEXT.

colon begins the definition, while the semicolon ends it.

Figure 1 (page 44) shows what this definition looks like when compiled into memory on an emulated Forth system. Such a definition consists of the following parts:

- The name field contains an ASCII representation of the word's name, so that the word can subsequently be found and executed.
- The link field points to the previous definition in the linked list of dictionary entries. (A search through the dictionary for a given word begins with the most recently defined word, and, using the link field for each word, proceeds backwards through the chain or chains of words.)
- The code pointer field points to machine-executable code that is generic to each "family" of definitions. As yet we have only discussed colon definitions; we will introduce other types later.
- The remainder of the definition is a list of addresses of other routines. Depending on the processor, these addresses are usually 16- or 32-bit numbers. In this example, the addresses of the routines STEP1, STEP2, and STEP3 are listed. At the end of the list is the address of another routine called EXIT, which is compiled by the semicolon and which terminates the definition when it is executed.

The unusual thing about threaded code is the absence of any CALL instructions—*everything* is a subroutine. Definitions consist simply of addresses of other definitions. As we'll see, even data structures are defined as dictionary entries, so that their addresses appear in definitions when referenced.

If you expand your imagination a bit, you can view these addresses as actual *instructions*, since each address is unique. This is exactly the way they are viewed by the Forth machine, and in particular, by the routine called the "address interpreter," also known as NEXT.

It is the function of the address interpreter to execute each "Forth instruction" in turn. This function is analogous to the operation of a machine processor wherein each opcode is interpreted and the program counter advanced. An "interpreter pointer," called **I** (usually a register in the real processor) operates analogously to the processor's program counter: during execution of any Forth instruction, I points to the next instruction to be executed; it is incremented by the address interpreter.

To begin with the simplest case possible, let's assume that each of the definitions STEP1, STEP2, and STEP3 is defined in machine code—they are not high level Forth definitions. (Forth allows routines to be coded in high-level code or in machine code *without distinction*; in practice, however, only a handful of time-critical or hardware-dependent routines must be machine-coded.) Here's what happens, in somewhat simplified form for now. I is pointing to the first address in the definition of PROCESS; this is the address of STEP1. The address interpreter, NEXT, first advances I to the next cell (the one containing the address of STEP2), then jumps to STEP1. STEP1 is a machine code definition, executed in the normal way by the

underlying processor. The final piece of code in STEP1 is NEXT once again (either in-line or as a jump instruction, depending on whether performance is a more or less important consideration than memory usage in the particular hardware configuration). This second NEXT increments I to the third cell, then jumps to STEP2, and so on.

Each machine-coded definition (called a primitive) must end by executing the code for NEXT to keep the Forth machine running.

In the preceding discussion, we omitted one important detail. The address interpreter does not jump to the address of each new instruction, but rather to the *code pointed to by the code field* of each new instruction—one added level of indirection. Figure 2 (at left) shows this relationship.

In the case of a machine code definition, the code field points to the body of the definition itself, where the actual machine instructions have been compiled. In the case of all colon (high-level) definitions, however, the code field points to a single routine in memory that we will name CALL. Essentially this routine handles the job of nesting definitions, so that one high-level word may invoke another word, which in turn invokes another, and so on.

Here is what CALL does, step by step. Using our previous example, let's now suppose that STEP2 has been defined as a colon, or high-level, definition. We are executing PROCESS, and have just completed STEP1. I (the interpreter pointer) is pointing to the address of STEP2. The address interpreter advances the pointer in anticipation of the return, then gets the address of STEP2. Using this address, it jumps to the address pointed to by the code pointer field of STEP2. Since STEP2 is a high-level definition, the routine at this address is CALL. This routine saves I on a special stack for return addresses called the return stack. It then places into I the address of the beginning of the instruction list in STEP2 (see Figure 3, at left). This done, CALL executes NEXT. Thus the Forth machine continues as before, but has nested in a level.

The opposite of the CALL function is EXIT, which terminates each high-level definition. When the EXIT in STEP2 is encountered, it pops the top of the return stack back into the interpreter pointer, thus resuming execution of PROCESS at the address of STEP3.

Because of the extra level of indirection offered by the code pointer field, the architecture we've described has been called indirect threaded code (ITC).

The value of the code pointer field is that each word knows what kind of procedure it is, and how to invoke itself. This is what makes it possible in Forth to design a routine first in high-level code, then to rewrite it in machine code without changing any code elsewhere that invokes the routine.

Each family of data structures in Forth uses a unique prologue routine, pointed to by the code fields of all members of that family. A simple example is the constant, as illustrated by Figure 4 (page 45). A Forth constant is a dictionary entry, possessing the same type of header structure as all dictionary entries. Its body consists of a code pointer for all constants, along with the value of the

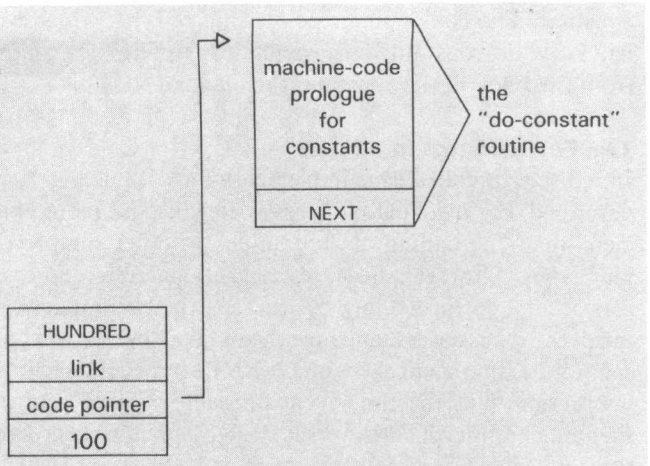


Figure 4
A Forth constant is a normal dictionary entry whose code pointer points to a routine common to all constants.

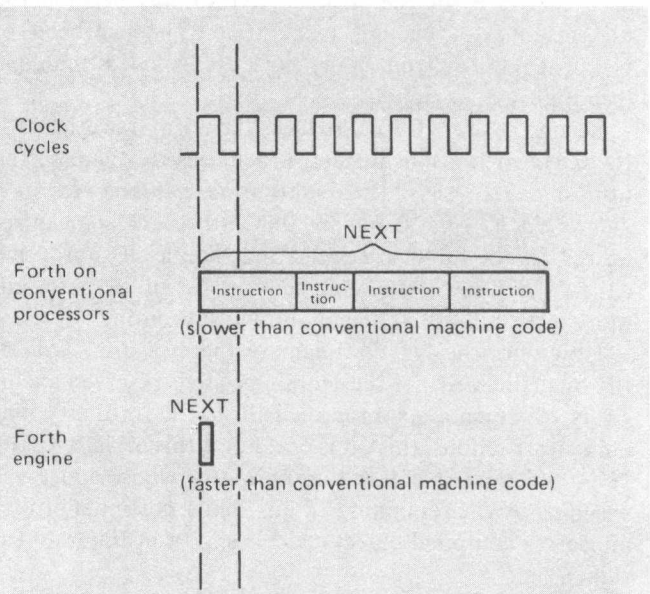


Figure 5
Comparison of overhead for NEXT

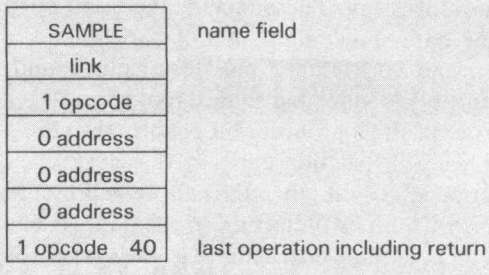


Figure 6
A definition list on the NC4000.

constant. The code pointer points to a routine, which we will call "do-constant," whose job it is to fetch the value from the body of the constant, then invoke NEXT.

The Forth Engine in Silicon

In indirect threaded code implementations of Forth as just described, the invoking of words is elegant and more efficient than the calling of subroutines in other high-level languages. However, because conventional microprocessors are not optimized for ITC, running the Forth machine involves some performance overhead. This overhead consists of the time spent executing NEXT after each machine instruction, plus the time spent for each call and return. Running high-level Forth versus straight machine code can increase execution time by as much as 100 percent (which is still better than other high-level languages).

A primary goal of the NC4000 architecture is to allow the execution of threaded code with as little overhead as possible. This goal is achieved first by eliminating NEXT as a software routine. Instead, NEXT has become the *background* to the chip's operation. The overhead of NEXT is reduced from many clock cycles to less than one (see Figure 5, at right).

Secondly, the NC4000 reduces the normal overhead of the CALL operation. In fact, it achieves this goal to the utmost degree possible: a subroutine call executes in a *single clock cycle*. Compare this with the 15 clocks required by the 8086 to effect a machine-level subroutine call. A high-level language such as C will require still more instructions to effect a jump to subroutine.

How does the NC4000 achieve these results? One of the bits (the MSB) of the 16-bit opcode is reserved exclusively to indicate a subroutine call. For all ordinary machine instructions, this bit is one; but if this bit is zero, the NC4000 effects a call to the subroutine whose address is specified by the remaining 15 bits. Thus, colon definitions appear as compiled object code as shown in Figure 6 (at right).

As you can see, compiled object code for the NC4000 consists of a mixture of actual machine instructions and single-instruction subroutine calls to definitions. In other words, the NC4000 executes high-level Forth threaded code as its native machine language.

Of actual machine instructions the NC4000P boasts over 170 combinations, not including permutations of register addressing. These include the usual Forth primitives for data-stack manipulation, arithmetic, memory, register, and I/O fetches and stores, jumps and loops—even multiply, divide, and square root steps.

The use of the high-order bit enables the chip's logic to decode each instruction early in the clock cycle. If the instruction is a CALL, then the address represented by the remaining 15 bits will appear on the address bus in time for the appropriate instruction (in the called routine) to be latched and executed on the next cycle.

The NC4000 does even better with the EXIT routine. Another of the 16 bits in the opcode (octal 40) is reserved exclusively to indicate the return operation. The return can occur simultaneously with other operations of the ALU. In a single clock cycle, a word can perform its last operation and return to the word that invoked it. Thus, in most cases, there is *no overhead* for the EXIT operation.

One consequence is that program space is limited to 32K addresses. Since the NC4000 uses *word addresses*, not byte addresses, the program may occupy 64K bytes of memory. Given the extreme compactness of threaded-code systems in general, and of this instruction set in particular (each instruction can perform up to five operations simultaneously), 64K of program memory will contain a considerable amount of code compared to traditional systems. Since memory operators use full 16-bit addresses, program size may be further compacted by placing data buffers and tables above the 32K word boundary instead of in the program region. For those who still aren't satisfied, Novix is currently considering development of a 32-bit version of the NC4000 architecture.

A necessary ingredient in this design is the implementation of a return stack in hardware. The return stack can be expressly manipulated by the programmer, but it is also controlled automatically by the CALL and EXIT operations of the processor. The NC4000 features a unique parallel architecture that allows the return stack to operate simultaneously with other CPU operations.

Notice in Figure 6 that the code field is no longer needed, because Forth is the native machine code of the NC4000. The CALL routine, for instance, which serves as the prologue for colon definitions, now exists in processor logic, and is invoked whenever bit 15 is zero. In the case of constants, the pointer to the old "do-constant" routine can now be replaced by a literal-fetch instruction.

In fact, each dictionary entry, whether a definition, a constant, variable, array, or what-have-you, is most efficiently compiled as a definition and directly executed. Thus, the Novix processor uses "direct-threaded-code" (DTC), not ITC.

To summarize, the NC4000 realizes the Forth machine directly in silicon logic, without even resorting to microcode. As a result, it supports a subroutine call and return with an overhead of one clock cycle in most instances, and never more than two. Comparable CALL and RET instructions on the 8086 require a total of 34 clock cycles (and each 8086 cycle is two to five times slower than that of the Novix prototype).

The Software Implications

What does all of this mean to the programmer? It is easily demonstrated that the NC4000 runs faster than conventional micros. Because Forth instructions execute in a single clock cycle, the chip runs Forth code about 100 times faster than Forth running on a conventional processor. A benchmark using the Sieve of Eratosthenes reveals that the NC4000 runs Forth over 10 times faster than the 68000 runs its own machine code. (This benchmark was made on the NC4000P, the initial CMOS gate-array prototype; future revisions will increase the speed considerably.)

But these benchmarks only begin to show the potential power of the NC4000 architecture. A benchmark such as the Sieve represents a straightforward algorithm, which is best coded in the linear terms of conventional processors.

There is nothing to be gained by decomposing such an algorithm into smaller modules. Although a 10-times speed improvement is nothing to be ashamed of, the Sieve benchmark doesn't let the NC4000 show its true stuff.

The real wonder of the Novix chip is that it allows execution of an elegant, high-level, modular language directly in the logic of the CPU.

Forth and the NC4000 offer three distinct benefits in terms of programming methodology. The first is the interchangeability of a word without affecting any code that invokes the word. The second is the ability to use words, which tend to be shorter and simpler, instead of subroutines which tend to be longer. The third is that more efficient subroutines encourage greater freedom in the use of modular programming. Let's examine these points more closely.

Interchangeability

In the world of software development, it's no longer enough just to get a program running. The program must also be able to be maintained and easy to change. Why would a program ever need to change, once it's running? You may need to upgrade the program to run on newer equipment. You may need to add more sophisticated features to keep up with the competition. In fact, most software groups find themselves writing families of programs; that is, they write many versions of related programs in their application field, each a variant on an earlier program.

To be maintainable, a program must be easy to read, easy to understand, and, most importantly, structurally durable. By this I mean that you can rewrite a section of the program without having the whole thing collapse.

Here is an example of how Forth encourages structurally durable code. Suppose we have a variable called STATUS. In Forth this is defined as

```
VARIABLE STATUS
```

One component of our application, a device driver, for example, writes into STATUS, another component reads STATUS. References to STATUS are present throughout the program.

In Forth, variables are locations that can be acted upon. For instance, in the phrase

```
STATUS  @
```

the word **@** (pronounced "fetch") fetches the value of STATUS. In the phrase

```
STATUS  !
```

the word **!** (pronounced "store") stores a given value into STATUS. In either phrase, the execution of the word STATUS is the same: it merely returns the address of the location of its data. This address is then used by the **@** or **!** operators (or by any other memory manipulation operation).

Now suppose we must write the next generation of the program. This time we are monitoring ten devices instead of just one and we must handle a STATUS variable for each of them. What we need is a ten-cell array and a pointer to the cell in the array for the current device. In Forth we merely define:

```
CREATE STATUSES   20 ALLOT   ( 10 cells)
VARIABLE DEVICE ( number of current device)
: STATUS   ( -- adr)
  DEVICE @ 2*     STATUSES + ;
```

On the first line we have defined a 10-cell array (20 bytes) called STATUSES. On the second line we've created a variable that will contain the number of the current device (between 0 and 9). On the third line we've defined the word STATUS to fetch the number of the current device, multiply this by two (to compute the offset into the array in bytes), then add it to the address of the top of the array. The result is the address of the cell containing the status for the current device.

Now the other components of the application may write

```
STATUS   @
```

and

```
STATUS   !
```

as before, even though we have changed the definition of STATUS from that of a variable to a colon definition—from a data structure to a procedure. Forth allows us to hide the details of how STATUS is defined from the code that uses it. What appears to be a thing (a variable) to the original code is actually defined now as an action.

Words, Not Subroutines

Strictly speaking, Forth words are no different than ordinary subroutines. However, because they are optimized, and because they require no special calling command in the code that invokes them, Forth encourages a finer granularity in the decomposition of an application. In other words, Forth words tend to be much shorter and more numerous than typical subroutines in other languages. This characteristic imparts a style that is unique to Forth, and it is one of its most difficult characteristics to describe.

Here is an example of a collection of subroutines that might be written in a conventional language:

```
ENABLE-LEFT-MOTOR
ENABLE-RIGHT-MOTOR
DISABLE-LEFT-MOTOR
DISABLE-RIGHT-MOTOR
ENABLE-LEFT-SOLENOID
ENABLE-RIGHT-SOLENOID
DISABLE-LEFT-SOLENOID
DISABLE-RIGHT-SOLENOID
```

In Forth one might replace these eight subroutines with these six words:

```
ENABLE
DISABLE
LEFT
RIGHT
MOTOR
SOLENOID
```

Now our syntax reduces to simple, English words, combined in useful phrases:

```
ENABLE LEFT MOTOR
ENABLE RIGHT MOTOR
Etc.
```

Not only do we reduce the number of routines from eight to six, we also make each word's definition much shorter and simpler. The following listing shows how little code we might need to define the syntax:

```
-1 CONSTANT ENABLE ( all "one"s)
 0 CONSTANT DISABLE ( all zeros)
 1 CONSTANT LEFT (bit mask)
 2 CONSTANT RIGHT (bit mask)
 : MOTOR (action side - -) 'MOTOR SET-BIT ;
 : SOLENOID (action side - -) 'SOLENOID SET-BIT ;
```

We have defined the word SET-BIT to require three arguments: a flag, either all ones or all zeros; a bit-mask, with a "1" indicating the desired bit; and an address. SET-BIT sets the bit indicated by the mask at the given address according to the given flag. The words 'MOTOR and 'SOLENOID provide the hardware addresses of the ports that communicate to the motor and solenoid respectively; bits zero and one of these addresses control the left and right devices, respectively.

In well-written Forth code, the average definition consists of only seven words, and fits within a single line of code (or two lines if a long comment is used). Few other languages encourage you to create subroutines called LEFT and RIGHT, merely to act as "adjectives. " Nor would you find routines such as MOTOR that consist of only two other words.

In conventional programming languages, subroutines tend to be longer and more complex, covering many possible options. A conventional subroutine will contain a number of conditional branches, so that it can decide how to behave depending on the desired function. Typically, many of these decisions shouldn't have to be made at runtime; they arise only because the subroutine is too general. It has not been decomposed sufficiently.

The product of good decomposition is simplicity. Simplicity is the key not only to efficient, compact programs, but also to programs that are easy to write and, therefore, easy to test and debug. They are also easy to understand and, therefore, easy to maintain.

Modularization

The third way that optimized subroutines can enhance program maintainability is by allowing the program designer to break the application into very small, useful modules. Modularization is the key to well-written Forth code.

Modularization is more than merely chopping code up into small pieces. There is no way to automate good modularization. It is something that arises from the programmer's intimate understanding of the problem being solved. Allowing the problem to decompose itself naturally is a critical part of the early design phase—one that requires the programmer's experience, sensitivity, and artistry.

The best use of modularization is to decompose the application into components. A component is a resource, such as a UART driver, a queue, a linked list, a stack, etc. Each resource will typically consist of a number of functions and data structures.

The idea of component programming is to define all these functions and data structures in the same place, not scattered throughout the program as they are *needed.* The primary advantage is the ability to change the component if necessary by editing code in only one location.

In Forth, component programming means creating a set of names (words) to *represent* the functions and structures of each component. The set of words veil the internal algorithms and structures, the "how it works." They present only a conceptual model of the compenent described in natural language, the "what it does."

These words then become the language for describing the data structures and algorithms of components written at a higher level. The "what" of one component becomes the "how" of a higher-level component. A Forth application is nothing but a collection of components.

Thus, Forth programming consists of extending the root language toward the application, providing new commands that can be used to describe the problem at hand. Forth is an environment for creating "application-oriented languages," i.e. , programming languages designed for particular applications. Examples include robotics languages, control languages, data acquisition languages, and music languages.

Conclusion

Charles Moore, the creator of Forth and chief architect of the NC4000 processor, has claimed that the Forth chip represents "a landmark in the evolution of hardware and software. This article has only tapped the surface of the many revolutionary hardware features of the NC4000. Its high-speed computation and I/O characteristics have made the processor immediately interesting to engineers concerned with bandwidth in applications such as telecommunications and signal processing.

Its virtues are not limited to hardware features and real-time applications, however. I hope that this article has at least suggested some of the elegance of the processor's native language, and that you will have a chance to experience its unique style on your own.

Note

For more information on threaded code languages, refer to: "An Architectural Trail to Threaded-Code Systems," by Peter M. Kogge, in *Computer*, March 1982; also to *Threaded Interpretive Languages* by Ronald Loeliger, McGraw-Hill/Byte books. For further reading on the Novix processor, see "Fast processor chip takes its intructions directly from Forth," *Electronic Design*, March 21, 1985, or *Programmer's Introduction to the Novix NC4000P Microprocessor* available from Novix, 10590 N. Tantau Ave. , Cupertino, CA 95014. For a discussion of Forth methodology, see *Thinking Forth* by Leo Brodie (Prentice-Hall, 1984).

Design of a Forth Target Compiler

by Howard H. Robinson, Philip D. Morse II and Sidney A. Bowhill

Forth is a programming language that is both flexible and extensible. It also generates threaded code that is extremely compact and executes rapidly. Forth also allows the programmer to stay within a higher-level programming environment, while at the same time making use of all of the low-level capabilities of the computer. These are ideal characteristics for a language to be used in the development of programs for machines with their code in PROM.

We had been developing code for PROM-based applications for several years. Primarily we used assemblers. After we became acquainted with FIG-Forth, we realized that it had the potential to be an excellent programming environment for the development of code for PROM applications. There were, however, two serious drawbacks. Firstly, there are large sections of the compiled threaded Forth code that are not needed for the functioning of the PROM-based application, but are difficult to exclude from the code (such as the Forth compiler itself). Secondly, sections of the code must be in RAM (such as self-modifying code, variables and pseudo-registers).

By recompiling Forth programs, you can develop tight, PROMable code that executes at the top speed the target microprocessor will allow.

What was needed to rectify these problems was a post-compiler. A post-compiler, or recompiler, selects from the compiled threaded Forth code just those segments that are necessary for the execution of the application program. It also directs any code that must be resident in RAM to the RAM area of the target machine.

H. Robinson et al., *University of Illinois, 1406 W. Green St., Urbana, IL 61801*

The General Design of the CICFIG Recompiler

The program described in this article is a recompiler of the type specified above. It allows programs to be developed in Forth on a host machine and recompiled to produce the target application code. It was developed by the Central Illinois Chapter of the Forth Interest Group (CICFIG) and works in conjunction with the native host Forth compiler.

To produce the machine executable target code, source code for the application program is compiled by the standard Forth compiler within the host system. The recompiler (itself already compiled into the host Forth system) selects parts of the host system along with the compiled application code to be included in the target code. The recompiler relocates all the addresses within this code to conform to the memory map of the target machine, supplied by the application programmer. The resulting minimum length code can be executed from PROM, and potentially can perform tasks at the maximum speed of the microprocessor. The code also conforms to the memory map of the target machine, and can stand alone (i.e. does not necessarily require an operating system or monitor).

One of Forth's greatest advantages is that its interpreter can execute new-

ly compiled code. This property greatly simplifies debugging: compiled parts of the program can be tested and debugged individually with the interpreter. To extend this advantage to the process of debugging the target application, we have written the recompiler so that the target machine code can optionally include the interpreter. When the program is fully operational, the minimum code can be generated by excluding the interpreter (and the verb headers required for the interpreter to operate). In addition, once the interpreter has been excluded, the user has access only to those parts of the program allowed by the application programmer; the fact that the program is compiled Forth code should be completely transparent to the user.

Compiled Forth Code

The recompiler has as input the application program already compiled by the host Forth system. To understand what the recompiler must do, it is necessary to understand the structure of compiled FIG-Forth-model code. The model includes three parts: the start-up code, an inner interpreter, and the dictionary. The start-up code and the inner interpreter are contiguous in low memory in the host Forth system and thus can be treated as one block that must be included in the target machine code.

The start-up code initializes the registers required for the operation of the Forth inner interpreter and then jumps to the inner interpreter. The inner interpreter uses these registers to arrange the sequential execution of verbs from the dictionary.

The dictionary is a set of verbs in one linked list. Each verb defines a sequence of operations to be executed by the machine. Each verb has three parts: a header, a code field, and a parameter field. The header specifies the name of the verb (so the verb can be identified within the dictionary), a link address (so the dictionary forms one linked list of verbs), and certain special bits that are signals to the compiler of the host Forth system.

The code field contains the address of the machine executable code to which the inner interpreter will cause the processor to jump when that verb is executed at run-time. The parameter field contains the information that is necessary for the function of the verb to be accomplished. This may be data, as in the case of variables (and arrays and constants), or a list of addresses of other verbs that are to be performed sequentially (e.g. a colon definition). Alternatively, the parameter field may contain machine executable code that performs the function of the verb directly. In this case, the code field contains the address of the parameter field.

Since Forth code consists largely of lists of addresses, we have devised a novel algorithm for identifying all addresses in the host Forth system that may need to be redirected for inclusion in the target machine code. The procedure calls for two complete copies of the host Forth system, including the compiled application program. These two Forth systems are identical except that one Forth system has its origin exactly 201Hex locations higher in memory than the other (i.e. one has its origin at 100H and the other at 301H). When equivalent locations in both images are compared, all numbers (pairs of bytes) differing by 201H must be addresses. This algorithm identifies all absolute addresses, whether they are embedded in machine executable code, or used as data (such as the list of code field addresses in the parameter field of a colon definition).

Forth Systems for Target Recompilation

The recompiler described here is a program written entirely in high level Forth (MVP-Forth). The recompiler is itself compiled into an existing Forth system. The application program is also compiled into the same host Forth system. The recompiler examines the compiled application code and determines all code segments necessary for the proper functioning of the application program in the target machine. A file is generated that contains a copy of those necessary code segments. All addresses in the code segments in that file have been corrected to reflect the squeezing and the relocation of the host Forth system to generate the target code image. This file then contains the code for the application program, which can be loaded into the target machine and executed.

The code generated by the recompiler is intended to be executed on a target machine with a microprocessor that is code-compatible with the host microprocessor; thus, the recompiler does not provide the function of a cross-compiler. At present, the recompiler can only generate code to be executed on either an 8080/Z80 compatible or an 8086/88 microprocessor based target machine. Development machines for these two families of microprocessors are common and can be quite modest (for instance, only one floppy disk drive is required).

We have considered adapting other Forths to allow the use of other microprocessors (such as the 6502). Time limitations, however, have precluded this. In addition, not all Forth systems are easily adapted for recompilation of this kind. The code to be included by the recompiler in the target machine code must comply with the following three rules.

Rule #1: All non-addresses in the two compiled Forth images discussed above must be equal. Thus, data storage areas, such as arrays, must be initialized. This rule is only relaxed in the case of variables, due to the possible use of certain system variables in both the host Forth system and in the target machine code (such as PREV, USE and SEC).

Rule #2: All inter-verb relative addressing mode jumps are forbidden (intra-verb relative jumps are allowed). This rule is necessary because the algorithm for detecting addresses does not detect relative jump addresses. The versions of the target compiler available from CICFIG (see the section on Availability, page 68) have all the inter-verb relative jumps in their kernels removed.

Rule #3: Both the host and the application compiled code must not contain machine code jumps and calls from one code level verb to another code level verb. This practice, com-

mon in FIG-Forth model systems, can be eliminated by recoding certain parts of the kernel.

MVP-Forth implemented on CP/M-80 and MSDOS machines seemed to conform most nearly to all our requirements. In order to make it comply with the above stated rules, we changed the kernel of the MSDOS version of MVP-Forth to replace all relative jumps to the inner interpreter with absolute jumps. The verbs 0BRANCH, <LOOP> and <LOOP> were recoded to remove jumps to BRANCH. A relative jump in <+LOOP> to <LOOP> was changed to an absolute jump. Both <VOCABULARY79> and <VOCABULARYFIG> were changed to contain an absolute jump to DODOES. DOES> was changed to put in an absolute jump to DODOES.

The kernel of the CP/M-80 version of MVP-Forth was changed to complete the following verbs without inter-verb jumps: −, 0BRANCH, <, <LOOP>, <?TERMINAL>, <CR>, <KEY>, <LOOP>, SEC-READ, and U*. Two instructions were added to the beginning of the start-up code that read and load an initial value into the cell that is to function as the pseudo-register for the return-stack pointer. This was necessary to allow this cell to be initialized in RAM when the start-up code segment is in PROM.

In both versions, the verb <+LOOP> has an inter-verb jump to BRANCH. This case is handled as an exception by the recompiler. We intended that the compiler of Forth should not be included in the resultant target machine code. Attempts to use the compiler in the target machine will have unpredictable results.

User Interface

Generation of the target machine code with this recompiler is a two-step process. The two steps are performed by the two host Forth systems TARG1 and TARG2. These two Forth systems have the recompiler already compiled into them and are identical except that they have their origins 201H locations apart.

The application programmer specifies the memory map of the target machine by loading the constants indicated in Figure 1 (page 57). These constants indicate where the origin of the target machine code is and to what value pointers to any desired RAM areas of the target machine are initialized. For simplicity, this memory map conforms to the memory map of MVP-Forth. The application programmer may rearrange this map to suit the nature of the particular problem.

The application programmer must specify the verb in the host dictionary that is to serve as the cold start verb of the target machine code. This is done by loading the constant TCOLDCFA with the code field address (cfa) of the target machine cold start verb. This verb should perform any initialization of the Forth system that may be required by the application program (for example, the "user area" in RAM may need to be initialized). The examples given in the following sections explain other initializations that may be required, depending upon the nature of the application program. The cold start verb must invoke the application program. In this way the recompiler can detect and include in the target code all verbs necessary for the operation of the cold start verb and the application program.

Upon start-up of the target machine code, the start-up routine will first be performed. This will start the inner interpreter, which executes the cold start verb and then the application program.

The procedure for the generation of target code is outlined on page 76. The recompiler is initiated with a zero on the parameter stack. This signals to the recompiler that headers should be removed from all verbs to be included in the target machine code. Alternatively, if the parameter field address (pfa) of the root verb of the application is on top of the stack, then headers will be included in the target machine code. This allows verbs in the target machine code to be "found" (by -FIND). In this case the cold start verb usually invokes an interpreter such as the verb QUIT (see example #3).

In example #1 (see screens 200–201, page 76) the recompiler is used to generate headerless code with the Forth outer interpreter excluded for a disk-based task. This program accepts HEX numbers from the keyboard and sends them to the MSDOS device PRN (usually the printer). This is useful for setting up dot-matrix printers in various modes.

Screen #200 loads a 1 into the constant MAPFILE to signal the generation of a DOS file, <codefile>.MAP, which will contain (in ASCII) the addresses and names of all verbs in the target machine code. This screen also loads the constants that define the memory map of the target machine code (see Figure 1). Loading these constants is required prior to the initiation of the recompiler.

Screen #201 contains the application program for example #1. The verb TINPUT is for numerical input from the keyboard. This avoids the Forth verb ABORT" that threads to the outer interpreter (which we are trying to exclude). LP sends a byte to the PRN device and APP is the router and root verb of the application program. Screen #201 also has the verb TCOLD which performs the initialization required to load the "user area" of RAM. The constant TCOLDCFA has been loaded with the code field address of the verb to be executed upon start-up. The code generated by this program is less than 2K.

Example #2 (screens 202–203, see page 76) demonstrates the minimum code length that can be generated by the MSDOS MVP-Forth version of the recompiler. Screen #202 loads the memory map for the target machine. Screen #203 has the verb TCOLD, which simply performs a return to the operating system. The code generated is 412 bytes, and includes the Forth start-up code, the inner interpreter, a group of mandatory verbs (which includes the run-time routine DOCOL, for instance), and INTCALL. Eleven verbs are included in the target machine code in this example.

Example #3 (screens 204–207, see page 76) demonstrates that the Forth interpreter can be included in the target machine code. The verbs in the target code must have headers so that they can be found by the outer inter-

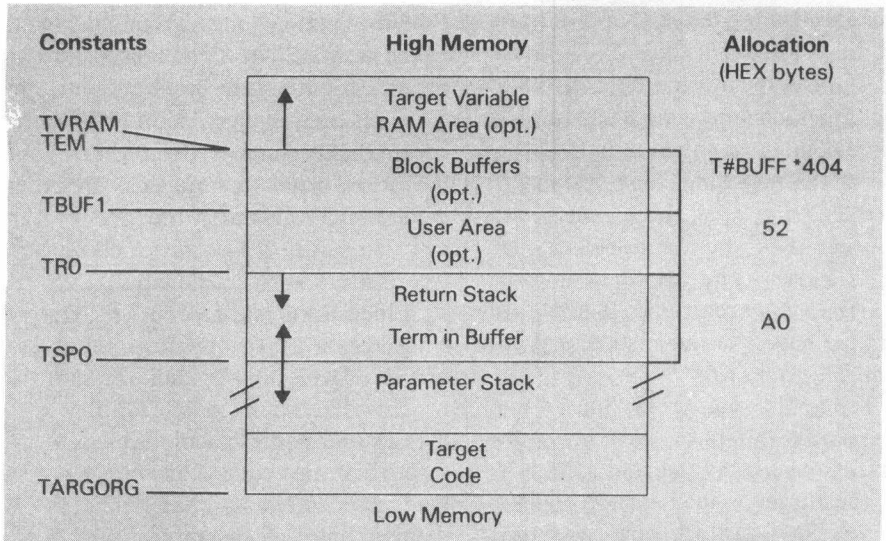


FIGURE 1
Constants Loaded by Application Programmer to Specify Memory Map

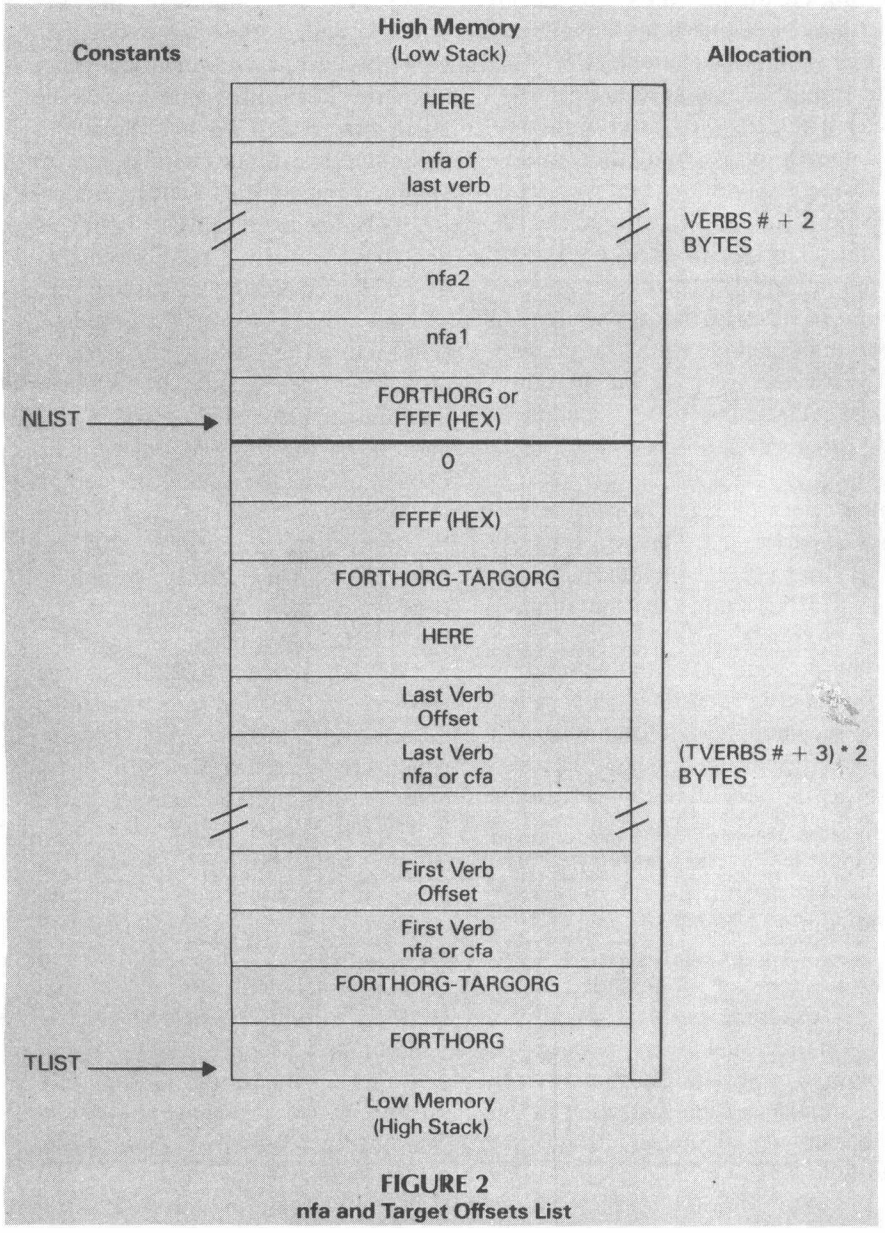


FIGURE 2
nfa and Target Offsets List

preter. Care must be taken to ensure that the Forth virtual memory block disk verbs are not included. Screens #205–206 have the Forth outer interpreter tailored to exclude references to the disk block verbs. Screen #207 has a test application program (APP) and the cold start verb TCOLD. The vocabulary link in Forth must also be initialized. The recompiler is started by typing ' APP TARGET, rather than 0 TARGET, as in a headerless case. The cfa of the root verb of the application program is on the parameter stack; knowledge of it is necessary to the recompiler since the verb TCOLD pointed to by the constant TCOLDCFA does not thread to APP.

Example #4 (screens 208–212, page 78) shows that the verbs of the Forth virtual memory disk block handlers can be included without the outer interpreter (disconnecting certain groups of routines within the host Forth system can be a bit tricky). This program reads in a number of Forth blocks (in ASCII) and writes their contents to a DOS file (in ASCII). This is done so that the file can be manipulated by your favorite word processor (such as WordStar; you must set the page length to 27 to manipulate 16 line Forth screens in document mode from the DOS file generated from this program).

In screen #208, the target memory map is loaded to indicate that 2 disk block buffers are to be allotted. Screens #209–210 set up the DOS file and transfer the information from the blocks to the DOS file. Trailing spaces are calculated by CHARS and removed by BLOCK>FOUT. Screen #211 has the verb TCOLD, which initializes the program. Several constants are loaded in TCOLD. It is important to be sure that the verbs referenced indirectly in TCOLD as literals (FIRST, LIMIT, #BUFF, and C/L) are included in the target machine code by direct references elsewhere in the program.

The Forth verb <R/W> has ABORT" in its definition. Consequently, ABORT, QUIT, etc. would all be included, although they would be quite useless, since the code generated is headerless. Thus, <R/W> should be revectored to a new verb

which does not have this problem. Screen #212 has the verb <TR/W>, which is identical to the normal <R/W> except that the ABORT" construction has been removed. <TR/W> is re-vectored through 'R/W so that <TR/W> will be included by the recompiler rather than <R/W> (and the entire Forth outer interpreter).

The program in example #5 (screens 212–217, page 78) provides the inverse function of the program in example #4. In the DOS-ASCII files the <cr> <lf> pair is present at the end of each line of text, and unnecessary spaces are not present. These programs allow interchange between Forth blocks and DOS files. Example #5 removes the <cr> <lf> pairs and inserts spaces at the end of each line to fill-out each line of the Forth block. In screen #215, the verb TVARIABLE demonstrates how the RAM variable storage area can be used.

It may be necessary to generate code for a low-memory jump table to handle a start-up jump and interrupt jumps. In simple cases, this can be handled by hand patching. In more complex cases the recompiler can be instructed to generate a file (by setting the constant MAPFILE, 1 ' MAPFILE !) containing the memory map (in ASCII) of all verbs in the target machine code. The code for the low-memory jump table can be generated by a program that queries the memory map file for addresses.

Error Messages and Warnings

During the process of target compilation, if the recompiler finds numbers that are not equivalent in the two Forth images and are not identifiable as addresses (or the two bytes of the data area of a variable), then an error is indicated ("Images not matched at <address>"). A brief dump is then produced, the dictionary is "unsmudged", and control is returned to the interpreter. This error is generally the result of an uninitialized data area within a verb. The application programmer must then inspect the dump to determine the offending verb and recode or initialize the verb. After repairing the problem, the target compilation process must be reinitiated from the beginning.

A temporary DOS file (TARGET.TMP) is generated on the default drive by TARG1 to hold an image of itself. TARG2 expects to find this file on the default drive. If this file is missing, TARG2 requests that it be replaced ("TARGET.TMP not found on default drive.").

A warning ("<address> <ID.> —Code verb assumed.") is issued by both TARG1 and TARG2 if there is a verb with an address in its code field that the recompiler cannot recognize. The offending verb will be included in the target machine code and the recompiler will continue.

The most common problem we have encountered while using this target compiler has been caused by a verb missing from the application code that seemingly should have been included. This situation arises because the missing verb is referenced only indirectly (as a literal for instance). That is, the missing verb's pfa is compiled into a verb as a literal by the verb ' at compile-time. If all references to a verb are of this type then the recompiler will not include this verb in the application code. The recompiler was designed this way because there is no way to anticipate the intention of the application programmer for the literals compiled into the definition of a verb.

If the application program crashes, the programmer should first inspect the source code to see if there are any missing verbs. This can be done by checking all verbs referenced by literals (from ') against the verbs included in the application code listed in the DOS application code map file. If a verb is missing, then the application code must be changed to force inclusion of the missing verb by explicitly referencing that verb.

It is our intention that verbs not required for the run-time activity of the application program not be included in the target machine code. We have found that the outer interpreter verbs or the disk block virtual memory verbs are sometimes included in the code when they have no intended use in the application. The most common source of this problem is run-time error handling of Forth through the verb <ABORT">. This verb leads the recompiler far into the outer interpreter and BLOCK verbs of the host Forth system. To avoid including unwanted verbs, the application programmer needs to identify an unwanted verb (e.g. QUIT or R/W etc.). A program has been devised (to be described elsewhere) to determine the threads that lead to the unwanted verbs. The application programmer can redefine the particular verb that leads to the unwanted verbs. Example #3 demonstrates this situation. In this example some of the verbs of the interpreter have been redefined to exclude any reference to the disk block virtual memory verbs. This is quite a general procedure and demonstrates the great flexibility of Forth in allowing procedures that are central to the functioning of the host system to be redefined for inclusion in the target code.

Inside the Target Compiler

The verb TARGET sequentially invokes the major tasks required by the recompilation process. The first of these tasks, APPTAG, tags (sets bit 5 of the header, the SMUDGE bit) all verbs in the host that are to be included in the application code.

The constant HEADERS, if it is non-zero (that is, the cfa of the application root verb), signals that all verbs necessary for the functioning of both the application root verb and the cold start verb (referenced by the constant TCOLDCFA) are to be tagged. In this case, headers will be included in the target code. The cold start routine must lead to an interpreter (such as QUIT).

If the constant HEADERS is zero, then only those verbs needed for the functioning of the cold start verb are tagged and the cold start verb must thread to the application program. In this case, headers will not be included in the target code. If the interpreter or other unwanted verbs are unintentionally tagged, the source code must be modified to exclude references to unwanted verbs.

To search the tree structure defined by the application program, the recursive verb TREETAG is initiated with the cfa of the cold start of the applica-

tion on the stack. TREETAG determines the type of the verb being examined at that node in the tree. Variables, constants, and machine-code verbs are simply tagged and TREETAG terminates at that level of recursion. If TREETAG is examining a colon verb, then each entry in the parameter field is the cfa of another verb that must be searched. TREETAG puts a cfa on the stack and then calls itself. This process is repeated for all entries in the parameter field. The verb PFSKIPS handles entries in the parameter field that are exceptions (entries that are to be skipped, such as branches and character strings).

Verbs that are "CREATE DOES>" defining verbs are tagged by having VERBTAG find their name field addresses with the verb MAXNFA. MAXNFA recursively searches the entire host dictionary (when initiated properly; see the glossary, page 69) and returns the maximum nfa less than the address of the DOES> part of the "CREATE DOES>" construction. Further searching of the DOES> portion of the defining verb is performed by the verb CREATE-DOES>. This verb functions similarly to TREETAG and recursively calls TREETAG by the forward-reference-executing verb >TTAG. At present, the CREATE portion of the CREATE-DOES> construction will be included (wastefully) in the target code; however, no searching or tagging of the verbs referenced in this portion is performed. The exact form of "CREATE DOES>" defining verbs varies with different Forth implementations; some adjustments to the recompiler may be necessary with other FIG-Forth model systems.

APPTAG directs TFIX;CODE and TFIXNULL to tag certain verbs that are required in the target code; also the root verb of the application is searched if headers are requested.

VERBLISTS directs the creation of two lists on the parameter stack. The first list (proceeding down in memory) is a numerically sorted list of all name field addresses in the host dictionary. This list reflects the physical order of the verbs in memory rather than the order defined by the links within headers of the verbs. This physical ordering is useful during the elimination of unwanted verbs. The second verb list contains pairs of numbers. There is one pair for each tagged verb in the host. The first number is the address of the beginning of the memory segment for that tagged verb in the host system. This is either a cfa or an nfa, depending on whether or not headers are to be included in the target code. The second number of each pair is the numerical difference between the first number of the pair and the address of the start of that verb in the target machine code. This information will be used to recalculate all of the addresses in the target code to reflect the squeezing out of the unnecessary verbs from the host Forth system.

The verb FRAMEIN sets aside the space (see Figure 2, page 57) on the stack for the nfa list and the target offsets list. The size of this space is determined by the verb COUNTVERBS which recursively searches all vocabulary branches of the host dictionary, determining the total number of verbs in the host and how many of those verbs are tagged (the variables VERBS# and TVERBS#). FRAMEIN initializes certain pointers to these lists and also initializes the beginning and ending entries that are used as signals for the list searching verbs.

The verb OFFSETS enters into the target offsets list the starting address of the verbs to be in the target code. A running count of the verb locations to be in the target code is calculated. This number is used to calculate the difference between the host address and the target code address of each verb. For each verb that will be in the target, the host code address, paired with the offset (the difference calculated above), is entered into the target offsets list.

The name field addresses of all verbs in the host are entered into the nfa list by the verb NFALISTFILL. NFALISTFILL fills this list in the sequence followed by the recursive-branch-searching routine when searching the host dictionary (starting at the verb pointed to by FORTH CONTEXT @ @). This list is not necessarily in the same sequence as the physical order of the dictionary, and thus must be sorted. Since the list usually has large areas that are already in sequence with just the order of these blocks out-of-sequence, the sort can be performed rapidly by the verb WINDOWSORT. This routine detects the limits of the out-of-sequence blocks with WINDOWTOP, FRAMEBOTTOM, and WINDOWBOTTOM. The blocks are then moved into sequence by OPENMOVE.

The relocation routines compare the host Forth image generated by TARG1 (in the DOS file TARGET.TMP) with the current resident image (TARG2). These images should differ only for addresses, and the addresses should be different by exactly 201H. All addresses within the verbs being included in the target machine code must be recalculated to account for the squeezing out of unwanted verbs in the host. Certain addresses at the beginning of the target code must also be patched to reflect the memory map of the target machine.

Either of two pairs of routines direct this relocation operation, depending on whether headers are to be included in the target code (either RELOCATE and VERB>TARG, for headers, or RELOCATENH and VERB>TARGNH, for no headers). RELOCATE directs DISCARD to advance the file containing the image of TARG1 past areas of the file that are to be discarded. The verb VERB>TARG then copies the verb currently being examined by RELOCATE to the target image file. All addresses detected by VERB>TARG are examined by REPAIRADD, which returns the address reflecting the relocation of the current verb. If a link address is being adjusted, REPAIRADD directs NEWLINK to determine a new link address to account for intervening verbs in the host that are not to be included in the target code. In this process, the host dictionary structure is lost and the target dictionary links will reflect the physical order of its verbs. REPAIRADD detects whether the addresses need to be patched by PATCHADD to take into account the memory map of the target machine. REPAIRADD then consults the target offsets list to adjust the addresses to reflect the relocation of code.

If numbers in the two images are

not equal, the verb VAR examines the code to determine if the number is in the data area of a variable. If this is not true, an error is issued by VAR and a dump produced by DUMP/=. In this case the tagged verbs are all untagged to allow examination of the host system from the outer interpreter by the application programmer.

If no headers were to be included, the same steps as above occur except that RELOCATENH and VERB> TARGNH are used. Because headers are not to be included in the target code, the vocabulary links do not need to be recalculated.

After all tagged verbs have been relocated and the application code file closed, a map of the memory allocation of the target machine is displayed by TARGETMEMORY. During the relocation process, each verb to be included in the target code is displayed on the screen along with its address in the target machine. This information is also copied to a file if requested (1 ' MAPFILE !) by the application programmer.

The DOS file interface is identical for both the CP/M-80 and MSDOS versions. The primary verbs are FC@ and FC!. These verbs fetch or store a byte from the sequential DOS file specified by the file specification address on the stack. The defining verb FILE creates a verb that functions like a verb created by VARIABLE. The verb created by the defining FILE has three parts: first, a count index byte; second, a disk memory area (128 bytes); and third, a DOS filename specification area (file control block). When the verb is executed at run-time, the address of the count byte is left on the stack. The verb DMA will advance this address to the disk memory area, and the verb FCB will advance the count byte address to the DOS filename specification area (file control block). The verbs FILE@ and FILE! are used by FC@ and FC! to fetch or store the next sequential segment to or from the disk memory area. Note that the file areas can be reused as often as desired. The verbs SPACEFILE, SPACESFILE, CRFILE, TYPEFILE and U.RFILE perform output functions to a file that are the same as analogous screen display verbs.

There are other verbs involved in maintaining the DOS file directory and in opening and closing DOS files. These verbs all return status flags except SETDMA. The verb FILENAME? queries a DOS filename from the keyboard and then sets up the filename specification area to be ready to search for or open a DOS file. File specification can also be loaded by the string-handling verb PUT$ with string verbs created by the defining verb STRING.

Complex Forth Constructions

Forth allows great flexibility in generating new programming constructions. Rather than trying to anticipate all possible new constructions, we decided to have the recompiler handle simple cases, but to allow the application programmer easy access to change the recompiler to handle more complex cases. Examples of cases that the recompiler does not currently handle are: 1.) a new verb with the function of colon; 2.) verbs that have an unusual effect on the parameter field of a verb being compiled (such as branches and literals).

The recompiler does properly handle simple CREATE-DOES> constructions (such as the defining verb STRING, screen #125). A verb such as F$ (screen #128) would be included in the application code if it had been referenced. The application code would also include the defining verb (STRING) since this verb contains the run-time action of the defined verb (F$). All other verbs necessary for the function of the DOES> portion of the defining verb would also be included.

If the recompiler finds a value in a code field of a verb that it does not recognize, then that verb will be included in the application code; however, no further searching of this verb occurs. If this verb was the result of a new colon-type definition, then not searching this verb may result in some verbs that are needed not being included in the application code.

The recompiler may not generate the proper application code if there are bytes in the parameter field of a colon-defined verb that are not a code field address (or special instances that the recompiler has been instructed to search for, for example, branches and literals).

The recompiler has been designed to allow cases such as those described above to be easily included in the recompiler by the application programmer. The verb PFSKIPS on screen #137 must be augmented to handle special bytes within the parameter field of colon-defined verbs. The verb TREETAG on screen #139 can be augmented to handle cases such as newly defined colon-type definitions.

The verb EXIT must not be used since the verb tagging routines assume that the cfa of the verb EXIT terminates each colon defined verb. The cfa of EXIT must be the last cfa in the parameter field of a colon verb and must not appear anywhere else in the definition.

Availability of the Ready-to-Run Target Compiler from CICFIG

This project grew out of the needs of several of the CICFIG members for a target compiler to use to generate PROMable code for our work on various instrumentation control applications. We are placing the result of our efforts in the public domain because we believe there is a broad-based need for such a package. This target compiler fulfills a large part of our initial desires. It has not been ported to as large a variety of microprocessors as we had initially hoped, but this may change after the release of this code.

The address relocation process is not as fast as we had hoped; however, the modular design of the recompiler will allow us easily to replace these routines with more efficient ones in a later release.

Since the generation of TARG1.COM and TARG2.COM may be a bit complicated due to the recoding of certain parts of the kernel of the host Forth system, we have decided to distribute (at a cost of $30) a package containing the target compiler and all the parts of the target compiler generator. This package contains: the ready-to-run target compiler (TARG1 and TARG2), the verb tag tracer, and a program to filter the CP/M-80 target code to move the return stack pointer to RAM. Also included are the sources (Forth blocks and DOS files) for the recompiler, the examples, and the kernels. Documentation on the generation of the target

compiler and its use to target-compile application code are also included.

At present we are supplying this material for use on MSDOS compatible computers and on CP/M-80 compatible systems. Other systems may become available later and we should be contacted. Disk formats which are available are MSDOS Version 2 IBM-PC DSDD 5¼-inch 8-sector, CP/M-80 Version 2.2 standard 8-inch and CP/M-80 NorthStar Horizon SSDD 5¼-inch.

Address of CICFIG

CICFIG
Dept. of Electrical and Computer Engineering
University of Illinois
1406 W. Green St.
Urbana, Illinois 61801

Acknowledgements

We thank the members of the Target Compiler Group and the other members of CICFIG for their input into the design and testing of this target compiler. Everett Dade provided invaluable assistance by rigorously examining the source code. This work was supported in part by: a grant from the U.S. Department of Energy (DOE DEAC02 80ER10701) to Dr. Antony Crofts, a grant from the National Foundation for Cancer Research to Dr. Harold Swartz, and by a grant from the National Aeronautics and Space Administration under grant NsG–7506.

Glossary for the CICFIG Recompiler Source Code

The following abbreviations are used in the stack pictures.

a	address (16 bit addressing)
b	byte
cfa	code field address
f	flag (0 = false)
fa	file address (the address of the first byte of a data area created by the defining word FILE)
n	number (16 bit signed)
nfa	name field address
pfa	parameter field address

!FIX (--- a) scr# 122 A variable used to avoid patching the link field of the verb ! when headers are to be included in the target code.

0BRANCHCFA (--- cfa) scr# 124 A constant that leaves the cfa of 0BRANCH.

;S (---) scr# 126 Put a zero (null) at the next byte of the input stream to terminate interpretation of a screen.

<+LOOP>CFA (--- cfa) scr# 124 A constant that leaves the cfa of <+LOOP>.

<."">CFA (--- cfa) scr# 124 A constant that leaves the cfa of <."">.

</LOOP>CFA (--- cfa) scr# 124 A constant that leaves the cfa of </LOOP>.

<;CODE>CFA (--- cfa) scr# 124 A constant that leaves the cfa of <;CODE>.

<ABORT">CFA (--- cfa) scr# 124 A constant that leaves the cfa of <ABORT">.

<LOOP>CFA (--- cfa) scr# 124 A constant that leaves the cfa of <LOOP>.

>TTAG (---) scr# 136 Execute the verb TREETAG. This allows a forward reference to TREETAG during compilation of the recompiler.

APPTAG (---) scr# 140 This verb routes the verb tagging routines.

ASCII (--- b) scr# 127 Leave the number that is the ASCII code for the next character after space in the input stream.

BRANCHCFA (--- cfa) scr# 124 A constant that leaves the cfa of BRANCH.

CASE (---) scr# 127 Defined in "Eaker Case Statement" *FORTH Dimensions* II/3 p 37, 9/80.

CLEANUPDOS (---) scr# 155 Flushes and closes the DOS files used by the relocation routines.

COUNTVERBS (nfa ---) scr# 134 Search the dictionary of the host Forth system from the verb with name field address nfa to the beginning of the dictionary. Keep a running count of the number of verbs encountered in the variable VERBS#, and the number of tagged verbs in the variable TVERBS#. This verb is recursive and is started by leaving the nfa obtained from FORTH CONTEXT @ @. The entire host Forth system is searched through recursive calls to COUNTVERBS to include all vocabularies.

CREATE-DOES> (a ---) scr# 138 Tag the verb with the largest nfa less than the address a. If a points to the DOES> portion of a verb that is a CREATE-DOES> construction, continue to TREETAG all verbs which the DOES> portion of this verb requires. Otherwise, indicate an exception ("--- Code verb assumed") and continue without searching the verb that the address a points into.

CRFILE (fa ---) scr# 134 Append a <cr> to the DOS file specified by the file address fa.

DDRIVE (---) scr# 141 Prompt the operator, and pause until a key is pressed.

DISCARD (n ---) scr# 152 Discard n bytes from the file IMG (TARGET.TMP).

DJMPLEN (--- n) scr# 120 An implementation specific constant that leaves the number of bytes of the machine code sequence that is a jump to DODOES at the start of the DOES> portion of a "CREATE DOES>" defining verb.

DMA (fa --- a) scr# 128 Leave the address of the disk transfer area of the file specified by the file address fa.

DOCODE (--- a) scr# 122 A variable used in TREETAG to determine if the verb being currently examined is a code level verb.

DOCOL (--- a) scr# *124 A con*stant that leaves the address of DOCOL.

DOCON (--- a) scr# 124 A constant that leaves the address of DOCON.

DOESJMP (--- b) scr# 120 An implementation specific constant that leaves the first byte of the machine code sequence that is a jump to DODOES at the start of the DOES> portion of a "CREATE DOES>" defining verb.

DOUSE (--- a) scr# 124 A constant that leaves the address of DOUSE.

DOVAR (--- a) scr# 124 A constant that leaves the address of DOVAR.

DOVAR? (--- a) scr# 122 A variable used as a flag by the code relocation routines to signal that a variable is currently being relocated.

DOVOC (--- a) scr# 124 A constant that leaves the address of DOVOC.

DOWN (--- a) scr# 122 A variable whose value is used as a pointer to within the name field address list.

DUMP (a n ---) (defined in MVP-FORTH Utilities) Display the contents of the n bytes of memory starting at address a.

DUMP/= (a ---) scr# 148 Display the contents of memory in the vicinity of the address a. This will help to identify the section of the Forth system that was not equivalent to the image in the DOS file TARGET.TMP. In addition, the DOS files are closed, all tagged verbs are untagged, and then the program aborts.

ENDCASE (---) scr# 127 See CASE.

ENDOF (---) scr# 127 See CASE.

EXITCFA (--- cfa) scr# 124 A constant that leaves the cfa of the verb EXIT.

**F$** (--- a n) scr# 128 A string created by the defining verb STRING. There are n bytes currently in F$, starting at address a. This string holds the name of the DOS file TARGET.TMP.

FC! (b fa ---) scr# 133 Append the byte b to the file specified by the file address fa.

FC@ (fa --- b) scr# 133 Get the next byte b from the file specified by the file address fa.

FCB (fa --- a) scr# 128 Leave on the stack the address of the file control block area specified by the file address fa.

FCLOSE (fa --- f) scr# 131 Close the DOS file specified by the file address fa. Leave the flag FFFF hex if the file cannot be found, 0 if successful.

FCREATE (fa --- f) scr# 130 Make (and open) a new DOS file with the name specified by the contents of the FCB area of the file specified by the file address fa. Leave FFFF hex if no more directory space is available, 0 if successful.

FDELETE (fa --- f) scr# 130 Delete the DOS file with the name specified by the contents of the FCB area of the file specified by the file address fa. Leave FFFF hex if the file was not found, 0 is successful.

FILE (---) scr# 128 A defining word that creates a verb that functions like a variable with 162 bytes. Use like VARIABLE: FILE <filename>. In the parameter field of <filename>, the first byte is used to keep a count on the disk memory transfer area from DOS (the 128 bytes where data transfers from DOS are stored), and the last 33 bytes are used as the DOS file control block for the DOS filename.

FILE! (fa --- f) scr# 132 Append the 128 bytes in the file transfer area of the file specified by the file address fa. Leave FFFF hex if the disk was full, 0 if the write was successful.

FILE@ (fa --- f) scr# 132 Read the next 128 bytes into the DMA area from the DOS file named in the FCB area of the file specified by the file address fa. Leave FFFF hex if no more data can be read, 0 if the read was successful.

FILECREATE (fa --- f) scr# 132 Make (and open) a new DOS file deleting if necessary an existing file with the same name. The DOS file is named by the contents of the FCB area of the file specified by the file address fa. Leave FFFF hex if no directory space was available, 0 if successful.

FILEEOF (fa --- f) scr# 132 Leave FFFF hex if the last allocated byte of the DOS file specified by the file address fa has been read; 0 otherwise.

FILENAME? (fa ---) scr# 129 Prompt the keyboard for a DOS filename. The filename is loaded into the FCB area of the file specified by the file address fa.

FINDTAG (a1 --- a2) scr# 146 Search the name field address list starting at address a1 and leave the the address a2 of the next location in the list that has a value that points to a tagged verb.

FIXLINK? (--- a) scr# 122 A variable used by the relocation routines when the target code is to contain headers. A non-zero value indicates that the next encountered address is a link and must be patched to correct for the possible absence in the target code of the verb to which the link used to point to.

FIXSMUDGE (---) scr# 147 This verb sets to zero the smudge bit of all verbs which have been tagged by the tagging routines.

FLIP (n1 --- n2) scr# 126 Swap the high and low bytes of the word on the stack.

FNSEARCH (fa ---) scr# 130 Have DOS search for the next file that matches the file named (wildcards are usually used) in the FCB area specified by the file address fa. The matching DOS filename is returned in the disk memory transfer area last assigned by the verb SETDMA. Leaves FFFF hex if no match is found, 0 if a match is found.

FOPEN (fa --- f) scr# 131 Open a DOS file with the name contained in the FCB area of the file specified by the file address fa. Leaves FFFF hex if the file cannot be found, 0 if the file was successfully found.

FORTHORG (--- a) scr# 120 An implementation specific constant that leaves the address of the origin of the current host Forth system.

FRAMEBOTTOM (a1 n --- a2) scr# 144 Search down the name field address list from the pointer a1 until the value in the list is less than the number n. Leave the pointer a2 for this entry.

FRAMEIN (--- n1...nj) scr# 142 Construct the framework for the verb lists on the stack by leaving zeros for the j cells required. Certain locations within the lists and certain pointers to the lists are initialized. The verb RAZE will return the stack to its depth prior to FRAMEIN. The number j is calculated from the variables VERBS# and TVERBS#.

FREAD (fa --- f) scr# 131 Read the next 128 bytes from the file specified by the file address fa. These bytes are transferred to the disk transfer area specified by the last call to the operating system (SETDMA). Leave FFFF hex if there is no more data available, 0 if the read was successful.

FSEARCH (fa --- f) scr# 130 Have DOS search for the file named in the FCB area of the file specified by the file address fa. Leave FFFF hex if no match is found, 0 otherwise.

FWRITE (fa --- f) scr# 131 Append 128 bytes from the last disk transfer area set (SETDMA), to the file named in the FCB area specified the file address fa. Leave FFFF hex if the disk was full, 0 if the write was successful.

HEADERS (--- n) scr# 123 A constant that leaves either 0 if the recompiler is to generate headerless code, or the pfa of the root verb of the application program if headers are to be included in the target code.

HITKEY (---) scr# 141 Pause until a key is depressed on the console.

ID.FILE (nfa fa ---) scr# 135 Copy a verb name to the file specified by the file address fa.

IMAGESAVE (---) scr# 141 Copy an image of the current Forth system from FORTHORG to HERE to the DOS file TARGET.TMP.

IMG (--- fa) scr# 128 A file specifier created by the defining verb FILE. Leaves the address fa of the disk transfer area count byte.

ISFF (f1 --- f2) scr# 130 Leaves FFFF hex if f1 is FF hex, 0 otherwise.

ISZERO (f1 --- f2) scr# 130 Leaves 0 if f1 is 0, FFFF hex otherwise.

LB (n --- b) scr# 128 Leave the low byte of the number n.

LITCFA (--- cfa) scr# 124 Leave the cfa of the verb LIT.

LOADFCB (a1 n a2 ---) scr# 141 Zero 33 bytes starting at address a2 (an FCB area), and move n bytes from address a1 to address a2+1.

LTIB (--- a) scr# 120 A constant that leaves the address of a cell in low memory that contains a pointer to the terminal-input-buffer.

**M$** (--- a n) scr# 128 A string created by the defining verb STRING. There are n bytes currently in M$, starting at address a. This string holds the DOS filename attribute MAP.

MAP (--- fa) scr# 128 A file specifier created by the defining verb FILE. Leaves the address of the disk transfer area count byte.

MAPFILE (--- f) scr# 123 A constant that leaves a zero if no map file is to be generated; otherwise, signal the generation of a DOS ASCII file <filename>.MAP with the names and addresses of the verbs included in the target code.

MAXNFA (nfa1 a nfa2 --- nfa3 a) scr# 135 This verb recursively searches the entire host dictionary starting at nfa2 (FORTH CONTEXT @ @) and returns nfa3 which is the maximum nfa that is less than or equal to a. When started, nfa3 should be 0.

NEWLINK (nfa1 --- nfa2) scr# 148 Find the host name field address nfa2 of the verb that when moved to the target code, should be pointed to by the link field of the verb with the host name field address nfa1.

NEXTNFA (nfa1 --- nfa2) scr# 150 Leave the name field address nfa2 of the next verb in physical order in the host Forth system after the verb with the name field address nfa1. If the verb with name field address nfa1 is the last verb in the dictionary (in physical order), then leave HERE.

NFALISTFILL (nfa ---) scr# 143 Search the dictionary of the host Forth system from the verb with name field address nfa, to the beginning of the dictionary. Enter the name field address of all verbs encountered in the name field address list. This verb is recursive and is started by leaving the nfa obtained from FORTH CONTEXT @ @. The entire host Forth system is searched through recursive calls to NFALISTFILL to include all vocabularies.

NFTAG (nfa ---) scr# 136 Set the smudge bit of the verb with name field address nfa to 1.

NLIST (--- a) scr# 123 A constant that leaves the address of the first entry in the name field address list.

OF (---) scr# 127 See CASE.

OFFSETS (---) scr# 146 Enter into the tagged verb offsets list the name field addresses (or code field addresses) of all tagged verbs in the host Forth system. With each entry, store the difference between this address in host Forth system and the address that verb will be at in the target code.

OPENMOVE (a1 a2 a3 ---) scr# 145 Insert at list pointer a2, the section of the name field address list that starts at pointer a1 and ends at pointer a3. See WINDOWSORT.

PATCHADD (a1 --- a2 f) scr# 149 If the flag f is 1, then the address a2 found at the address a1 should be corrected to a new address for the target code. If the flag f is 0, then the address a2 should not be corrected. This is used to allow loading of target code RAM addresses.

PATCHMAX (--- a) scr# 120 An implementation specific constant

that leaves the address above which in the host Forth system no addresses will need to be patched by PATCH-ADD for inclusion in the target code.

PFSKIPS (a1 --- a1 or a2 0) scr# 137 Check to see if the cfa (which is an entry in the parameter field of a colon verb) pointed to by a1 is in need of special handling, such as branches, loops and text. Leave the address a1 if this cfa is not an exception. Leave the address a2 and a 0 flag to bypass searching the verb referenced by a1, and continue searching at the pointer a2 + 2.

PUT$ (a n ---) scr# 125 Move the bytes (string) in the input stream starting after the next space character to the next $ character (exclusive) to the address a. The byte before the address a stores the current number of bytes in the string and the byte below that holds the maximum number of bytes in the string (see STRING). QUIT if the string is too long. The number n is ignored.

RAZE (n1...nj ---) scr# 142 Remove the verb lists (j cells) from the parameter stack (see FRAMEIN). The number j is calculated from the *variables* VERBS# and TVERBS#.

RELOCATE (--- n) scr# 152 Used when headers are to be included in the target code. This word organizes the relocation of the necessary sections of the host Forth system to generate the target code. The total number of bytes n in the target code is left on the stack.

RELOCATENH (--- n) scr# 154 Used when headers are not to be included in the target code. This word organizes the relocation of the necessary sections of the host Forth system to generate the target code. The total number of bytes n in the target code is left on the stack.

REPAIRADD (a1 --- a2) scr# 150 Leave the address a2 which is the value at the host Forth address a1, corrected to reflect the address relocation of the code to be included in the target code.

ROOF (--- a) scr# 123 A constant that leaves the address of last name field address in the name field address list.

SETDMA (fa ---) scr# 131 Instruct DOS that the disk transfer area is at memory location fa+1.

SETUPDOS (---) scr# 155 Setup the DOS files for the relocation routines of the recompiler.

SFLAG (f1 n --- f2 n) scr# 136 This verb changes the flag f1 to f2 (f2=f1-1), This is used to determine the none-of-the-above case of the CASE construction in the verb TREETAG.

SPACEFILE (fa ---) scr# 134 Append a space character to the file specified by the file address fa.

SPACESFILE (n fa ---) scr# 134 Append n space characters to the file specified by the file address fa.

STRING (n ---) scr# 125 A defining word used to create a dictionary entry for the verb with the name of the next character string in the input stream. The new verb will be allotted n+2 bytes. The first byte is the maximum allotted space for the string, the next byte is the current length of the string. At run-time the new verb will leave on the stack the address of the first byte of the string and the number of bytes in the string.

T'STREAM (--- a) scr# 129 Functions as 'STREAM except that the verb BLOCK has been removed

T#BUFF (--- n) scr# 123 A constant that leaves the number of disk block buffers required by the application program.

TARG (--- fa) scr# 128 A file specifier created by the defining verb FILE. Leaves the address of the disk transfer area count byte.

TARGET (pfa or 0 ---) scr# 157 This verb routes the operation of the target compiler. If the top of the stack has a zero, then no headers will be included in the target code; otherwise, this value is the parameter field address of the root verb of the application program and headers will also be included in the target code.

TARGETMEMORY (n ---) scr# 156 Display a memory map of the target code that has n bytes.

TARGMAP (n nfa ---) scr# 148 Display the verb name with name field address nfa and display the target code address (n + TARGORG) of that verb. Copy this same information to a DOS file if a map file has

been requested.

TARGORG (--- a) scr# 123 A constant that leaves the address of the origin of the target code.

TBUF1 (--- a) scr# 123 A constant that leaves the address of the start of the disk block buffer area of the target code.

TCOLD1 (--- a) scr# 120 An implementation specific constant that leaves the address of the cell in low memory that contains the address of the first verb to be executed upon start-up of the target code.

TCOLDCFA (--- cfa) scr# 123 A constant that leaves the cfa of the first verb to be executed upon start-up of the target code.

TEM (--- a) scr# 123 A constant that leaves the address of the top of the disk block buffer area in the target code.

TFIX;CODE (---) scr# 140 This verb tags certain ;CODE verbs that are mandatory in the target code.

TFIXNULL (---) scr# 140 This verb tags the verb X (null) and then patches the header for this verb.

TLIST (--- a) scr# 123 A constant that leaves the address of the start of the target verb offset list.

TPICK (n1 --- n2) scr# 126 Functions as PICK except that ABORT" has been removed.

TR0 (--- a) scr# 123 A constant that leaves the address of the start of the target code return stack pointer for initialization of the inner-interpreter of the target machine.

TROLL (n ---) scr# 126 Functions as ROLL except that ABORT" has been removed.

TREETAG (cfa ---) scr# 139 This recursive verb tags (sets the smudge bit to 1) all verbs in the host Forth system that are required for the function of the verb whose code field address is cfa.

TREETAGCFA (--- cfa) scr# 123 A constant that leaves the cfa of TREETAG. Used to allow forward referencing at compilation time within TARGET.

TSP0 (--- a) scr# 123 A constant that leaves the address of the start of the target code parameter stack pointer for initialization of the inner-interpreter of the target machine.

TVARIABLE (---) scr# 126 A defining word used in the form: TVARIABLE <varname>. When TVARIABLE is executed, it creates the verb <varname> in the host dictionary with its parameter field initialized to the next allotted cell in the variable RAM area of the target machine. This allows variables to be implemented in RAM when the target code is in ROM. The application programmer must set up the target variable RAM area by loading the constant TVRAM.

TVERBS# (--- a) scr# 122 A variable used to hold the number of verbs that have been tagged by the tagging routines.

TVRALLOT (n ---) scr# 126 Add the number n to the variable TVRAMOFF. This allows target variable RAM bytes to be allotted.

TVRAM (--- a) scr# 123 A constant that leaves the address of the start of the target variable RAM area. This value must be loaded by the application programmer if needed.

TVRAMOFF (--- a) scr# 122 A variable that contains the current number of bytes of the target variable RAM area that have been allotted.

TWORD (b --- a) scr# 129 Functions as <WORD> except that ABORT" has been removed.

TYPEFILE (a n fa ---) scr# 134 Copy n characters starting at address a to the file specified by the file address fa.

U.RFILE (n1 n2 fa ---) scr# 134 Convert the number n1 to an unsigned ASCII number right justified n2 spaces. Append these ASCII characters to the file specified by the file address fa.

U8 (--- a) scr# 120 A constant that leaves the address of the cell in low memory that contains the value to initialize the eight user variable.

VAR (a ---) scr# 149 If the address a is the pfa of a VARIABLE defined verb (or the return stack pointer cell address) then copy the contents of that cell to the DOS file specified by TARG. Otherwise, dump the contents of the host Forth system in vicinity of the address a and indicate an error.

VERB>TARG (nfa ---) scr# 151 The host Forth verb (including the header) with name field address nfa is copied to the DOS file specified by TARG. Address relocation and patches are performed.

VERB>TARGNH (cfa ---) scr# 153 The host Forth verb (excluding the header) with code field address cfa is copied to the DOS file specified by TARG. Address relocation and patches are performed.

VERBLISTS (---) scr# 147 This routine routes the construction and filling in of the verb lists on the parameter stack.

VERBS# (--- a) scr# 122 A variable used to hold the number of verbs that are in the host Forth system dictionary.

VERBTAG (a ---) scr# 136 Tag (set the smudge bit to 1) the verb whose code field address or DOES> jump address is a.

WINDOWBOTTOM (a1 a2 --- a3) scr# 144 Examine the values in the name field address list starting with the element pointed to by a1. Continue so long as they are decreasing and greater than the contents of the element of the list pointed to by a2. When this is not true, leave a3, which points to the element just beyond this point.

WINDOWSORT (a ---) scr# 145 Sort the list of unsigned numbers (name field addresses) between the high memory address a and low memory location specified by the value FFFF hex in the list. At the end of the sort, change the entry at the bottom of the list to FORTHORG.

WINDOWTOP (a1 --- a2) scr# 144 Index down the list of unsigned numbers starting at pointer a1 until the values in the list are no longer monotonically decreasing. Leave the pointer a2 to where this occurs. See WINDOWSORT.

X (---) scr# 140 This is a pseudonym for null that does not have the compile-time parts of the host system Forth verb *null*. This version of null will be present in the target code.

DDJ

Procedure

TARG1 (from TARG1.COM DOS)

(A) load screen with memory map (scr# 200, example #1)
(B) load screen(s) with application (scr# 201, example #1)
(C) load screen with cold start (scr# 201, example #1)

0 TARGET (0 => no headers)

(control will be returned to the operating system)

TARG2 (from TARG2. COM DOS)

(A-C) exactly as above (200 201 THRU)

0 TARGET (0 => no headers)

(upon request, supply the DOS filename for the target code)
(control will be returned to the interpreter when complete)

Forth Compiler Listing **(Text begins on page 52)**

```
SCR #200
 1 ( No headers, No block verbs; Target Memory Map Constants) HEX
 2 ( This program outputs to printer, ASCII chars typed in in hex)
 3
 4 1 ' MAPFILE !
 5
 6 6000 ' TVRAM !
 7 6000 ' TEM !
 8 TEM ' TBUF1 !
 9 TBUF1 52 - ' TRO !
10 TRO A0 - ' TSPO !
11 100 ' TARGORG !
12 DECIMAL
13
14
15
16


SCR #201
 1 ( LP TINPUT APP TCOLD) HEX
 2
 3 : LP  ( b --- )  5 SWAP SYSCALL ;
 4
 5 : TINPUT ( --- n )  0 0 ." ? " QUERY 1 TWORD CONVERT DROP DROP ;
 6
 7 : APP  ( --- ) HEX
 8  ." To send an ASCII character to the printer," CR
 9  ." type a HEX number and (cr) after '?' {FF to terminate}." CR
10  BEGIN TINPUT DUP FF = NOT WHILE SPACE LP REPEAT 0 0 SYSCALL ;
11
12 : TCOLD ( --- ) 0 EPRINT !
13  INIT-USER UP @ 6 + 30 CMOVE PAGE APP ;
14
15 ' TCOLD CFA ' TCOLDCFA !
16 DECIMAL


SCR #202
 1 ( No headers, No block verbs; MINIMUM Test; Target Memory Map)
 2 HEX
 3 0 ' MAPFILE !
 4
 5 6000 ' TVRAM !
 6 6000 ' TEM !
 7 TEM ' TBUF1 !
 8 TBUF1 ' TRO !
 9 TRO A - ' TSPO !
10 100 ' TARGORG !
11 DECIMAL
12
13
14
15
16


SCR #203
 1 ( TCOLD) HEX
 2
 3 : TCOLD  0 0 0 0 21 INTCALL ;
 4
 5 ' TCOLD CFA ' TCOLDCFA !
 6 DECIMAL
 7
 8
 9
10
11
12
13
14
15
16


SCR #204
 1 ( Headers, No block verbs; Target Memory Map Constants) HEX
 2
 3 1 ' MAPFILE !
 4
 5 6000 ' TVRAM !
 6 6000 ' TEM !
 7 TEM ' TBUF1 !
 8 TBUF1 52 - ' TRO !
 9 TRO A0 - ' TSPO !
10 100 ' TARGORG !
11 DECIMAL
12
13
14
15
16
```

Forth Compiler Listing (Listing continued, text begins on page 52)

```
SCR #205
 1 ( TABORTCFA TNABORT TNUMBER T-FIND)
 2
 3 0 CONSTANT TABORTCFA
 4
 5 : TNABORT ." NOT RECOGNIZED" TABORTCFA EXECUTE ;
 6
 7 : TNUMBER  ( a --- d )
 8  0 0 ROT DUP 1+ C@ ASCII - = DUP )R + -1 DPL !
 9  CONVERT DUP C@ BL )
10  IF DUP C@ ASCII . = NOT IF TNABORT THEN 0 DPL !
11     CONVERT DUP C@ BL ) IF TNABORT THEN
12  THEN DROP R) IF NEGATE THEN ;
13
14 : T-FIND  BL TWORD CONTEXT @ @ (FIND) ;
15
16


SCR #206
 1 ( TINTERPRET TQUIT TABORT)
 2
 3 : TINTERPRET  ( --- )
 4  BEGIN T-FIND
 5     IF DROP CFA EXECUTE
 6     ELSE HERE TNUMBER DPL @ 1+ NOT IF DROP THEN
 7     THEN
 8  AGAIN ;
 9
10 : TQUIT  ( --- )
11  BEGIN CR RP! QUERY TINTERPRET ." OK" AGAIN ;
12
13 : TABORT ( --- )
14  SP! [COMPILE] FORTH TQUIT ;
15
16 ' TABORT CFA ' TABORTCFA !


SCR #207
 1 ( APP TCOLD)
 2 HEX
 3 : APP CR ." HELLO FROM APP" BEGIN ?TERMINAL ?DUP UNTIL ;
 4
 5 : T.R  ( n1 n2 --- ) )R 0 SWAP OVER DUP D+- (# #S ROT SIGN #)
 6  R) OVER TYPE ;
 7
 8 : TCOLD ( --- )  0 EPRINT !
 9  INIT-USER UP @ 6 + 30 CMOVE
10  PAGE ." QUIT OK" CR
11  INIT-FORTH @ ' FORTH 2+ !
12  DECIMAL TABORT HEX T.R ;
13 DECIMAL
14
15 ' TCOLD CFA ' TCOLDCFA !
16


SCR #208
 1 ( Block verbs, No headers; BLOCKASC; Target Memory Map) HEX
 2 ( Generates DOS-ASCII files of Forth disk block areas)
 3
 4 1 ' MAPFILE !
 5
 6 2 ' T#BUFF !
 7
 8 6000 ' TVRAM !
 9 6000 ' TEM !
10 TEM 404 T#BUFF * - ' TBUF1 !
11 TBUF1 52 - ' TRO !
12 TRO A0 - ' TSPO !
13 100 ' TARGORG !
14 DECIMAL
15
16


SCR #209
 1 ( CHARS BLOCK)FOUT) HEX
 2
 3 : CHARS  ( a --- n )
 4  DUP C/L + 1-
 5  BEGIN DUP C@ 20 = OVER 4 TPICK ) AND
 6  WHILE 1-
 7  REPEAT SWAP - 1+ ;
 8
 9 : BLOCK)FOUT  ( ba fa --- )
10  SWAP DUP 400 + SWAP
11  DO I CHARS I + I DO I C@ OVER FC! 1 /LOOP
12     D OVER FC! A OVER FC! C/L
13  /LOOP DROP ;
14 DECIMAL
15
16


SCR #210
 1 ( FOUT TINPUT BLOCKASC) HEX
 2
 3 FILE FOUT
 4
 5 : TINPUT ( --- n )  0 0 ." ? " QUERY 1 TWORD CONVERT DROP DROP ;
 6
 7 : BLOCKASC  ( --- )
 8  CR ." Starting block # (DEC)" TINPUT
 9  ." # of blocks" TINPUT
10  ." DRIVE (0,1)" TINPUT 0 MAX 1 MIN IF DR1 ELSE DR0 THEN
11  CR ." DOS file out name" FOUT FILENAME? FOUT FILECREATE DROP
12  OVER + SWAP
13  DO I BLOCK FOUT BLOCK)FOUT LOOP
14  1A FOUT FC! FOUT FILE! DROP FOUT FCLOSE DROP
15  CR ." Task complete." CR BYE ;
16 DECIMAL


SCR #211
 1 ( TCOLD) HEX
 2
 3 : TCOLD ( --- ) 0 EPRINT !
 4  INIT-USER UP @ 6 + 30 CMOVE PAGE
 5  TBUF1 ' FIRST ! TEM ' LIMIT ! T#BUFF ' #BUFF ! EMPTY-BUFFERS
 6  CR I DENSITY ! FIRST USE ! FIRST PREV !
 7  DR0  40 ' C/L !
 8  DECIMAL BLOCKASC ;
 9
10 ' TCOLD CFA ' TCOLDCFA !
11 DECIMAL
12
13
14
15
16


SCR #212
 1 ( (TR/W))
 2
 3 : (TR/W)  ( n --- ) USE @ )R ROT USE  ! SWAP MAX-DRV 0
 4  DO I DR-DEN DENSITY ! DUP BPDRV - -1 )
 5     IF BPDRV - I 1+ MAX-DRV = IF BYE THEN
 6     ELSE I DRIVE ! LEAVE
 7     THEN
 8  LOOP SPBLK * SPBLK 0
 9  DO DDUP T&SCALC
```

(Continued on page 81)

Forth Compiler Listing (Listing continued, text begins on page 52)

```
10     IF SEC-READ ELSE SEC-WRITE THEN
11     1+ 1024  SPBLK / USE +!
12  LOOP DDROP R) USE ! ;
13
14 ' (TR/W) CFA 'R/W !
15
16


SCR #213
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16


SCR #214
 1 ( Block verbs, No headers; ASCBLOCK; Target Memory Map) HEX
 2 ( Generates Forth disk block areas for a DOS-ASCII files)
 3
 4 1 ' MAPFILE !
 5
 6 2 ' T#BUFF !
 7
 8 6000 ' TVRAM !
 9 6000 ' TEM !
10 TEM 404 T#BUFF * - ' TBUF1 !
11 TBUF1 52 - ' TRO !
12 TRO A0 - ' TSPO !
13 100 ' TARGORG !
14 DECIMAL
15
16


SCR #215
 1 ( CURBLOCK BCOUNT FIN BC! LINE>BLOCK) HEX
 2
 3 TVARIABLE CURBLOCK  TVARIABLE BCOUNT
 4
 5 FILE FIN
 6
 7 : BC!  ( b --- ) BCOUNT DUP @ 400 =
 8  IF 1 CURBLOCK +! UPDATE CURBLOCK @ BUFFER DROP 0 OVER ! THEN
 9  DUP @ DUP 1+ ROT ! PREV @ 2+ + C! ;
10
11 : LINE>BLOCK  ( b --- ) DUP D =
12  IF 0 ELSE 7F AND BC! 1
13     BEGIN FIN FC@ 7F AND DUP D = NOT
14     WHILE BC! 1+
15     REPEAT DROP FIN FC@ DROP C/L SWAP -
16  THEN BEGIN DUP WHILE 1- 20 BC! REPEAT DROP ; DECIMAL


SCR #216
 1 ( TYPEFILENAME TINPUT ASCBLOCK) HEX
 2 : TYPEFILENAME  ( fa --- )
 3  DUP FCB 1+ 8 TYPE ." ." FCB 9 + 3 TYPE ;
 4
 5 : TINPUT ( --- n )  0 0 ." ? " QUERY 1 TWORD CONVERT DROP DROP ;
 6
 7 : ASCBLOCK  ( --- )
 8  CR ." DOS in file name" FIN FILENAME? FIN FOPEN
 9  IF CR FIN TYPEFILENAME ."  can't be opened." BYE THEN
10  CR ." Starting block # (DEC)" TINPUT ." Drive # (0,1)" TINPUT
11  0 MAX 1 MIN IF DR1 ELSE DR0 THEN
12  DUP CURBLOCK ! 0 BCOUNT ! BUFFER DROP
13  BEGIN FIN FC@ DUP 1A = NOT WHILE LINE>BLOCK REPEAT DROP
14  BEGIN BCOUNT @ 400 < WHILE 20 BC! REPEAT
15  UPDATE SAVE-BUFFERS FIN FCLOSE DROP
16  CR ." Task complete." CR BYE ; DECIMAL


SCR #217
 1 ( TCOLD) HEX
 2
 3 : TCOLD ( --- ) 0 EPRINT !
 4  INIT-USER UP @ 6 + 30 CMOVE PAGE
 5  TBUF1 ' FIRST ! TEM ' LIMIT ! T#BUFF ' #BUFF ! EMPTY-BUFFERS
 6  CR 1 DENSITY ! FIRST USE ! FIRST PREV !
 7  DR0  40 ' C/L !
 8  DECIMAL ASCBLOCK ;
 9
10 ' TCOLD CFA ' TCOLDCFA !
11 DECIMAL
12
13
14
15
16


SCR #120
 1 ( Implementation specific constants for CP/M) HEX
 2
 3 INIT-FORTH 12 - CONSTANT FORTHORG   ( Host origin)
 4 148 CONSTANT PATCHMAX   ( end of patch area)
 5 110 CONSTANT TCOLD1     ( address for patch for jump to cold)
 6 116 CONSTANT U8         ( address for patch for user#8)
 7 118 CONSTANT LTIB       ( address for patch for Term-in-buffer)
 8 CD  CONSTANT DOESJMP    ( first byte of run-time create-does))
 9 3   CONSTANT DJMPLEN    ( # of bytes for run-time c-d jmp)
10 DECIMAL
11
12
13
14
15
16


SCR #121
 1 ( MS-DOS Implementation specific addresses) HEX
 2
 3 INIT-FORTH 1B - ' FORTHORG !
 4 151 ' PATCHMAX !                                    ( MS-DOS)
 5 119 ' TCOLD1 !                                      ( MS-DOS)
 6 11F ' U8 !                                          ( MS-DOS)
 7 121 ' LTIB !                                        ( MS-DOS)
 8 88  ' DOESJMP !                                     ( MS-DOS)
 9 5   ' DJMPLEN !                                     ( MS-DOS)
10 DECIMAL
11
12
13
14
15
16
```

Forth Compiler Listing (Listing continued, text begins on page 52)

```
SCR #122
 1 ( VARIABLES) HEX
 2
 3 VARIABLE TVRAMOFF  0 TVRAMOFF !
 4 VARIABLE DOCODE    0 DOCODE !
 5 VARIABLE VERBS#    0 VERBS# !
 6 VARIABLE TVERBS#   0 TVERBS# !
 7 VARIABLE DOWN      0 DOWN !
 8 VARIABLE FIXLINK?  0 FIXLINK? !
 9 VARIABLE !FIX      0 !FIX !
10 VARIABLE DOVAR?    0 DOVAR? !
11 DECIMAL
12
13
14
15
16
```

```
SCR #123
 1 ( CONSTANTS see Examples and Glossary) HEX
 2 0 CONSTANT MAPFILE     ( These constants are loaded by the user)
 3 0 CONSTANT T#BUFF
 4 0 CONSTANT TVRAM
 5 0 CONSTANT TEM
 6 0 CONSTANT TBUF1
 7 0 CONSTANT TRO
 8 0 CONSTANT TSPO
 9 0 CONSTANT TARGORG
10 0 CONSTANT TCOLDCFA
11
12 0 CONSTANT ROOF    ( These constants are initialized by TARGET)
13 0 CONSTANT NLIST
14 0 CONSTANT TLIST
15 0 CONSTANT HEADERS
16 0 CONSTANT TREETAGCFA DECIMAL
```

```
SCR #124
 1 ( CONSTANTS for detection of special cases)
 2 ' (+LOOP)  CFA CONSTANT (+LOOP)CFA
 3 ' LIT      CFA CONSTANT LITCFA
 4 ' (.")     CFA CONSTANT (.")CFA
 5 ' (ABORT") CFA CONSTANT (ABORT")CFA
 6 ' EXIT     CFA CONSTANT EXITCFA
 7 ' OBRANCH  CFA CONSTANT OBRANCHCFA
 8 ' (LOOP)   CFA CONSTANT (LOOP)CFA
 9 ' (/LOOP)  CFA CONSTANT (/LOOP)CFA
10 ' (;CODE)  CFA CONSTANT (;CODE)CFA
11 ' BRANCH   CFA CONSTANT BRANCHCFA
12 ' :      CFA @ CONSTANT DOCOL
13 ' SPO    CFA @ CONSTANT DOUSE
14 ' 0      CFA @ CONSTANT DOCON
15 ' PREV   CFA @ CONSTANT DOVAR
16 ' FORTH  CFA @ CONSTANT DOVOC
```

```
SCR #125
 1 ( STRING PUT$ from A.Winfield "The Complete FORTH" Sigma 1983)
 2 HEX
 3 : STRING  ( n --- )
 4  CREATE DUP C, 0 C, ALLOT
 5  DOES) 2+ DUP 1- C@ ;
 6
 7 : PUT$  ( a n --- )
 8  DROP 1-
 9  DUP 1- C@
10  24 WORD DROP
11  HERE C@ <
12  IF ." String too big" DROP QUIT THEN
13  HERE DUP C@ 1+
14  ROT SWAP CMOVE ;
15 DECIMAL
16
```

```
SCR #126
 1 ( TVRALLOT TVARIABLE TPICK TROLL ;S FLIP) HEX
 2
 3 : TVRALLOT  ( n --- )  TVRAMOFF +! ;
 4
 5 : TVARIABLE  ( --- )  TVRAMOFF DUP @ 2 ROT +! TVRAM + CONSTANT ;
 6
 7 : TPICK  ( n1 --- n2 )  2* SP@ + @ ;
 8
 9 : TROLL  ( n --- )  1+ DUP TPICK SWAP 2* SP@ +
10  BEGIN DUP 2- @ OVER ! 2- SP@ OVER U< NOT UNTIL DDROP ;
11
12 : ;S  ( --- ) 0 'STREAM ! ; DECIMAL
13
14 ;S   HEX
15 : FLIP  ( n --- n )
16  0 100 U/MOD SWAP 100 * + ; DECIMAL
```

```
SCR #127
 1 ( EAKER CASE STATEMENT FROM CEE 9/80 FORTH DIMENSIONS II/3 P 37)
 2
 3 : CASE   ?COMP CSP @ SP@ CSP ! 4 ; IMMEDIATE
 4
 5 : OF  4 ?PAIRS COMPILE OVER COMPILE = COMPILE OBRANCH HERE 0
 6       , COMPILE DROP 5 ;  IMMEDIATE
 7
 8 : ENDOF  5 ?PAIRS COMPILE BRANCH HERE 0 , SWAP 2
 9         [COMPILE] THEN 4 ; IMMEDIATE
10
11 : ENDCASE  4 ?PAIRS COMPILE DROP BEGIN SP@ CSP @ = 0=
12            WHILE 2  [COMPILE] THEN REPEAT CSP ! ; IMMEDIATE
13
14 : ASCII  BL WORD  1+ C@  [COMPILE] LITERAL ; IMMEDIATE
15
16
```

```
SCR #128
 1 ( F$ M$ FILE IMG TARG MAP DMA FCB LB) HEX
 2
 3 B STRING F$ F$ PUT$ TARGET  TMP$
 4 3 STRING M$ M$ PUT$ MAP$
 5
 6 : FILE  ( --- ) CREATE HERE A2 ALLOT A2 0 FILL ;
 7
 8 FILE IMG
 9 FILE TARG
10 FILE MAP
11
12 : DMA  ( fa --- a ) 1+ ;
13 : FCB  ( fa --- a ) 81 + ;
14 : LB  ( n --- b ) FF AND ;
15 DECIMAL
16
```

```
SCR #129
 1 ( T'STREAM TWORD FILENAME?) HEX
 2 : T'STREAM  ( --- a )  TIB @ >IN @ + ;
 3
 4 : TWORD  ( b --- a )  T'STREAM SWAP ENCLOSE DDUP >
 5  IF DDROP DDROP 0 HERE !
 6  ELSE >IN +! OVER - DUP >R HERE C! + HERE 1+ R> 1+ CMOVE
 7  THEN HERE ;
 8
 9 : FILENAME?  ( fa --- )
10  ." ? " QUERY
11  TIB @ 1+ C@ ASCII : =                    ( default or select?)
12  IF  3A TWORD 1+ C@ 40 - 0 MAX OF MIN OVER FCB C! ( select disk)
13  ELSE 0 OVER FCB C! THEN                    ( default disk)
14  DUP FCB 0C + 15 0 FILL DUP FCB 1+ 0B BLANK       ( clear FCB)
15  2E TWORD COUNT 8 MIN 3 TPICK FCB 1+ SWAP CMOVE    ( load FCB)
16  BL TWORD COUNT 3 MIN 3 TROLL FCB 9 + SWAP CMOVE ; DECIMAL
```

```
SCR #130
 1 ( ISFF FSEARCH FNSEARCH FDELETE FCREATE) HEX
 2
 3 : ISFF  ( f --- f )  FF = IF FFFF ELSE 0 THEN ;
 4
 5 : ISZERO  ( f --- f )  0= IF 0 ELSE FFFF THEN ;
 6
 7 : FSEARCH  ( fa --- f )  FCB 11 SWAP SYSCALL LB ISFF ;
 8
 9 : FNSEARCH ( fa --- f )  FCB 12 SWAP SYSCALL LB ISFF ;
10
11 : FDELETE  ( fa --- f )  FCB 13 SWAP SYSCALL LB ISFF ;
12
13 : FCREATE  ( fa --- f )
14  DUP 81 0 FILL FCB 16 SWAP SYSCALL LB ISFF ;
15 DECIMAL
16
```

```
SCR #131
 1 ( FOPEN FCLOSE SETDMA FREAD FWRITE) HEX
 2
 3 : FOPEN ( fa --- f )
 4   DUP 80 SWAP C! FCB F SWAP SYSCALL LB ISFF ;
 5
 6 : FCLOSE  ( fa --- f )
 7  FCB 10 SWAP SYSCALL LB ISFF ;
 8
 9 : SETDMA  ( fa --- )
10  DMA 1A SWAP SYSCALL DROP ;
11
12 : FREAD  ( fa --- f )
13  FCB 14 SWAP SYSCALL LB ISZERO ;
14
15 : FWRITE  ( fa --- f )
16  FCB 15 SWAP SYSCALL LB ISZERO ; DECIMAL
```

```
SCR #132
 1 ( SETDMA FREAD FWRITE FILE@ FILE! FILEEOF) HEX
 2
 3 : FILE@  ( fa --- f )
 4  DUP SETDMA DUP FREAD IF FFFF FF ELSE 0 0 THEN ROT C! ;
 5
 6 : FILE!  ( fa --- f )  DUP C@
 7  IF DUP SETDMA DUP FWRITE SWAP 81 0 FILL ELSE DROP 0 THEN ;
 8
 9 : FILEEOF ( fa --- f )
10  C@ FF = IF FFFF ELSE 0 THEN ;
11
12 : FILECREATE  ( fa --- f )
13  DUP FSEARCH NOT IF DUP FDELETE DROP THEN FCREATE ;
14 DECIMAL
15
16
```

```
SCR #133
 1 ( FC@ FC! IMG TARG MAP F$ M$) HEX
 2
 3 : FC@  ( fa --- b )
 4  DUP C@ 80 = IF DUP FILE@ DROP THEN
 5  DUP C@ 1+ DUP 3 TPICK C! + C@ ;
 6
 7 : FC!  ( b fa --- )
 8  DUP C@ 80 = IF DUP FILE! DROP THEN
 9  DUP C@ 1+ DUP 3 TPICK C! + C! ;
10 DECIMAL
11
12
13
14
15
16
```

```
SCR #134
 1 ( SPACEFILE SPACESFILE TYPEFILE U.RFILE) HEX
 2 : SPACEFILE  ( fa --- ) 20 SWAP FC! ;
 3
 4 : SPACESFILE  ( n fa --- )  SWAP 0 MAX ?DUP
 5  IF 0 DO DUP SPACEFILE LOOP THEN DROP ;
 6
 7 : CRFILE  ( fa --- )  D OVER FC! A SWAP FC! ;
 8
 9 : TYPEFILE  ( a n fa --- )  ROT ROT DUP 0)
10  IF OVER + SWAP DO I C@ 7F AND OVER FC! 1 /LOOP
11  ELSE DDROP THEN DROP ;
12
13 : U.RFILE  ( # r fa --- )  ROT ROT
14  0 SWAP )R SWAP OVER DUP D+-
15  (# #S ROT SIGN #) R) OVER -
16  4 ROLL DUP ROT SWAP SPACESFILE TYPEFILE ; DECIMAL
```

```
SCR #135
 1 ( ID.FILE MAXNFA) HEX
 2
 3 : ID.FILE  ( nfa fa --- )  SWAP COUNT 1F AND ROT TYPEFILE ;
 4
 5 : MAXNFA  ( nfa1 a nfa2 --- nfa3 a ) [ SMUDGE ]
 6  BEGIN DUP DUP 0= SWAP @ A081 = OR NOT
 7  WHILE DUP PFA DUP CFA @ DOVOC = SWAP 4 + @ 0= NOT AND
 8     IF ROT ROT 3 PICK PFA 2+ @ MAXNFA ROT THEN
 9     OVER OVER U( NOT
10     IF ROT DDUP U( NOT IF DROP DUP THEN ROT ROT THEN
11     PFA 4 - @
12  REPEAT DROP [ SMUDGE ] ;
13 DECIMAL
14
15
16
```

```
SCR #136
 1 ( NFTAG VERBTAG SFLAG )TTAG) HEX
 2
 3 : NFTAG ( nfa --- )
 4  DUP C@ 20 OR SWAP C! ;
 5
 6 : VERBTAG ( cfa --- )
 7  DUP 2- @ (;CODE)CFA =
 8  IF 0 SWAP [COMPILE] FORTH CONTEXT @ @ MAXNFA DROP
 9  ELSE 2+ NFA
10  THEN NFTAG ;
11
12 : SFLAG  ( f --- f ) SWAP 1- SWAP ;
13
14 : )TTAG  ( --- ) TREETAGCFA EXECUTE ;
15 DECIMAL
16
```

```
SCR #137
 1 ( PFSKIPS)
 2
 3 : PFSKIPS ( a1  --- a1 / a2 0 )
 4  DUP @
 5  CASE  LITCFA OF LITCFA )TTAG 2+ 0 ENDOF
 6        (.")CFA OF (.")CFA )TTAG 2+ COUNT + 2- 0 ENDOF
 7        (ABORT")CFA OF (ABORT")CFA )TTAG 2+ COUNT + 2- 0 ENDOF
 8        OBRANCHCFA OF OBRANCHCFA )TTAG 2+ 0 ENDOF
 9        BRANCHCFA OF BRANCHCFA )TTAG 2+ 0 ENDOF
10        (LOOP)CFA OF (LOOP)CFA )TTAG 2+ 0 ENDOF
11        (/LOOP)CFA OF (/LOOP)CFA )TTAG 2+ 0 ENDOF
12        (+LOOP)CFA OF (+LOOP)CFA )TTAG (LOOP)CFA )TTAG 2+ 0 ENDOF
13  ENDCASE ;
14
15
16
```

Forth Compiler Listing (Listing continued, text begins on page 52)

```
SCR #138
 1 ( CREATE-DOES)) HEX
 2
 3 : CREATE-DOES)  ( cfa --- )
 4  @ DUP VERBTAG DUP C@ DOESJMP =
 5  IF ' DOES) CFA VERBTAG DJMPLEN 2- +
 6     BEGIN 2+ DUP @ EXITCFA = NOT
 7     WHILE PFSKIPS ?DUP IF DUP @ )TTAG THEN
 8     REPEAT
 9  ELSE DUP DUP HEX CR U. 2+ NFA ID.
10       ." ---Code verb assumed." CR DECIMAL
11  THEN DROP ;
12 DECIMAL
13
14
15
16


SCR #139
 1 ( TREETAG) HEX
 2 : TREETAG  ( cfa --- )  [ SMUDGE ]
 3  DUP 2+ NFA C@ 20 AND NOT
 4  IF 0 SWAP DUP 2+ DOCODE ! DUP VERBTAG DUP @
 5    CASE DOCODE @ OF SFLAG ENDOF
 6      DOCOL OF
 7        BEGIN 2+ DUP @ DUP EXITCFA = SWAP (;CODE)CFA = OR NOT
 8        WHILE PFSKIPS ?DUP IF DUP @ TREETAG THEN
 9        REPEAT SFLAG ENDOF
10      DOUSE OF DUP 2+ @ DUP 14 ) SWAP 36 ( AND
11        IF DUP EXECUTE @ TREETAG THEN SFLAG ENDOF
12      DOVAR OF SFLAG ENDOF
13      DOCON OF SFLAG ENDOF
14    ENDCASE SWAP NOT
15    IF DUP CREATE-DOES) THEN
16  THEN DROP [ SMUDGE ] ; DECIMAL


SCR #140
 1 ( TFIX;CODE X TFIXNUL APPTAG) HEX
 2 : TFIX;CODE  (  --- )
 3  ' CREATE CFA VERBTAG ' CONSTANT CFA VERBTAG
 4  ' : CFA VERBTAG ' USER CFA VERBTAG ' EXIT CFA VERBTAG
 5  HEADERS IF ' BYE CFA TREETAG THEN ;
 6
 7 : X R) DROP ;
 8 : TFIXNULL ( --- )
 9  ' X CFA TREETAG  80E0 ' X NFA ! ;          ( patch for null)
10
11 : APPTAG  ( --- )  CR CR ." Start tagging."
12  TFIX;CODE TCOLDCFA TREETAG
13  HEADERS ?DUP IF TREETAG TFIXNULL THEN
14  ."    Finished tagging." ; DECIMAL
15
16 ' TREETAG CFA ' TREETAGCFA !


SCR #141
 1 ( HITKEY DDRIVE LOADFCB IMAGESAVE) HEX
 2
 3 : HITKEY  ( --- )  ."  and hit any key. --- " KEY DROP ;
 4
 5 : DDRIVE  ( --- )
 6  CR CR ." Insert DOS disk in default drive," HITKEY ;
 7
 8 : LOADFCB  ( a n fcba --- )
 9  DUP 21 0 FILL 1+ SWAP CMOVE ;
10
11 : IMAGESAVE  ( --- ) DDRIVE
12  F$ IMG FCB LOADFCB
13  IMG FILECREATE DROP
14  HERE FORTHORG DO I 1- SETDMA IMG FWRITE DROP 80 /LOOP
15  IMG FCLOSE DROP CR CR ." Step 1 complete." DDRIVE BYE ;
16 DECIMAL


SCR #142
 1 ( FRAMEIN RAZE) HEX
 2
 3 : FRAMEIN  ( --- )
 4  SP@ 2- 2- ' ROOF !
 5  VERBS# @ 2+ 0 DO 0 LOOP
 6  SP@ ' NLIST !
 7  HERE ROOF 2+ ! FFFF NLIST !
 8  TVERBS# @ 3 + 2 * 0 DO 0 LOOP
 9  SP@ ' TLIST !
10  FORTHORG DUP TLIST !
11  TARGORG - TLIST 2+ !
12  HERE NLIST 8 - !
13  FFFF NLIST 4 - ! ;
14
15 : RAZE  ( --- )
16  VERBS# @ 2+ TVERBS# @ 3 + 2 * + 0 DO DROP LOOP ; DECIMAL


SCR #143
 1 ( COUNTVERBS NFALISTFILL) HEX
 2
 3 : COUNTVERBS   ( nfa --- ) [ SMUDGE ]
 4  BEGIN DUP DUP 0= SWAP @ A081 = OR NOT
 5  WHILE DUP PFA DUP CFA @ DOVOC = SWAP 4 + @ 0= NOT AND
 6     IF DUP PFA 2+ @ COUNTVERBS THEN
 7     DUP C@ 20 AND IF 1 TVERBS# +! THEN PFA 4 - @ 1 VERBS# +!
 8  REPEAT DROP [ SMUDGE ] ;
 9
10 : NFALISTFILL   ( nfa --- ) [ SMUDGE ]
11  BEGIN DUP DUP 0= SWAP @ A081 = OR NOT
12  WHILE DUP PFA DUP CFA @ DOVOC = SWAP 4 + @ 0= NOT AND
13     IF DUP PFA 2+ @ NFALISTFILL THEN
14     DUP PFA 4 - @ SWAP DOWN @ ! -2 DOWN +!
15  REPEAT DROP [ SMUDGE ] ;
16 DECIMAL


SCR #144
 1 ( WINDOWTOP FRAMEBOTTOM) HEX
 2 : WINDOWTOP  ( a1 --- a2 )
 3  BEGIN DUP DUP @ SWAP 2- @ U< NOT
 4  WHILE 2-
 5  REPEAT 2- ;
 6
 7 : FRAMEBOTTOM  ( a1 n --- a2 ) SWAP
 8  BEGIN DUP @ 3 PICK U< NOT
 9  WHILE 2-
10  REPEAT SWAP DROP ;
11
12 : WINDOWBOTTOM  ( a1 a2 --- a3 ) @ OVER
13  BEGIN DUP @ DUP 4 PICK U< NOT SWAP FFFF = NOT AND
14  WHILE 2-
15  REPEAT SWAP DROP SWAP WINDOWTOP DDUP U<
16  IF SWAP THEN DROP ; DECIMAL


SCR #145
 1 ( OPENMOVE WINDOWSORT) HEX
 2
 3 : OPENMOVE  ( a1 a2 a3 --- )
 4  DUP DUP 2+ SWAP 5 PICK SWAP - DUP )R PREV @ SWAP BMOVE
 5  3 PICK 2+ SWAP 2+ 3 PICK 5 ROLL - BMOVE
```

```
 6  R@ - 2+ PREV @ SWAP R) BMOVE ;
 7
 8 : WINDOWSORT  ( a ---  )
 9  BEGIN DUP WINDOWTOP DUP @ FFFF = NOT
10  WHILE OVER OVER @ FRAMEBOTTOM
11     DDUP WINDOWBOTTOM OPENMOVE
12  REPEAT DROP DROP FORTHORG NLIST ! ;
13 DECIMAL
14
15
16
```

```
SCR #146
 1 ( FINDTAG OFFSETS) HEX
 2
 3 : FINDTAG  ( a1 --- a2 )
 4  BEGIN 2+ DUP @ C@ 20 AND UNTIL ;
 5
 6 : OFFSETS  ( --- ) CR CR ." Calculating offsets."
 7  NLIST 2+ @ TLIST NLIST TVERBS# @ 0
 8  DO FINDTAG DUP )R @ HEADERS NOT IF PFA CFA THEN
 9     ROT - SWAP 4 + SWAP OVER 2- @ + OVER
10     2+ ! R@ @ HEADERS NOT IF PFA CFA THEN OVER !
11     R@ 2+ @ SWAP R)
12  LOOP
13  DROP DUP 2+ @ SWAP 6 + ! DROP
14  ."   Offsets calculated." ;
15 DECIMAL
16
```

```
SCR #147
 1 ( VERBSLIST FIXSMUDGE) HEX
 2
 3 : VERBLISTS  ( --- ) 0 VERBS# ! 0 TVERBS# !
 4  CR CR ." Creating verb lists."
 5  [COMPILE] FORTH CONTEXT @ @ COUNTVERBS FRAMEIN
 6  ROOF DOWN ! CONTEXT @ @ NFALISTFILL
 7  I BUFFER DROP ."   Sorting." ROOF WINDOWSORT
 8  DECIMAL CR CR VERBS# @ U. ." (DEC) Verbs in host Forth."
 9  CR TVERBS# @ U. ." (DEC) Verbs in target code."
10  OFFSETS ;
11
12 : FIXSMUDGE  ( --- )
13  TLIST 4 + TVERBS# @ 0
14  DO DUP I 4 * + @ HEADERS NOT IF 2+ NFA THEN
15     DUP C@ DF AND SWAP C!
16  LOOP DROP ; DECIMAL
```

```
SCR #148
 1 ( NEWLINK TARGMAP DUMP/=) HEX
 2
 3 : NEWLINK  ( nfa1 --- nfa2 )
 4  TLIST 2- 2-
 5  BEGIN 2+ 2+ DDUP @ = UNTIL
 6  2- 2- @ SWAP DROP 0 FIXLINK? ! ;
 7
 8 : TARGMAP  ( n nfa --- )
 9  SWAP TARGORG + DUP 4 HEX U.R SPACE MAPFILE
10  IF 4 MAP U.RFILE MAP SPACEFILE DUP MAP ID.FILE MAP CRFILE
11  ELSE DROP THEN OUT @ SWAP ID. OUT @ - A + DECIMAL
12  BEGIN 1- DUP 0< NOT WHILE SPACE REPEAT DROP
13  50 OUT @ F + < IF CR 0 OUT ! THEN ;
14
15 : DUMP/=  ( a --- )  CR 20 - 50 DUMP TARG FILE! DROP
16  TARG FCLOSE DROP IMG FCLOSE DROP FIXSMUDGE ABORT ; DECIMAL
```

```
SCR #149
 1 ( PATCHADD VAR) HEX
 2
 3 : PATCHADD  ( a1 --- a2 f ) DUP
 4  CASE TCOLD1    OF  TCOLDCFA 1 ENDOF
 5     INIT-USER   OF  TSPO     0 ENDOF
 6     U8          OF  TRO      0 ENDOF
 7     LTIB        OF  TSPO     0 ENDOF
 8     UP          OF  TRO      0 ENDOF
 9     RPP         OF  TRO      0 ENDOF
10     DUP         OF  DUP @    1 ENDOF
11  ENDCASE ROT DROP ;
12
13 : VAR  ( a --- )  DOVAR? @ NOT OVER RPP = NOT AND
14  IF CR ." Images not matched at " DUP HEX 4 U.R DECIMAL DUMP/=
15  THEN PATCHADD DROP DUP FF AND TARG FC! FLIP FF AND TARG FC! ;
16 DECIMAL
```

```
SCR #150
 1 ( NEXTNFA REPAIRADD) HEX
 2
 3 : NEXTNFA  ( nfa1 --- nfa2 )
 4  NLIST 2- BEGIN 2+ DDUP @ U< UNTIL @  SWAP DROP ;
 5
 6 : REPAIRADD  ( a1 --- a2 )
 7  DUP @ DOVAR = IF 2 DOVAR? ! THEN
 8  DUP PATCHMAX < IF PATCHADD ELSE @ 1 THEN
 9  IF FIXLINK? @ ?DUP IF SWAP DROP NEWLINK THEN
10     TLIST
11     BEGIN DDUP @ U< NOT
12     WHILE 2+ 2+
13     REPEAT 2- @ -
14  THEN ;
15 DECIMAL
16
```

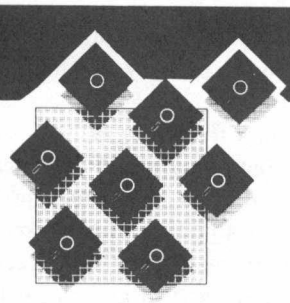
```
SCR #151
 1 ( VERB)TARG) HEX
 2
 3 : VERB)TARG  ( nfa --- )
 4  DUP NEXTNFA SWAP
 5  DO IMG FC@ DUP I C@ =
 6     IF I FIXLINK? @ =
 7        IF DF AND !FIX @ IF 0 FIXLINK? ! 0 !FIX ! THEN
 8        THEN TARG FC! 1
 9     ELSE IMG FC@ 100 * + 201 - I @ =
10        IF I REPAIRADD DUP FF AND TARG FC!
11           FLIP FF AND TARG FC! 2
12        ELSE I VAR 2
13        THEN
14     THEN DOVAR? @ IF -1 DOVAR? +! THEN
15  /LOOP ;
16 DECIMAL
```

```
SCR #152
 1 ( DISCARD RELOCATE) HEX
 2
 3 : DISCARD  ( n --- )

 4  ?DUP IF IMG SWAP 0 DO DUP FC@ DROP LOOP DROP THEN ;
 5
 6 : RELOCATE  ( --- n )  0 DOVAR? !
 7  0 OUT ! 0 !FIX ! 0 FIXLINK? !
 8  CR CR TARGORG 4 HEX U.R ."  HEX (target origin)" CR CR
 9  MAPFILE IF TARGORG 4 MAP U.RFILE MAP CRFILE THEN
10  FORTHORG 0 NLIST 8 - TLIST
11  DO I TLIST = NOT IF DUP I @ TARGMAP I @ FIXLINK? ! THEN
12     I @ [ ' ! NFA ] LITERAL = IF 1 !FIX ! THEN
13     SWAP I @ SWAP - DISCARD
14     I @ VERB)TARG I @ DUP NEXTNFA DUP ROT - ROT + 4
15  /LOOP SWAP DROP DECIMAL ;
16 DECIMAL
```

Forth Compiler Listing (Listing continued, text begins on page 52)

```
SCR #153
 1 ( VERB)TARGNH) HEX
 2
 3 : VERB)TARGNH  ( cfa --- )
 4  DUP NEXTNFA SWAP
 5  DO IMG FC@ DUP 1 C@ =
 6     IF TARG FC! 1
 7     ELSE IMG FC@ 100 * + 201 - I @ =
 8        IF I REPAIRADD DUP FF AND TARG FC!
 9           FLIP FF AND TARG FC! 2
10        ELSE I VAR 2
11        THEN
12     THEN DOVAR? @ IF -1 DOVAR? +! THEN
13  /LOOP ;
14 DECIMAL
15
16

SCR #154
 1 ( RELOCATENH) HEX
 2
 3 : RELOCATENH  ( --- n )
 4  0 OUT ! 0 FIXLINK? ! 0 DOVAR? !
 5  CR CR TARGORG 4 HEX U.R ."  HEX (target origin)" CR CR
 6  MAPFILE IF TARGORG 4 MAP U.RFILE MAP CRFILE THEN
 7  FORTHORG 0 NLIST 8 - TLIST
 8  DO I TLIST = NOT IF DUP I @ 2+ NFA TARGMAP THEN
 9     SWAP I @ SWAP - DISCARD
10     I @ VERB)TARGNH
11     I @ DUP NEXTNFA
12     DUP ROT - ROT + 4
13  /LOOP SWAP DROP DECIMAL ;
14 DECIMAL
15
16

SCR #155
 1 ( SETUPDOS CLEANUPDOS) HEX
 2
 3 : SETUPDOS  ( --- )
 4  DDRIVE F$ IMG FCB LOADFCB
 5  BEGIN IMG FSEARCH
 6  WHILE CR ." TARGET.TMP not found on default drive" DDRIVE
 7  REPEAT IMG FOPEN DROP
 8  CR CR ." Target code filename"
 9  TARG FILENAME? TARG FILECREATE DROP MAPFILE
10  IF TARG FCB MAP FCB 21 CMOVE
11     M$ MAP FCB 9 + SWAP CMOVE MAP FILECREATE DROP
12  THEN ;
13
14 : CLEANUPDOS  ( --- ) IMG FCLOSE DROP
15  IMG FDELETE DROP TARG FILE! DROP TARG FCLOSE DROP MAPFILE
16  IF 1A MAP FC! MAP FILE! DROP MAP FCLOSE DROP THEN ; DECIMAL


SCR #156
 1 ( TARGETMEMORY)
 2
 3 : TARGETMEMORY  ( n --- ) CR CR
 4  ." (HEX) Target Memory Map" HEX
 5  CR TVRAM TVRAMOFF @ + 1- 4 U.R ."  Last needed byte of RAM."
 6  CR TVRAM 4 U.R ."  Target variable RAM start {TVRAM}."
 7  CR TEM 4 U.R ."  Target-end-of-memory {TEM}."
 8  CR TBUF1 4 U.R ."  Start of block buffer area {TBUF1}."
 9  CR TRO 4 U.R ."  Initial return stack pointer {TRO}."
10  CR TSPO 4 U.R ."  Initial stack pointer {TSPO}."
11  CR TARGORG + 1- 4 U.R ."  Target code end address {last used}."
12  CR TARGORG 4 U.R ."  Target origin {TARGORG}."
13  DECIMAL ;
14
15
16


SCR #157
 1 ( TARGET) HEX
 2
 3 : TARGET  ( pfa or 0 --- )
 4  DUP IF CFA THEN ' HEADERS !
 5  TCOLDCFA 2+ NFA FENCE ! FREEZE
 6  APPTAG
 7  FORTHORG 301 = IF IMAGESAVE THEN
 8  VERBLISTS
 9  SETUPDOS
10  HEADERS IF RELOCATE ELSE RELOCATENH THEN
11  CR CR HEX DUP U. DECIMAL ."  (HEX) Bytes in target. "
12  TARGETMEMORY CLEANUPDOS FIXSMUDGE RAZE
13  CR CR ." Target compilation complete." ;
14 DECIMAL
15
16
```

End Listing



Neon, Version 1.0

Company: Kriya Systems, Inc.
505 N. Lake Shore Drive,
Suite 5510, Chicago, IL
60611, (312) 822-0624
Computer: Apple Macintosh (128K)
Price: $155.00
Circle Reader Service No. 101
Reviewed by Bruce Horn

This review is reprinted with the permission of the author, Bruce Horn, and The Club Mac News, *in which it originally appeared (Vol. 1, No. 12, May 1985).*

I have not written many programs in Forth. I've always considered Forth to be a sort of high-level assembly language, fun for hacking around in, but not suited to development of large programs. I have written Smalltalk programs, though, and when I heard that Charles Duff and the folks at Kriya had developed Neon, an object-oriented programming environment that was said to be a cross between Smalltalk and Forth, I was intrigued; is this the long-awaited favorite programming environment for the Macintosh?

History of Object-Oriented Programming

Object-oriented programming began in the early 1960's with the development of Simula by the Norwegian Computation Center in Oslo, Norway. Simula was the first language to implement the Class construct. In the early 1970's, the Learning Research Group at Xerox Palo Alto Research Center began implementation of the Smalltalk programming environment. Smalltalk was the first system to be designed completely around the Class/Object concept. Many other languages have since provided support for classes, objects, and subclassing, including CLU, Ada, C++ (an extended version of C), and several versions of LISP, though objects and classes are not as well-integrated into these languages as they are in Smalltalk. The ease of experimentation with objects was one of the key reasons that much of the interesting user interface designs borrowed for Macintosh and Lisa were created in the Smalltalk system. (See *Smalltalk-80: The Language and its Implementation,* by Adele Goldberg and Dave Robson, for more details.) The latest version of Smalltalk has been licensed to several universities and computer companies; Tektronix markets a 68000-based Smalltalk machine for around $14,000, and a version of Smalltalk-80 will soon be available for the Macintosh XL.

Classes and Objects—a Quick Overview

In Neon, classes define objects, a convenient packaging of 1) the data that defines the object and 2) the object's actions. For example, Class **Pen** defines what a pen object looks like (as far as its internal state) and acts like (the messages it responds to). The internal state, or instance variables of a **Pen** object, might include location, direction, and pen size; the messages that pens respond to would include **move:**, **turn:**, **goto:**, and **home:**. (The Neon tutorial has an excellent discussion of the implementation of a **Pen** class, with examples and line-by-line explanation of the source code). Class definitions are bracketed by **:Class** and **;Class** words, and include a list of instance variables (which may themselves be objects) and methods (the procedures run when a message is sent to the object).

Defining a class is like designing a little computer, with its own memory (the instance variables) and instruction set (the messages it can receive). In using a computer, usually you don't concern yourself with how the computer executes the instructions you give it. If you did, and you counted on the various nuances of the implementation of each instruction, programming the computer would be much more complicated. Classes protect you from the need to know about the implementation of the object (computer) by allowing you to talk to the object through a set of messages.

Responses to messages include returning a result on the stack, changing the internal state of the object, sending messages to other objects, or just producing a side-effect (such as drawing a line on the screen). **10 move: Bic** sends the message **move:** to the pen object named **Bic**, changing its location instance variable and possibly drawing a line on the screen; **45 turn: Bic** turns the pen 45 degrees to the right. One advantage of having a pen *object* do your line drawing on the screen instead of just calling Quickdraw directly is that you can have many separate pens, each with its own state, and you can send messages to them independently. To paraphrase Dan Ingalls, one of the developers of Smalltalk, programming with objects is like working with trained animals instead of pushing data around with a broom.

Classes may be defined that are subclasses of another class. This causes objects of that class to inherit both the instance variables and actions of the superclass. Subclassing provides the capability of saying, "I want something just like that, except a little different." For example, you might want to have a **Pen** that recorded its movements. Since the current class **Pen** handles everything except the recording, subclassing **Pen**

and creating a **RecordingPen** that provides the capability would make the most sense.

Since class **RecordingPen** is a subclass of **Pen**, **RecordingPen** objects will have both **Pen** and **RecordingPen** instance variables, and will respond to the union of all **Pen** and **RecordingPen** messages. In general, if the message is not defined in the class that the object belongs to, Neon will search up the superclass chain to find a class that can respond to the message, and runs the appropriate method. You may also add new messages and instance variables to distinguish your class from the superclass, and you can override superclass messages by providing them in your subclass.

To create the new **RecordingPen** class all you would need to do is add **startRecording:**, **stopRecording:**, and **playBack:** messages, and a new instance variable **penRecord**, which keeps the list of messages sent to the object since recording started. By overriding each message that the object wants to record, it can remember the message in **penRecord** and send the appropriate message to its superclass, **Pen**, to do the actual drawing.

Building a system from scratch based on classes and objects provides an extremely flexible, compact and modifiable system. Since the code that defines the system itself is accessible, extensions to the system may be made quickly and easily at even the lowest level.

(For an excellent discussion of classes, messages and object oriented programming, see Brad Cox's article "Message/Object Programming: An Evolutionary Change in Programming Technology" in the January 1984 issue of *IEEE Software*.)

Built-in Classes

Since Neon, like Smalltalk, was written in itself, using its own object and class concepts, the groundwork for building applications is already there and fully accessible to the programmer. The basic classes provided in Neon make it easy to support the Macintosh User Interface standard with minimal effort. In Pascal, a Mac program typically has several large case statements at the very highest level of the main program to classify events received and to dispatch them to handlers. Neon helps to take care of this by encapsulating most of the standard event handling in class **Event**, menu selection and dispatching in class **Menu**, and standard window actions in class (you guessed it) **Window**. Through these classes, Neon shields the programmer from the standard necessities and deals with most of the housekeeping that is required in a Macintosh application behind the scenes.

There are many other classes written to simplify interaction with the User Interface Toolbox. Class **File** provides access to the Toolbox File I/O and Save/Open dialogs, and class **String** defines messages for printing, insertion, and pattern matching of text strings (a nice interface to Munger, the Toolbox string handler). In addition, there is a set of nine classes to support Quickdraw, including **Rect**, **Oval**, **Picture**, and **GrafPort**. All Toolbox routines are directly accessible.

Using Neon

Neon is started by double-clicking on **neon.com**. **Neon.com** is a prebuilt environment in which a basic set of classes, messages, and objects have been installed. As you work in this environment, you will be creating classes, objects and messages, as well as new Neon words. Once you have modified the default environment you may save it on the disk. In this way you can build your own custom Neon system.

When Neon starts up, it comes up with a single teletype window, where you can type Neon commands. Output from some of the utilities appears there also. The Neon window does not update when windows are moved around above it, which is a minor nuisance. There is no text editing allowed in the Neon window other than backspacing.

It is fairly easy to create new windows and menus while in Neon and to attach Neon words to each menu item. For example, sending the message **Example:** to a new window creates a Neon window that will interpret any text typed to it. There are also example messages in several other classes, including **Control**.

The standard **Neon.com** system, as it comes from Kriya, has five menus:

1) The **Apple** menu has the standard list of desk accessories, an "About Neon . . ." item, and an extra accessory called "Editor." This accessory provides the text-editing facilities necessary for editing class and object definitions within Neon.

2) The **File** menu consists of five items:
Load... loads a text source file into the current environment, defining new classes, objects, and messages. You would load **grdemo.load** to define the special classes for the graphics demonstration program.
Save saves your current environment onto the disk. As you modify classes and customize the system to meet your needs, you can save the complete system and return to it by simply opening the appropriate Neon document. This is often called "taking a snapshot."
Save As... allows you to save your current environment under a new name.
Print... prints a specified text file to the printer. The print command produces a formatted ("prettyprinted") listing of the source code in the file.
Quit closes all files and returns to the Finder. All changes made to the Neon environment are lost (unless they had been previously Saved.)

3) The **Edit** menu provides the standard **Undo**, **Cut**, **Paste**, **Copy**, and **Clear** items, as well as a special **Editor** item to start up the editor desk accessory.

4) The **Utilities** menu has six items:
List Words lists all words currently defined in the dictionary.
List Objects lists all objects of a given type that currently exist in the environment.
List Classes lists all defined classes in the current environment, indenting to show subclass relationships.
Examine Memory is a feature that allows you to examine and search the Macintosh memory.

Grep... searches all text files on the disk for a given string. The results of the search (file name, line number, and text line) are printed in the Neon window.

Install lets you change the sizes of the stack, dictionary, and heap. The stack and dictionary are adjustable, while the heap gets what is left.

5) Finally, the **Neon** menu provides five items:

Echo to Printer echoes all text that appears in the Neon window to the printer, including text that is typed by the programmer. This is especially useful in that it allows the programmer to keep a record of the input and output of a program being debugged.

Echo During Load displays the text from a source file being loaded in the Neon window as it is being compiled. This is useful in helping the programmer debug a new module or class definition.

Show Free Space displays the amount of memory available in the dictionary and the heap in the Neon window.

The last two entries in the Neon menu deal with modules. Modules are separately compilable, relocatable code segments that are loaded and purged from memory dynamically. This allows large applications to run without having to be completely resident in memory. These modules act similarly to the Macintosh code segments in that they are dynamically loaded into memory and purged when not in use. Many of the menu items in Neon are defined by modules, including **Grep** and **Examine**.

Show Module Status lists all purgeable modules currently defined in the dictionary and shows whether they are loaded by printing their memory locations (or 0 if not loaded).

Purge Modules purges all modules from the heap, freeing up space for a memory-intensive operation.

Because the Neon window provides no editing capability, the system comes with an editor to edit class definitions within Neon. This editor is an enhanced MockWrite desk accessory, from CE Software. I found it easy and convenient to use, though I would prefer a built-in editor in which I could select, edit, and execute Neon statements directly. Once a class definition is created, it may be loaded into the system by either typing // followed by the filename or by selecting the **Load** item in the file menu. Either action begins compilation of the file. After the file has been completely compiled and defined in the system, you may create new objects, send messages to previously-created objects, and so on.

After a few hours with Neon, I felt quite at home creating objects, sending messages, and building graphical tools. The fact that Neon is an interactive programming environment is a great advantage in building applications for the Macintosh because the turn-around time is much shorter and experimentation is easier. Neon also provides a facility for "sealing off" an application, creating a double-clicking program without access to the Neon environment.

Forth Improvements

Besides support for objects and classes, Neon has other improvements to standard Forth. The main objection to Forth is usually the need for strange and unintelligible stack manipulation in function definitions; Neon avoids this through local variables and named parameters. Consider the example in the Neon tutorial for calculating **a2 + b2:**

```
:formula1 { a b -- solution }
a a * → a
b b *
a + . CR ;
```

The input parameters are named between the braces, and are pulled off the stack and stored in the local variables **a** and **b**. The arrow (→) stores the value on the stack into the named parameter. Local variables may also be defined within the braces by preceding the variable list with a backslash. Again, from the Neon tutorial, consider the formula **(a + b-3c)/(b + 3c):**

```
:formula2{ a b c \ num den -- result }
a b + 3 c * → num
3 c * b + → den
num den / ;
```

The numerator and denominator are calculated separately, stored in local variables, and then the formula is calculated. With local variables and named parameters, Neon programs are downright readable.

Summary

Neon is nicely designed. It is clear that the developers had an excellent understanding of what object-oriented programming is supposed to be. With the basic classes, examples, and tutorial, you could be writing interesting programs that support the Macintosh User Interface Standard in a matter of weeks. This compares very favorably with the months that most people take to build the support routines that a real application requires. Kriya is also planning to support a clearinghouse service for useful, public domain classes written in Neon for use by software developers.

Neon is also a good programming language for occasional use, though it has so many features and is so flexible that it could take a little while to learn what is provided in the system. The documentation is thorough and well-written, and the tutorial is interesting and useful.

As a programming environment, Neon is very well suited to the Macintosh. The built-in classes are well designed and useful, and provide a much improved interface to the Macintosh Toolbox routines. It is flexible, powerful, fast and a lot of fun. It is also the only implementation of an object-oriented system for the Macintosh so far. Because of the support for classes and objects, organization of large programs would be quite manageable, while with standard Forth working on a large system with thousands of words might become confusing. The only disadvantage that Neon has, in my opinion, is the backwards Forth syntax. If somebody comes up with a new infix parser-scanner to allow Smalltalk-like expressions, Neon could be a real winner in the programming language sweepstakes.

MacFORTH Level 1, Version 1.2; Level 2, Version 2.1; Level 3, Version 3.0

Company: Creative Solutions, Inc.
4701 Randolph Rd, Ste. 12
Rockville, MD 20852
(301) 984-0262
Computer: Apple Macintosh (128K)
Price: Level 1—$149, Level 2—$249 (includes Level 1), Level 3—$499 (includes Levels 1 and 2)
(Circle Reader Service No. 103)
Reviewed by Alan Clute

The Macintosh is a giant toolkit. An ideal language to take advantage of this toolkit is Forth. MacFORTH, from Creative Solutions, Inc., is an excellent Forth for the Macintosh. MacFORTH delivers the richness and power of the Macintosh without getting overly complex. The package, while being simple to use, still leaves the programmer enough breathing room to develop programs that are powerful and make good use of the Mac's capabilities.

MacFORTH is a production language. Current products written in MacFORTH include PeachTree's *Back to Basics Accounting*, Brainworks' *ChipWits*, and Ideaform's *MacLabeler*. An interesting new product is CSI's *StudioMac*, a music program that includes tutorials, source code, MIDI interface capability, and complete plans to build an inexpensive MIDI interface for the Macintosh.

Contents of Package

Table 1 (see page 101) shows the main contents of each level of MacFORTH. The three levels are nested, and upgrades are available for the difference in list price. Because Level 2 includes more and is in some ways easier to work with than Level 1, it is probably the best package to get first, especially at mail-order prices. For the record, this review concerns itself with Versions 1.2, 2.1, kernel 2.3, and, briefly, with Level 3 version 3.0.

MacFORTH has been around a fairly long time, as Macintosh software goes. Level 1 was first released in April of 1984, with the first version of Level 2 coming out in July of that year. MacFORTH was the first serious native language for the Macintosh. It makes good use of a 128K Macintosh and can be used with only one disk drive (if you are so inclined).

MacFORTH is not copy protected, so it can easily be backed up or installed on a hard disk. The accompanying license agreement says in no uncertain terms that you should not give or sell copies to others, but CSI is to be congratulated for not otherwise interfering with your use of the program.

Forth Implementation

MacFORTH is derived from the Forth-79 standard, optimized for the Macintosh environment. It is a 32-bit Forth, which is more suitable for the large address space of the Mac than 16-bit implementations. Considered just as Forth, MacFORTH is a good one. Converting programs from another standard version of Forth should be rather straightforward.

The internal implementation is direct threaded with a token *table* to keep all word references to 16 bits. Thus, program code requires about the same amount of dictionary space as in a 16-bit version of Forth. Variables occupy 32 bits. Fetch and store operators come in several flavors to support 8, 16, or 32 bit values.

MacFORTH uses traditional Forth blocks for source code, but these blocks are within block files. MacFORTH thus lives very comfortably within the Macintosh environment.

Programs can do disk I/O using

Level	Feature	Comments
1	• Going FORTH tutorial	Disk based, interactive.
	• Block-oriented editor	Similar to MacWrite.
	• Macintosh interface	Menus, windows, mouse, sound, event handler.
	• File system	Text, record, virtual, and block files. MacWrite text files.
	• Printer/Serial interface	Window or screen output, full serial I/O.
	• Programming aids	Trace, debug, error traps, direct calls to toolbox routines.
	• Source code	Forth extensions, editor, many examples, terminal program, download conversion, standard file-open dialogue.
2	• Level 1 included	
	• System tools	Heap management, 68000 assembler, snapshot (vocabulary save).
	• Advanced graphics	Polygons, regions, pictures, rectangle arithmetic and scrolling.
	• Text editing	TextRecords and TextFields.
	• Controls	Buttons, boxes, and scroll bars.
	• Clipboard I/O	Read and write from/to clipboard
	• Floating Point	IEEE-754 standard
	• Source code	Graphics demo, note card data management demo (fully annotated in manual), many examples, source for above extensions.
3	• Level 2 included	
	• Application workshop	Turnkey production tools and tutorial. Supports 512K Mac, overlays on 128K Mac.
	• Resource workshop	Macintosh resource customization and tutorial.
	• Kernel workshop	Forth kernel customization and tutorial.

Table 1:
Contents of MacFORTH Levels

block files, or can choose to use data files with text, fixed-length record, or byte-addressable virtual formats. The entire storage-management vocabulary includes about 90 words.

Documentation

MacFORTH is extremely well documented. Most of the documentation covers the Macintosh interface, but then, most of the Forth is quite standard. The level 1 manual is about 160 pages, plus glossary and appendices. Level 2 runs over 200 pages, plus glossary, and Level 3 adds 200 more pages, which show you step by step how to turn a running program into a production disk.

The manuals make clear presentations. The glossaries are well organized, well indexed, and well written. The only thing obviously missing is a reference in each glossary entry to the page where the word is discussed in context.

MacFORTH pretty much assumes you know Forth, although it comes with a Forth tutorial and a rich set of examples that would prove useful even to a beginner. As is often advised, the beginner really should arm him/herself with some instructional book like *Starting FORTH* or *Thinking FORTH* by Leo Brodie (Prentice-Hall, 1981 and 1984) instead of relying on a language manual.' In addition, a new book should be out by the time you read this, *MacFORTH: Programming for the Rest of Us* by Dave Colburn, one of the authors of MacFORTH (also available from CSI).

The Going Forth tutorial, a split-screen affair with paged text on one side and the live MacFORTH system on the other, provides a hands-on introduction to MacFORTH. Source is on disk for those who want to expand the tutorial for others.

You get plenty of source code with these packages, both "real-world" stuff, like the block editor, and "tutorial" material like the Note Card example. This last is part of Level 2 and includes 23 screens of source and 20 pages of detailed manual text. In all, Level 1 includes about 120 screens of source, and level 2 about 140 screens.

Forth and the Macintosh

Forth has been described as a language for the development of problem-oriented languages. MacFORTH is a Macintosh-oriented language. The "problems" of the Macintosh—how to use windows, pull-down menus, graphics, and other features—have been smoothly addressed by MacFORTH.

The MacFORTH vocabulary is huge, over 900 words in Level 1 with another 250 or so added in Level 2. Most of these are the "application words" that give you direct control over the Macintosh. The Macintosh and its fabled toolbox are made directly available, letting you do things "the Macintosh way" with relative ease. Don't be intimidated by the large vocabulary: flexibility comes from lots of little words, rather than a few large ones, and the Macintosh has many of these small units. The window-handling package alone has over 70 words associated with it.

Examples

Straightforward use of windows and pull-down menus, for example, turns out to be simplicity itself. The code in Figure 1 (at left) creates a window with a close box and a size box. Figure 2 (at left) shows an example menu definition, with 3 items. The

```
        ( create window )
15 CONSTANT testWindow
NEW.WINDOW testWindow

        ( define title )
    " Example Window"                testWindow W.TITLE
        ( define rectangle )
    50 50 100 300                    testWindow W.BOUNDS
        ( add boxes )
    CLOST.BOX SIZE.BOX +             testWindow W.ATTRIBUTES
        ( add window to the screen )
    testWindow ADD.WINDOW
```

Figure 1
Code to Create Window

```
        ( create menu, define title )
11 CONSTANT testMenu
0 " Example" testMenu NEW.MENU
        ( define item text )
" First;Second;Third(;" testMenu APPEND.ITEMS
        ( add menu to the screen )
DRAW.MENU.BAR
```

Figure 2
Menu Definition

```
        ( define menu handler )
testMenu MENU.SELECTION:
        ( CASE statement for items )
    CASE 1 OF FirstProgram ENDOF
         2 OF SecondProgram ENDOF
         3 OF ThirdProgram ENDOF
    ENDCASE
        ( turn off title hilite )
    0 HILITE.MENU ;
```

Figure 3
Selection of Menu Item

first two are "active" and the third is "inactive" (grayed out and incapable of being selected). The trailing **(** in **Third(** causes that item to be inactive. Type style (bold, underline, etc.) and command key equivalents are easily assigned using other character suffixes. Selection of a menu item is easily linked to the proper actions using a defining word (Figure 3, at left). Fancier uses of scrollbars, control buttons, and the like are implemented with the clarity and simplicity so essential to good Forth style.

The Level 2 manual includes a reference section giving the Pascal name, MacFORTH name, and stack usage for the over 150 toolbox traps directly callable by the user. For advanced applications, MacFORTH provides simple low-level words to access any toolbox routine.

Support

CSI maintains a technical hotline for owners of Level 1 and Level 2, and publishes a quarterly newsletter. The hotline is open during business hours and is a normal toll call.

An important part of the MacFORTH "safety net" is the Special Interest Group (SIG) on CompuServe. This is currently open to registered owners of MacFORTH only, and according to CSI about 10% of MacFORTH owners participate. As with most such SIGs, this one offers data libraries with public domain software written in MacFORTH. One data library is devoted to notices and patches for known bugs. A terminal program (with source code) is included on the Level 1 disk.

User Interface

You would expect a first-class Macintosh toolkit implementation to make very good use of the Macintosh, and, for the most part, MacFORTH does just that. Windows and menus for the MacFORTH environment are standard Macintosh. The text editor included in Level 1 is a subset of MacWrite, with source on disk if you want to study it or add features.

MacFORTH is so Mac oriented that rough spots in the user interface come as a bit of a shock. For example, the Forth text interpreter is plain vanilla Forth and doesn't seem to know about mousing around. Keystrokes just sort of lie there on the screen, and to edit a mistake you have to back up and retype. Gasp! Fortunately the CompuServe SIG comes to the rescue here with a handy little utility that stores the last ten commands, in editable form, in a pull-down menu. Nothing like a user's group to rescue one from primitive conditions.

Another minor annoyance is that text in the MacFORTH window is not restored after being overlaid by the editor window. When you exit the editor you are left with a blank rectangle chopped out of your dialogue with MacFORTH. This conserves memory, but is tacky. Those of you with 512K Macs will want to add screen save/restore to the editor with a few lines of code.

Developer Package, Level 3

I was only able to look at the manual for Level 3. This package is for developers who want to produce turnkey products for distribution. CSI used to charge a per-copy royalty, but the new Level 3 package replaces that policy and includes a royalty-free license agreement.

The Level 3 manual provides some excellent tutorial material and seems to be as concerned with helping you turn out a good stand-alone product as the Level 1 is with helping you make good use of the Macintosh.

The support situation is a bit different for Level 3. CSI is understandably wary of spending a lot of time on the phone with developers, so Level 3 owners are expected to use the CompuServe SIG instead of the CSI hotline when they need help. In practice, this seems to work out fine, to judge from the message traffic on the SIG (which includes heavy participation by CSI staff). If you are planning to use MacFORTH to develop a turnkey product or program for distribution, you might buy Level 2, use the hotline and the SIG while learning MacFORTH and during program development, and then upgrade to Level 3 when you are in the final stages.

Conclusion

It is clear that the authors of MacFORTH want you to get your hands on the Macintosh and use it well. The manuals, samples, and examples all support this "Power to the People" approach. Most of the manual is about how to use the Macintosh with the tools provided by MacFORTH. The result is that, for the most part, you don't have to spend hours or days poring over *Inside Macintosh* to get the Macintosh to do something, but can instead learn by example and hands-on experimentation. Because it is so easy to try things in Forth, using it on the Mac should lead to even greater reduction of program development time than would using Forth on a more limited machine.

DDJ

Last month (September 1985, #107 page 94) John Bass described how to put together your own hard-disk interface for the Macintosh for less than $500. The following listing, np_dvr.c, is the driver routine.

Mac Toolbox Listing

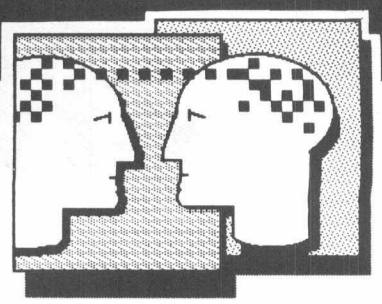
```
/*
 * np_dvr.c version 1.0, July 20, 1985
 * MacSCSI non-partitioned driver routine.
 * This version was based on the ramdisk.asm example found in
 * Aztec C 1.06 and was developed on the Mac under Aztec C.
 *
 * The changes are Copyright 1985 by John L. Bass, DMS Design
 * PO Box 1456, Cupertino, CA 95014
 * Right to use, copy, and modify this code is granted for
 * personal non-commercial use, provided that this copyright
 * disclosure remains on ALL copies. Any other use, reproduction,
 * or distribution requires the written consent of the author.
 *
 * Sources are available on diskette from Fastime, PO Box 12508
 * San Luis Obispo, Ca 93406 -- (805) 546-9141. Write for ordering
 * information on this and other Mac products.
 */

#asm
;:ts=8
;
DRVNUM  equ     5               ;change this if conflicts with others
;
Start
        dc.w    $4f00           ;locked, read, write, ctrl, status
        dc.w    0               ;no delay
        dc.w    0               ;no events
        dc.w    0               ;no menu
        dc.w    open-Start      ;open routine
        dc.w    rdwrt-Start     ;prime routine
        dc.w    control-Start   ;control routine
        dc.w    status-Start    ;status routine
        dc.w    tst-Start       ;close routine
        dc.b    8
        dc.b    ".MacSCSI"      ;name of driver
        ds      0               ;for alignment
;
MAXMAP          equ     640
;
drBlLen         equ     16
drNmAlBlks      equ     18
drAlBlkSiz      equ     20
drClpSiz        equ     24
drAlBlSt        equ     28
drFreeBks       equ     34
;
template
        dc.w    $d2d7           ;signature word
        dc.l    0               ;init time
        dc.l    0               ;backup time
        dc.w    0               ;attributes
        dc.w    0               ;# of files
        dc.w    4               ;first dir block
        dc.w    0               ;# of dir blocks
        dc.w    0               ;# of allocation blocks total
        dc.l    0               ;bytes/allocation block
        dc.l    0               ;bytes/allocation call
        dc.w    4               ;first data block
        dc.l    1               ;next free file #
        dc.w    0               ;# of unused allocation blocks
        dc.b    7               ;length of name
```

```
        dc.b    "MacSCSI"               ;name of drive
        ds      0                       ;for alignment
;
stabuf
        dc.w    0                       ;current track
        dc.b    0                       ;write protect in bit 7
        dc.b    1                       ;disk in place
        dc.b    1                       ;drive installed
        dc.b    0                       ;sidedness of drive
        dc.l    0                       ;next queue entry
        dc.w    0                       ;unused
        dc.w    DRVNUM                  ;drive number
drvref  dc.w    0                       ;driver ref num
        dc.w    0                       ;file system ID
        dc.b    0                       ;sidedness of disk
        dc.b    0                       ;needs flush flag
        dc.w    0                       ;error count
;
diskbase_
        dc.l    0                       ;base of disk memory
;
openflg
        dc.w    0                       ;once only open flag
;
open
        lea     openflg,a0              ;get once only flag
        tst.w   (a0)                    ;is it open already?
        bne     skip                    ;yes -- just exit
        st      (a0)                    ;set once only flag

        move.l  #14,d0                  ;size of drive queue entry
        dc.w    $a51e                   ;NewPtr - system heap
        clr.w   10(a0)                  ;local file system
        move.l  #DRVNUM,d0              ;drive number 5
        swap    d0
        move.w  24(a1),d0               ;driver reference number
        lea     drvref,a2               ;get address of driver reference number
        move.w  d0,(a2)                 ;set up status buffer
        dc.w    $a04e                   ;AddDriver

        move.l  (a1),-(sp)              ;push handle to resource
        dc.w    $a992                   ;DetachResource

        link    a6,#-50                 ;allocat the block on the stack
        move.l  sp,a0                   ;setup as arg to mountvol trap
        move.w  #DRVNUM,22(a0)          ;set our drive number in block
        dc.w    $a00f                   ;ask for it to be mounted
        unlk    a6                      ;clear block from stack

skip
        move.l  #0,d0                   ;set non-error return
        rts                             ;exit back to caller
;
rdwrt
        movem.l d0-d7/a0-a6,-(sp)       ;save regs
        move.l  a0,Pbp_
        move.l  a1,Dp_                          ;save arguments
        rts
restore_
        move.l  Dp_,a1                          ; pass DCEptr
        rts
#endasm

#include        <memory.h>
#include        <resource.h>
#include        <desk.h>
#include        <pb.h>
/*
 * varibles used by asm routines
```

```
 * Dp and Pbp are device driver arguments handled by save/restore
 * ptr and dvrarg needed by AddDriver call
 */

DCEPtr          Dp;
ParmBlkPtr      Pbp;

/*
 * Local Stuff
 */

int     DvrState;
        jsr     ScsiRdWr_               ;Call the C stuff
        movem.l (sp)+,d0-d7/a0-a6       ;restore regs
        bra     tst                     ;exit via IOdone

control
        cmp.w   #1,26(a0)       ;is it KillIO
        bne.s   tst             ;no, exit
        move.l  #0,d0
        rts                     ;just return
;
status
        cmp.w   #8,26(a0)       ;is it status request??
        bne.s   tst
        movem.l a0/a1,-(sp)     ;save regs
        lea     28(a0),a1       ;get dest address
        lea     stabuf,a0
        move.l  #22,d0          ;move 22 bytes
        dc.w    $a02e           ;BlockMove
        movem.l (sp)+,a0/a1     ;restore regs
;
tst
        move.l  #0,d0                   ;okay return
        movem.l d4-d7/a4-a6,-(sp)       ;save regs going into IOdone
        move.l  $8fc,a0                 ;get IOdone address
        jsr     (a0)                    ;call IOdone
        movem.l (sp)+,d4-d7/a4-a6       ;restore them after IOdone
        rts                             ;and jump to it

        public  _Uend_,_Dorg_,_Cend_

save_
        lea     Start+(_Uend_-_Dorg_)+(_Cend_-Start),a4 ; setup baseadr
/*
 * ScsiRdWr -- do a driver read/write function.
 */
ScsiRdWr() {
        register struct ioParam *ip;
        long    len,part;
        short   blkno;
        char    *addr;
        save();
        ip = & Pbp->u.iop;
        part = Dp->dCtlPosition & 0xFFFFFE00;
        len = (ip->ioReqCount + 511) & 0xFFFFFE00;
        addr = ip->ioBuffer;
        while(len >= 512) {
                blkno = part>>9;
                if((Pbp->ioTrap & 0xff) == 2) {
                        ScsiRead(blkno, addr);
                 } else {
                        ScsiWrite(blkno, addr);
                }
                len -= 512;
                addr += 512;
                part += 512;
        }
        ip->ioActCount = ip->ioReqCount;
        restore();                      /* exit to I/O Done */
}
```

End Listings

CP/M EXCHANGE



by Bob Blum

The CP/M Exchange RCP/M system is available for your use 24 hours a day, 7 days a week. Reach it by dialing (404) 449-6588.

My queue runneth over. It seems that I always have far more projects underway than I will ever have time to complete. My workmates characterize my problem as simply not being able to manage my time efficiently. I know better, I know the real reason that this phenomenon constantly haunts me.

My time is being slowly pilfered away by the software developers who continue to write programs for CP/M. It's an insidious plot. What reasonable person would continue to support an outdated operating system such as CP/M and the archaic 8-bit microprocessors it runs on? The schemers know of my weakness for trying every new program that's advertised, and they use it to their best advantage. Some are even so bold as to send me their addicting wares of slavery without even letting me call to volunteer myself for human sacrifice.

So, as you can see, I'm not at fault for falling victim to my weaknesses.

All fun aside, more and more new 8-bit CP/M software is being introduced each day. I applaud all of those who have the foresight not to abandon a strong 8-bit market. By some standards I may be misguided, but I rest assured that I am in the best of company.

In the Queue

I'm anxiously waiting for my review copy of the latest single board computer from Micro Mint. This little barn burner, the size of a mini-floppy drive, sports the HR64180, the newest 8-bit chip from Hitachi. I'm told that this device is at least as powerful as an 8-Mhz Z80 and offers a much higher level of integration, including I/O ports and other features that normally require external devices. The most exciting part is its capability to address directly 512K of memory. My plans call for a review and then for porting CP/M Plus over to it, taking full advantage of all of its new advanced features. I think in a short time that a new level of performance will be established.

Even though DRI has decided to put CP/M Plus on the mature list, and will no longer actively support it, I am continuing to port it to different computers. My most recent adventure is with the Intercontinental 8-bit S-100 master board; on-board memory management and many other performance features make the ICM board a natural for CP/M Plus.

Among the top ranking features of CP/M Plus is its capability to time and date stamp files automatically at each access. This feature is especially important to me because I have the untidy habit of not religiously tracking my disk files during a heated development session. I wait for the project to conclude before trying to make good sense out of my files and then sometimes have trouble recalling the order in which I created them, and, most important, which is the most current. Until Plu*Perfect Systems began to sell DateStamper there was no software available to add this desirable feature to CP/M 2.2. I've been using DateStamper for several weeks and have found it a most useful tool and safeguard. I will be reviewing it in a future issue of this column.

Finally, my stack of things to do wouldn't be complete without a new programming tool. I have been using DSD80, a DDT-compatible debugger with many new and useful added functions. It's a very solid package that turns the drudgery of debugging into an art form. I will be reviewing it in the future.

Encore, Encore

After producing the fastest full featured Z80 assembler, SLR Systems could do nothing for an encore but bring out the fastest fully featured 8080 assembler.

SLRMAC shares all the operational features and configuration options of its family member Z80ASM. A total of 32 permanently configurable options combined with 20 runtime command line parameters make for the ultimate in flexibility and ease of use. I found several of the configuration options to be most interesting.

A maximum error threshold count can be set that automatically aborts the assembly when reached. This option precludes resetting your computer or in some other way manually aborting an assembly if excessive errors are present.

Another option dealing with errors sets a limit on the number of errors that can be displayed on the CRT before the display is frozen. It's no longer necessary to sit poised to strike the CTL-S key combination to stop the CRT display in an attempt to catch a glimpse of an error flying by at lightning speed. Instead, the display automatically stops upon reaching the set error count and waits for the operator's signal to continue.

One other option that I particularly like is that addresses on the listing can be *generated in either* logical order, high byte first, or the normal low byte-high byte fashion. The mental step of swapping the address bytes slows me down as I move around in a program listing.

Command line arguments allow almost all of the permanent configuration options to be overridden and the

specification of many more options that are significant to an individual assembly.

One particularly important option deals with the type of binary file that is output from the assembly. The H option directs the output file to be in the most common .HEX format. To execute a program written in this format, of course, requires that it first be converted to a .COM file by processing it through the LOAD utility program. If the program consists of a single phase (all routines are self-contained) then another option can be used that will dramatically speed the process of converting the source program to an executable .COM file. By including the A option on the command line, SLRMAC will assemble the program in a single pass through the source file and create a .COM output file ready for execution. This option will also work for programs that contain forward references. It seems that SLR went the extra mile and included forward reference resolution logic in the assembler so that a single pass assembly is now a workable and timesaving reality.

If you're a modular programmer, as I am, then the three options dealing with relocatable output files will be of keen interest to you. Two are variations of the pseudo-standard Microsoft naming convention, while the other permits the design maximum of seven characters to be utilized. To provide advanced features not otherwise available in other binary formats, an SLR format can also be selected. This format is not only more compact than the others but it also provides for a full 16-character external name. This increased name size has proven to be very valuable to me in extremely complex systems containing many hundreds of individual modules, not to mention how much more quickly the link editing operation runs.

The restrictions pertaining to statement labels found in most assemblers have been removed from SLRMAC in most cases. It's not at all uncommon in some assemblers to be forced to follow all labels with a colon. In others, the archaic practice of requiring that only certain types of statements have labels followed by a colon is followed. SLRMAC doesn't require that any label be followed by a colon except in the case where a duplication exists between a reserved word or a macro name. In all other instances the use of the colon is entirely optional. One final word about labels in SLRMAC: a label of any length not exceeding the maximum line length of 128 characters is permitted and the beginning 16 characters are significant.

Many problems can occur when passing relocatable expressions from program to program. Most assemblers only allow the most simple uses of this type of expression. For instance, when loading the external value into a register, any type of math involving this type of expression is prohibited, which many times requires that a great deal of extra code be written. SLRMAC allows full expression usage via a special relative data type. In this way, arithmetic operations involving multiple relocatable expressions can be resolved properly at link editing time. But a word of caution is in order; the relative data type is only accepted by SLR's own link editing program.

The standard program counter maintenance pseudo-ops are available in SLRMAC. These include ORG, CESG, DSEG, ASEG, and COMMON, to name a few. A modified form of COMMON is available to assign up to 5 additional address spaces that can be assigned beginning addresses at link editing time. This can be very important if the program being written is targeted for a machine control environment in which dedicated portions of the address space must be defined. Again, this is an extension to the standard binary output files and as such is only available in the SLR format relocatable file.

When defining data storage areas it is not at all uncommon to need a string delimiter placed at the end of, for example, a console message. The DEFZ data definition statement will automatically generate a byte containing binary zeros at the end of the quoted string. And the DC statement will automatically turn bit 7 on in the last byte of the defined string.

A full complement of conditional assembly pseudo-ops and a full macro capability are provided. Two pseudo-ops that I have found to be especially useful are IFIDN and IFDIF. IFIDN tests for equality between two strings and sets the true condition according to the result. IFDIF performs exactly the opposite function of testing for inequality between the two strings and sets the true condition according to the result.

The method used to process the MACLIB pseudo-op is unique and worthy of comment. Intel mnemonic codes were designed to be used with the 8080, and there is no provision made for the additional instructions available on the Z80. When the program is being written for the Z80 and full use of its instruction set is desired, the extended instruction mnemonics are usually added to the assembly through a macro library. This solves the problem, but at the expense of slower run time and the necessity to maintain another macro library. SLRMAC, on the other hand, directly recognizes the extended set of Z80 mnemonics. Upon encountering the MACLIB Z80 statement the extend-

ed set of mnemonics are simply turned on, resulting in no speed degradation due to the loading of a macro library and macro generation.

SLRMAC is another notable product from SLR Systems. It not only does the job that it's advertised to do but it is also free of the many petty annoyances and inaccuracies that plague so many similar products. Its many advanced features, ease of use, and blinding speed make it a must for anybody in need of an Intel mnemonic assembler.

Your Attention Please

Before progressing further with the discussion of the Ampro Little Board computer begun last month, a few words on Z80 interrupts are in order. The Z80 and its family of related peripheral support chips have an excellent built-in interrupt system. Each support chip that has more than one active element has its own internal interrupt priority scheduler and the capability to be daisy-chained into a system. This structure does not require assistance from interrupt priority schedulers in all but the biggest systems.

The main controlling element in the system, the Z80 CPU, can handle an interrupt in one of three ways depending on the setting of the interrupt mode. This fact permits the software to determine, to a large degree, how complex the interrupt system will be.

The most basic interrupt mode, mode 0, operates identically to that of the 8080 interrupt response mode and was included to provide full compatibility with that earlier and less powerful CPU. This mode is set by executing the IM 0 instruction. With this mode, the interrupting device can place any instruction on the data buss and the CPU will execute it. Thus, the interrupting device, not the memory, supplies the next instruction to be executed.

Mode 1, when enabled by the IM 1 instruction, causes the CPU to execute a restart to memory location 38h in response to an interrupt. This action is identical to that of the nonmaskable interrupt except the call location in memory is 38h instead of 66h.

Finally, mode 2, the most powerful interrupt mode, combines three system elements to resolve the effective memory address loaded into the program counter. With a single 8-bit value supplied by the interrupting peripheral device an indirect call can be made to any memory location.

With this mode the CPU forms a 16-bit memory address. The upper eight bits are supplied from the Z80 interrupt register, and the lower eight bits are supplied by the highest priority interrupting peripheral device. This 16-bit address is used as an indirect memory address pointing to a stored table of pointer words that point to the processing routine responsible for servicing the interrupt. Actually, only seven bits are required from the interrupting device because bit 0 is always zero. This structure might seem overly complex until you consider that through software the service routine tables can be changed at will, providing the ultimate in flexibility.

Interrupts and the Little Board

The Ampro Little Board computer satisfied all the requirements I set for my RBBS system except the need for a real-time clock. My first impulse was to attach one through a serial port, but I soon remembered that both ports were busy; one with the CRT and the other with the modem. The lack of any unused I/O ports stymied any chance I had of using an external clock. The only option open to me was to look to the Little Board for a solution.

Digging into the schematics of the Little Board revealed a fairly straightforward solution to my problem. The engineers responsible for designing the Little Board used the Z80 and its related support chips exclusively to minimize interface logic requirements and arranged them to allow full use of their daisy-chained interrupt capabilities. So it seemed that all I needed was to find an unused timer, or some other source of regular interrupts, and a little software to implement a real-time clock.

Fortunately, my search was a short one. Channel 2 of the Clock Timer Circuit (CTC), used for generating the baud rates for the serial ports, was left totally unused. All that was needed to start it working as a real-time clock was the few lines of code of Listing One (page 120).

My first task was to define a table of pointer values (section 2 in the listing) addressing the interrupt processing routines. The address of the first entry of this table would later be loaded into the interrupt vector register. Following this table are the processing routines. I chose to place the clock values in low memory locations: 08h hours, 09h minutes, 0Ah seconds, and 0Bh tenths of seconds.

Next, I had to insure that the constant flow of interrupts from the CTC would not adversely affect any other time-sensitive system component such as the floppy disk controller. The only positive way to do this was to disable the interrupts (sections 3 through 6 in the listing) prior to entering the data transfer routines. In practice, this proved to be absolutely necessary because of excessive disk errors when running with interrupts enabled.

The side effect of this, however, is to destroy the clock's accuracy. Since it's unknown exactly how long the clock will be stopped *during a disk* data transfer, it's impractical to add in a fudge factor upon exiting the disk routines. By comparing the computer clock against a stopwatch I found that during a highly disk intensive period the computer clock would run approximately 50% slow. Under more normal circumstances the clock wasn't accurate, but was close enough for my purpose of timing how long a user was on the RBBS.

To achieve my goal of at least 95% accuracy I would have to run the floppy disk controller with interrupts enabled, but, as I already pointed out, that produces an unbearable error rate. My only choice was to modify the hardware slightly to allow both the CTC and the floppy disk controller to run with interrupts enabled. That is a project I am currently working on. Once complete, I will publish the software and other modifications in this column.

Getting back to the real-time clock

routines, the last addition I had to make to the BIOS (sections 7 and 8 *in the listing*) was to initialize the CTC and the interrupt register with the proper constants at cold boot time.

If you are inclined to make these changes to your machine, be sure that you are running version 2.3 of the BIOS, also known as the hard disk BIOS. Have a CTC reference manual handy to use when setting initial values.

One final comment: My system has a hard disk that ports into the Little Board through a SCSI peer buss adapter. I have found that this configuration runs completely error free because the hard disk controller is a buffered device able to transfer data while other tasks are active.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**

CP/M Exchange Listing

(Text begins on page 114)

```
1) Add at line number 551 after statement FDALV  EQU  50

INTRPT  SET     YES             ; SET INTERRUPTS TO NO AS DEFAULT
TIMER   EQU     YES             ; USE CHANNEL 2 OF CTC AS A TIMER
TYPER   EQU     NO              ; USE BUFFERED KEYBOARD

2) Add at line number 1028 after statement WTBLEN  EQU  $-WINCH3

        IF      TIMER

;=========================================================
;
;       MODE 2 INTERRUPT VECTORS FOR THE CTC
;
;=========================================================

        ORG     ($+17) AND 0FFF0H

TIMER$INT$TABLE:

        DW      BAUD$A
        DW      BAUD$B
        DW      TIMER$RTN
        DW      DSK$INT
        ENDIF

TIMER$RTN:

        PUSH    PSW             ;SAVE THOSE REGISTERS THAT WE DESTROY
        PUSH    H               ;*

        LXI     H,12            ;POINT AT HUNDREDS
        INR     M               ;INCREMENT IT
        MOV     A,M             ;GET HUNDREDS DIGIT
        SUI     10              ;AT LIMIT
        JNZ     TIMER$EXIT      ;NO
        MOV     M,A             ;ZERO HUNDREDS

        DCX     H               ;POINT AT TENS
        INR     M               ;INCREMENT IT
        MOV     A,M             ;GET TENS DIGIT
        SUI     10              ;AT LIMIT
        JNZ     TIMER$EXIT      ;NO
        MOV     M,A             ;ZERO TENS

        DCX     H               ;POINT AT SECONDS
        INR     M               ;INCREMENT IT
        MOV     A,M             ;GET SECOND DIGIT
        SUI     60              ;AT LIMIT
        JNZ     TIMER$EXIT      ;NO
        MOV     M,A             ;ZERO SECOND

        DCX     H               ;POINT AT MINUTES
```

 (Listing continued, text begins on page 114)

```
        INR     M               ;INCREMENT IT
        MOV     A,M             ;GET MINUTE DIGIT
        SUI     60              ;AT LIMIT
        JNZ     TIMER$EXIT      ;NO
        MOV     M,A             ;ZERO MINUTE

        DCX     H               ;POINT AT HOURS
        INR     M               ;INCREMENT IT
        MOV     A,M             ;GET HOUR DIGIT
        SUI     24              ;AT LIMIT
        JNZ     TIMER$EXIT      ;NO
        MOV     M,A             ;ZERO HOUR

TIMER$EXIT:

        POP     H
        POP     PSW

BAUD$A:
BAUD$B:
DSK$INT:

        EI
        RETI
```

3) Add at line number 3144 after label WR:

```
        IF      INTRPT
        DI
```

4) Add at line number 3159 after label WREXIT:

```
        IF      INTRPT
        EI
        ENDIF
```

5) Add at line number 3199 after label RD:

```
        IF      INTRPT
        DI
        ENDIF
```

6) Add at line number 3214 after label RDEXIT:

```
        IF      INTRPT
        EI
        ENDIF
```

7) Add at line number 4431 before label LOGMSG: and before JMP GOCPM.

```
        IF      TIMER
        XRA     A               ;GET A ZERO
        STA     8               ;CLEAR TIMER SAVE AREAS
        STA     9               ;*
        STA     10              ;*
        STA     11              ;*
        STA     12              ;*
        MVI     A,(LOW TIMER$INT$TABLE) AND 0F0H
        OUT     CTCA0           ;SET INTERRUPT VECTOR LOW 8 BITS
        MVI     A,HIGH TIMER$INT$TABLE
        STAI                    ;SET INTERRUPT VECTOR HIGH 8 BITS
        IM2                     ;SET INTERRUPT MODE 2
        MVI     A,0A7H          ;TIMER MODE - 256 PRESCALER - CONSTANT
                                ;FOLLOWS - RESET - CONTROL
        OUT     CTCA2
```

```
        MVI     A,156           ;TIME CONSTANT
        OUT     CTCA2
        ENDIF
```

8) Add at line number 4458 and after label LOGMD:

```
        IF      TIMER
        DB      CR,LF,'Real Time Clock Active'
        ENDIF
```

End Listing

Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

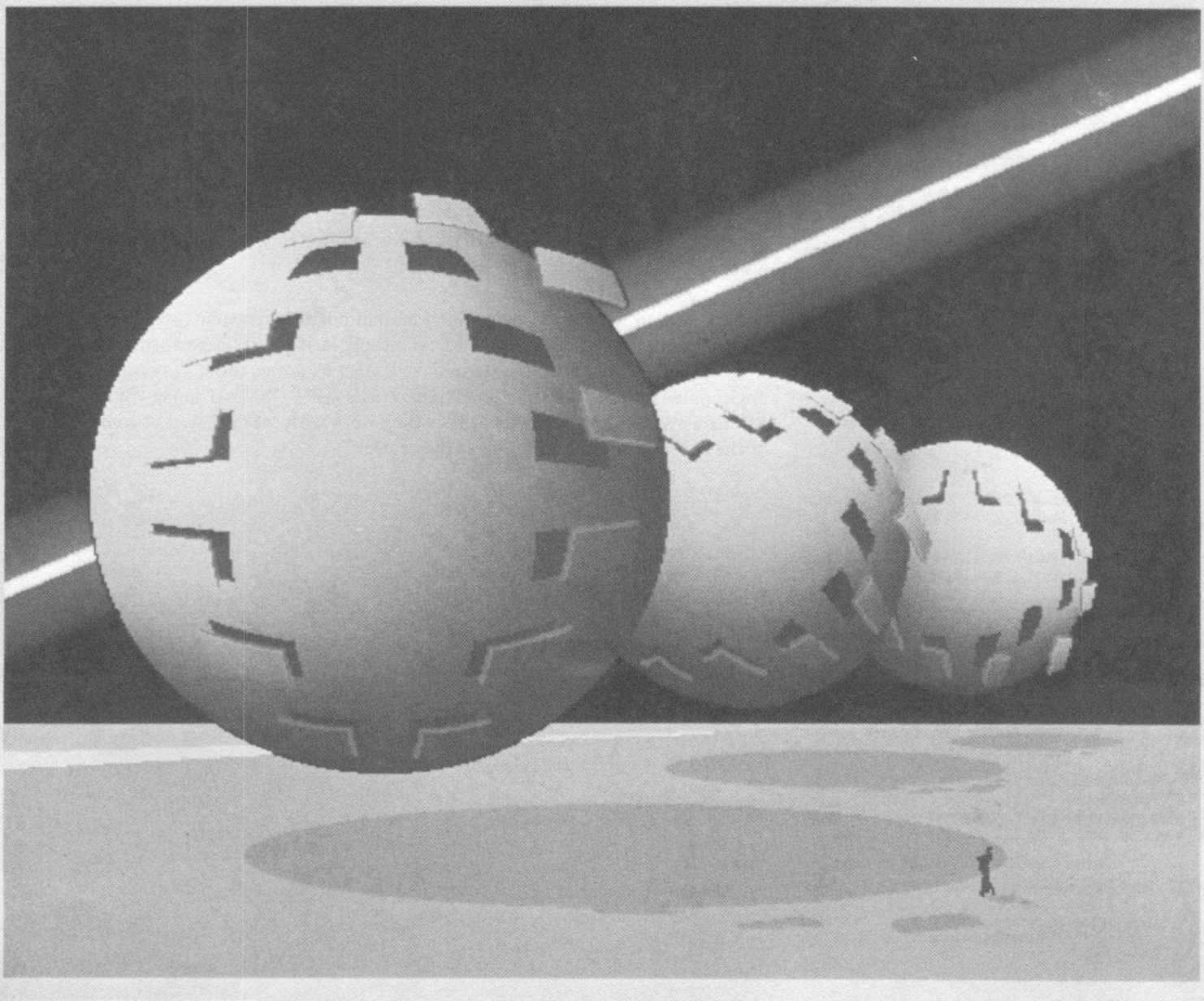
#109 NOVEMBER 1985 $2.95 ($3.95 CANADA

Modula Tools

Programmable Editors

An MSDOS Filter

68K and 286 Tips



11

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Vince Leone*
Typesetter *Jean Aring*

Production
Production Manager *Bob Wynne*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover Artist *Kazuo Morita*

Circulation
Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Kaminski*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Stephanie Barber*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*
Accountant *Marilyn Brown*
Adm. Coordinator *Kobi Morgan*

Advertising
Advertising Director
Shawn Horst *(415) 424-0600*
Systems Manager *Ron Copeland*
Administrative Manager *Anna Kittleson*
Advertising Sales
Walter Andrzejewski *(617) 567-8361*
Lisa Boudreau *(415) 424-0600*
Beth Dudas *(714) 643-9439*
Michele Beaty *(317) 875-0557*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0278-6508**

Customer Service: For subscription problems call: outside CA 800-321-3333; within CA 619-485-9623 or 566-6947. For book, back issue, or disk order problems call 415-424-1474.

Subscription Rates: $25 per year within the United States, $46 for airmail to Canada, $62 for airmail to other countries. Foreign subscriptions must be pre-paid in U.S. Dollars, drawn on a U.S. Bank.

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 686-1520 TELEX: 620430 (WUI)

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

ABC Audited Paid Circulation

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

November 1985
Volume 10, Issue 11

CONTENTS

In this Issue

Mark Edwards's review of ten programmable editors is the centerpiece of this issue, and we think it is another step in the direction we want to take in reviewing technical software tools. Edwards has asked all the questions we would ask about a programmer's editor and has tabulated the results; and he's done it for ten editors.

This month we also focus on Modula-2, the language that Pascal's father Niklaus Wirth thinks will replace Pascal for serious programming. Brian Anderson gives a little lesson in Modula-2 programming while presenting his bit-manipulation routines, and David Carroll's comparison of Modula-2 and Pascal introduces the language to those who have not yet encountered it. We've also supplied a brief list of sources of Modula-2 compilers and tools.

This issue's cover image, courtesy of Aurora Systems, is a work of computer art by Kazuo Morita of Tokyo.

Next month we present our second annual operating systems issue. We expect to offer a variety of tools to extend operating system performance for a variety of operating systems. We'll also examine three "operating environments" for 8088-type machines: DRI's GEM, IBM's TopView, and Microsoft's Windows, looking at how they work with MSDOS and at some of the subtler programming issues they raise.

Dr. Dobb's Journal

ARTICLES

Modula-2 vs. Pascal for Microcomputers: An Update — 28
by David W. Carroll
Why you might want to program in Modula-2, how it differs from Pascal in general and from Turbo Pascal in particular, and where to find Modula-2 development tools. **Reader Ballot No. 191.**

Bit Manipulation in Modula-2 — 38
by Brian R. Anderson
In Modula-2 the definition and implementation components of a process are implemented separately. Watch one programmer develop definition and implementation modules for some simple Modula-2 tools. **Reader Ballot No. 192.**

REVIEW

Programming Editors, Programmable Editors — 60
by Mark Edwards
From BRIEF to XyWrite: ten editors contrasted and evaluated as program-development tools. **Reader Ballot No. 196.**

COLUMNS

Dr. Dobb's Clinic — 14
by D.E. Cortesi
Readers and the Resident Intern analyze the Ross puzzle; *Wired Tales*, an anthology of computer-maintenance horror stories. **Reader Ballot No. 190.**

The Software Designer — 48
by Paul Heckel
The author of *The Art of Friendly Software Design* discusses the design decision he made in developing a new software metaphor. **Reader Ballot No. 193.**

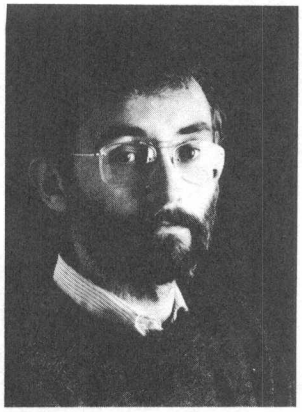
16-Bit Software Toolbox — 88
by Ray Duncan
68000 routines for square roots, binary search, and pseudo-random number generation, the 80286 vs. the 8086; an MSDOS DUMP filter. **Reader Ballot No. 194.**

CP/M Exchange — 102
by Bob Blum
A look at the U.S. Robotics modem, with a sample program showing how to implement all its features. **Reader Ballot No. 195.**

DEPARTMENTS

Editorial 6
Letters 8
Of Interest 122
Advertiser Index 128

EDITORIAL



In late August and early September, with no trace of embarrassment, *PC Week* announced the "first tangible results of AI research;" *Electronics* described new products that "move AI closer to the computing mainstream;" and *Computerworld* proclaimed that "AI is on the road to reality."

These claims that AI is once more just around the corner were prompted by product announcements at the International Joint Conference on Artificial Intelligence in August and expectations of product announcements at Comdex this month. We've heard such claims before, but this time they may be accurate.

It's not just the fact that Lisp machines and AI workstations are getting cheaper or the fact that Lotus, Ashton-Tate, and Microsoft are ostentatiously spending money on AI development. There are real products coming to market that employ real AI features.

Two recently announced database programs, ANSA Software's Paradox and Symantek's Q&A, employ AI techniques, as does Javelin Software's Javelin financial-modeling program. Q&A's natural-language interface is a descendant of Symantek founder Gary Hendrix's Ladder system that he began developing while at SRI in the 1970s. (These developments and the Borland-Analytica merger, with its prospect of a powerful $99 database package, should at least make the database field more interesting.)

Hendrix's work was originally done in Lisp and, although a germ of Lisp survives in the product, major portions are written in C and assembly language. This is a direction that other AI programs are taking as well. Teknowledge is porting its S.1 knowledge-based system-development software from Lisp to C for memory efficiency, performance, and—most notable—for compatibility with what they refer to as "conventional computing practice." "Conventional computing practice," Teknowledge is telling us, means C.

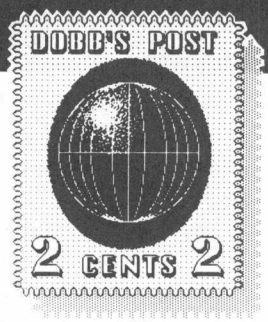
Neither Teknowledge nor any other firm thinks C is a good language for developing AI applications, but it's obvious they regard it as the appropriate language for delivering systems that require programmer involvement. Lisp is still the leading language in AI development—a status that won't be hurt by the current shift toward Common Lisp (which the trades will inevitably and casually label the standard, de facto or otherwise). HP, Apollo Computer, Intellicorp, DEC, and Xerox have all announced Common Lisp implementations or products based on Common Lisp. Prolog, the other significant AI programming language (which was written explicitly for AI work) gained support when IBM introduced a Prolog implementation for its VM operating system. But, as Michael Genesereth and Matthew Ginsberg point out in September's *Communications of the ACM*, the resolution principle is probably not an adequate deductive method for really effective logic programs, and Prolog is little more than a software implementation of the resolution principle. It's possible that we don't yet have the AI language we need to develop powerful AI-based commercial software.

Then again, IBM's new expert-system development product was written in Pascal—a reminder that you don't necessarily need just the right tool for the job.

At any rate, we at *DDJ* are optimistic enough about the near-term applicability of AI techniques that we're publishing another issue with this theme in April. But a good application of AI techniques is appropriate to any issue, so consider yourself invited to send us yours.

Michael Swaine

Michael Swaine



Replies to "Turbo Pascal vs. the Standard"

Dear *DDJ*,

We are responding to "Dr. Dobb's Clinic" on TURBO PASCAL (July 1985). Hopefully, the following will clarify some of the statements made by Mr. Cortesi in the article.

• The comments about TURBO PASCAL running under CP/M 2.2 and CP/M Plus are not accurate. Please refer to your own review of TURBO PASCAL in your June 1984 issue (referenced below).

• The discussion about the way the compiler treats available memory storage needs clarification. The programmer has total control over the amount of storage allocated by a TURBO PASCAL program. A brief glance at the CP/M-80 memory map (on page 291 of the Version 3.0 Reference Manual) reveals why the amount of available memory must be fixed at compilation time: static variables start at the end address and stretch downwards. For this reason, both Version 2.0 and 3.0 allow you to lower the end address of your program and thus enable .COM files to work on systems with less memory available.

• Regarding TURBO PASCAL and educational institutions, our compiler is well received and widely used. Computer science departments at more than 400 universities around the world have chosen TURBO PASCAL to be their "standard" programming language; ACM has selected TURBO PASCAL as the language for its national and international programming contests; and the Advanced Student Placement Testing Service has designated TURBO PASCAL its official testing language.

This brings us to the main thrust of Mr. Cortesi's article. He points out the ways in which TURBO PASCAL is nonstandard and states that "Borland has an obligation" to support all standard Pascal features. We give careful consideration to our users' requests for enhancements and additions to TURBO PASCAL and, unlike most software companies, we have already implemented many of these features in our two major upgrades. We will consider Mr. Cortesi's suggestions within this context.

Why do we use the Jensen/Wirth report? We consider it the most important standard and, for the most part, follow its guidelines. We are not the only ones to downplay the differences between ISO and Jensen/Wirth. *Dr. Dobb's* says, "For most users, the difference between the two standards will be small. . . . The compiler [TURBO PASCAL] accepts almost all standard Pascal statements. It is therefore possible to write highly portable programs" (June, 1984). We have carefully designed efficiency into our compilers—there are plenty of large, cumbersome systems available that provide total standard support.

TURBO PASCAL provides all the tools and utilities necessary for the vast majority of programmers to produce fast, high-quality scientific, business and personal software. This speed and power have made TURBO PASCAL the industry standard and the choice for more than 400,000 programmers. TURBO PASCAL's price, documentation and support have made it the best software value in the business.

BORLAND INTERNATIONAL
4113 Scotts Valley Dr.
Scotts Valley, CA 95066

I overstated the problems with CP/M storage allocation—see Winograd's letter below. Regarding all other points, I stand by my prior statements.—D.E.C.

Dear *DDJ*,

I read with interest D. E. Cortesi's comments about TURBO PASCAL in the July "Dr. Dobb's Clinic." He is correct in noting that because TURBO PASCAL doesn't look at the BDOS entry address at location 6, a program compiled with a larger TPA will not run in an environment with a smaller one. I originally experienced the same problem. Text processing programs that I wrote in TURBO PASCAL wouldn't run from the WordStar no-file menu, and programs I compiled on my Kaypro 10 crashed on my Otrona, which has a smaller TPA.

Fortunately, there is a very simple solution to the problem. When you command TURBO PASCAL to compile to disk (to do this you use the O and C options from the main menu), you will see a screen such as is found in the Table, page 12. This screen reflects the memory arrangement of my Kaypro 10.

The Start Address is the address for the first byte of the program code. This is one byte higher than the end of the TURBO run-time library. The End Address is the highest address for program code, which is normally (BDOS − 1), minus 700 to 1,000 bytes for the loader which allows other programs to be executed from the TURBO main menu (see p. 261 of the TURBO PASCAL 3.0 manual).

To solve the problem raised by Mr. Cortesi, simply reset the End Address from the above screen. Start by pressing E. TURBO PASCAL will then prompt

End address: A100

As shown by the entry above, I usually choose A100h as my End Address. This ensures that a program will execute on any system with a 40K TPA. This should be enough even for ancient homebrew systems, and it should allow programs to run on almost any machine from the WordStar no-file menu or with memory-resident utilities, such as keyboard enhancers, installed. If not, simply go back to the above menu and increase the End Address until the "Not enough memory" message stops appearing when the compiled program is run.

Ed Winograd
4704 Edison Lane
Boulder, CO 80301

All true and I should have known it. But this is still very restrictive: you have to size your program to the worst possible case, and it can never take advantage of storage over that minimum.—D.E.C.

Dear *DDJ*,
While it is apparent that Mr. Cortesi possesses a working knowledge of Pascal, it is also apparent that he has not worked with standards organizations. He makes such statements as "the standard" and "There is a Pascal standard," giving one the impression that there exists "one and only one" standard for Pascal. There could be nothing farther from the truth! Secondly, having been personally involved with several projects keyed against emerging standards, "draft," "final draft," and "standard" do *not* have the same meaning, practically or in any other sense of the word!

Mr. Cortesi then proceeds to tell me that ANSI, IEEE, and ISO are one big happy family. First, he ignores the fact that each one of these independent standards organizations has *published* their "standard" for Pascal separately, identified by each organization's individual "numbering/labeling" plan. Mr. Cortesi should have identified each standard by appropriate label/name, organization, and date, and then identified which of these was "the standard" for the purposes of his article.

Finally, I find it particularly difficult to accept the premise that merely because a programming language is standard (adheres to some recognized body's standards) it is in some way inherently portable. I would be extremely pleased to see your journal tackle this very volatile issue in the near future.

Bruce Hutfless
Tacoma, WA

There is just one U.S. domestic Pascal standard. It was the joint work of the IEEE and ANSI and was published in identical form by both bodies under the title American National Standard Programming Language Pascal. *The foreword to the document details the close liaison that was maintained between the U.S. joint committee and its international counterparts. There is a formal statement: "Differences of Technical Substance Between the Standard and the International Standard, as Represented by ISO/DIS 7185." Except for the U.S. omission of conformant arrays, these are exceedingly narrow technical points. There's a careful statement of how "compliance with this standard is equivalent to compliance at level 0 with the international standard." The joint committee's intent is quite clear: they mean for the U.S. and international standards to agree with each other. For all practical purposes, there is true worldwide agreement on a single standard for Pascal.—D.E.C.*

Dear *DDJ*,
Allow me to congratulate you on your excellent critique of TURBO PASCAL that appeared in the July issue of *DDJ*. Frankly, I wish that someone else had written it, since many people still remember Dave Cortesi's review of JRT Pascal and his reputation of being picky about Pascal implementations may lessen the impact. Nevertheless, he certainly said things that desperately needed to be said. All the uniformly glowing praise for TURBO was beginning to bother me. Finally someone has realized (in print) that, as implementations of Pascal go, TURBO is very mediocre.

My own experience with TURBO PASCAL has been rather unpleasant. Over the past three years I have written roughly 60,000 lines of Pascal code for a wide variety of applications. I do almost all my work under VAX/VMS, which has an absolutely first-rate Pascal compiler (full support of the ISO standard with optional warnings on nonstandard features, highly optimizing, 2000 lines/minute average compile time on a VAX-11/750, source-level debugging, etc.).

I get a request to port code to MSDOS about once a week. When TURBO first became available, it sounded great. The reviews made it sound like the ultimate compiler. I borrowed a manual from a friend and read it. It was obviously Pascal implemented by people who did not understand the basic philosophy of the language. Worse, the implementation was completely unusable. I write strictly standard-conforming code most of the time, but I frequently use all the features TURBO elected to omit, many of which are virtually impossible to work around. The result is that TURBO is absolutely useless. This is very difficult to explain to neophyte TURBO owners who want to use my code. "Buy a real Pascal compiler instead of a compiler for some random language invented by Borland" does not tend to go over well.

After countless arguments about standards (and more trips to the bookshelf) my customary response now is to simply give people the code they want and tell them to go for it. To forestall the inevitable arguments I also give them a listing with all nonstandard features flagged by the VAX Pascal compiler. After a few attempts they usually give up or just switch to some other implementation.

Edwin Earl Freed
Box 430
Perkins, OK 74059

Dear *DDJ*,
I read your Clinic column in the July 1985 issue with particular interest because I teach an introductory programming course to engineering students using TURBO PASCAL. Your points have generally been well

made, and I can only hope that Borland does the necessary to improve compatibility. Although Version 1.0 undoubtedly was written on a shoestring and in a hurry, there has been plenty of time (and money) for improvements to Version 3.0. In fact, I expect to continue using Version 2 because there has been so little improvement in Version 3.0.

I do expect to continue to use it for teaching, and I don't agree with your objection to its use for this purpose. Because of the fast compilation and built-in editor, students learn faster than with any alternative I have seen. What they learn appears to be valid "standard Pascal". You have not identified any features that are violations. There is more than enough to keep student programmers busy for a semester (or two) without ever getting to the New/Dispose and Get/Put problems, although the omission of the Page procedure is disappointing. They should be able to progress to another, more complete compiler if they really need such advanced features.

Edward H. Wiser
North Carolina State University
School of Agriculture and Life Sciences

Dear Mr. Kahn,
I recently purchased a number of copies of the MSDOS implementation of Version 3.0 of TURBO PASCAL for our department. Although there is much to be pleased with, I have a number of concerns that are very substantial. Fortunately, as I was gathering my thoughts on the subject, there appeared in the July issue of *Dr. Dobb's Journal* an article, by D. E. Cortesi, which expresses my views to perfection. I assume you will have seen it. Personally, I find the absence of Get and Put an enormous handicap. Their replacements are really pernicious.

To Mr. Cortesi's implied requests for a version of TURBO PASCAL that implements Pascal in a more "standard" way, let me add my own. The current implementation will not be acceptable for use in classes that teach Pascal. I will advise those I know to avoid using it.

Phillip Emig
Department of Mathematics
California State University, Northridge
Northridge, CA 91330

Dear *DDJ*,
I sent you a letter last month in which I demonstrated the way to correct a problem in TURBO PASCAL's code structuring. However, it seems that the letter must have been lost in the shuffle somewhere, as D. E. Cortesi's column in the July issue claims that TURBO PASCAL is "... unusable under CP/M Plus, and dubious under CP/M 2.2". The problem he describes is real, but as can be seen in my letter, the fix is extremely simple. And to date, no one else has offered or published this procedure.

Sincerely,
James R. Shiflett
P.O. Box 1236
Stafford, TX 77477

Mr. Shiflett had sent a discussion of how the CP/M version of TURBO allocates storage, including an involved method by which you can trick it into using all available storage. He's put the same information up on the Borland SIG on CompuServe. I thought that, while he'd done a first class job of systems work, his solution was a typical expert's hack: unsafe, prone to error, dependent on a particular compiler version and a particular operating system and thus absolutely nonportable, and in general not something I would want to use or publicize.—D.E.C.

TEX Features

Dear *DDJ*,
Thank you for the fine review you gave our product, PCTEX. We hope you will give Personal TEX a chance to mention recent enhancements to our product. We might add that these are not proposed additions, but actual features that are part of the product being shipped today.

- We have, now, a TEX driver for the Corona Laser Printer. It produces 300-dot-per-inch TEX finals at two 8½-by-11 pages per minute on an XT and six pages per minute on an AT.
- We offer a complete TEX publishing package for the PC consisting of PCTEX, the Corona driver, and a Corona Laser Printer, all for only $3,395.
- We carry the complete line of Textset device drivers (Apple LaserWriter, screen preview, and more) and OCLC Metafoundry fonts (Helvetica, Copperplate, Schoolbook, and others). (Lack of screen preview was one of the complaints of the reviewers.)

Our unbundled basic TEX package is still only $279—a little over half of the competition's bundled price. And we've got Michael Spivak, the gadfly author of *The Joy of TEX*, on our team. Michael has written a delightful introduction to TEX together with an easy-to-use macro package he calls VANILLA that comes only with PCTEX.

We would like to let potential buyers of document-formatting software know that TEX is all we do. When you talk TEX with us, you've got 100% of our attention.

By the way, we will soon be shipping our implementation of TEX, Version 1.5.

Lance Carnes
President, Personal TEX, Inc.
20 Sunnyside Avenue
Suite H
Mill Valley, CA 94941

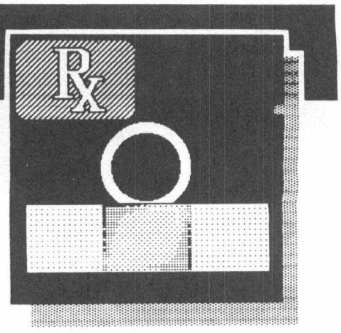
We have received an announcement of an agreement under which Addison-Wesley, distributors of MicroTEX, will also license and distribute the Textset laser printer and screen preview drivers.—Ed.

```
                    Memory
compile →           Com-file
                    cHn-file
Start address:      20E2 (min 20E2)
End address:        CA42 (max CD06)
Find run-time error Quit
>
```

Table
Adjusting TURBO Memory

DDJ

DR. DOBB'S CLINIC



by D. E. Cortesi

Watt Did Ross Dew?

In June we published a short BASIC program by David Ross and asked "Watt duzzit dew?" Several of you wrote to explain it to us. Some provided improvements; we'll get to those in a bit.

Ross's original program appears in Listing One (below). Watt duzzit dew? Richard Oakland of Fon Du Lac, Wisconsin, says, "In theory, Ross's program determines the digits of the number 2^k and places the character representations of those digits into successive elements of a string array of dimension $k+1$. It concludes by printing the array." Why is it so complicated, when

```
130 PRINT 2^K
```

would accomplish about the same thing? "It seems likely," Oakland says, "that the object was to display with full precision numbers that have more digits than single precision can generally handle . . . the author might even have hoped to display numbers that overflow single precision, numbers like 2^{400}, for example."

Why $k+1$ digits? David Barker of Mt. Clemens, Michigan, says that this number "can be justified because it requires $k+1$ digits to represent 10^k, and, because 2^k is less than 10^k, you will always be able to express 2^k in $k+1$ digits or less."

How did Ross want his program to work? Let's go through it line by line, as several readers did. Look first at lines 110 and 140, which are closely related:

```
110 DEF FNL(X)=LOG(X)/T
140T=LOG(10)
```

We don't all work with logarithms every day, so here's a brief review. The BASIC function LOG( ) returns the *natural log* of a number, the logarithm to the base *e*. This function is usually written ln( ) to distinguish it from the *common log*, or logarithm to the base 10, that we met first in school. Why does BASIC provide only natural log? Because it can be used to obtain a logarithm to any base: the log of a number x to any base b is

$$\ln(x)/\ln(b)$$

Function FNL(X) is taking advantage of that identity to provide the common log of its argument X. It would have been clearer if written

```
DEF FNL(X)=LOG(X)/ LOG(10)
```

although slightly slower in execution.

Line 120 gets the power to which 2 is to be raised. It should be forced to be an integer because the program is clearly meant to produce an integer output. You should also test to see whether it is positive; 2^0 is just 1 and a negative power will cause errors later.

Line 130 creates the array of digits and clears it to zeros. Clearing is necessary, says David Shochat of Los Angeles, because "there is one sense in which this algorithm is efficient. The digit calculated at each iteration is actually the next *nonzero* digit. If we use K=10 the '0' in '1024' will be skipped over. This is why the buffer has to be initialized to zeros at the start."

Line 160 sets up the target of the conversion, the number 2^k:

```
160 A0=K*FNL(2)
```

The crux of Ross's program is that this number, which might be very large indeed, can be represented safely in BASIC by its common logarithm. Because of the identity

$$\log(x^k)=k*\log(x)$$

Ross can get the log of 2^k without ever evaluating the number itself.

The Achilles' heel of the program is that, in most editions of Microsoft BASIC, LOG and all the other transcendental functions are evaluated only in single precision, so the value

```
100 REM BASIC PUZZLE BY D. ROSS
105 REM MODIFIED FOR MBASIC BY DEC
110 DEF FNL(X)=LOG(X)/T
120 INPUT "K=",K : DIM A$(K+1)
130 FOR I=1 TO K+1 : A$(I) = "0" : NEXT I
140 T = LOG(10)
150 Z = ASC("0")
160 A0 = K*FNL(2)
170 B = 1
180 WHILE B > 0
190     A = A0+FNL(B)
200     N = INT(A)
210     IF N<0 THEN 280
220     L = A-N
230     X = 10^L
240     D = INT(X)
250     A$(K-N+1) = CHR$(D + Z)
260     B = B-10^(N-A0+FNL(D))
270 WEND
280 FOR I=1 TO K+1 : PRINT A$(I); : NEXT I
```

Listing One

of 2^k is known only to six significant digits, regardless of its size. As Dave Shuman of Pendleton, Indiana puts it, "sizable numbers rapidly exceed the capacity of significant digits calculated in line 160. Even if the rest of the program contained infinite significance, the necessary information has been lost." No doubt, that's why Ross didn't put a DEFDBL at the head of his program. Some BASIC implementations, however, *do* support LOG with greater precision, as we'll see later.

Now that we have some idea of what the program tries to do, let's look at the loop in which it does it. Its first step is

```
190 A=A0+FNL(B)
```

David Shochat's analysis helped us see that here Ross is using another logarithmic identity

$$\log(x\times y)=\log(x)+\log(y)$$

to compute

$$A=\log(B\times 2^k)=\log(2^k)+\log(B)$$

The first time through B=1, so A is just A0. Now the program takes that value apart:

```
200 N=INT(A)
210 IF N<0 THEN GOTO 280
220 L=A-N
```

Shochat says "N and L are the characteristic and mantissa, respectively, of the log of $B\times 2^k$, that is:

$$B\times 2^k=10^{(N+L)}=10^N\times 10^L$$

Dave Shuman puts it this way: "Line 200 determines which digit N is next (the integer portion of a logarithm determines the decimal position of its antilog); line 210 terminates the loop in case the logarithm didn't come out exact; and line 220 gets the rest of the logarithm, L, which tells what the digits are beginning at position N."

The next lines process the number further:

```
230 X=10^L
240 D=INT(X)
```

Shochat analyzes this as setting up so that

$$B\times 2^k=X\times 10^N,\ 1\le X<10$$

"which is just scientific notation." Shuman seconds him: "the antilog of X is the number in the form a.bcde. . . ." Its integer part, assigned to D, is the next digit to display. The next line,

```
250 A$(K-N+1)=CHR$(D+Z)
```

assigns that digit as an ASCII character to the output array. It isn't hard to see how this code might pick off the *initial* digit of A0, but how does it get the rest of them? The next line is the key:

```
260 B=B-10^(N-A0+FNL(D))
```

Shuman says it removes the new digit by adjusting the factor B: "At each cycle, if you multiply B times A0, the original power of 2, you should get the rightmost N-1 digits of A0."

Shochat tries to explain how. "On the first iteration B=1, so the first D we get is the leading digit in the decimal representation of 2^k. Now the new B value has to lop off somehow the digit just found. It is

$$B-10^{N-A0+\log(D)}$$

which is

$$B-10^N\times(1/10^{A0})\times 10^{\log(D)}$$

which is

$$B-10^N\times(1/2^k)\times D$$

Now the trick is to think of this multiplied through by 2^k, which gives

$$(B\times 2^k)-(D\times 10^N)$$

Thus the new $B\times 2^k$ is just the old $B\times 2^k$ with D's contribution removed. At the end of the iteration where D is the last digit, this will bring B down to zero and end the loop." If it doesn't work out even because of round-off errors, the value of A calculated in line 190 will be less than 1, so the characteristic of its log assigned to N will be less than zero and line 210 will bail out.

Why Doesn't It Work?

This is all very ingenious but, as you know if you've tried the program, it doesn't work very well. Using MBASIC on CP/M or IBM BASIC before Version 3, it produces correct answers for only K=1, 2, and 3 (producing 2, 4 and 8). Then it gets progressively less accurate. How come?

There are two problems. Shuman says, "The most immediate problem is the conversion in line 240, where the real number X is converted to the integer D. For the rightmost digit we run into the classic problem of fixed/float equivalence. The next digit might be 4, for instance, but X was calculated as 3.99578 and D will then be 3. In all but the rightmost digit, the number X will be higher than the digit value, reflecting the digits yet to come, and the conversion is OK.

"I inserted a new line 280. Because we can tell by inspection that all results for K greater than zero should be even numbers, the new line 280 adds 1 to any results that end in an odd digit."

When this fix is made, the program produces correct answers up to the point where the number of digits in 2^k exceeds the precision of the values in A0, A and B, which are in turn limited by the precision of LOG. It can do K=20 in our CP/M system. Jack Fay of Seattle got to K=24 in Applesoft BASIC. Larry Manns of Trappe, Pennsylvania, got right answers to K=35 on a Wang. By using double precision on all variables in Microsoft BASIC 3.0, Shuman got up to K=51 on an IBM PC/AT. Shochat worked it to K=100 on a VAX.

The fundamental constraint is that, although logarithms allow Ross to cope with the *magnitude* of very large numbers, the limited *precision* of float variables keeps the program from displaying their many digits accurately. "My hat is off to David Ross," Shuman says. "In one elegant little program he gives a demonstration of two classic problems in computer number representation: loss of significance and inexact representation of whole numbers."

The program in Listing Two (page 21) shows our attempt to stretch the precision of MBASIC to its outer limits. It squeezes out two more digits to reach K=24 by using double precision and literals for ln(10) and LOG(2). By some fluke it also gets $K=30$ right, but not the numbers above or below it.

Other Ways

Two readers tried to do Ross one better. The first was Orin Safier. He simply automated what you'd do to calculate a large power of 2 by hand: you'd write down the largest power of 2 you knew or could look up; then you'd sit there doubling it over and over, producing something like the following table

65,536
131,072
262,144

With a little practice you can compute the digits of the next number almost as fast as you can write them. But why not let a program do the doubling?

Our rewrite of Safier's program appears in Listing Three (page 22). We changed the original only for longer variable names and to use a Boolean expression, not an IF, to handle the carry. Safier says the program "will print 2^k with perfect accuracy" but "involves many iterations for high values of k, and is thus abstractly less satisfactory than Ross's." That's a good point. Can you work out an approximate expression for how many iterations Ross's loop will make as a function of the input, K? What about Safier's?

And here's an extra-credit puzzle for APL users. Can you implement Safier's method, replacing the inner loop with non-looping array operations? You'll have to find a way to handle the carries in parallel.

David Barker also set out to better Ross, but used Pascal instead of BASIC. Like Safier, he aimed to beat the limited precision of float numbers by implementing his own ultralong integers, storing each decimal digit as one entry in an array. But where Safier arrived at 2^k by doubling 1, K times, Barker looked deeper. He explained it to us in terms of binary multiplication. We'd prefer to explain it to you in terms of one of the identities of the exponential function, namely that

$$x^y = x^p \times x^q$$

provided that $y=p+q$ (for example, $2^5=2^3\times2^2$). Next, notice that any odd number y can be decomposed into a p of 1 and a q of the even number one less than y, whereas any even number y can be decomposed into equal p and q simply by breaking it in half.

Those facts, plus the identity that anything to the first power is itself, practically lead us by the nose to a recursive definition of a power function. The case of x^1 is the one whose answer we know immediately. The rules for decomposition supply us with a way to make the magic recursive statement, "If I knew the answer to the simpler cases x^p and x^q, I could compute x^y:"

```
PowerOf(x,y)
   if y = 1 then return x
   if odd(y) then
      let t = PowerOf(x,y-1)
      let t = t * x
   else
      let t = PowerOf(x,y/2)
      let t = t * t
   endif
   return t
```

Actually, the two legs of the "if" can be collapsed into one to make a simpler function:

```
PowerOf(x,y)
   if y = 1 then return x
   let t := PowerOf(x,y div 2)
   let t := t * t
   if odd(y) then
      let t := t * x
   return t
```

That's a general power function. When we know that x is 2, it gets simpler:

```
TwoToThe(k)
   if k = 1 then return 2
   let t = TwoToThe(k div 2)
   let t = t * t
   if odd(k) then
      double(t)
   return t
```

Although these are recursive functions, no level of recursion actually does anything until the next lower level has finished. Therefore, the variable t needn't be local to the function; each completed level may leave its result in a global t for the next level to work on.

Barker did our TwoToThe(k) function in Pascal, implementing t as a many-digit integer expressed as an array. If you give it 9865 digits, he says, you can evaluate 2 raised to the power 32767 in a couple of hours—and he sent a solid page of digits to prove it.

We liked his program so much we couldn't let it alone. We typed it in and then kept tweaking away at it, trying to make the multiplication of big integers (which dominates the run time) go faster. The much-abused result, so far from Barker's original that he might not recognize it, appears in Listing Four (page 22). It's for Turbo Pascal. Some of the tweaks may be beneficial only in the 8-bit version of Turbo, whose integer mul, div, and mod functions are decidedly lethargic.

Wired Tales

We asked if anyone had maintenance experiences to share, good or bad. It seems the bad experiences are easier to recall, or more fun to recount. Your responses put us in mind of *Weird Tales*, the old magazine of supernatural horror. Consequently, we decided to collect them together to form an anthology of tales of the *sub*-natural horrors of hardware. We call it *Wired Tales.*

No Parking

When he rested his head
he got it in the neck!

After working with computers for eleven years and using my S-100 system for seven, *says Michael M. Dodd of Northbrook, Illinois,* I upgraded to a Compaq Deskpro. I'd had it just three weeks when I blew the power

supply by turning the computer off, then on, too quickly. Are all switching power supplies this fragile? In seven years I never had a problem with the old Processor Technology supply.

Two weeks later I turned the Deskpro on and the 33Mb Rodime hard disk wouldn't run. It wasn't even turning. The dealer replaced it at no charge, then told me that I should *never* use the head-parking program. It seems this disk automatically parks its head when power is removed. Apparently, the supplied parking program runs the head into the bolts that mount the platters, jamming its rotation like a stick in the spokes of a bike.

I restored all my data from tape backup to the new disk and was back in business—almost. dBase III wouldn't run because of its copy-protection scheme. Yes, Ashton-Tate will replace the diskette, but this means I can never restore data from tape without uninstalling dBase III. I paid money for this?

The Victim

They rolled him for $60 and left him catatonic!

Let me tell you about my printer, *croaks Glenn English of Austin, Texas.* It's a Dataproducts B-300. I've run several trees of paper through it, but recently it has developed quirks.

The first was in the ribbon rollers. The ribbon lives in a plastic box. It's pulled out one end, dragged through the print head, and stuffed back in the box by a pair of rollers made of something that looks like transparent rubber. Mine wore out. Dataproducts carries them—for $60 a pair. They last three years, so that's OK, but the delivery schedule is not. Dataproducts insists it takes two months to process an order for parts they have in stock. (They have an 'expedited' service for *many* more bucks.)

I went to the local DEC office and asked for rollers for an LP-25 (a B-300 in a new box). They couldn't figure out what I was talking about. If I had the Dataproducts part number, just maybe they could cross-reference it, but they couldn't find it under their DEC model number. Luckily, a DEC maintenance tech happened by and found them for me. Price? $60.

But now the machine is down in a most peculiar way. It runs its power-on self tests OK and its internal ripple-print test as well. Put it on-line and its on-line LED lights up. But send it data and it just sits there.

Dataproducts does have a service department, but they sent me to TRW. TRW won't work on it because it is in my house. (I have no idea why.) One of their techs offered to swap boards with me if I'd leave a deposit of $700. When I plugged in the new board the printer went completely catatonic. That's where it stands now. In the future I think I'll stay away from Big Kids hardware. If my ProWriter breaks, I can get a new one for what it's going to cost to get the B-300 back on the air!

Satisfied At Last

The first one wanted too much and died!

I chose Shugart 851 drives, *Jay S. Rouman of Mt. Pleasant, Michigan, tells us,* because they were supposed to be the standard of the industry, and besides (I heard) they just never failed. Wrong. A plague of intermittent 'bad sector' errors settled in and I began looking for a cure. I've spent more than a few hours at the repair bench but thought that a disk drive should be best left to the folks at the factory.

They'd be glad to repair my drive, they said, for a flat fee of $250. It warmed my heart to know they'd suffer no financial loss; nevertheless I looked elsewhere. An independent repair outfit offered a flat fee of $200.

There was a message in this, I thought. With the help of a friend I tracked the problem to the index sensor. This circuit consisted of one IC and a few resistors. Resistors never fail, so I replaced the IC. Wrong.

Now, the resistors were in a pack, and replacement ought to have been a snap; after all, resistor packs are pretty simple. Wrong. This one had very strange values in a strange configuration and was custom made for Shugart.

I jumped for joy when the clerk in Shugart's parts department said the pack was just $1.50 and they'd be glad to ship it to me. Then he told me about the $50 minimum order. I complained that 32 and a third spare packs was a bit much for my two-drive system, but it got me only sympathy. He suggested I contact repair outfits or that I fabricate the part myself.

The homemade resistor pack (a wire sculpture of ⅛-watt resistors) got the drive in service, but I was steaming. I asked at several repair shops, but the best I could get was a promise to batch my order with one of theirs whenever they next might send one.

Then I happened on an ad for Hamilton/Avnet. That's a big company with many offices but still, the ad had the words *Shugart* and *parts* on the same page and gave a toll-free number. I called and it was wonderful! They had the part in stock. They would sell me a single one. They'd ship it UPS. They wouldn't insist on a purchase order or even a credit card—they'd bill me! And they were as good as their word.

Since then, Shugart has gone out of business (I like to think it's because of that $50 minimum-order policy) and I haven't needed Hamilton/Avnet again. Still, they're the first place I'll call if I ever need parts.

DDJ

Dr. Dobb's Clinic (Text begins on page 14)

Listing Two

```
100 REM Logarithm-based display of 2^K after D. Ross
110 DEFDBL A-Z : DEFINT I
120 REM if your LOG() is double, use LN10=LOG(10) here
130 LN10 = 2.302585093# 'natural log of 10
140 REM get max mileage out of short LOG()
150 DEF FNCLOG#(X#) = LOG(X#)/LN10
160 REM if your LOG() is double, use CLOG2=FNCLOG#(2#)
170 CLOG2 = .301029996# ' common log of 2
180 REM get integral power IK and set up
190 INPUT "Power of 2";IK : IF IK<=0 THEN END
200 DIM A$(IK+1) : FOR I = 0 TO IK : A$(I)="0" : NEXT
210 IZ = ASC("0")
220 A0 = IK * CLOG2
230 B = 1
300 WHILE B > 0
310     A = A0 + FNCLOG#(B)
320     IN = INT(A) : IF IN<0 THEN 399
330     ID = INT(10# ^ (A-IN))
340     A$(IK-IN+1) = CHR$(ID+IZ)
350     B = B - 10# ^ (IN - A0 + FNCLOG#(ID+0#))
390 WEND
399 REM early exit point
400 IF (ID AND 1) AND (ID<9) THEN A$(IK+1) = CHR$(ID+IZ+1)
410 FOR I=IK-INT(A0)+1 TO IK+1 : PRINT A$(I); : NEXT
420 PRINT : CLEAR : RUN
```

End Listing Two

Listing Three

```
100 REM display 2^K by doubling
110 REM after Orin Safier
120 DEFINT A-Z
130 INPUT "Power of 2"; K
140 SIZE = 1+ INT( K * LOG(2)/LOG(10))
150 DIM DIGITS(SIZE)
160 DIGITS(1) = 1 :REM 2^0 = 1
170 MSD = 1 :REM count of live digits
200 FOR I = 1 TO K
210     C = 0
220     FOR J = 1 TO MSD
230             Q = DIGITS(J)
240             Q = Q+Q+C
250             C = - (Q>9) :REM "TRUE" is -1
260             DIGITS(J) = Q - (C*10)
270     NEXT J
280     IF C THEN MSD = MSD+1 : DIGITS(MSD) = 1
290 NEXT I
300 FOR I = MSD TO 1 STEP -1
310     PRINT CHR$(DIGITS(I) + ASC("0"));
320 NEXT I
330 PRINT : CLEAR : RUN
```

End Listing Three

Listing Four

```
program TwoPower;
{$A-  8-bit turbo, allow recursion }
{$R-  turbo, range checks off after test }

const maxdigit = 2048;

type
    digindex = 1..maxdigit;     { index/count over digits   }
    digit = 0..9;               { stored digit of "number"  }
    addig = 0..99;              { temporary arith. result   }
{ type of a multi-digit decimal integer: it consists of an  }
{ array of digits with digits[1] being the LEAST signifi-   }
{ cant and digits[msd] being the MOST significant.          }
    number = record
        msd : digindex;
        digits : array[digindex] of digit
```

```
    end;

var
    nØ, nl : number;     { one is current result, depending  }
    which : boolean;     { ..on value of this switch }
    power : integer;
    timestab : array[digit,digit] of addig;

{ procedure to initialize timestab. see MULBYN.PAS for use}

procedure initab;
var x,y : digit;
begin
    for x := Ø to 9 do
        for y := Ø to 9 do
            timestab[x,y] := x * y
end;

{ procedures to manipulate "numbers" ...first, one to write }
{ a number to standard output as in write(n).               }

procedure Shownum(var n: number);
var j : digindex;
begin
    write(' ');
    for j := n.msd downto 1 do
        write(chr(ord('Ø')+n.digits[j]))
end;

{ ...then one to set a "number" to a one-digit constant.    }

procedure Setnum(var n: number; d: digit);
begin
    with n do begin
        msd := 1;
        digits[1] := d
    end
end;

{ ..one to clear out k digits to zero (before multiply) and }
{ also set number to one-digit zero.                        }

procedure Zernum(var n: number; k: digindex);
var j: digindex;
begin
    with n do begin
        msd := 1;
        for j := 1 to k do digits[j] := Ø
    end
end;

{ ...one to double a number by adding it to itself.     }

procedure Double(var n: number);
var j : digindex; dig, Cy : addig;
begin
    with n do begin
        Cy := Ø;
        for j := 1 to msd do begin
            dig := digits[j];
            dig := dig + dig + Cy;
            if (dig < 1Ø) then Cy := Ø
            else begin
                dig := dig - 1Ø;
                Cy := 1;
            end;
            digits[j] := dig
        end; {for j}
        if (Cy <> Ø) then begin { number gets longer }
            if (msd = maxdigit) then
                begin
                    writeln('Oops, overflow on double');
                    Halt {turbo terminator}
                end;
            msd := msd+1;
            digits[msd] := 1
        end {if cy}
    end {with number n}
end;
```

Listing Four

```
    { ...and a general integer-multiply routine after that sent }
    { by David Barker: z gets x*y in the style of pencil and    }
    { paper multiplication.  Used in this program only to square}
    { a number as in z gets x*x.                                }

    procedure Mulnum(var x, y, z: number);
    var
        i,j,k : digindex;
        xdig, dig, Cy : addig;
    begin

    if ((x.msd+y.msd) > maxdigit) then
        begin
            writeln('Oops, multiply overflow!');
            Halt {turbo-specific termination}
        end;

    Zernum(z,x.msd+y.msd);
    k := 2; {in case x=0, which it can't in this program}
    for i := 1 to x.msd do begin
        Cy := 0;
        xdig := x.digits[i];
        if (xdig<>0) then begin
            k := i;
            if (xdig > 1) then
                {$I MULBYN.PAS mul-by-2-to-9 code}
            else
                {$I MULBY1.PAS mul-by-1 code}
        end {xdig > 0}
        else
            {x.digit[i] = 0, nothing to do}
    end {for i};
    with z do begin
        if digits[k]=0 then k := k-1;
        msd := k
    end

end;

{ In this program we have only 2 "numbers," n0 and n1.  We  }
{ start out using n0 and flop between it and n1 as we have  }
{ to square each temporary result.  The following procs use }
{ boolean "which" to apply Shownum, Mulnum, and Double to   }
{ "it" -- "it" being the one of n0, n1 currently holding the}
{ result of computations.                                   }

procedure Showit;
begin
    if which then Shownum(n1) else Shownum(n0)
end;

procedure Doublit;
begin
    if which then Double(n1) else Double(n0)
end;

procedure Squarit;
begin
    if which
        then Mulnum(n1,n1,n0)
        else Mulnum(n0,n0,n1);
    which := not(which)
end;

{Here is the recursive power-of-two algorithm,almost exactly }
{ as worked out in the text. }

procedure TwoToThe(K : integer);
begin
    if (K < 4) then begin
    { bottom of recursion, initialize number }
        which := false; { using n0 }
        case K of
        0 : Setnum(n0,1);    { doesn't happen in this prog. }
        1 : Setnum(n0,2);
```

```
    2 : Setnum(n0,4);
    3 : Setnum(n0,8);
    end
end
    else begin
    { not at bottom, recurse to initialize, then square }
        TwoToThe(K div 2);
        Squarit;
        if odd(K) then Doublit
    end
end;



begin
    initab;
    repeat
        write('Power of 2 (0 to end): '); readln(power);
        if (power > 0) then begin
            TwoToThe(power);
            Showit; writeln
        end
    until (power <= 0)
end.

{ --------- MULBY1.PAS ---------------------------------- }
{ code to multiply 1 * y.digits into z.digits, placed in }
{ an out-of-line include file for clarity }
begin
    for j := 1 to y.msd do begin
        dig := y.digits[j] + Cy + z.digits[k];
        if (dig < 10) then Cy := 0
        else begin
            Cy := 1;
            dig := dig-10
        end;
        z.digits[k] := dig;
        k := k+1
    end {for j};
    z.digits[k] := Cy
end {xdig=1}
\NP
{ --------- MULBYN.PAS ---------------------------------- }
{ code to multiply xdig * y.digits into z.digits when xdig>1}
{ placed out-of-line for clarity }
begin
    for j := 1 to y.msd do begin
{ note: in next statement, a direct "xdig mul y.digits[j]"  }
{ is replaced by a table lookup.  This reduces execution    }
{ time by 15% in 8-bit turbo but might be SLOWER in other   }
{ pascals or in PC turbo.                                   }
        dig := timestab[xdig,y.digits[j]] + Cy + z.digits[k];
        Cy := 0;

{ note: in 8-bit turbo the following ... }
        while (dig>9) do begin
            dig := dig-10;
            Cy := Cy+1
        end;
{ ...cut execution time by 50% compared to the more direct  }
{"if (dig>9) then begin Cy:=dig div 10, dig:=dig mod 10 end"}
{ ...this might NOT be true in other pascals or in PC turbo }

        z.digits[k] := dig;
        k := k+1
    end {for j};
    z.digits[k] := Cy
end {xdig>1}
```

End Listings

Modula-2 versus Pascal for Microcomputers: An Update

by David W. Carroll

Even Niklaus Wirth admits that current implementations for Pascal restrict its usefulness to writing academic or "toy" programs. Will Modula-2 validate the structured Pascal model for serious software development?

David W. Carroll is the author of Telecommunications for the PCjr *(Micro Text/Prentice Hall) and* Programming with Turbo Pascal *(Micro Text/McGraw-Hill). He operates the High Sierra RBBS (209-296-3534), a bulletin board dedicated to the exchange of information about Turbo Pascal and other high-level languages.*

The May 1984 issue of *DDJ* (#91) featured the article "Introduction to Modula-2 for Pascal Programmers" by Hugh McLarty and David Smith[1]. In that article the authors covered the differences between Standard (ISO) Pascal and Modula-2. The present article is also aimed at programmers familiar with Pascal, but discusses more practical aspects of using Modula-2 in the microcomputer environment.

Developers of microcomputer software have been slow to adopt structured, high-level programming languages. PL/M and PL/I-80, Pascal and, most recently, C have been used to develop large applications programs, compilers and assemblers, but only a few machine-level systems have been written in these languages, notably, parts of CP/M in PL/M and all of Unix in C. There are two main reasons for this: first, some of these languages do not allow direct access to the machine; secondly, the code generated by available compilers is often inefficient.

To see how inefficient compiled code can be, you need only observe the difference in operating speed between Lotus 1-2-3, which is coded in assembly language, and Context MBA, which is coded in C. The speed problems experienced by Microsoft in using C to develop Windows finally caused them to revert to assembly language for large segments of the program.

PL/M and PL/I-80 are powerful languages, but have never caught on for development of commercial applications. PL/M has been confined to the Intel development system environment and PL/I-80 suffers from the same problems as its older sibling for mainframes, PL/I, namely, size and complexity.

Pascal was originally intended to be a teaching tool, a model language to demonstrate and foster structured programming in computer science courses. It was not meant to be a language in which large systems programs were written, and so does not include many of the features that are helpful, or necessary, for that task. Because the language is well structured and easy to learn, many software vendors extended it to include more features. However, because a formal standard for Pascal was a long time coming, many different versions were developed, resulting in the current incompatibilities between most extensions of the language.

C is a very popular high-level *hanguage that* does provide access to the machine level. It does not, however, enforce typing or offer the other structured concepts that Pascal does (for better or for worse). C code can also become very obscure (some may read elegant) and difficult to maintain.

Copyright © 1985 by David W. Carroll, 23522 Shake Ridge Rd., Volcano CA 95689.

Origins of Modula-2

Niklaus Wirth, who originally designed Pascal in 1971, created Modula-2 in 1977 and released the first technical report on the language in March, 1980. Modula-2 is based largely on Pascal and Mesa, a modular language developed by Xerox; the support for concurrent processing comes from an earlier Wirth language for real-time programming called Modula. Modula-2 eliminates most of the deficiencies of Standard Pascal. It also provides features that allow large systems to be broken down into small components that can be developed independently in a truly top-down fashion. Although other structured languages try to achieve this goal through the use of procedures and functions, Modula-2 achieves a much higher level of modularization and data isolation. Dr. Wirth noted at conferences at Stanford University and Sacramento, CA in June 1985: "I [developed] Modula-2 because all the compilers that were becoming available [for] Pascal were fine for toy [small] programs, fine for introductory courses . . . I [wanted] to show that structured programming languages [were] not just for the school. Their real value comes when you do big systems. For that you need efficient compilation."

Pascal's Limitations

If you program in Pascal, you are familiar with its limitations. It is true that extended versions of Pascal have been developed. There exist, however, no standards for these extensions, so that any application that uses them is not portable. A summary of the things missing from Standard Pascal is presented below:

1. Separate Compilation
2. Machine level interface
   - Bit-wise operators
   - Direct port and memory access
   - Absolute addressing
   - Interrupt structure
3. Dynamic strings
4. Multitasking
5. Procedure libraries
6. Definable abstract data types
7. Programmer definable scope of objects
8. An elegant way to exit loops before completion

The Benefits of Modula-2

The major advantage of Modula-2 is the concept of the module. Modules are stand-alone, self-contained units. Although they are compiled *individually*, they are not *independent*. A module consists of two parts, a definition module and an implementation module. The definition module declares all portions of the module that are visible from the outside, that is, all exported objects, as well as any imported objects that are needed to declare the exported objects. This allows the compiler to check in the library files for all required import symbols and to generate a symbol file for the new module. In addition, the inner components of the module are invisible to other modules, except for the items exported. The implementation module contains the actual program code. It is used to generate an object code file to be combined with other modules when the program is ready to be linked together.

The following example shows how Modula-2 can be useful in developing a large system. Suppose that a defense system is being built in which all files must be encrypted before being written to disk. With Modula-2, once the parameters have been specified, the encryption routine can be developed independently by a small group of programmers with a high security clearance. The rest of the module will be invisible. In the meantime, other programmers can develop the rest of the system using a dummy file access module in the library. When the encryption module is completed, it is included in the library, and the system program is re-compiled. A summary of the benefits of Modula-2 is presented below:

1. Modules
   - Separate Compilation with parameter checking
   - Control over visibility
   - Declarations vs. code sections
2. Definable abstract data types
3. Multitasking capability
4. Interrupt handling
5. Low-level definitions
6. Libraries
7. Larger programs

Using Modula-2

Modula-2 programs are generated in stages. First, the source file is compiled and the symbol (SYM) and link (LNK) files are produced. At this time, all external objects to be imported are checked. If they are not currently available in the library modules, they are flagged. When all modules have been compiled, the master program module is linked to all other needed modules. Finally, the output file is produced in whatever form is supported by the particular compiler (assembly, object [.COM or .EXE], or load file).

Pascal Programmers and Modula-2

Experienced Pascal (and C) programmers will have little difficulty acquainting themselves with the basic features of Modula-2 within a few hours; they should be familiar with the large number of standard library modules within a few weeks. However, the features that support the development of multitasking/multi-user programs go beyond the scope of Pascal and will naturally take longer to learn.

Differences from Pascal

An excellent reference book for programmers making the transition from Pascal to Modula-2 is *Modula-2 for Pascal Programmers*, by Richard Gleaves[2]. I will present here a brief summary of major differences between Pascal and Modula-2. However, a discussion of the added support for multitasking is beyond the scope of this article.

• Modula-2 specifically uses the ASCII character set. This

eliminates problems in porting to mainframes that use other schemes. In Modula-2 all identifiers are case sensitive. This means that GETDATA, GetData, and getdata are three different identifiers. Also, underscores are not allowed in identifiers. All reserved words must be written in upper case. Comments use only (* and *) as delimiters and may be nested.

- A new type has been added, the unsigned integer or CARDINAL type, used for counting numbers. Reals and Integers cannot be mixed in Modula-2, as the implicit type conversion has been eliminated.
- Program structure is very similar to that found in Pascal, but there are some minor differences. The reserved word BEGIN is no longer used to delimit blocks (except for procedure blocks); instead, all control structures allow multiple statements and require an END delimiter (except REPEAT . . . UNTIL blocks). The FOR structure has eliminated the DOWNTO keyword and now allows the step value to be declared. The declaration section of a procedure block no longer requires a specific order of declarations (in Pascal, LABEL, CONST, TYPE, VAR, PROCEDURE or FUNCTION). The last statement in a procedure must be an END with the same identifier as the procedure heading. FORWARD declarations are no longer required; procedures may be referenced before they are declared. PROGRAM declarations are no longer used; instead, the MODULE is the compilation unit.
- FUNCTIONs are not available in Modula-2; instead, a typed procedure is used that must include a parameter list. Procedures and typed procedures may have an empty parameter list. RETURN may be used to terminate a procedure, and a value may be included.
- A new LOOP structure is included, delimited with an END. The loop continues until an EXIT statement is encountered, at which point control transfers to the statement following the END.
- Formal open array parameters are allowed in procedures in the form ARRAY of T. The size of the array is determined at the time of activation by using the HIGH utility procedure.
- A significant change in Modula-2 is that all utility and file functions are now performed by library modules. Thus, even the simplest program module must IMPORT I/O procedure identifiers and EXPORT values.

Small Pascal programs can be converted easily to Modula-2. One major difference, as noted above, is the case sensitivity of Modula-2: all keywords must be written in upper case and other identifiers must be consistent. The program structure is nearly the same. Control structures simply do not use a BEGIN, although they require an END. IMPORT and EXPORT statements are required to access library routines like I/O. Functions must be changed to typed procedures.

Longer programs will benefit from separate compilation of modules and the use of hidden objects allowed in Modula-2.

Modula-2 vs. Turbo Pascal

Ever since the release of Turbo Pascal two years ago by Borland International, there has been a resurgence of interest in the use of Pascal on microcomputers. With over 350,000 copies of Turbo Pascal sold, a large base of new Pascal users has been created. In fact, one prominent analyst of the software market has suggested that the influence of Turbo Pascal will cause Pascal or a Pascal-like language (Modula-2 or Ada) to become the language of choice for educators, engineers, software developers, and hackers, and possibly even to replace BASIC through the 1980s and 1990s. So, now that so many people are using Turbo Pascal, why should you consider switching to Modula-2?

Well, first let's get one thing straight: Turbo Pascal is not a true Pascal. As D.E. Cortesi pointed out in the July 1985 *DDJ*[3], Turbo does not conform to ISO Standard Pascal with respect to several required features. *[See this month's Letters section for response to Dave's column. Ed.]* In addition, Turbo has a large number of extensions designed to overcome many of the limitations of Standard Pascal. These two facts limit the portability of all programs written in Turbo Pascal.

Moreover, Turbo Pascal has limitations of its own. The most obvious one is that on the PC/MSDOS version both the program and the formal data area are limited to 64K. Of course, the heap area consists of all remaining memory, but not all data elements can be made up of pointer-variables!

Another limitation of Turbo Pascal is that it only produces .COM files, .CHN (chain) files, and overlay files. Furthermore, it is difficult to link separately compiled Turbo programs or separately assembled machine language programs. All parts of a Turbo program must be compiled together at the same time. There is no facility for separate compilation. If you are writing a 10,000 line program, this can be a significant factor.

What, then, is the value of Turbo Pascal. It is useful for writing small programs, but not large systems. It is also very important for demonstrating structured programming concepts, as Wirth originally intended. Borland is expected to release Turbo Modula-2 by the end of 1985 (Beta versions of the compiler for Z80 CP/M systems are currently being tested). The 8086 version will probably solve one of the problems with Turbo Pascal by allowing the use of the entire available memory in an IBM PC for both program and data storage.

The design of Modula-2 virtually requires separate steps for compilation and linking, as well as extensive library checking during compilation. This means that Modula-2 compilers will be somewhat slower to use for pro-

Tools for Modula-2 Programmers

Modula-2 compilers are currently available for many large and small systems. In addition to compilers, some companies (such as Information Systems, listed below) are developing lines of Modula-2 utilities. Modula Corporation is even selling a commercial version of the Lilith, the Modula-2 workstation developed by Wirth's team in Zurich. (Some 200 of the original Liliths were built.) The following list, prepared with help from contributors Brian Anderson and Ed Joyce, contains addresses of vendors of Modula-2 compilers and programming tools for microcomputers and some minicomputers. We expect to be able to add to this list in February, when we hope to review several Modula-2 compilers.

Modula Tools

Borland International
4585 Scotts Valley Dr.
Scotts Valley, CA 95066
(800) 556-2283
Borland has a Modula-2 compiler for CP/M in Beta test, scheduled for release this year, with an MSDOS port to follow.

Fachbereich für Informatik
Universität Hamburg
Schluterstrasse 70
D-2000 Hamburg 13
West Germany
This is a source for a Modula-2 compiler for VAX/VMS systems.

Hochstrasser Computing AG
Leonhardeshalde 21
CH-8001 Zurich
Switzerland
01-47-55-48
Four graduates of ETH, where Wirth developed Modula-2, have written a Modula-2 compiler for Z80 CP/M systems.

Information Systems
1901 N. Fort Myer Drive
Arlington, VA 22209
(703) 522-8898
Thomas Woteki has developed a suite of Modula-2 programming tools using Logitech's Modula-2/86.

Interface Technologies
3336 Richmond, Suite 200
Houston, TX 77098
(800) 922-9049 and (713) 523-8422
M2SDS is a low-priced compiler for MSDOS systems; SDS-XP is a more expensive version. Preliminary reports indicate that these products still contain a number of bugs. ITS has established a bulletin board (713-523-7255, 300/1200 baud) to provide information and services for Modula-2 programmers.

Logitech
805 Veterans Blvd.
Redwood City, CA 94063
(415) 365-9852
Logitech has roots in early Modula-2 and Lilith development in Switzerland; we reviewed an early version of its Modula-2/86 compiler for MSDOS systems in February of this year. The latest version is 1.10. Logitech also publishes the *MODULA-2 Newsletter* to provide information about its product line.

Modula Corporation
1673 West 820 North
Provo, UT 84601
(801) 375-7400 and (800) LILITH2
Modula Corporation has compilers for Apple II, Lisa, Macintosh, and MSDOS. Its MacModula runs on 512K or 128K machines.

Maritime Infosystems, Ltd.
6660 Reservoir Road
Corvallis, OR 97333
(503) 929-2552
The Mosys Modula-2 System is a fully extensible, adaptable Modula-2 programming support environment for Sage, Stride and Pinnacle computers.

Dr. Josef A. Muheim
BBC AG
Abteilung ESL
Werk Turgi
CH-5401 Baden
Switzerland
Dr. Muheim has a Modula-2 compiler for PDP-1/RSX-11 systems.

P. Robinson
Computer Laboratory
University of Cambridge
Corn Exchange St.
Cambridge CB2 3QG
England
Mr. Robinson has a Modula-2 compiler for VAX/Unix systems.

Scenic Computer Systems, Corp.
14852 NE 31st Circle
Redmond, WA 98502
(206) 885-5500
Scenic distributes a Modula-2 compiler for the 68000.

BENCHMARK	Compile	Link	Execute
Turbo Pascal Ver 2.0	0:01	n/a	0:16
Logitech Modula-2 Ver 1.10	1:27	1:34	0:25*

All times in min:sec
*Includes loading runtime module
Test run on TAVA PC XT from hard disk.

Table
Compile, Link, and Execution Times for Sieve of Eratosthenes Benchmark

gram development than fast Pascal compilers like Turbo. Short programs for finding prime numbers (using the Sieve of Eratosthenes algorithm) are shown in Pascal in Listing One (page 34) and Modula-2 in Listing Two (page 34). The compile and link times for Logitech's Modula-2 compiler (Version 1.10) and Turbo Pascal (Version 2.0) are shown in the first two columns of the Table (above). Here, as expected, Turbo Pascal is the clear winner. The third column shows the time required for the execution of the compiled code. Although the code generated by the Logitech compiler seeems to run more slowly, part of the figure consists of the time required to load the runtime module, about 10 seconds. Thus, in execution time the numbers are actually comparable.

Notes

[1] H. McClarty and D.W. Smith, "An Introduction to Modula-2 for Pascal Programmers," *DDJ* #91, May 1984 pp. 22–27.

[2] Gleaves, Richard, *Modula-2 for Pascal Programmers*, Springer-Verlag, 1984.

[3] D.E. Cortesi, "Turbo Pascal vs. the Standard," *Dr. Dobb's Clinic*, *DDJ* #105 July 1985, pp. 12–18.

Further Reading

Cashman, M., "Extensions and Performance Improvements Keep Pascal Computing," *Digital Design*, August 1984, pp. 106–107.

Cooper, Doug, *Standard Pascal User Reference Manual*, Norton 1983.

Jensen, Kathleen and Wirth, Nikalus, *Pascal User Manual and Report*, Springer-Verlag 1974.

Joyce, Edward, *Modula-2: A Seafarer's Guide and Shipyard Manual*, Addison-Wesley, 1985.

Meng, B., "Ada and Modula-2: True Systems Languages?" *Digital Design*, August 1985 pp. 74–79.

Wirth, N. *Programming in Modula-2*, Springer-Verlag, 1983.

——— "Modula-2, An Overview," *Micro Cornucopia* #25, August-September 1985 pp. 76–79.

(Listings begin on next page)

DDJ

Modula-2 vs. Pascal (Text begins on page 28)

Listing One

```
{ Eratosthenes Sieve for Prime numbers in Pascal }
program prime;
const
  size = 8190;
var
  flags : array[0..size] of boolean;
  i, prime, k, count, iter : integer;
begin
  writeln('10 iterations');
  for iter := 1 to 10 do            { do program 10 times }
  begin
    count := 0;                     { prime counter }
    for i := 0 to size do
      flags[i] := true;             { set flags all true }
    for i := 0 to size do
      if flags[i] then              { found a prime }
      begin
        prime := i + i + 3;         { twice the index + 3 }
        {writeln(prime);}
        k := i + prime;             { first multiple to kill }
        while k <= size do
        begin
          flags[k] := false;        { zero a non-prime }
          k := k + prime            { next multiple }
        end;  {while}
        count := count +1           { primes found}
      end;  {for, if}
  writeln(iter);
  end; {for }
writeln(count,' primes')          { # of primes found in 10th pass }
end.
```

End Listing One

Modula-2 vs. Pascal

Listing Two

```
(* Compute a table of the first n primes numbers.  Print
  primes (optional). Use the the Sieve of Eratosthenes algorithm.*)

MODULE sieve;

FROM Terminal IMPORT WriteString, WriteLn;
FROM InOut    IMPORT WriteCard;

CONST size = 8190;

VAR flags: ARRAY[0..size] OF BOOLEAN;
    i,j,prime,k,count: CARDINAL;

BEGIN
  FOR j := 1 TO 10 DO
    WriteString('Pass = ');
    WriteCard(j,2);
    WriteLn;
    count := 0;
    FOR i := 0 TO size DO flags[i] := TRUE END;
    FOR i := 0 TO size DO
      IF flags[i] THEN
        prime := i+i+3;
        k := i+prime;
        WHILE k <= size DO
          flags[k] := FALSE;
          INC(k,prime)
        END;
        INC(count);
   (*   WriteCard(prime,8);
        WriteLn *)
      END
    END
  END;
  WriteLn; WriteCard(count,0);
  WriteString(' primes'); WriteLn
END sieve.
```

End Listings

Bit Manipulation in Modula-2

by Brian R. Anderson

Though Modula-2 provides a facility to manipulate data bits directly (via BITSETs), the methods available are not always convenient. Possibly, the most common use of this facility in high-level language programming is manipulating character data (bytes). A trivial example would be clearing the high bits of all characters in a WordStar file. In C, this is an easy matter of a single instruction (for each character):

```
ch &= 0x7F; /* bitwise and */
```

In Modula-2 the same operation takes five instructions:

```
I := ORD (ch);
   (* I is a CARDINAL *)
B := BITSET (I);
   (* B is a BITSET *)
EXCL (B, 7); (* Exclude bit 7 *)
I := CARDINAL (B);
ch := CHR (I);
```

There has to be a better way! Though I knew from the outset that it was impossible to match C's simplicity, I set out to write a collection of procedures that would make it easier and more intuitive to perform common and important bit-manipulation operations in Modula-2

The first step was to write a definition module with eight common bit manipulations (set a bit, reset a bit, test a bit, shift left all bits, shift right all bits, AND/OR/XOR two bytes). This definition module is shown in Listing 1 (page 40) and provides the user (i.e., programmer) with an interface to the implementation module.

The first implementation module that I wrote was in Modula-2. My intention was eventually to write the code in assembly, but I wanted a working module to which I could compare the assembler version during the debugging phase. The Modula-2 implementation module is shown in Listing 2 (page 40). Despite the awkwardness of the Modula-2 syntax, this module was somewhat easier to write than the corresponding assembler module.

The Modula-2 compiler that I am using (Hochstrasser Z80) allows integration of standard MAC/REL files with Modula programs. This is accomplished by way of a conversion program. The assembly code is developed in the usual manner and then assembled using M80 or RMAC. A name translation file may be specified during conversion, at which time the REL file becomes an MRL file (Modula Object Code). (M80 shortens all identifiers to 6 characters and maps all characters to upper case, whereas this Modula-2 allows 24-character identifiers of both upper and lower case.) The assembler implementation module is shown in Listing 3 (page 44); the name conversion file is shown as Listing 4 (page 46).

With either implementation, the operation of clearing the high bit of

The implementation module can be coded in assembly language for speed without affecting the definition module.

Brian R. Anderson, 2977 E. 56th Ave., Vancouver, B.C. Canada

characters in a WordStar file now becomes as simple for the Modula-2 programmer as it is for the C programmer:

```
Reset (ch, 7); (* reset bit 7 of ch *)
```

In fact, this is simpler and clearer than the C code to perform the same function.

Performance

From the point of view of the programmer the two implementation modules look identical. This is guaranteed because they both use the same definition module. Though no error checking is done, the procedures behave sensibly. If they are asked to set bit 9, for example, they simply circle around and set bit 9 – 8 (i.e., bit 1). You should note that these procedures consider the least significant bit to be bit zero.

The assembler module performs considerably better with regard to code size and speed. The Modula-2 implementation compiles to 555 bytes of code and 10 bytes of data, whereas the assembly implementation results in only 129 bytes of code and no data. The procedures in the assembler version are also 4 times faster (on average).

Conclusions

Bit manipulation in Modula-2 need not be as cumbersome as suggested by the definition of the language. The provision of assembly language interface for high-level language compilers allows significant performance improvements.

I would like to hear from other programmers if the facilities provided by the Bits module are useful enough to be considered for inclusion in standard libraries. Perhaps other procedures should be included (Rotate?). Let us know what you think before we approach the Modula Users Society (MODUS) with a proposal.

(Listings begin on next page)

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 192.**

Bit Manipulation (Text begins on page 38)

Listing One

```
bits.def

DEFINITION MODULE Bits;
(* bit manipulation module *)
(* BA June 10, 1985        *)

   EXPORT QUALIFIED
      BYTE, Set, Reset, Test, ShiftLeft, ShiftRight, And, Or, Xor;

   TYPE
      BYTE = CHAR;

   PROCEDURE Set (VAR A : BYTE; bit : CARDINAL);

   PROCEDURE Reset (VAR A : BYTE; bit : CARDINAL);

   PROCEDURE Test (A : BYTE; bit : CARDINAL) : BOOLEAN;

   PROCEDURE ShiftLeft (VAR A : BYTE);

   PROCEDURE ShiftRight (VAR A : BYTE);

   PROCEDURE And (A, B : BYTE) : BYTE;

   PROCEDURE Or (A, B : BYTE) : BYTE;

   PROCEDURE Xor (A, B : BYTE) : BYTE;

END Bits.
```

End Listing One

Listing Two

```
bits.mod

IMPLEMENTATION MODULE Bits;
(* bit manipulation module *)
(* BA June 10, 1985        *)

   VAR
      cA, cB : CARDINAL;    (* cardinal value for parameters A & B *)
      bA, bB : BITSET;    (* bitset value for parameters A & B *)
      Result : BITSET;    (* result for and/or/xor *)


   PROCEDURE Set (VAR A : BYTE; bit : CARDINAL);

      BEGIN
         cA := ORD (A);    (* convert BYTE to CARDINAL *)
         bA := BITSET (cA);    (* 'coerce' CARDINAL to BITSET *)
         INCL (bA, bit MOD 8);    (* set the bit (make sure in range
                                                    0 --> 7) *)
         cA := CARDINAL (bA);    (* 'coerce BITSET back to CARDINAL *)
         A := CHR (cA);    (* convert CARDINAL back to BYTE *)
      END Set;


   PROCEDURE Reset (VAR A : BYTE; bit : CARDINAL);

      BEGIN
         cA := ORD (A);
         bA := BITSET (cA);
         EXCL (bA, bit MOD 8);
         cA := CARDINAL (bA);
         A := CHR (cA);
      END Reset;


   PROCEDURE Test (A : BYTE; bit : CARDINAL) : BOOLEAN;

      BEGIN
         cA := ORD (A);
         bA := BITSET (cA);

         IF (bit MOD 8) IN bA THEN
            RETURN TRUE;
         ELSE
            RETURN FALSE;
         END;
      END Test;
```

Bit Manipulation (Listing continued, text begins on page 38)

Listing Two

```
PROCEDURE ShiftLeft (VAR A : BYTE);

   BEGIN
      cA := ORD (A);
      cA := cA * 2; (* Shift Left is equivalant to Multiplication by 2 *)
      A := CHR (cA);
   END ShiftLeft;
PROCEDURE ShiftRight (VAR A : BYTE);

   BEGIN
      cA := ORD (A);
      cA := cA DIV 2;
      A := CHR (cA);
   END ShiftRight;


PROCEDURE And (A, B : BYTE) : BYTE;

   BEGIN
      cA := ORD (A);    (* both BYTEs must be forced to BITSET *)
      bA := BITSET (cA);
      cB := ORD (B);
      bB := BITSET (cB);

      Result := bA * bB;    (* AND the two bitsets *)

      cA := CARDINAL (Result);    (* force Result back to BYTE *)
      A := CHR (cA);
      RETURN A;
   END And;


PROCEDURE Or (A, B : BYTE) : BYTE;

   BEGIN
      cA := ORD (A);
      bA := BITSET (cA);
      cB := ORD (B);
      bB := BITSET (cB);

      Result := bA + bB;

      cA := CARDINAL (Result);
      A := CHR (cA);
      RETURN A;
   END Or;


PROCEDURE Xor (A, B : BYTE) : BYTE;

   BEGIN
      cA := ORD (A);
      bA := BITSET (cA);
      cB := ORD (B);
      bB := BITSET (cB);

      Result := bA / bB;

      cA := CARDINAL (Result);
      A := CHR (cA);

         RETURN A;
       END Xor;

END Bits.
```

End Listing Two

(Listing Three begins on page 44)

Bit Manipulation (Listing continued, text begins on page 38)

Listing Three

```
;IMPLEMENTATION MODULE Bits;
;(* bit manipulation module *)
;(* BA June 13, 1985        *)
       NAME    ('BITS')
;
       .Z80
;
;   EXPORT QUALIFIED
;      BYTE, Set, Reset, Test, ShiftLeft, ShiftRight, And, Or, Xor;
;
       PUBLIC SET, RESET, TEST, SHL, SHR, ANDB, ORB, XORB
;
;   TYPE
;      BYTE = CHAR;
;
       CSEG
;
MOD8   EQU     00000111B       ;Mask to calculate bit MOD 8
;
;
;   PROCEDURE Set (VAR A : BYTE; bit : CARDINAL);
;
SET:   POP     IY              ;Return Address
       POP     BC              ;bit #
       EX      (SP),IY         ;BYTE's address <---> Return Address
       LD      A,MOD8          ;Make sure bit
       AND     C               ; in range 0 --> 7
       LD      B,A             ;'Safe' bit # in B
       XOR     A               ;Clear Accum. & CY
       CCF                     ;Set CY to make mask
       INC     B               ;Adjust count
FINDS: RLA                     ;Rotate mask
       DJNZ    FINDS           ; until count is zero
       OR      (IY)            ;Set the bit
       LD      (IY),A          ;Return it to BYTE
       RET

;
;   PROCEDURE Reset (VAR A : BYTE; bit : CARDINAL);
;
RESET: POP     IY              ;Return Address
       POP     BC              ;bit #
       EX      (SP),IY         ;BYTE's address <---> Return Address
       LD      A,MOD8          ;Make sure bit
       AND     C               ; in range 0 --> 7
       LD      B,A             ;'Safe' bit # in B
       XOR     A               ;Clear Accum. & CY
       CCF                     ;Set CY to make mask
       INC     B               ;Adjust count
FINDR: RLA                     ;Rotate mask
       DJNZ    FINDR           ; until count is zero
       CPL                     ;Invert mask
       AND     (IY)            ;Reset (clear) the bit
       LD      (IY),A          ;Return it to BYTE
       RET

;
;   PROCEDURE Test (A : BYTE; bit : CARDINAL) : BOOLEAN;
;
TRUE   EQU     00000001B       ;
                               ;
TEST:  POP     HL              ;Return Address
       POP     BC              ;bit #
       EX      (SP),HL         ;BYTE <---> Return Address
       LD      IY,0            ;Clear a Pointer
       ADD     IY,SP           ;Make Copy of Stack Pointer
       LD      A,MOD8          ;Make sure bit
       AND     C               ; in range 0 --> 7
       LD      B,A             ;'Safe' bit # in B
       XOR     A               ;Clear Accum. & CY
       CCF                     ;Set CY to make mask
       INC     B               ;Adjust count
FINDT: RLA                     ;Rotate mask
       DJNZ    FINDT           ; until count is zero
       AND     L               ;Check if bit set
       JP      Z,FALSE         ;If zero, return FALSE
       LD      A,TRUE          ; else, return TRUE
FALSE: LD      (IY+2),A        ;Store function Return Value
       RET

;
;   PROCEDURE ShiftLeft (VAR A : BYTE);
```

(Continued on page 46)

Bit Manipulation (Listing continued, text begins on page 38)

Listing Three

```
;
SHL:    POP     HL              ;Return Address
        EX      (SP),HL         ;BYTE's address <---> Return Address
        SLA     (HL)            ;Shift Left (Arithmetic)
        RET

;
;   PROCEDURE ShiftRight (VAR A : BYTE);
;
SHR:    POP     HL              ;Return Address
        EX      (SP),HL         ;BYTE's address <---> Return Address
        SRL     (HL)            ;Shift Right (Logical)
        RET

;
;   PROCEDURE And (A, B : BYTE) : YTE;
;
ANDB:   POP     HL              ;Return Address
        POP     DE              ;'B'
        EX      (SP),HL         ;'A' <---> Return Address
        LD      IY,0            ;Clear a Pointer
        ADD     IY,SP           ;Make Copy of Stack Pointer
        LD      A,L             ;Move 'A' to Accum.
        AND     E               ;And with 'B'
        LD      (IY+2),A        ;Store function return value
        RET
;
;   PROCEDURE Or (A, B : BYTE) : BYTE;
;
ORB:    POP     HL              ;Return Address
        POP     DE              ;'B'
        EX      (SP),HL         ;'A' <---> Return Address
        LD      IY,0            ;Clear a Pointer
        ADD     IY,SP           ;Make Copy of Stack Pointer
        LD      A,L             ;Move 'A' to Accum.
        OR      E               ;Or with 'B'
        LD      (IY+2),A        ;Store function return value
        RET


;
;   PROCEDURE Xor (A, B : BYTE) : BYTE;
;
XORB:   POP     HL              ;Return Address
        POP     DE              ;'B'
        EX      (SP),HL         ;'A' <---> Return Address
        LD      IY,0            ;Clear a Pointer
        ADD     IY,SP           ;Make Copy of Stack Pointer
        LD      A,L             ;Move 'A' to Accum.
        XOR     E               ;Xor with 'B'
        LD      (IY+2),A        ;Store function return value
        RET

;
;END Bits.
;
        END                     ;
```

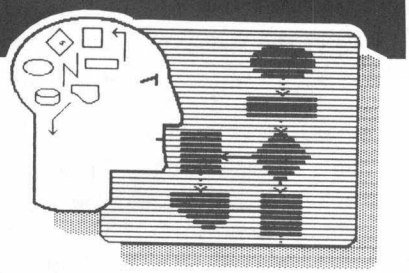
End Listing Three

Listing Four

```
(* Translation Table for converting Identifiers *)
(* from Assembler format to Modula format. *)
(* Table is used by module converter, which converts *)
(* standard REL file into MRL (Modula ReLocatable) Object file. *)

BITS        Bits
SET         Bits.Set
RESET       Bits.Reset
TEST        Bits.Test
SHL         Bits.ShiftLeft
SHR         Bits.ShiftRight
ANDB        Bits.And
ORBB        Bits.Or
XORB        Bits.Xor
```

End Listings

THE SOFTWARE DESIGNER



Zoomracks: Designing a new Software Metaphor

by Paul Heckel

The poet's eye, in a fine frenzy rolling,
Doth glance from heaven to earth, from earth to heaven;
And as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them into shapes, and gives to airy nothing
A local habitation and a name.
A Midsummer Night's Dream

In my book *The Elements of Friendly Software Design*[1] I treat the design of applications software as a communications craft such as writing, filmmaking, or advertising. For over two years at Quickview Systems, we have been refining the ideas expressed in this book and putting them into practice in a product called Zoomracks. Zoomracks offers users a new environment and a new productivity tool. We have incorporated in this product four innovations in user-interface design:

1) A new computer metaphor: the rack. A rack is a familiar object. People know how items are organized on a rack. Consequently, a rack is an effective organizational model for a computer program. We think that Zoomracks will become a conceptual extension of the physical rack in the same way that VisiCalc became a conceptual extension of the physical spreadsheet.
2) A new viewing mechanism called Smart Zooms. Unlike windows, Smart Zooms shows the "big picture" rather than the details.
3) A method of using toggles and inverses to give even the novice a feeling of control in using the software.
4) A concept, called Fieldscrolls, that allows a user to construct a wide variety of database schemes by specifying a minimum amount of information.

Though the first two concepts were fundamental from the beginning, the last two evolved almost accidently into something more powerful than we had expected. This article is intended to explain what our thought processes were in designing and developing Zoomracks and not just what the product is.

The Opportunity

When you develop a new product, you should limit the number of fundamental goals that you hope to achieve. These goals will determine where thought, energy, and time are spent. They will determine what gets put in and what is left out—at least until the final stages of product development. In developing Zoomracks our fundamental goals were:

1. We wanted to present information on a computer screen effectively, compactly, and in a way that was independent of screen size[2]. Although Zoomracks is being introduced on the IBM PC, it was designed with the lap and hand-held computers of the future in mind. As semiconductor technology advances, computers will get smaller and smaller. By 1990, a circuit board the size of a credit card will hold 4 megabytes of RAM[3]. The microscopic size of electronic components will allow the manufacture of computers that are much smaller than those available today. Yet, the lap and hand-held computers will become practical and popular only if software is available that makes effective use of smaller screens. Programs that can operate only with larger and more expensive displays will naturally become less attractive. Of course, software that can utilize smaller displays must work at least as effectively with standard size computer screens.

The Japanese are experts at making things small. Their approach is to distill out only the essence and to eliminate what is not essential[4]. We tried to do the same thing in developing our software.
2. We wanted to provide an organization mechanism that was general enough to handle many kinds of information but was also simple and comprehensible to the user.

When there are pocket computers that have the power of a desktop computer, what will you want them to do? We think you will want them to keep track of the personal information that you want with you at all times: names and addresses, appointments, things to do, notes, memos, and the records of business expenses and automobile mileage that the IRS demands. Zoomracks was designed to organize and manage this kind of information. It was designed to be a general purpose tool as opposed to a specific application.
3. We wanted to give users a variety of ways to view their information.
4. We wanted Zoomracks to be easy to use.

. . . the idea itself probably is the most important element of the entire illustration. Certainly, if the idea is not good and if it does not interest and intrigue people, any other good qualities which the picture may possess will be lost because they will not be seen. It is an utter waste of effort to paint a beautiful, story-telling picture unless it is based on a good central idea—one which can be readily understood.
Norman Rockwell

It may be a good thing to copy reality; but to invent reality is much, much better.
Giuseppe Verdi.

VisiCalc demonstrates how a good unifying concept works. At the point of sale, the question, "Can I do X with VisiCalc?" is transformed into "Could you do X on a manual spreadsheet?" In the office, "How would I do X with VisiCalc?" is turned into "How would you do X on a manual spreadsheet?" The user can do a lot just on the basis of information he already has.

Our unifying concept is a rack similar to magazine racks or the racks that hang next to factory time clocks. We started with racks because it was a simple, familiar concept that we felt could be developed both visually and in terms of complexity. First, we will describe a single rack, but imagine it as one of several side by side. Later we will describe how Smart Zooms display several racks on the screen at once.

Every single object shown in a picture should contribute directly to the central theme. All other things should be ruthlessly discarded.

Norman Rockwell

Consider a time-card rack. The first line of each card is always visible. Any card can be removed and the details examined. Cards can be inserted, removed, and moved into other slots in the same or different racks. Typically, cards in a rack have the same form (time cards), but are different in content (people and hours worked). They are probably arranged in some order (by employee number or name). Several time-card racks might be next to each other. These are the essential features of time-card racks. This is what people know and expect. This is the idea that serves as the unifying concept for Zoomracks.

Once we have selected the metaphor, we can begin to extend and transform our computerized racks to eliminate the limits that physical racks have. First, we can make racks arbitrarily long. A Zoomrack can hold 2 cards or 2,000, growing or shrinking to meet the number of cards in it. Second, a rack can automatically keep cards in order by the first field. People expect cards and lists to be ordered by the first field, as in a library catalog or a phone book. In a physical rack, inserting a new card in proper order requires moving all the cards below it to make room. A card inserted in a Zoomrack automatically goes to its alphabetic place. Third, although all the cards in a physical time-card rack are the same size, the cards in a Zoomrack can be of different lengths. At one extreme, a Zoomrack can hold a set of cards that contain only one brief item on the first line. If you remove the card, you see that the rest of it is empty. In this instance, the set of cards serves as a list of brief items of information. At the other extreme, a Zoomrack can be a file system. In this case, the first line contains the file label; when you remove the card, you find that the rest of it is filled by a lengthy document. Thus, you can use racks for lists, card files or for document files as you wish.

Exaggerate the essential and leave the obvious vague.

Vincent van Gogh

By striking one key, you can toggle between multicard mode, where you get a display of the first line of several cards in the Zoomrack, and single-card mode, where you view the details of one card. This toggling becomes a reflex action, like switching a light on and off. By making it easy to flip cards in and out we hoped to make Zoomracks more like a physical rack.

Acting is a great profession as long as no one catches you at it.

Spencer Tracy to Burt Reynolds

One of our design objectives was to modify the perceptions of the user. Ordinarily, he perceives himself as giving instructions to a computer; we wanted him to perceive himself as manipulating real, familiar objects. If you see someone on television, that person seems real. From the evidence of your eyes you believe in his existence as much as you believe in the existence of the person sitting next to you in the same room. A good writer makes his characters seem real. You can probably conjure up a mental

You start with an idea or vision. From there you proceed to the techniques and technology required to bring that vision to life. A painter's first concern is not with his paints and canvas, but with the theme and mood of his picture. He imagines something in his own mind that he wants to convey to the minds of his audience. The tools of his profession are merely means (often uncooperative) to that end.

The programmer, like the painter, is a communicator. The same principles govern his creative efforts; his activities are arranged in the same hierarchy. The most important task for him is to choose a unifying idea. This unifying idea is like the spine of a book that holds all the pages together. It is only after this idea has been fixed that the programmer can think about techniques of presentation or technologies for communication. Our first concern, then, will be the unifying idea.

When an actor studies a play, he looks for a unifying concept around which he can build his entire portrayal. A program, too, must have a simple, communicable concept that holds everything together. The concept behind a spreadsheet program is the array of cells, each of which contains a number, an equation or text. Everything else is detail—important detail, but detail nonetheless. A good concept is not merely a reproduction of reality (like an image in a mirror), but creates a new reality. It might take time to attract an audience, but a good unifying concept will find resonance among users. VisiCalc, and more recently ThinkTank, opened up important new markets because they embodied such concepts.

The unifying concept becomes the basis for communication with the audience. Once chosen, it defines an easily intelligible framework into which everything else fits. It is the metaphor that strikes a familiar chord in the audience and allows them to talk about your product in familiar terms: "It's just like a spreadsheet." It generates word-of-mouth reports, creates new markets, and brings people in to purchase your product.

picture of Ebenezer Scrooge or Huckleberry Finn without much effort. They have been transported from the imagination of Charles Dickens and Mark Twain to your imagination. They may be fictional characters, but they are real to you.

Just as the job of a communicator is to create a reality in the mind of his audience, the job of the software designer is to create a reality in the mind of the user. When an author creates the "willing suspension of disbelief," the reader becomes oblivious to the fact he is reading a book. The software designer helps to suspend disbelief by minimizing the distractions caused by the mechanics of using the software.

One way to lessen distractions and make racks more real is to ensure that commands work the same in both multicard and single-card display modes (and, as we'll see later on, in multirack and single-rack modes). For example, the command for going to the next card in the rack—and thus the mental process required to perform this task—is always the same. When the form of a command depends on what mode you are in, you have to focus on the mechanics of the operation. This either actually disrupts your perception that the racks are real, or at least makes them psychologically more distant.

Each rack has a format that specifies its fields of information (e.g., name, phone, city). All the cards in one Zoomrack are of the same format, but different racks can have different formats. One Zoomrack could be used for names and addresses, another for appointments (sorted by date and time), a third for notes, a fourth for memos, and so on.

You can specify a format for any Zoomrack. More importantly, you can easily change it after the rack is already loaded with cards. This has two advantages.

1. People don't want to have to get it right the first time. You want to be able to try something, and when you see a way to do it better, change it. Adding, deleting, or moving fields should be easy.
2. People might want to concentrate information of special interest on the first line of a card where it can be viewed in multicard display mode. For example, if you have a card with the name and telephone number of an individual on the first line, you might want to move the name of the person's company or the date of an important meeting with him onto the same line.

One command toggles the display of labels on and off. This helps you use screen space efficiently because labels are left off most of the time. You only need labels when you are entering information or are first using Zoomracks. Usually labels are background noise: you are interested in your information, not the name of your information. Again, we are trying to provide the ability to view the essential.

Seeing is forgetting the name of the thing you see.

Paul Valery

Zoomracks consists of several racks. Each rack contains an unlimited number of cards. Each card has fields and a format that specifies the number of fields and how they are to be displayed. Each field has unlimited lines. Everything else is detail. (Our current implementation limits you to 10 Zoomracks, 29 fields per Quickcard, 80 characters per line, and the amount of RAM memory.)

Within this framework, Zoomracks offers three different capabilities: you can have short fields like those found in databases, text fields for multipage notes, and columns of information for forms such as sales orders or spreadsheets. The obvious way to do this is to have three types of field for columns, documents, and database entries and a bunch of rules for the user to follow. Unfortunately, this doesn't allow for easy use. While seeking a solution to this problem, we happened upon the Fieldscroll concept.

Each Quickcard is made up of Fieldscrolls that contain the text of the fields. To use a chemical analogy, if the Fieldscroll is an element, a Quickcard is a molecule, and a rack is a set of identical molecules. The format command lets you change the chemical structure.

The format command lets you specify how your Fieldscrolls are arranged, or, more precisely, which line each Fieldscroll is hung from, and, in the case of multiple Fieldscrolls on one line, the position of each within the line. Additional lines in a Fieldscroll are displayed successively on blank lines underneath. Thus, a Fieldscroll can be used in any of the three different ways mentioned above:

1. Database Mode: You display one field (typically, a name or phone number) on one line. Here the Fieldscroll consists of the single field on the single line.
2. Document or Text Mode: You display a document across the entire width of the display on consecutive lines. Here the Fieldscroll consists of the first line and the several lines for text underneath it.
3. Column Mode: You display several narrow fields on the first line. Here each narrow field is a Fieldscroll. Now, if you hang blank lines from each field, these lines end up being the same width and you have transformed your set of narrow fields into columns.

You determine the meaning of your Fieldscrolls on a card. The format specification (where the name of the field and its location in the display are stored) is merely an aid to your understanding. Inconsistencies between cards and format (e.g., wrong field type, line too long) cannot occur.

Fieldscrolls were developed accidently. Zoomracks was evolving from an electronic Rolodex product into a multirack database product. We decided to add word processing. Because our database supported fields with subfields (or lines), we decided to treat the fields as text scrolls and the subfields as lines in the scroll. We had a move command in our electronic Rolodex that moved fields around to change the format. The quick and dirty (and, we thought, temporary) solution was to display additional

lines of a Fieldscroll on succeeding blank lines that did not have Fieldscrolls on them. We thought that after we had used the product for a while we would get a better feeling for the problems involved and design a much better mechanism for specifying complex formats.

We ended up keeping the original solution (although we did polish it up a bit). The basic interface is simple. All the user really has to do to format a rack is to move Fieldscrolls around. If you put several on a line, they become narrow columns; if you put only one on a line, it becomes wide. You can insert blank lines to make them as long or as short as you want. The display is updated immediately.

Most importantly, virtually all requests that a user can make are legal and are transformed without loss of data into visible results that can be examined and modified. Any format can display any information because there are no limitations on type or size. As a result, several sources of error are eliminated. Specifically:

1. Cutting a card from one rack and pasting it into another is always a visible operation with no source of error even though the formats of the racks might be radically different.
2. Practically (but not strictly) speaking, there is no such thing as an improperly formatted external ASCII file.
3. The only important format operation is Move (move this Fieldscroll to there). It is always allowed and has no special cases. For example, a text field can be on a line with other fields. Other less important operations are: insert Fieldscroll, cut Fieldscroll, split line, join line, and edit Fieldscroll label.

When you use Fieldscrolls, a single phone number might appear in a several line text field, or the text of a memo might be crammed into a phone number field, but these are not errors. No information is lost. The user can see the results and fix his problem just by changing his format. People find error messages annoying. They puncture the illusion of reality. We have attempted to make Zoomracks as flexible as possible to reduce the number of error messages with which the user must deal.

Of Mice and Menus

At the bottom of the hierarchy of things with which the designer of an effective user-interface must be concerned are what I call technologies for communication. These are the hardware and software technologies that help you communicate with a computer on a technical level: mice, voice recognition, menus, icons, high-resolution graphics, color, and so on. These technologies can help to improve the presentation of an idea, but can never transform a bad idea into a good one or a dull presentation into an interesting one. Good ideas that are well-presented are much more important than flashy technologies.

New technologies are merely tools that a software designer must manipulate in his attempt to communicate with the user a little better. They are never magic panaceas that eliminate the need to solve fundamental communication problems. Technologies can be seductive—they can distract the designer from his real concerns. Making a color film is easy. Making a *good* film—whether it be black and white or color—is hard. The same is true of software products.

Presentation and Staging

The task I am trying to achieve above all else, is to make you see.

D.W. Griffith

We have seen that the most important element in a good user-interface is the unifying idea and that the least important is the technology used for communication. In between these two in degree of importance is the technique used for presentation. In theater this is staging: how the scenery is arranged, where the players stand, and how they face each other and the audience. It is central to how an audience sees a play. Before the time of D.W. Griffith, people made movies by showing one continuous long-range shot of the action. To introduce a variety of perspectives into his presentations and to get us to see, Griffith invented the language of movies that we know today. He added close-up, medium-range, and panorama shots and edited them into a montage that controlled what his audience saw.

We need to find ways to add a variety of perspectives to our software products, just as Griffith did with movies. Currently, the main presentation mechanism is windows. Though many graphic systems give you the capability to zoom in and out, windows in text-oriented systems generally have the same unchanging depth that movies had before Griffith. If early movies provided the unvarying remoteness of the long shot, windows generally provide the unvarying intimacy of the close-up.

I want to reach that state of condensation of sensations that constitutes a picture.

Henri Matisse

Zoomracks lets you have several racks on the screen at once. If you have a rack of appointments, another of names and addresses, and a third of notes, you might want to display them all at the same time. If windows are used to display files, you *do* see detail. However, when a window gets smaller or is hidden by other windows, you lose the overall view. Smart Zooms allow you either to obtain a panoramic view of several racks of information or to zoom in on any one of them to see its details.

Zoomracks lets you have up to ten racks on the screen at once, each of which is displayed with a Smart Zoom. If there are eight racks on the screen, then each of them is compressed to ⅛ of the screen width. Essential information is distilled from the cards on a rack and displayed. Vowels are deleted and words and fields are truncated to fit. You will find this works quite well when three or four racks are shown in 80 columns; it is useful even when all ten racks are on. You get a visual overview of the information in each rack and your imagination fills in missing details. The characters that are present suggest the full text—especially if you are already familiar with it.

The interface with the Zoomracks

display consists of just three commands:

1. Typing any digit from 0 to 9 will turn on the corresponding rack (add it to the working set of racks displayed in multiple rack mode) and cause that rack to become the current rack (to be highlighted and be the object of any subsequent commands).
2. The Zoom command toggles between a panoramic display of all the racks that are turned on and a screen-sized display of the current rack.
3. One last command turns off the current rack (removes it from the working set of racks).

Just as all the commands work the same in single-card and multicard display mode, so all commands work identically in single-rack and multi-rack mode.

Zoomracks does not have a work line at the bottom of the screen. Rather, it positions the work line over the current field. This is likely to be controversial, because many people are accustomed to command lines. We feel, however, that most users focus their attention on the field where they are working and that something that pops out of a field seems more natural than something on a line at the bottom of the screen.

Toggles and Inverses

The user-interface for Zoomracks is table driven: for every key input there is a corresponding name in the command menu and a subroutine that is invoked. Reconfiguring this command network is a simple matter of restructuring the table. Consequently, during development we were able to make major changes in the user-interface without modifying the program proper. This enabled us to postpone the creation of a final user-interface until all the program functions were fully operational. During this time we used Zoomracks and carefully observed what we liked and did not like.

Quite late in the development we decided that the first target computer would be the IBM PC and that function keys should therefore be central to the user-interface. Early in development we had two toggle commands, Zoom Rack and Zoom Quickcard, that we liked very much. We assigned these two functions to F1 and F2. We assigned several other functions to the other function keys. At this time we wondered whether it would be possible to make all the function keys toggle. The answer was yes. As a result of making all the function keys toggle we can tell the user, "You can hit any function key at any time and it will do something visible someplace on the screen. If you do not like what it does, press it again and it will undo what it did."

Part of the reason for doing this was that we realized that people often lose function key templates. We thought it would be nice to let the user hit any key until he found the right one.

In an early implementation of Zoomracks we had an exit command. When we looked at it closely, we found it was really three different commands: accept input (as in carriage return), return up the menu tree, and exit from a major command mode such as Edit or Format. It seemed that exit was a command concerned with the mechanics of doing things rather than with what was being done. We didn't like this. Therefore, we eliminated all the exits, largely by turning them into toggles. Both Format and Edit Format became modes that could be toggled on and off. Command menus such as Zoom or Modify became toggles that turned on a series of options.

As we were discovering the value of toggles, we realized that they were just a special type of inverse command. Almost everyone is familiar with the undo command, which cancels the effect of the last command (or in a few implementations the last several commands). We felt that every command could have an inverse or reversing command. For example, the command TAB=next Fieldscroll had its inverse BACKTAB=previous Fieldscroll. We placed inverse commands next to each other on the menu and used small symbols to identify them as pairs.

What is important is the feeling that users get when using toggles and inverses. Because things are reversible, they aren't worried about making a mistake. Just as important is the fact that toggles make them more adventurous; they know if something does not work, they can get out of it.

From our experience with Zoomracks in Alpha and Beta test we have learned the following:

1. Zoomracks is good at presenting information on the screen in a distilled form—particularly when you are dealing with several racks of information.
2. The commands for toggling between single-card and multicard modes and single-rack and multi-rack modes allow you to view your information in different ways.
3. Zoomracks provides a useful organizational metaphor[5]. It is a two-dimensional file system that can be extended to store macros, spreadsheets, communication cards, electronic mail files, and several other kinds of information. A Zoomrack can even be treated as a relational database table. Of these potential capabilities, only macros are currently implemented in Zoomracks.

   Zoomracks lets you define a rack of macros. The first character of the first field in each card is used for the macro name; another field contains its definition. Because a macro is a Quickcard in a rack, the user knows how to view, copy, delete, edit, and comment macros. He only has to learn the commands to execute and define them. Macros enhance the basic power of Zoomracks—the ability to work with several racks at once. For example, a macro can find and display all your appointments, to-do items, and notes.
4. Users of Zoomracks find that they end up doing many operations in multirack or multicard mode. Operations such as inserting a card, editing, or changing a format are done without the mental effort of going to single-rack or single-card mode.
5. As lap and hand-held computers become smaller and more powerful, we expect Smart Zooms to become particularly useful because they allow computers to make efficient use of smaller displays and the same user

interface to be used on desktop computers as well.

Conclusion

The development of Zoomracks was not as linear and smooth as this article might suggest. Things that are clear and obvious now were fuzzy in the early stages of design. The thought processes described here were more important in recognizing, selecting, and polishing ideas, than in generating them in the first place.

I always have two things in my head—I always have a theme and the form. The form looks for the theme, theme looks for the form, and when they come together, you're able to write.

W. H. Auden

If you look at many of the early movies you will notice that the actors use exaggerated gestures. They seem strange and create a distance between us and what happens on the screen. We also see this when techniques appropriate for stage—where the actors must be at a distance from their audience—are used in film. Griffith realized that film was a more intimate medium and set about developing the techniques that are appropriate to it.

A similar thing happened in popular music when Bing Crosby recognized that the microphone opened up the possibility for more intimate communication with an audience. The techniques of the theater, where songs were belted out to large audiences, were inappropriate for radio and the phonograph. Crosby developed new techniques to go with the new medium. As computers get smaller and are used in a more personal way by more people, we have to develop techniques of making software that can reduce the psychological distance between computers and their users.

Zoomracks, Quickcards, Fieldscrolls, and Smart Zooms are trademarks of Quickview Systems. Rolodex is a trademark of Rolodex Corporation.

Notes

[1] Heckel, Paul, *The Elements of Friendly Software Design,* Warner Books, 1984.

[2] Tufte, *The Visual Display of Quantitative Information,* Graphics Press, 1983.

[3] Finke, Douglas, "Major Trends in Memory", *Transportable and Battery-power Personal Computers Proceedings, Future Computing,* 1984.

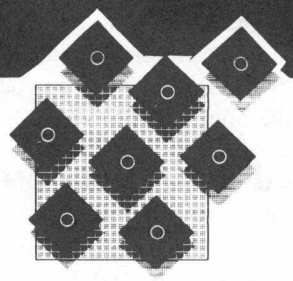
[4] O-Young Lee, *Smaller is Better: Japan's Mastery of the Miniature,* Kodansha Press.

[5] The idea of the *organizational metaphor* is well discussed by Chuck Clanton in "The Future of Metaphor in Man-Computer Systems" in the December, 1983 issue of *Byte*, pp. 263–280.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 193.**

Programming Editors, Programmable Editors



As a programmer, you spend many hours with your editor. If it is a good one and well-suited to your style of editing, it can greatly speed your production of code. If it is a bad one, or forces you to adopt an alien style, it can impede your work, spoil your outlook, and raise your blood pressure.

Beyond the basic functions, it is a matter of opinion what features an editor should offer. To choose an editor that is right for you, you need to know both what the different editors offer and what you yourself want your editor to do. In this review I describe some of the features, strengths and weaknesses of ten editors for the IBM PC and PC clones. It is up to you to select the editor with the style, performance, and price that meet your needs.

Features

What features might be of importance to the programmer? The tables that accompany this review list and evaluate many of the specifications and features of the various editors. They are by no means comprehensive, but they do cover some important aspects:

Size (see Table 1, page 62)

If you are working on a machine with limited memory, such as the PCjr, you need an editor that fits. Similarly, if you use floppy diskettes, you probably want your editor to be compact, so that it can be placed on the same diskette as your compiler and linker. If you work with both CP/M-80 and MSDOS, you may prefer an editor available for both eight-bit and sixteen-bit machines. If you work with large files, you may wish to choose an editor with virtual memory.

Mark U. Edwards, Department of History, University Hall, Purdue University, West Lafayette, Indiana 47907.

Documentation and Help (see Table 2, page 62)

You may not care whether an editor comes with a tutorial. On the other hand, a manual that does not have a good index or fails to group topics conveniently will cause you to lose a significant amount of time when searching for a particular piece of information.

Editing Commands (see Table 3, page 62)

With some editors you perform most fundamental tasks while in edit mode: you move your cursor around the screen, insert, delete, and so on, with sequences of control, alt, and function keys. Other editors add a command mode. This is used to execute more complicated commands and even macro language programs. Of the other editing functions noted in the table, the ability to reconfigure the keyboard is particularly important. You may not like the way the editing commands are assigned to keys. If the keyboard can be reconfigured, you can change the assignments to reflect, say, your preference for WordStar's keyboard assignments. If the keyboard cannot be reconfigured, either you must employ a keyboard enhancer, like SuperKey or ProKey, (assuming that the enhancer is compatible with your editor and does not steal space you need for files) or you must adapt to the configuration of the editor.

Search and Replace (see Table 4, page 66)

Here there are many choices. Do you need an editor that can search from the cursor backward toward the beginning as well as forward toward the end of the file? Is the ability to search for a string as you type it, one character at a time, an essential feature, or just a nice extra? Wild cards in a search allow you to locate general textual *patterns* as well as specific strings. Regular expressions, familiar to users of the Unix utility grep[1], allow the user to specify practically any pattern, including alternate patterns. You can, for example, search for all occurrences of "int" or "long int" in one search. Furthermore, if the editor supports regular expressions in search-and-replace operations, you have a powerful translating tool. This enables you, for example, to replace all Pascal if-then statements with the equivalent C construct. In Table 16 (page 74) you can compare the various wild cards or regular expressions offered by the different editors.

File and Window Management (see Table 5, page 66)

Most, but not all of these editors, allow you to edit files larger than RAM. Some do this automatically, through some virtual memory scheme. Others require (or, if you see this as an advantage, allow) you to page a large file manually. It can be extremely helpful to be able to edit one file while looking at, and excerpting pieces from, another. You must consider, however, how many files you need to be working on, and looking at, simultaneously.

Text Formatting Commands (see Table 6, page 66)

For the most part, a programmer's editor need not be a full-featured word processor. Even so, some text formatting capability can be useful on occasion, for example, for adding

comments to code. One formatting command that is especially valuable for producing well-structured code is the ability automatically or manually to vary the line indentation.

Printing (see Table 7, page 66)
Do you need the ability to print while editing? Most editors allow you to print all or part of the active file. Some can print one file while you edit another.

Undo (see Table 8, page 68)
Some editors allow you to undo deletions. Some also allow you to undo commands other than deletions. Obviously, this can be a useful feature. If you can undo everything you do, you are protected from most mistakes. Such a capability can, however, eat up memory and slow performance.

Keystroke Macros (see Table 9, page 68)
If you have ever used one of the keyboard enhancers, such as ProKey or SuperKey, you are familiar with keystroke macros: a series of keystrokes is assigned to a single key; then, when that key is pressed, the keyboard enhancer plays back the keystrokes as if you were typing them in yourself. Several of the editors have this or a similar capability built-in.

Macro Language (see Table 10, page 69)
Keystroke macros can take you only so far. To automate complicated editing tasks, such as complex translations or the automatic balancing of opening and closing parentheses in C or Pascal, you need a *macro language* that includes full conditional branching and the ability to set, manipulate, and evaluate variables. Actually, a macro language can be used to extend the capabilities of an editor in whatever direction you choose. In effect, with a good macro language you can create your own, customized editor. This does mean, however, that you have to master another programming language. The macro languages incorporated in the editors under review are based on one of two models: some resemble variants of Lisp and normally must be "compiled," or at least "tokenized," before use; others resemble Digital Equipment Corporation's TECO and are made up of short one or two letter commands that can be created and executed on the fly. In my opinion, the Lisp-like style is easier to learn and to use, at least for complex macros. Yet, programmers who are familiar with the terse commands of a TECO-like editor can make real gains in speed of creation, at least for simpler macros. To help you decide which kind of language is best-suited to your needs, I have provided sample programs using the different macro languages (see Listing, page 83).

Subprocesses (see Table 11, page 69)
MSDOS 2.0 and above supports an exec function, that is, it allows one program to execute another program, even command.com. An editor that takes advantage of this capability allows the user to run a compiler, linker or the MSDOS built-in commands, either from the editor itself or while the editor is still resident in memory. Of course, there must be enough memory left in your system for the second program.

Error Handling (see Table 12, page 69)
At the very least, you should expect your editor not to fail if you leave your drive door open. Also, if you attempt to quit the editor without saving your file, the editor should warn you that you will lose all your modifications if you proceed. This is error handling at a fairly elementary level, but there is some variation among the editors on this score.

Benchmarks (see Table 13, page 70)
Speed is not everything. Still, if you have to wait too long for routine editing tasks to be completed, you may become frustrated and more likely to commit errors. To test speed of execution I chose several tasks that a programmer might well undertake in a session. If your compiler says that there is an error on line 436 of your file, you want to get to that line as quickly as possible. So, using a dual drive Zenith 161 running MSDOS 2.1, I timed how long it took the editors to load a 28K text file and jump to line 436. I also counted the number of keystokes that were needed to accomplish this task. Then, I timed how long the editor took to write the file to disk with a backup. I also created a five hundred line file that simulated an assembly language program with comments on every line. Comments were preceded by text, spaces, and tabs in various combinations. To test an editor's search and replace command, I replaced the five hundred semi-colons with "REM." Next, after restoring the semi-colons, I had the program change all the assembly language comments

```
;<comment>
```

to C-style comments

```
/*<comment>*/
```

For some editors, this required writing a keystroke macro or a macro lan-

	BRIEF 1.3	EC 2.1	EDIX 4.02	EMACS 0.8	Epsilon 2.03	MIX 1.2	Pmate 3.37	VEDIT+ 2.01	XTC 3.0	XyWrite II plus
A. size of program	73K	60K	81K	237K	79K	47K	29K[3]	26K	100K	72K
B. "full system" size	170K	271K	106K	488K	118K	86K	29K[3]	51K	122K	106K
C. minimum memory	192K	128K	192K	384K	192K	128K	64K[3]	64K	192K	
E. DOS	2.0	2.0	2.0	2.0	2.0	2.0[4]	2.0[4]	2.0[4]	2.0	2.0
F. file size limit	disk	memory	250K	memory	disk	disk[1]	disk	disk	disk[1]	disk
G. maximum line length	512	1K	255	avail. memory	media limited	255	250	1000	255[2]	255

Notes:

[1] Must manually page larger files. Cannot page backwards.

[2] Set at startup.

[3] This is the basic model. In Version 4.0 there are three other models: Menu model is 33K, Fortran model 32K, and C model 55K. The C model requires at least 128K.

[4] Also has a CP/M version.

Table 1
Size

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. pages	223	36[1]	164	258	65	98	196	302	91	518
B. table of contents	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
C. index	fair[2]	fair[1]	good	Y	fair[2]	fair[2]	none	fair[3]	good	good
D. reference card	Y	N	N	N	N	N	N	N	N	N
E. on-line help	excellent	excellent	good	fair	fair	fair	fair	fair	fair	fair
F. tutorial	43 pp.	on-line	on-line	N	on-line	N	N	37 pp.	N	157 pp.

Notes:

[1] The current manual (8-1-85) is atrocious, but the on-line help and tutorial are so good that a manual is hardly necessary!

[2] Largely a list of commands with little or no cross references or topical entries.

[3] Good index for VEDIT manual, no index for VEDIT PLUS.

Table 2
Documentation and Help

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. modes	Edit	Edit	Edit	Edit	Edit	Cmmd. & Edit	Cmmd. & Edit	Cmmd. & Edit	Cmmd. & Edit	Cmmd. & Edit
B. cursor movement	full	full	full	full	full	full	full	full	full	full
C. deletion	full	full	full	full	full	full	full	full	partial[1]	full
D. block movement	full	full	partial[2]	full	full	full	full	full	partial[1]	full
E. reconfigure keyboard	Y	partial[3]	Y	Y	Y	Y	Y	Y	N	Y
F. extensible	8	4	5	8	5	5	7	6	7	7

Notes:

[1] Deletion and movement of blocks is done by whole lines.

[2] Block copies and block moves leave newline characters before and after the inserted block.

[3] Single control keys and twenty function keys can be assigned macros.

[4] Keyboard macros that can be assigned to a select number of keys.

[5] Keyboard macros that can be assigned to any key.

[6] Macro programs that can be executed from the command line.

[7] Macro programs that can be executed from the command line or assigned to a select number of keys.

[8] Keyboard and macro programs that can be assigned to any key or executed from the command line.

Table 3
Cursor Movement, Inserting, Deleting

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. search backwards	Y	N	[5]	Y	Y	N	Y	N	N	Y
B. incremental search	Y	N	N	Y	Y	N	N	N	Y	N
C. query replace	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
D. wild cards	Y	partial[1]	Y	Y	N	N	Y	Y	Y	Y
E. regular expressions	Y	N	Y	Y	N	N	N	partial[2]	N	N
F. wild cards in replacements	Y	partial[1]	partial[6]	Y	partial[3]	N	N	N	partial[4]	N
G. undo replacements?	Y	N		Y	N	N	N	N	N	N

Notes:

[1] Has one wild card to match (and replace) a single character.
[2] Limited alternate pattern matching (A or B or C etc.).
[3] Can do a recursive edit upon finding an entry.
[4] Can place macros in the replace text.
[5] Searches only forwards but automatically wraps around to beginning of buffer and continues back to where it started.
[6] Only one wild card that can substitute for everything found in the search pattern.

Table 4
Searching and Replacing

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. number of buffers or files	Disk Capacity	5	12	avail. memory	Disk Capacity	2	11	37	11	2
B. number of windows	52	5	4	8	11	2	1	1	8	2
C. file merging	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
D. virtual memory	Y	N	N	N	Y	N[1]	Y	Partial[2]	N[1]	Y
E. Access to DOS file management	Y	Y	Y	Y	Y	Y	N[3]	Y	Y	Y
F. optional backup	Y	Y	Y	Y	N	Y	Y	Y	N	Y

Notes:

[1] Can page a large file in and out manually.
[2] Can page a large file in and out automatically.
[3] Available in version 4.0.

Table 5
File and Window Management

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. tab setting	variable	fixed	variable	fixed	fixed	variable	variable	variable	fixed	variable
B. margin setting	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
C. centering	Y	Y	N	Y	Y	Y	N[1]	N	Y	Y
D. word wrap	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
E. indenting	auto	auto	auto	Y	manual	manual	manual	manual	manual	

Notes:

[1] Manual gives a macro that could be placed on one of the select macro keys.

Table 6
Text Formating Commands

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. print active file	Y	Y	Y	Y[1]	Y[1]	Y	Y	Y	Y	Y
B. print while editing	N	Y	N	N	N	N	N	N	Y	Y

Notes:

[1] Must open a PRN file and write to it.

Table 7
Printing

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. undo deletions										
1. character	Y	N	N	Y	N	partial[2]	N	partial[2]	N	N
2. word	Y	partial[1]	N	Y	Y	partial[2]	Y	partial[2]	N	Y
3. line	Y	partial[1]	partial[3]	Y	Y	partial[2]	Y	partial[2]	Y	Y
4. block	Y	partial[1]	partial[3]	Y	Y	N	Y	Y	Y	Y
B. undo other commands	Y	N	N	Y	N	N	N	N	N	N

Notes:
[1] Only if using Alt-D command.
[2] Only if cursor has not been moved from the line.
[3] Only with certain commands.

Table 8
Undo

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. "on the fly"	Y[1]	Y[2]	N	Y	Y	Y	N	N	Y	N
B. save and restore	N	Y	Y	Y	Y	Y	N	N	Y	Y
C. pause for user	N	N	N	Y	Y	N	N	N	Y	Y
D. templates	Y	Y	Y	Y	Y	Y	N	N	Y	Y

Notes:
[1] Only one "on-the-fly" macro.
[2] For twenty function keys.

Table 9
Keystroke Macros (built-in keyboard enhancer)

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. model	Lisp	none	none	Lisp	none[1]	none	Teco	Teco	Teco	Teco[2]
B. full conditionals	Y			Y			Y	Y	Y	Y
C. "on the fly"	N[3]			N[3]			Y	Y	Y	Y
D. ease of use for simple macro	fair			fair			good	good	good	fair[4]
E. speed of creation for simple macro	fair			fair			good	good	good	fair[4]
F. ease of use for complex macro	excellent			excellent			fair	fair	poor[5]	poor[4]
G. speed of creation for complex macro	good			good			fair	fair	poor[4]	poor[4]

Notes:

[1] The Beta version I saw had a C-like macro language with full conditionals. I did not test it extensively.

[2] Not a complete macro language. Limited documentation and no telephone support.

[3] Can be compiled and loaded automatically, making them almost "on the fly."

[4] Some macros, especially complicated ones, are likely to be impossible. See note (1)

[5] Cannot use complex boolean expressions in a single branch test.

Table 10
Macro Language

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. invoke DOS	Y	Y	N[4]	Y	Y	Y	Y[3]	N	Y	Y
B. concurrency	N	N	N	N	Y	N	N	N	N[2]	N
C. run compiler	Y[1]	Y	N[4]	Y[1]	Y	Y	Y[3]	N	Y	Y
E. Specify program memory	Y	N	N	Y	N	N	N	N	N	N

Notes:

[1] Automatic compilation and flagging of errors, if any.

[2] Can do fore- and background processes within the editor.

[3] In Version 4.0.

[4] The accompanying *Professional Writer's Package* can be used to invoke EDIX, WORDIX, INDIX, and other programs.

Table 11
Subprocesses

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. disk error recovery	good	poor[1]	good	poor[2]	good	poor[2]	none[3]	poor[2]	poor[2]	fair[4]
B. Abandon edit confirm	Y	Y	Y	Y	Y	N	Y	Y	N	Y

Notes:

[1] See capsule review.

[2] MSDOS does the error recovery. Ignore or Abort can lead to loss of edit.

[3] Hung the machine completely. This was fixed in Version 4, but MSDOS still did the recovery.

[4] Recovered OK, but said there was no such file on read.

Table 12
Error Handling

guage program; with others, a regular expression in a replace did the job. Next, after once more restoring the file to its original condition, I had the editor strip out all the comments along with any blanks or tabs preceding the comments. Finally, for those editors with a true macro language, I wrote a simple program to count opening and closing curly braces and tested the program on my IBM PC/AT. This last benchmark gives you a rough idea of both the speed (on an AT) and the intelligibility of the different macro languages (see Listing, page 83).

Miscellaneous Features (see Table 14, page 72)

I also tested whether the editor worked with Borland's SideKick and SuperKey. Here I assumed that the results could be generalized to other memory resident productivity tools. I have also noted whether IBM's extended ASCII character set can be accessed and displayed. Finally, since several editors include a built-in calculator, I have added that to my list.

Overall Evaluation (see Table 15, page 72)

The last table consists of my own subjective evaluation of these various specifications and functions. I explain some of my judgments in the capsule reviews of each editor (see below). One might expect that the price of an editor (both list and after discount, if it can be purchased at discount) should bear some relation to its features, but that is not always the case.

Editor-specific Comments

In each of the following capsule reviews (see Table 17, page 83, for specific product information) I focus on the particular strengths and weaknesses of a given editor along with selected special features. Together with the tables and the listing, these capsule reviews should give you a sense of what an editor is like.

One note about the versions reviewed. The tables display the specifications, features and performance of the *production* version of the editor that was available on August 1, 1985. Several of the editors had significant enhancements in the works, and I was able to see Beta test versions of the enhanced editors. In such cases I have added a paragraph on the enhancements to my capsule review. The tables, however, show the figures for the production version rather than the Beta version.

BRIEF (Solution Systems)

BRIEF's built-in commands are easy to use and to learn. Still, if you do not like a particular key assignment or the effect of a command, you can change it. The keyboard is fully reconfigurable and any macro program written in the powerful Lisp-like macro language can be assigned to

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. loading itself and test file and move to line 436 (seconds)	29	23	32	83	29	44[1]	16	28	86	35[10]
(keystrokes)	5	5	5	5	5	8	6	8	9	7
B. writing test file and backup (seconds)	30	[2]	18	24	9	[1]	23	30	23[3]	7
C. simple search and replace (seconds)	28	7	107[4]	16	23	49	38	17	108[4]	18[5]
D. change ASM comments to C (seconds)	47	244[7]	689[9]	36	53	341[8]	65	125	192[4]	328[4]
E. stripping comments (seconds)	30	—	355[4]	22	—	—	234	5	25 min.[4]	[6]
F. Scroll back 5 screens (seconds)	< 3	< 2	< 4	< 7	< 3	< 4	< 2	< 2	< 2	< 3
G. Brace count macro program (min:sec)	0:06	—	—	0:11	—	—	4:16	5:36	54:00	

Notes:

[1] MIX loaded only about 15K of test files.
[2] An earlier version of EC did this in 8 seconds; the current version was unable to do it at all. See capsule review.
[3] Does not do an automatic backup.
[4] Updates screen after every operation.
[5] Using "Change Invisible" command. With regular replace, which updates the screen, it took 125 seconds.
[6] I could not create a program that stripped comments from <space><tab><comment> lines.
[7] Done by holding the function key down until the keystroke buffer was filled. Could not search for either tab or space before the semicolon.
[8] Overflowed the keyboard buffer and put in extra, unwanted comment delimiters.
[9] This is the time for a "fully automated" keyboard macro. Using a more simple macro and repeatedly holding the envoking key down until the keyboard buffer was full. I did the conversion in 322 seconds. Updates the screen after every operation.
[10] Must specify page and line number on the page.

Table 13
Benchmarks

any key or invoked by name.

BRIEF has an outstanding undo facility. The default configuration allows the last 30 editing commands to be undone. This number can be raised to a maximum of 300 commands. Until you reach this maximum or run out of RAM, every command you issue can be undone. So if you make ten changes and then realize that the first one was an error, you can undo all the changes back to the mistake. This ability probably slowed BRIEF down on the benchmarks, but it meant that I was able to undo the results of each test with one keystroke! Needless to emphasize, this facility can save endless grief.

BRIEF has powerful search and translate commands. These use regular expressions and function much like the Unix grep utility (see Table 16 for a list of the expressions that are supported). For example, the string

```
m?l¦de
```

would match "male" or "made" or "mile" or "mode," and so on. The power of these operations is most apparent when you perform translation tasks. When you use regular expressions it is easy, for example, to change every Pascal if-then statement into its C equivalent. Unfortunately, BRIEF cannot search for strings that extend over several lines.

As with several of the other editors, you can, given sufficient memory, create from within BRIEF a subprocess that runs DOS. Within this subprocess any valid DOS command can be executed and other programs can be run. BRIEF comes with several macro packages that use this ability to compile source files. With one keystroke you can tell BRIEF to write the

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. compatibility with Sidekick and SuperKey	Y[1]	Y	Y	Y	N[2]	Y	Y	Y	Y	N[3]
B. IBM extended ASCII supported	Y	Y	Y	N	Y	N	partial[2]	Y	Y	Y
C. Built-in calculator	N	Y[4]	N	N	N	N	N	Y[5]	N	Y[6]

Notes:
[1] Must use -p flag on startup.
[2] Control-Alt, used to invoke Sidekick, is a prefix for some Epsilon commands. Does not work with SuperKey.
[3] Works with Sidekick, but not with SuperKey.
[4] Has logical, arithmetic, and shift functions as well as ASCII conversion.
[5] Has logical and arithmetic functions.
[6] Has four arithmetic functions.

Table 14
Other Features

	BRIEF	EC	EDIX	EMACS	Epsilon	MIX	Pmate	VEDIT+	XTC	XyWrite
A. price	$195.00	$49.50	$195.00	$495.00	$225.00	$29.95	$225.00	$225.00	$99.00	$295.00
B. discounted?	Y	N	Y	N	N	N	Y	Y	N	Y
C. ease of use	good	good	good	fair	good	fair	fair	good	good	good
D. documentation	good	good[1]	good	fair	fair	fair	poor	good	good	fair[2]
E. editing power	excellent	good	good	excellent	excellent	fair	excellent	excellent	good	excellent
F. formatting power	good	good	fair[5]	good	good	good	good	good	good	excellent
G. translating power	excellent	fair	good	excellent	fair	fair	good	good	good	good
H. file handling	good	good	good	fair	good	fair	good	good	fair	good
I. error handling and undoing	excellent	poor	good	good[3]	good	poor	poor	fair	fair	good
J. macro power	excellent	fair	fair	excellent	good	fair	good	good	good	fair
K. subprocesses	good	good	none	good	excellent	none	none	none	good	good
L. speed	good	good	fair	fair[4]	good	fair	good	good	poor[6]	good

Notes:
[1] On-line documentation is excellent, current written documentation inadequate.
[2] Documentation is good except for the macro language.
[3] Error handling is relatively poor, but the undo facility is excellent.
[4] On memory bound applications, quite good. But accesses the disk frequently to load various macros.
[5] With WORDIX combination, it is, of course, excellent.
[6] Its macro language is so slow as to be unusable for even simple tasks.

Table 15
Summary

source code file to disk, to invoke the compiler, and, if any errors are generated, to return to the source file and place the cursor on the offending line of code! This same procedure is used to "tokenize" programs written in BRIEF's Lisp-like macro language and load them into BRIEF.

BRIEF comes with a library of macros, including a brace checker, routines for automatic indentation of C code, incremental search, and keystroke to ASCII translation. I have used BRIEF's macro language to write a programmer's calculator, C language templates and editing routines, disk utility routines, and a program to translate one formatting language into another. I found the language easy to use and more than adequate for even complex editing needs.

With its macro and help libraries and its macro language compiler, BRIEF takes up half a double-sided IBM diskette. I have used it extensively on my two-drive Zenith without any difficulty, but it is better suited to a hard-disk system.

EC (C Source)

Except for a macro language and virtual memory buffering of files, EC offers most of the functions a programmer needs at a very affordable price. If you have sufficient RAM, you can work on five files at once, moving quickly and easily between them. The keyboard is partially reconfigurable, allowing you to create keystroke macros for twenty function key combinations and the twenty-six alphabetic control keys. There is a nicely implemented interface to DOS that enables you (once again if you have sufficient RAM) to run programs from within EC. The on-line help and tutorial (and associated menu system) were the best of the bunch.

In addition to the standard editing commands found on most editors, EC offers a number of unusual features. There is, for example, a List command that searches for a specified string and then presents you with a list of the lines containing the string together with the line number. Find, Replace, and List can be made to work over specified areas of marked text. There is, however, no backward Find or Replace and only one wild card, substituting for any single character, is supported.

EC implements a calculator that does arithmetic, logical, and shift operations and also translates characters into their ASCII codes. EC also has routines for comparing files and matching opening and closing braces, parentheses, and square brackets. If you wish, EC will automatically indent and cancel indent after opening and closing braces to facilitate the production of structured code.

EC offers no file buffering to disk, either manual or automatic, so you must have enough RAM for your files. This is even more important if you intend to run other programs from within EC.

Over the course of this review I was sent at least a half dozen updates. Each new revision had additional, useful features but also a complement of new bugs and quirks. The final review version was unable to do a backup and save on my test file or on my chart of features; I lost an hour's worth of work as a result. The authors of EC are available by phone, to the middle of another. You can, however, write a macro to eliminate this problem.

EDIX is easy to learn and to use. It comes with both on-line help and tutorial. It is not very fast—probably because it is constantly updating the screen. For example, in the search and replace command that I used to strip comments from my test file, every character deletion occurred one at a time and the screen was updated accordingly!

Emacs (UniPress Software)

The original Emacs was written by Richard Stallman at MIT. Over the years it has spawned a host of imitations. Two of the editors reviewed in this article, BRIEF and Epsilon, are spiritual kin. The Emacs distributed by UniPress Software is based on James Gosling's Unix version of Emacs. It is most faithful to the full glory (and complexity) of the original Emacs that I used for years on a DEC-20. It has the capacity to handle multiple files and windows, to run scores of macro programs, to define and redefine keys and macros on the fly, to transform itself into a specialized editor for specialized chores, in short, to do just about anything you might want an editor to do. In order to have this much power, however, you must pay a price: disk accesses are frequent, a massive amount of memory and disk capacity is required and the documentation is less than complete. You should also keep in mind that a couple of the editors under review here are almost as powerful as Emacs.

The original Emacs was an extension of the TECO editor and used an extended TECO as its macro language. UniPress' Emacs uses "mock lisp" (MLisp) instead. I personally find Mlisp much easier to use. Emacs comes with a rich collection of Mlisp programs, including such delights as "electric-c", which substantially automates the structuring of C code, "abbrev," which allows you to specify abbreviations for Emacs to expand automatically as you type, and "undo", which allows you to undo the last 100 commands or 1000 characters, whichever is reached first. The search and replace commands can use regular expressions that allow complex conditional searches and translations.

You need at least 384K to run Emacs, and it really isn't comfortable with less than 512K. The program, which weighs in at 237K before allocating any of its many buffers, is totally memory resident and has no virtual memory management. The files you edit must fit into the remaining memory. Emacs also frequently accesses the disk for various MLisp programs that it needs, so, although you can run it on a two-drive floppy disk system, I would not recommend it. It performs fairly well on my Enhanced PC-AT.

The documentation I received is incomplete. A number of the macro packages included with the editor come with little or no explanation beyond the comments in the code files. Because I had become familiar with Emacs on the DEC-20, I found that I could fill in many of the gaps, but I would not want to begin my exposure to Emacs with this manual. There was mention of the "Info" data base for

on-line help and a "Teach" tutorial, but neither was included. This Emacs is really closer to a pre-release or Beta and eager and willing to respond to suggestions and complaints, so I assume this problem will be quickly fixed. If they would settle down with a stable version and get all the bugs out, EC would be an excellent editor for the price.

EDIX (Emerging Technology)

EDIX is the editor in Emerging Technology's *Professional Writer's Package*. The other modules in this package are WORDIX, a text formatter, SPELLIX, a spelling checker, and INDIX, an automatic indexer. The separation of editor and formatter will be familiar to all who have used the Unix editor vi and text formatter nroff. As a text formatting system, the EDIX/WORDIX combination is powerful and flexible. It has served as the kernel of such sophisticated applications as AcademicFont, a multilingual text formatter that provides output in Greek, Russian and other European languages with non-Roman characters. As a stand-alone programmer's editor (available separately), EDIX offers most of the commands that a programmer would need, along with some extras such as regular expressions in searches. It does not support virtual memory, although it can use all 640K of RAM in your machine. It can handle twelve files simultaneously with four windows on the screen. You can change directories, see directory listings, and erase files from within the editor. You can execute DOS internal commands from within EDIX, but you cannot run subprocesses such as a compiler.

EDIX supports keyboard macros. When you define a keyboard macro in EDIX, you write it into a buffer using various mnemonics for the keystroke commands. For example, to assign to Alt-F1 a macro that places C-style comments around a single line and then moves down a line, you would place in an empty buffer:

```
f 104 @aly/*@alz */@lnd
```

and then invoke the config command. If you wished permanently to assign this keystroke macro to the key combination Alt-F1, you could add this line to your initialization file and EDIX would read it in automatically at startup. Using macros such as these, you can completely reconfigure the keyboard. EDIX will not, however, support assignments involving multiple keystrokes such as WordStar's ^K^M move-block command.

EDIX allows you to move or copy lines, partial lines, and columnar data. The latter feature is particularly nice. Unfortunately, if you wish to move or copy parts of lines, you must subsequently remove the newline characters that EDIX automatically places at the beginning and end of blocks. This may be acceptable when using WORDIX, which will rejustify text lines, but is an inconvenience in moving blocks of code such as parameter lists from the middle of one line

Command	Function
?	Matches any single character.
*	Matches any number of occurrences of any character.
@	Matches any number of occurrences of the preceding character or expression.
¦	Matches either the preceding or following character or expression (alternative).
{	Begins a group of expressions.
}	Ends a group of expressions.
\	Escape character; if the following character has a special meaning, it is treated as a normal character, and vice-versa.
\n	Matches a newline (carriage-return, line-feed) sequence.
\t	Matches a tab character.
\c	Places the cursor under the following matched character
< or %	Matches the beginning of a line.
> or $	Matches the end of a line.
\#	Substitute the actual text matched by the #th matched group of expressions (# is a digit from 0 to 9).
[. . .]	Matches any *one* of the characters between [and].
[~ . . .]	Matches if the character is not any one of the characters between [~ and].
a–b	Matches (or does not match) any character from the range a to b when within [. . .] or [~ . . .] (a and b can be any character).

Table 16a
BRIEF's Regular Expressions

Command	Function
?	Matches any single character.
*	Matches any number of occurrences of any character.
@	Matches any number of occurrences of the preceding character or expression.
Alt-N	Matches a newline (carriage-return, line-feed) sequence.
%	Matches the beginning of a line.
$	Matches the end of a line.
Alt-L	Substitute the actual text matched by the search pattern. This is the only special character that can be used in the replacement string.
[. . .]	Matches any *one* of the characters between [and].
[^. . .]	Matches if the character is not any one of the characters between [~ and].
a–b	Matches (or does not match) any character from the range a to b when within [. . .] or [^. . .] (a and b can be any character).

Table 16b
EDIX's Regular Expressions

version, which its version number (0.80) also suggests. UniPress also supplies a version of its Emacs for VAX/VMS and Unix systems.

Epsilon (Lugaru Software)

Epsilon closely resembles Emacs. The one major difference is that the current release does not have a macro language. It does, however, have a full-featured keystroke macro facility, which can be used to extend the editor in useful ways. The keyboard is completely reconfigurable, and custom-made keystroke macros can be assigned to any key or given a name. You can easily edit a number of large files simultaneously and have up to eleven windows open at once. Epsilon handles the virtual memory management automatically. It offers several types of search command, including incremental search both forward and backward. It does not, however, have wild cards. It handles editing functions with more speed than many other editors.

Epsilon is the only editor under review that allows you to continue editing while running a second program concurrently. Not all programs can be run in this way, but most compilers and linkers will work fine. There is some degradation in editing speed when the two programs are running simultaneously, especially if the background program is accessing the disk with any frequency. Nevertheless, serious editing can still be done. I did encounter one significant problem with the concurrent processing. In the current version there is no way to limit the amount of memory that Epsilon uses. Instead, Epsilon takes memory as it needs it and does not release it when it is no longer being used. So, if you edit a couple of large files and use, say, the help facility, you quickly run out of memory for your second process.

Emacs, and hence Epsilon, has an especially rich set of editing commands. In a few cases, however, I wish that the the folks at Lugaru had gone beyond the original Emacs. For example, when you mark a block of text in Emacs, the only way you can see exactly what you have marked is to switch the cursor and the invisible mark back and forth. Epsilon works the same way. Other editors use reverse video or some other visual means to identify the marked block. This is a minor point, but it illustrates Epsilon's close dependence upon Emacs.

I also have had the opportunity to see a version of Epsilon in Beta test that includes a C-like macro programming language and regular expressions in searches. I was told that the final version would have regular expressions in both search and search-and-replace operations. Using the new macro language, I wrote and timed a routine to count braces. It handled my test file in 10 seconds. Using regular expressions in a search-and-replace, it took 131 seconds to strip out the comments in my test file of 500 assembly language comments. You may compare these times with other entries in Table 13, remembering that the version I examined was in Beta test.

MIX (MIX Software)

MIX is the least expensive of the edi-

Command	Function
.	Matches any character except newline.
\w	Matches any word character as defined by the syntax table.
\W	Matches any non-word character.
\b	Matches at a boundary between a word and a non-word character.
\B	Matches anywhere but at a boundary between a word and a non-word character.
\!	Matches the regular expression preceding or following it.
\`	Matches at the beginning of the buffer.
\'	Matches at the end of the buffer.
\<	Matches anywhere before dot.
\>	Matches anywhere after dot.
\=	Matches at dot.
\(. . .\)	Matches what it brackets. Used with the next command.
\n	Matches a copy of the string that the bracketed regular expression beginning with the nth \(matched.
[. . .]	Matches any *one* of the characters between [and].
[^. . .]	Matches if the character is not any one of the characters between [^ and].
*	Matches zero or more matches of the regular expression that precedes it.
^	Matches the beginning of a line.
$	Matches the end of a line.
\n	In replacement, causes the string matched by the nth bracket expression to be inserted.
&	Causes the string matched by the entire search string to be inserted.

Table 16c
EMACS Regular Expressions

Command	Function
Control-E	Match any character
Control-L	Take the next character literally. This allows a wild card character to be searched for.
Control-N	Match anything but the following character.
Control-S	Matches either a space or a blank.
Control-W	Matches any word terminator, that is, any character other than a letter or a digit.

Table 16d
Pmate's Wild Card Characters

tors. Even so, it comes with an extensive range of features: you can split the screen and edit two files at once, fully reconfigure the keyboard, and run other programs as subprocesses from within the editor. Furthermore, although MIX does not provide a full macro language, you can still create relatively sophisticated keyboard macros using combinations of the many commands built into MIX.

Unfortunately, all this power is handicapped by the small work space. MIX requires you to page larger files through memory manually. Once the working buffer is full, you must write lines out before you can load additional lines in. Other editors under review also work this way, but MIX has a particularly small buffer space. It appears to be only about 15K in size. Most commands, such as the replace command, only work over the portion of the file in the buffer. To perform a replace or to run a macro over a large file, you must read in a portion of the file, execute the commands, write out that portion, read in the next one, execute the commands, and so on. To go back to a portion already written to disk, you must write out the whole file, close it, and then read it in again. This could be done with a macro; nevertheless, it is inconvenient.

It is fairly simple to create keyboard macros to perform editing chores. The macros can be assigned to a key or combination of keys or can be given a name and executed from the command line. You can also repeat a command or macro a specified number of times. I used these two features to change all the assembly language comments in my test file to C-style comments. While doing this, I encountered several difficulties. The repeat command apparently works by placing copies of the command keystroke(s) in the keyboard buffer. *This* works fine as long as the number of repetitions is small, but, when that number becomes large, you run out of memory. Moreover, once the repetition of a command begins, there appears to be no way to abort, either from within a keyboard macro or from the keyboard itself. In the assembly-to-C test, I exhausted memory twice and then ended up with a series of extra */ characters at the end of the file. MIX updated the screen after every operation, which added to the time the whole process took. MIX is also available for 8-bit CP/M computers.

Command	Function
¦A	Matches any alphabetic letter, upper or lower case.
¦B	Matches a blank or tab.
¦C	Matches any control character.
¦D	Matches any numeric digit (0 through 9).
¦F	Matches any alphanumeric (letter or digit).
¦L	Matches a line terminator (carriage-return linefeed, form feed, or end of file).
¦M	Matches any sequence of zero or more characters.
¦N	Matches any character except the following character or pattern.
¦Pr	Access contents of text register r as pattern set.
¦Rr	Access contents of text register r as search string.
¦S	Matches any separator (not a letter or digit).
¦T	Matches selected separators (terminators).
¦U	Matches any upper case letter.
¦V	Matches any lower case letter.
¦W	Matches a single or multiple spaces or tabs.
¦X	Matches any character.
¦Y	Matches multiple characters until next pattern matches.

Table 16e
VEDIT PLUS's Wild Card Characters

Command	Function
Control-A	Matches any alphabetic character.
Control-B	Matches a space (ASCII 32).
Control-C	Matches a capital letter.
Control-D	Matches a digit.
Control-L	Matches a lower case letter.
Control-N	Matches any alphanumeric character (letter or digit).
Control-W	Matches a space or a tab.

Table 16f
XTC's Wild Card Characters

Command	Function
Control-Alt-A	Matches any letter or digit.
Control-Alt-L	Matches any letter.
Control-Alt-N	Matches any digit.
Control-Alt-W	Matches any arbitrary string of characters up to 80 characters in length.
Control-Alt-X	Matches any character.

Table 16g
XyWrite's Wild Card Characters

Pmate (Phoenix Computer Products)

Pmate is both lean and fast. It is an assembly language program originally written for CP/M-80 and machines with no more than 64K. Configured for a thousand byte garbage stack and a five thousand byte permanent macro area, Pmate comes to about 29K of code. Obviously, it can easily fit inside even an anemic PCjr!

Pmate operates in two modes: an

editing and a command mode. Although the editing mode is full-featured, it is in command mode that Pmate shows its muscle. Any of the more than 120 commands available in Pmate's macro language can be entered singly or as a macro program at Pmate's command line and executed immediately. Commands can also be entered into one of 10 additional text buffers and Pmate can be directed to take its commands from there. These buffers can be saved to disk, restored, and even saved inside Pmate's permanent macro area.

Pmate has a useful set of wild cards for searches (See Table 16) and a LIFO stack that holds deletions. This stack can be popped to restore deleted items back into the text. Ten macros can be placed on the ten function keys.

Pmate's macro language is powerful, but cryptic. The language is designed to be used "as you go." There is, therefore, a premium on terseness. The shorter the commands, the more quickly you can type them—provided you remember them! Most of the commands in Pmate are either one or two letters long. With only two letters, it is difficult to come up with meaningful mnemonics and, as a result, the commands are hard to remember. Compounding the problem is Pmate's documentation, which is disorganized and lacking an index. Pmate's error recovery also leaves much to be desired: it hung my machine on the disk door open test!

The preceding comments apply to Version 3.37. I also had the opportunity to use a preliminary copy of Version 4.0. This new version comes in three configurations: a menu driven version, and versions specialized for C and Fortran. Included with the package were programs that allow multitasking (Pmate and one other program running simultaneously) and support for the use of a mouse. On-line help is also available. The extensions to Pmate are based on Pmate's macro language and demonstrate its power. The menu driven and mouse versions can shield the user somewhat from Pmate's rather arcane command language. The C language version adds a number of useful macros that can speed the generation and debugging of C code. I have not programmed in Fortran for more than a decade and do not have a Fortran compiler, so I did not try the Fortran version. The other versions worked as advertised. The supplied macro-language file for the C routines is uncommented, and demonstrates how cryptic Pmate's language can be. Documentation was scant and I did encounter a few bugs, but my overall impression was positive. Version 4.0 also partially corrected the bug in error trapping; it still let DOS handle the error and the abort option returned me to the operating system, but at least the machine did not hang.

VEDIT PLUS (Compuview)

VEDIT PLUS is an enhanced version of VEDIT, an assembly language program originally written for 8-bit CP/M machines. VEDIT PLUS adds to VEDIT expanded text registers, a complete programming language, and the ability to edit multiple files. In theory, you can edit up to thirty-seven files at once. Given VEDIT PLUS' memory limitations, however, this would not be practical, unless the files were very small.

VEDIT PLUS has both an edit and a command mode. The commands in

Brief, Version 1.3
Solution Systems
335-D Washington St.
Norwell, MA 02061
(617) 659-1571 or (800) 821-2492
Price: $195.00

EC Editor, Version 2.1
C Source
12801 Frost Rd.
Kansas City, MO 64138
(816) 353-8808
Price: $49.50

Edix, Version 4.02
Emerging Technology Consultants, Inc.
4760 Walnut Street
Boulder, CO 80301
(303) 447-9495
Price: $195.00

AcademicFont
University MicroComputers
665 Monte Rosa Dr. #915
Menlo Park, CA 94025
(415) 854-8845
Price: from $275.00

Emacs, Version 0.80
UniPress Software, Inc.
2025 Lincoln Highway
Edison, NJ 08817
(201) 985-8000 or (800) 222-0550
Price: $495.00

Epsilon, Version 2.03
Lugaru Software, Ltd.
P.O. Box 110037
5227 Fifth Ave., Ste. 12
Pittsburgh, PA 15232
(412) 621-5911
Price: $195.00

MIX Editor, Version 1.2
MIX Software
2116 E. Arapaho
Suite 363
Richardson, TX 75081
(214) 783-6001 or (800) 622-4070
Price: $29.95

Pmate, Version 3.37
Phoenix Computer Products Corp.
1420 Providence Highway, Ste. 115
Norwood, MA 02062
(617) 762-5030
Price: $225.00

Vedit Plus, Version 2.01 and 2.31
Compuview
1955 Pauline Blvd.
Ann Arbor, MI 48103
(313) 996-1299
Price: $225.00

XTC, Version 3.0F
Wendin, Inc.
Box 266
Cheney, WA 99004
(509) 235-8088
Price: $99.00

Xywrite II Plus, serial number above 1 million
Xyquest
P.O. Box 372
Bedford, MA 01730
(617) 275-4439
Price: $295.00

Table 17
Product Information

the edit mode can be assigned to whatever keys you wish, with one important limitation: VEDIT PLUS allows you to designate only two prefix keys. Many editors use prefix keys. For example, in the WordStar command ^Q^D, which means go to the end of the current line, ^Q is a prefix key. Another frequently used prefix key in WordStar is ^K. If you were to reconfigure your keyboard with VEDIT PLUS to resemble the WordStar keyboard, you could designate only two prefix keys, say, ^Q and ^K. In that case, you could not use the function and cursor keys because they also require a prefix key.

VEDIT PLUS provides a full TECO-like macro language. You can use this language to write editing routines either on the command line or in a text register from which the editor can be directed to take its commands. These macro programs can be edited, saved to disk, and restored for later use. In the current version of VEDIT PLUS these macros cannot, however, be assigned to keys. VEDIT PLUS also offers a particularly robust set of wild cards that approach the functionality of regular expressions, at least for searches (See Table 16). You can even search for alternate patterns; that is, search, say, for "int" or "char" at the same time. You cannot, however, search backward. When you combine the macro language with the wild cards, you can do sophisticated translations. To give you an idea of the language's potential, there is a separate product from CompuView, a macro program, that allows VEDIT PLUS to translate Z80 assembly code to 8086 code. With VEDIT PLUS itself comes a file comparison macro, a mailing list sorter, and an on-line calculator.

Like Pmate's macro language, VEDIT PLUS' language is terse and somewhat cryptic. As I mentioned before, it is impossible with only one or two letters to give meaningful names to as many commands and internal variables as a true macro language offers. VEDIT PLUS' macro language is, fortunately, reasonably well documented, although the macro language manual needs an index.

The current version of VEDIT PLUS still lacks a number of features found in other editors. Although you can use automatic disk buffering as you move forward and backward in a large file, there is no virtual memory *per se* and the total memory reserved for all text and macro buffers is a restricting 64K. You cannot view two files at one time, but must switch between them. You also cannot run a program from within VEDIT PLUS, and DOS 2.0 pathnames are not yet supported.

I also previewed a version of VEDIT PLUS in Beta test that adds brief on-line help, pathname and subdirectory support, and reverse search. In the Beta version each buffer can allocate up to 60K of RAM, a significant improvement over the older version. I am told that additional enhancements, including true virtual memory management, are planned.

XTC (Wendin, Inc.)

XTC is the only editor of the nine to have its source code included! If you wish, you can can change the code itself and recompile the editor. The code is also a fine source for useful Pascal, C, and assembler routines.

The editor itself is full-featured and powerful, boasting its own macro language and the ability to do multitasking within the editor itself. This latter feature is unique among the nine editors. This is the ability to run a macro program in the background and continue with editing in the foreground. In Version 3.0, which is a substantial upgrade of Version 2.0, you can invoke DOS and run compilers and other programs from within the editor. This version also allows wild cards in search strings (see Table 16) and auto-indention for C or Pascal programming. Searches proceed only forward, from the current cursor location. The keyboard cannot be reconfigured.

The editor is line-oriented in many of its commands. For example, with XTC's block move commands you move whole lines around rather than portions of lines. Similarly, the Undo command restores only deleted lines rather than characters or words deleted within a line. This limitation prevents you from, for example, copying just the parameter list of a procedure from one location to another; instead you must copy the whole line.

The macro language includes the full range of loops and *conditional* structures and an extensive collection of functions that allow you to keep track of the editor environment. Macros can either be entered in their keystroke form and used immediately, or written out in a longer form and compiled with a separate macro compiler. They are limited, however, to eighty keystrokes each, so even fairly simple macros must be broken into several pieces. Macros can be saved by name or assigned to function keys. In branching tests only one Boolean operator is allowed. In other words, you cannot evaluate a compound Boolean expression in one test. It is this limitation, and the fact that the program is constantly updating the screen during macro programs, that probably accounts for XTC's incredibly slow performance on some of the benchmarks. Whereas, for example, BRIEF took 6 seconds to count all the curly braces in my test file and VEDIT PLUS, which had the next slowest time, took 5 minutes and 36 seconds, XTC took over 54 minutes! Until Wendin substantially speeds up its macro language, it is, for all practical purposes, unusable.

XyWrite II Plus (Xyquest)

XyWrite II Plus is sold primarily as a word processor, and as such has garnered some outstanding reviews[2]. But it can also be used successfully as a programming, and programmable, editor, thanks to its reconfigurable keyboard, keystroke programs, and its macro programming language. It also allows you to run a compiler from within the editor.

Written in assembler, XyWrite is fast and compact, weighing in at only 72K, yet offers *all* the features of a powerful "what you see is what you get" (WYSIWYG) word processor. It boasts a useful set of wild cards for searches (see Table 16). The last deletion can be restored, be it word, line, or block. To jump to a particular line in a file, you must calculate the page on which the line would reside.

Using XyWrite's extensive list of functions, you can produce editing programs with conditional branching. These programs can be saved to disk, reloaded, and executed from XyWrite's command line. Unfortunately, the programming language lacks certain crucial commands (such as a command to determine the character on which the cursor rests) and is scantily documented. XyQuest explicitly states that it will not provide phone support for the features of its macro language.

XyWrite takes over the keyboard, so it cannot be run with SuperKey and some other memory resident programs. It will work with SideKick, if you are careful to follow certain rules about opening and closing SideKick's windows.

Notes

[1] For an MSDOS version of grep, see Allen Holub's article "grep.c: A Unix-like Generalized Regular Expression Parser" in *DDJ*, October 1984, #96.

[2] See, for example, "Xywrite: Way to Go" by Peter H. Weil in *PC Magazine*, June 12, 1984.

DDJ

Reviews Listing (Text begins on page 60)

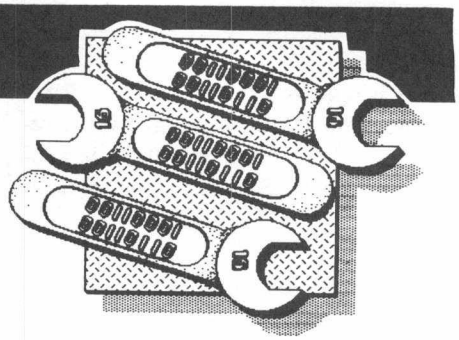
```
;**
;**   BRIEF -- demonstration program: brace counter
;**
(macro bracecnt
   (
      (int  count
            cur_char
      )
      (string  msg_pattern
               msg_text
      )
      (= msg_pattern "Excess { count = %d")
      (top_of_buffer)
      (while (search_fwd "[\\{\\}]")            ; look for { or }
         (
            (= cur_char (index "{}" (read 1)))  ; determine which you found
            (if (== cur_char 1)                 ; IF cur_char == 1 THEN
               (++ count)

               ; ELSE
               (-- count)
            )
            (next_char)
         )
      )
      (sprintf msg_text msg_pattern count)
      (message msg_text)
   )
)




; **
; ** EMACS -- demonstration program: brace counter
; **
(defun
    (bracecnt count
        (beginning-of-file)
        (while (! (eobp))                       ; while not at buffer end
            (re-search-forward "{\\|}")         ; find either { or }
            (if (= (preceding-char) '{')        ; IF == { THEN
                (++ count)
                ; ELSE
                (-- count)
            )
            (forward-character)
        )
        (message (concat "Excess { count = " count))
     )
)




;**
;**  PMATE -- demonstration program: brace counter
```

Reviews Listing (Listing continued, text begins on page 60)

```
;**
Ø V1              ; set variable 1 to Ø
ua                ; go to beginning of file
[                 ; begin loop
   @CV2           ; put character number in variable 2
   @t="{          ; if char == {
      [VA1]       ;       increment variable 1
   @t="}          ; if char == }
      [-1VA1]     ;       decrement variable 1
   m              ; move forward one character
@C=@2             ; until at end of buffer
]                 ; end loop
                  ; Since there is apparently no easy way to display
                  ; the number on the status line
b2e               ; go to buffer 2
@1\               ; insert the result into buffer 2




R*
R*      VEDIT+ -- demonstration program: brace counter
R*
B                 ! go to the top of the register !
ØXS1              ! set numeric register 1 to Ø!
[                 ! begin loop

                  ! IF char == { THEN increment variable 1 !
(.c = 123)[
1XA1
]
                  ! IF char == } THEN decrement variable 1 !
(.c = 125)[
-1XA1
]
c                 ! move a character !
(.c = 26)JL               ! IF end of file exit loop !
]                 ! end loop
@YT/Excess { = /! type result!
XT1               ! stored in numeric register 1 !
--
-- XTC demonstration program: brace counter
--

macro "braces" is
   i1 := Ø;               -- integer variable 1 is set to Ø
   bottom_of_file;
   i2 := LN;              -- number of last line, used to check if done
   top_of_file;           -- go to beginning
   BRACES2;               -- call BRACES2
end macro;


macro "IN_LINE" is        -- must be all caps to work
   goto_column (1);       -- go to beginning of line
   repeat loop
      if CC = 123 then    -- if current character is {
         i1 := i1 + 1;
      end if;
      if CC = 125 then    -- if current character is }
         i1 := i1 - 1;
      end if;
      right_character;
   until CN > LC
   end loop;              -- until past last column with non-blank character
end macro;

Macro "BRACES2" is
   IN_LINE;               -- do first line
   repeat loop
      down_line;
      IN_LINE;
   until i2 = LN
   end loop;             -- until on bottom line of file
   display_variable (i1);    -- print out the result
end macro;
```

End Listing

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

The programs published in this month's column are available for downloading from the Laboratory Microsystems RBBS *at (213) 306-3530 (300 baud or 1200 bps).*

68000 Square Roots

Mike Morton of Boston writes: "I was fascinated by the speed and elegance of Jim Cathey's 68000 square root routine (*DDJ* May 1985). If Jim is really concerned about speed he might consider these simple changes; perhaps your readers would be interested in them too. . . .

"1. Small change: the move.w #15,D4 can be changed to moveq #15,D4, assuming you don't care about the upper half of D4 (dbra is a work-oriented loop). This saves four cycles—big deal.

"2. Never shift when you can add: change each asl.l #1,Dn to add.l Dn,Dn. This reduces a ten-cycle instruction to six, saving four cycles. There are three such instructions in each loop, each executed sixteen times. Total savings, 4×3×16 = 192 cycles.

"3. Let's get desperate: surprisingly, you can trade in the roxl.l #1,Dn instructions for addx.l Dn,Dn (I haven't tested this). This saves two cycles in two different places in the loop, for a total of 64 cycles.

"The total savings are 260 cycles. This is an improvement of 14% to 17% over Jim's times. Suppose we apply the changes to the unrolled word-version of the loop that Jim suggests. The only change is for technique 3, in which roxl.w #1,Dn turns into addx.l Dn,Dn 15 times. Each of these saves four, not two, cycles—this means the savings are 2×15 cycles more than in the longword version. The total difference from the original word-size routine to the new one is 290 cycles, almost a 50% improvement."

For those who missed it, Jim Cathey's original code is run in this issue as Listing One (page 90).

68000 Binary Search & PRNG

Dr. Michael P. McLaughlin of McLean, Virginia, writes: "I am submitting the two following programming 'quickies' in response to your repeated call for 68000 code. I was a bit hesitant to send them in because I suspected they might be a bit too trivial for your typical reader. Nevertheless, I have used them extensively in studies in machine learning and am certain of their usefulness.

"The first [Listing Two, page 90] is a slightly modified binary search routine. The modification [consists in the fact that the search routine returns] the place where the missing item would be. This enables one to insert the missing item and keep the list in order. A short example of the proper use of the search routine is also included [Listing Three, page 90].

"The second routine [Listing Four, page 91] was prompted by Dave Cortesi's discussion of pseudo-random number generators [PRNGs (see the February and October '85 issues of *DDJ*)]. Because this is a well-studied field there is no reason to be uncertain of the quality of the generator. The one here is statistically very good, with a very long period. It could have been written in straightforward double precision, but that code, albeit 50% shorter, requires more time per random deviate."

80286 vs 8086

Ross P. Nelson of San Jose, California, writes: "Someone recently showed me a copy of your column in which the relative performance of the Intel 80286 was being discussed [see the April '85 issue of *DDJ*]. As a former Intel employee who has worked directly with the 286 for some time, I hope I can clear up some of the confusion.

"Let me begin by comparing the 8088 and 8086. They run at approximately the same speed when performing register-to-register operations. When memory references are included, however, the 8088 requires an extra four clocks for each 16-bit memory operation. This translates to a 20% to 30% performance degradation in a common instruction mix.

"When comparing the 80286 (real mode) to the 8086 you see about a 150% performance increase in arithmetic register-to-register operations. Memory reference performance varies, depending on how the operand is specified. I've enclosed a Table [page 89] that shows the instruction timing for different memory reference instructions. The base time is the time to execute the instruction only, the EA time is the setup time required to compute the Effective Address. All times are listed in clocks.

"As is shown in the table, memory reference instructions on the 286 show a 200% to 500% performance improvement when compared to their 8086 counterparts. For an average instruction mix, therefore, it is reasonable to claim that a 286 will run about twice as fast as an 8086 operating at the same clock rate and about 2.5 to 3 times as fast as an 8088.

"The standard instruction timings shown in the table are not affected by placing the 286 into Protected Mode. In Protected Mode, however, the 286 takes a substantial performance hit every time a segment register is loaded. For example, the instruction

MOV DS, AX

which requires 2 cycles on the 8088, 8086, and 80286 in Real Mode, requires 17 cycles on the 80286 in Protected Mode. Similarly, the instruction

INT 21H

which requires 71 cycles on the 8088, 51 cycles on the 8086, and 24 cycles on the 80286 in Real Mode, takes 79 cycles on the 80286 in Protected Mode. The INT 21H figures assume an operating system call traps through a call gate to a higher privilege level."

This Month's Filter: DUMP

Richard G. Markley of La Crescenta, California, contributed the MSDOS filter that accompanies this month's column as Listing Five (page 92). It transforms the Standard Input stream into a hex and ASCII dump and writes it to the Standard Output; both input and output can be redirected. You can therefore use this filter to dump a file or character device input stream in object format to another file or to a character device such as the printer. Here are some examples:

DUMP <file

This gives a continuous scrolling dump

DUMP <file >PRN

This dumps a file on the printer

DUMP <file1 >file2

This dumps a file into another file

Pagination can be achieved with the MORE filter that is supplied with MSDOS. Here is an example:

DUMP <file | MORE

Richard writes, "MORE.COM doesn't give a consistent display. The first display has 24 lines of new text and subsequent displays have 23. This gives an unbalanced appearance when used *in conjunction* with DUMP. I have found the problem can be solved by this sequence of actions:

A >DEBUG MORE.COM

Use DEBUG.COM to load MORE.COM into memory.

-E 1DB 00

Replace the byte at offset 1DBH with 00H. This resets MORE's row counter to zero instead of 1.

-W

Write the change to disk.

-Q

Exit from DEBUG.

"The above patch has worked successfully on MORE.COM of DOS Versions 2.0 through 3.0."

(Listings begin on next page)

DDJ

Reader Ballot

Vote for your favorite feature/article.
Circle Reader Service **No. 194.**

	8086			80286		
Instruction	**Base**	**EA**	**Total**	**Base**	**EA**	**Total**
MOV BX,[SI]	8	5	13	3	0	3
MOV CX,[1A22]	8	6	14	3	0	3
MOV AX,[BP+4]	8	9	17	3	0	3
MOV SI,[BP][DI+4]	8	11	19	3	1	4
ADD AX,[1A22]	9	6	15	7	0	7
SUB BX,[BP+2]	9	9	18	7	0	7
JMP 3A0	15	0	15	8	0	8
JMP [73EE]	15	6	21	8	0	8

Figure 1
Ross Nelson's comparison of some instruction times on the 8086, and 80286.

16-Bit (Text begins on page 88)

Listing One

```
*  Integer Square Root (32 to 16 Bit)
*
*  (Exact method, not approximate)
*
*  Call with:
*         D0.L = Unsigned number
*
*  Returns:
*         D0.L = SQRT (D0.L)
*         D1.L <> 0 if not exact root
*
*  Uses:
*         D1-D4 as temporaries ---
*         D1 = Error term
*         D2 = Running estimate
*         D3 = High bracket
*         D4 = Loop counter
*
*  Notes:
*         Result first in D0.W, but is valid in longword.
*         Takes from 1480 to 1832 cycles (including RTS).
*         (Word version is from 548 to 660 cycles).
*
lsqrt     move.w    #15,d4     ; Loop count (bits-1 of result)
          moveq     #0,d1      ; Result in D1
          moveq     #0,d2
sqrt1     asl.l     #1,d0      ; Get 2 leading bits at a time and
          roxl.l    #1,d1      ; into Error term for extrapolation.
          asl.l     #1,d0      ; (Classical method, easy in binary)
          roxl.l    #1,d1
          asl.l     #1,d2      ; Running estimate * 2
          move.l    d2,d3
          asl.l     #1,d3
          cmp.l     d3,d1
          bls       sqrt2      ; New error term > 2* running est.?
          addq.l    #1,d2      ; Yes, we want 1 bit then.
          addq.l    #1,d3      ; Fix up new error term.
          sub.l     d3,d1
sqrt2     dbra      d4,sqrt1   ; Do all 16 bit-pairs.
          movel     d2,d0      ; Returns answer in D0.W
          rts
```

End Listing One

Listing Two

```
;BINARY SEARCH -- (USES D4-D7, A6)
;TO SEARCH A SORTED ARRAY OF SIGNED LONGWORDS BEGINNING WITH A DUMMY ENTRY
;SMALLER THAN ANY POTENTIAL ENTRY, INITIALIZE THE REGISTERS AS FOLLOWS:
;       A6 = BASE ADDRESS OF ENTIRE ARRAY (LONGWORD)
;       D4 = TARGET LONGWORD
;       D7 = LENGTH OF ARRAY, IN BYTES, LESS DUMMY (LONGWORD)
;THE SEARCH WILL RETURN, IN D6, THE DISPLACEMENT (NUMBER OF BYTES FROM BASE)
;OF THE START OF THE TARGET LONGWORD, IF PRESENT, OR -DISPLACEMENT IF THE
;TARGET IS ABSENT.
;
;
                ENTRY           BINSRCH
BINSRCH         CLR.L           D5              ;D5 = pointer to bottom
                CLR.L           D6
B1              CMP.L           D5,D7           ;bottom > top ?
                BMI.S           FAILURE         ;yes, exit
                MOVE.L          D7,D6           ;else D6 = (D5+D7) div 2
                ADD.L           D5,D6
                LSR.L           #1,D6
                AND.L           #0FFFFFFFDH,D6  ;back to longword boundary
                CMP.L           0(A6,D6.L),D4   ;is this it?
                BEQ.S           SUCCESS         ;yes
                BGT.S           B2              ;no, target is bigger
                SUBQ.L          #4,D6           ;target is smaller
                MOVE.L          D6,D7           ;try lower half
                BRA             B1
B2              ADDQ.L          #4,D6           ;try upper half
                MOVE.L          D6,D5
                BRA             B1
FAILURE         NEG.L           D5              ;return -displacement
                MOVE.L          D5,D6
SUCCESS         RTS
                END             BINSRCH
```

End Listing Two

Listing Three

```
;EXAMPLE -- WHEN APPROPRIATELY LINKED, THIS CALLING ROUTINE WILL GIVE THE
;RESULTS INDICATED.
;
;
                EXTERN          BINSRCH
EXAMPLE         LEA             ARRAY,A6
                MOVE.L          #-128,D4        ;would be entry #1
                MOVE.L          #28,D7
                BSR             BINSRCH         ;returns -4
                MOVE.L          #-14,D4         ;entry #1
                MOVE.L          #28,D7
                BSR             BINSRCH         ;returns 4
                MOVE.L          #492,D4         ;entry #3
                MOVE.L          #28,D7
                BSR             BINSRCH         ;returns 12
                MOVE.L          #987654321,D4   ;entry #7
                MOVE.L          #28,D7
                BSR             BINSRCH         ;returns 28
                MOVE.L          #1000000000,D4  ;would be entry #8
                MOVE.L          #28,D7
                BSR             BINSRCH         ;returns -32
                RTS
ARRAY           DC.L            -99999          ;dummy
                DC.L            -14,27,492,18768,10112255,30741234,987654321
                END             EXAMPLE
```

End Listing Three

Listing Four

```
;PSEUDO-RANDOM NUMBER GENERATOR -- (USES D3-D7)
;GIVEN ANY SEED (1 TO 2**31-2) IN D7 (LONGWORD), THIS GENERATOR YIELDS A NON
;REPEATING SEQUENCE (RAND(I)) USING ALL INTEGERS IN THE RANGE 1 TO 2**31-2.
;THE AVERAGE EXECUTION TIME IS 342 MICROSECONDS (AT 8 MHz). THIS GENERATOR,
;REFERRED TO IN THE LITERATURE AS "GGUBS," IS KNOWN TO POSSESS GOOD
;STATISTICS. THE ALGORITHM IS:
;
;       RAND(I+1) = (16807*RAND(I)) MOD (2**31-1)
;
;WHEN PROPERLY CODED, THIS ALGORITHM WILL TRANSFORM RAND(0) = 1 INTO
;RAND(10000) =522329230. THE FOLLOWING IMPLEMENTATION USES SYNTHETIC DIVISION,
;VIZ.,
;
;       K1 = RAND(I) DIV 127773
;       RAND(I+1) = 16807*(RAND(I)-K1*127773)-K1*2836
;       IF RAND(I+1) < 0 THEN RAND(I+1) = RAND(I+1) + 2147483647
;
;REFERENCE:
;       BRATLEY, P., B.L. FOX and L.E. SCHRAGE, "A GUIDE TO SIMULATION"
;       (SPRINGER-VERLAG, 1983).
;
;
RANDOM          MOVE.L          D7,D6           ;copy RAND(I)
                BSR.S           DIV             ;divide D6 by 127773
                MOVE.L          D4,D5           ;copy K1
                MULS            #-2836,D5       ;D5 = -2836*K1
                MULU            #42591,D4       ;multiply D4 by 127773
                MOVE.L          D4,D6
                LSL.L           #1,D4
                ADD.L           D6,D4
                SUB.L           D4,D7           ;D7 = RAND(I)-K1*127773
                MOVE            #4,D4           ;counter
RAN1            MOVE.L          D7,D6           ;multiply D7 by 16807
                LSL.L           #3,D7
                SUB.L           D6,D7
                DBRA            D4,RAN1
                ADD.L           D5,D7           ;D7 = RAND(I+1)
                BPL.S           EXIT
                ADD.L           #2147483647,D7  ;normalize negative result
EXIT            RTS                             ;D7 = RAND(I+1)
;RAND(I)  (31 BITS)  DIV 127773  (17 BITS)
DIV             LSL.L           #1,D6           ;shift out unused bit
                CLR.L           D4              ;quotient
                MOVE            #14,D3          ;counter
                MOVE            D6,D5           ;save low word of RAND(I)
                SWAP            D6
                AND.L           #0FFFFH,D6      D6 = RAND(I) DIV 2**15
DIV1            LSL             #1,D4           ;line up quotient
                LSL.L           #1,D6           ;and dividend
                LSL             #1,D5           ;shift in bit of low word
                BCC.S           DIV2
                ADDQ.L          #1,D6
DIV2            CMP.L           #127773,D6      ;trial subtraction
                BMI.S           DIV3
                SUB.L           #127773,D6      ;real subtraction
                ADDQ            #1,D4           ;put 1 in quotient
DIV3            DBRA            D3,DIV1         ;decrement counter and loop
                RTS
                END             RANDOM
```

End Listing Four

Listing Five

```
1                                       page    55,132
2                                       title   'DUMP --- Hex and ASCII Dump Filter'
3                               ;
4                               ; DUMP --- filter to transform the Standard Input stream into a
5                               ;          Hex and ASCII dump, which is written to the Standard Output.
6                               ;
7                               ; Version 1.0  Richard G. Markley  July 20, 1985
8
9       = 000D                  cr              equ     0dh             ;ASCII carriage return
10      = 000A                  lf              equ     0ah             ;ASCII line feed
11
12                              ; DOS 2.x predefined handles
13
14      = 0000                  Std_Input       equ     0               ;Standard Input device or file
15      = 0001                  Std_Output      equ     1               ;Standard Output device or file
16      = 0002                  Std_Err         equ     2               ;Standard Error device or file
17
18                              ; DOS function numbers
19
20      = 0030                  Get_Version     equ     030h            ;get current DOS version
21      = 003F                  Device_Input    equ     03fh            ;read from file or device
22      = 0040                  Device_Output   equ     040h            ;write to file or device
23      = 004C                  Exit            equ     04ch            ;exit with return code
24
25      0000                    code            segment para public 'CODE'
26
27                                              assume  cs:code,ds:code,es:code,ss:code
28
29      0100                                    org     100h
30
31      0100                    dump            proc    far
32
33      0100  E9 021F R                        jmp     start
34
35                              ; data area
36
37      0103  0D 0A             Column_Guide    db      cr,lf
38      0105     0B [                           db      11 dup (' ')
39                      20
40                           ]
```

```
41
42      0110  30 20 20 31 20 20                    db      '0  1  2  3  4  5  6  7  8  9  '
43            32 20 20 33 20 20
44            34 20 20 35 20 20
45            36 20 20 37 20 20
46            38 20 20 39 20 20
47      012E  41 20 20 42 20 20                    db      'A  B  C  D  E  F',cr,lf
48            43 20 20 44 20 20
49            45 20 20 46 0D 0A
50      = 003D                        Col_Guide_Size  equ     $-Column_Guide
51
52      0140     03 [                 Data_String     db      3 dup (' ')
53                  20
54                     ]
55
56      0143  30 30 30 30 30 30       Byte_Counter    db      '000000  '
57            20 20
58      014B     19 [                 Hex_String      dw      25 dup (?)
59                  ????
60                        ]
61
62      017D     08 [                 ASCII_String    dw      8 dup (?)
63                  ????
64                        ]
65
66      018D  0D 0A                                  db      cr,lf
67      = 004F                        String_Size     equ     $-Data_String
68      = 0021                        Appx_String_Len equ     (LENGTH Hex_String)+(LENGTH ASCII_String)
69
70      018F  0D 0A 0D 0A             New_Lines       db      cr,lf,cr,lf     ;two blank lines
71      = 0004                        New_Lines_Size  equ     $-New_Lines
72
73      0193  ??                      Line_Count      db      ?               ;Count of lines per block
74
75                                    ; error messages
76
77      0194  0D 0A 44 55 4D 50       Pre_DOS_Error   db      cr,lf,'DUMP: incorrect DOS version',cr,lf
78            3A 20 69 6E 63 6F
79            72 72 65 63 74 20
80            44 4F 53 20 76 65
81            72 73 69 6F 6E 0D
82            0A
83      = 001F                        Pre_DOS_Size    equ     $-Pre_DOS_Error
84
85      01B3  0D 0A 44 55 4D 50       Input_Error     db      cr,lf,'DUMP: input device error',cr,lf
86            3A 20 69 6E 70 75
87            74 20 64 65 76 69
88            63 65 20 65 72 72
89            6F 72 0D 0A
90      = 001C                        Input_Size      equ     $-Input_Error
91
92      01CF  0D 0A 44 55 4D 50       Empty_Error     db      cr,lf,'DUMP: missing input error',cr,lf
93            3A 20 6D 69 73 73
94            69 6E 67 20 69 6E
95            70 75 74 20 65 72
96            72 6F 72 0D 0A
97      = 001D                        Empty_Size      equ     $-Empty_Error
98
99      01EC  0D 0A 44 55 4D 50       Output_Error    db      cr,lf,'DUMP: output device error',cr,lf
100           3A 20 6F 75 74 70
101           75 74 20 64 65 76
102           69 63 65 20 65 72
103           72 6F 72 0D 0A
104     = 001D                        Output_Size     equ     $-Output_Error
105
106     0209  0D 0A 44 55 4D 50       Disk_Full_Error db      cr,lf,'DUMP: disk is full',cr,lf
107           3A 20 64 69 73 6B
108           20 69 73 20 66 75
109           6C 6C 0D 0A
110     = 0016                        Disk_Full_Size  equ     $-Disk_Full_Error
111
112
113     021F  B4 30                   Start:          mov     ah,Get_Version  ;check version of DOS
114     0221  CD 21                                   int     21h
115
116     0223  0A C0                                   or      al,al           ;is it DOS v. 2.0 or higher?
117     0225  75 03                                   jnz     Prepare         ;yes, proceed
118     0227  E9 02E0 R                               jmp     Error_1         ;no, output error message
119
120     022A                          Prepare:                                ;prepare for processing
121     022A  FC                                      cld
122     022B  BD FFFF                                 mov     bp,-1           ;prevent output if no input
123     022E  E8 0324 R                               call    Input           ;perform input of data
124
125     0231                          Main_Loop:                              ;output heading
126     0231  B9 003D                                 mov     cx,Col_Guide_Size
127     0234  BA 0103 R                               mov     dx,offset Column_Guide
128     0237  E8 0341 R                               call    Output
129
130     023A  C6 06 0193 R 08                         mov     Line_Count,8    ;number of lines per block
131     023F  EB 0C 90                                jmp     Do_Block
132
133     0242                          Output_Lines:                           ;send 2 blank lines
134     0242  B9 0004                                 mov     cx,New_Lines_Size
135     0245  BA 018F R                               mov     dx,offset New_Lines
136     0248  E8 0341 R                               call    Output
137     024B  EB E4                                   jmp     Main_Loop
```

```
                                ;initialize part of data
Do_Block:                       ;template with spaces
        mov     ax,2020h
        mov     cx,Appx_String_Len
        mov     di,offset Hex_String
        rep stosw
                                ;initialize pointers
                                ;and convert data
        mov     bx,offset Hex_String
        mov     dx,size ASCII_String
        mov     di,offset ASCII_String
        jmp     Do_Line

Output_Data:    mov     cx,String_Size  ;write a line to Std Output
        mov     dx,offset Data_String
        call    Output
                                ;set pointer to sixteen's
                                ;place of byte counter
        mov     di,offset Byte_Counter+4
                                ;number of places in byte
        mov     cx,5            ;counter minus one

Adjust_Counter:                 ;increment input offset counter
        inc     byte ptr [di]   ;incr current number place value
        mov     al,[di]
        cmp     al,'9'          ;is it 9 or less?
        jbe     Check_Line_Cnt  ;yes, check no. of lines output
        cmp     al,'A'          ;no is it larger than A?
        ja      Check_For_F     ;yes, jump
        mov     byte ptr [di],'A'
        jmp     short Check_Line_Cnt

Check_For_F:    cmp     al,'F'          ;is it 'F' or less?
        jbe     Check_Line_Cnt  ;yes, check number of lines output
        mov     byte ptr [di],'0' ;no, make the digit '0'
        dec     di              ;move ptr to next higher place
        loop    Adjust_Counter  ;caryy into next higher digit

                                ;check number of lines output
Check_Line_Cnt: dec     Line_Count      ;has block been output?
        jne     Do_Block        ;no, output another line
        jmp     Output_Lines    ;yes, output blank lines

                                ;place ASCII equivalent of
                                ;byte in string
Do_Line:        mov     al,[si]         ;get byte from buffer
        mov     ch,al           ;keep copy
        cmp     al,' '          ;control code?
        jb      Not_Printable   ;yes, jump
        cmp     al,'~'          ;no, is char a tilde or less?
        jbe     Printable       ;yes, use it

Not_Printable:  mov     al,'.'          ;substitute '.' if not printable

Printable:      stosb                   ;store into ASCII section of output

                                ;place hex equivalent of byte
                                ;in output string
        mov     ah,2            ;number of nibbles in a byte
        xchg    di,bx           ;get hex section offset
Swap_Nibbles:   mov     cl,4            ;size of nibble in bits
        rol     ch,cl           ;exchange nibbles
        mov     al,ch           ;save copy
        and     al,0fh          ;mask off high 4 bits
        add     al,'0'          ;convert to ASCII char.
        cmp     al,'9'          ;is it '0-9'?
        jbe     Place_Digit     ;yes, store result
        add     al,7            ;put it into range 'A-F'

Place_Digit:    stosb                   ;store into hex section of output
        dec     ah              ;any more nibbles?
        jnz     Swap_Nibbles    ;yes, convert again

                                ;reposition pointers
        xchg    di,bx           ;get ASCII section offset
        inc     bx              ;skip space in hex section
        inc     si              ;move input pointer

                                ;input data when buffer empty
        dec     bp              ;buffer used up yet?
        jnz     Check_Item_Cnt  ;no,check item count
        call    Input

                                ;check number of items stored
Check_Item Cnt: dec     dx              ;store more data in string?
        jnz     Do_Line         ;yes, get another byte
        jmp     Output_Data     ;no, send a line

                                ;output last data string
                                ;if necessary
Finish_Output:  or      bp,bp           ;was there any input?
        jnz     Error_3         ;no, send error message
        mov     cx,String_Size  ;yes get size & addr. of string
        mov     dx,offset Data_String
        call    Output          ;send the line

                                ;exit to DOS with ERRORLEVEL set
        sub     al,al           ;return code = 0 for success
```

 (Listing continued, text begins on page 88)

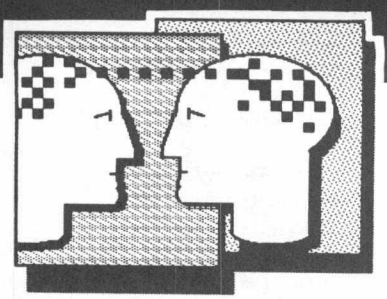
Listing Five

```
02DC  B4 4C                 Exit_to_DOS:    mov     ah,Exit          ;terminate with AL=return code
02DE  CD 21                                 int     21h

02E0  B9 001F               Error_1:        mov     cx,Pre_Dos_Size ;wrong DOS version
02E3  BA 0194 R                             mov     dx,offset Pre_Dos_Error
02E6  BD 0001                               mov     bp,1             ;save ERRORLEVEL value
02E9  EB 2E 90                              jmp     Output_Err_Msg  ;send error message

02EC  B9 001C               Error_2:        mov     cx,Input_Size   ;input device error
02EF  BA 01B3 R                             mov     dx,offset Input_Error
02F2  BD 0002                               mov     bp,2             ;save ERRORLEVEL value
02F5  EB 22 90                              jmp     Output_Err_Msg  ;send error message

02F8  B9 001D               Error_3:        mov     cx,Empty_Size   ;empty input stream
02FB  BA 01CF R                             mov     dx,offset Empty_Error
02FE  BD 0003                               mov     bp,3             ;save ERRORLEVEL value
0301  EB 16 90                              jmp     Output_Err_Msg  ;send error message

0304  B9 001D               Error_4:        mov     cx,Output_Size  ;output device error
0307  BA 01EC R                             mov     dx.offset Output_Error
030A  BD 0004                               mov     bp,4             ;save ERRORLEVEL value
030D  EB 0A 90                              jmp     Output_Err_Msg  ;send error message

0310  B9 0016               Error_5:        mov     cx,Disk_Full_Size ;output device is full
0313  BA 0209 R                             mov     dx,offset Disk_Full_Error
0316  BD 0005                               mov     bp,5             ;save ERRORLEVEL value

0319                        Output_Err_Msg:                          ;send message to Standard
                                                                     ;Error device & pass return
                                                                     ;code back to DOS
0319  B4 40                                 mov     ah,Device_Output
031B  BB 0002                               mov     bx,Std_Err       ;use handle for Standard Error dev.
031E  CD 21                                 int     21h
0320  8B C5                                 mov     ax,bp            ;recover return code
0322  EB B8                                 jmp     Exit_to_DOS      ;go terminate

0324                        Input           proc    near             ;get data from Standard Input
0324  53                                    push    bx               ;save hex section offset
0325  52                                    push    dx               ;save item count
0326  B4 3F                                 mov     ah,Device_Input
0328  BB 0000                               mov     bx,Std_Input     ;use handle for Standard Input
032B  B9 F000                               mov     cx,60*1024       ;size of input buffer
032E  BA 034F R                             mov     dx,offset Buffer;address of buffer for data
0331  CD 21                                 int     21h
0333  72 B7                                 jc      Error_2          ;jump if input device error
0335  0B C0                                 or      ax,ax            ;any input?
0337  74 94                                 jz      Finish_Output    ;no, send last string & exit
0339  BE 034F R                             mov     si,offset Buffer;set pointer to input
033C  8B E8                                 mov     bp,ax            ;length of input
033E  5A                                    pop     dx               ;restore item count
033F  5B                                    pop     bx               ;restore hex section offset
0340  C3                                    ret
0341                        Input           endp

0341                        Output          proc    near             ;send string to Standard Output
0341  B4 40                                 mov     ah,Device_Output
0343  BB 0001                               mov     bx,Std_Output    ;use handle for Standard Output
0346  CD 21                                 int     21h
0348  72 BA                                 jc      Error_4          ;jump if output device error
034A  3B C1                                 cmp     ax,cx            ;all requested bytes written?
034C  75 C2                                 jne     Error_5          ;jump, output device full
034E  C3                                    ret
034F                        Output          endp

= 034F                      Buffer          equ     $                ;beginning of input buffer

034F                        Dump            endp

034F                        Code            ends

                                            end     Dump
```

End Listings

CP/M EXCHANGE

U.S. Robotics Modem Review



by Robert Blum

The CP/M Exchange RCP/M system is available for your use 24 hours a day, 7 days a week. Reach it by dialing (404) 449-6588.

Second only to the microcomputer itself, the modem is the most useful piece of equipment brought to us by the electronics revolution. Never before has information been so readily available, or so easily accessed as it is today. From your living room you can transfer funds among bank accounts, leave a purchase order to be executed by your broker the next day, or gather information from the database on your corporation's mainframe for that last minute report.

Modems aren't new. They have been a vital part of commercial data processing for years. Only recently, however, has the owner of a personal computer been able to afford one, generally, a manual 300-baud modem. Today all this has changed. There are now available numerous intelligent 1200-bps modems for the individual to choose from. As this review is being written, 2400-bps devices are being introduced that promise better transmission quality at twice the speed. Incredible as it may seem, there are already indications that the next generation of modems will feature speeds as fast as 9600 bps and cost little more than the current crop (see the article "Modems: 2400 Bit/Sec and Beyond" by Dale Walsh in the June '85 issue of *DDJ*).

Modems come in two basic forms: stand-alone and internal. Stand-alone modems are enclosed in a case that is separate from the computer. This case contains only the modem itself and the power supply needed for independent operation. Attachment to the computer is via a serial port and appropriate cabling.

If an extra serial port is not available or the system designer is forced to enclose the communications board in the same case as the rest of the system, then an internal modem is needed. An internal, or bus-oriented, modem does not require a serial port for connection to the computer. It plugs directly into a system's backplane and responds across the bus like any other system board. Thus, it provides both the serial interface and the modem functionality.

There is one distinct disadvantage to the bus-oriented modem: it cannot be moved between systems of different types. For example, if an S-100 machine is replaced by an IBM PC or PC compatible, the S-100 modem and any driving software is rendered useless. The choice between a stand-alone or internal modem depends heavily on the needs and preferences of the user.

The S-100 Modem

Producing a bus-oriented intelligent modem that maintains reliable 1200-bps operation has, to my knowledge, been accomplished only by U.S. Robotics. The U.S. Robotics S-100 modem is totally self-contained and plugs directly into any slot of the S-100 bus. Processor services are provided through two dip switch selectable I/O port addresses that furnish performance and software compatibility with the company's Password line of stand-alone modems.

The designer of the U.S. Robotics S-100 modem divided the two separate logic sections, serial port and intelligent modem, into two separate physical sections on the board. The serial interface, USART, and the necessary support circuits occupy the left side of the standard-size S-100 board. The right side is fitted with several 40-pin chips and a handful of other support devices that form the modem section.

The Modem

The modem circuit, on the right side of the board, is the functional equivalent of the U.S. Robotics Autodial 212A or Password modems. It is a fully functional auto-dial/auto-answer modem that operates at 0 to 300 baud and 1200 bps according to the 212A protocol. It consists of two custom programmed 8049 8-bit microprocessors. The first of these interacts with a 2921 signal-processing microprocessor element that is responsible for directly controlling the phone line and the conversion of the analog transmission data into digital form. The other handles the digital interface with the serial section as well as the execution and interpretation of controlling commands sent to, or status signals returned from, the modem section.

Commands are sent to the modem via the on-board serial interface in the form of upper case ASCII text characters, numbers, and special symbols. The modem control commands are a subset of the DC Hayes command set, the de facto standard.

Because the S-100 modem is command driven, an escape sequence of characters is provided to interrupt the modem during data transmission. On reset or power-up the default escape sequence is set to +++. To ensure that the escape sequence is found among other data being transmitted, it must be bracketed by a period of at least one second during which no data is transmitted. The escape sequence is more important than might at first appear: no other method of returning the modem to command mode is available outside

A Answer mode. Puts the modem into answer mode waiting for a ring.

A/ Reexecute the last command.

AT Attention. This two character sequence must proceed all commands.

D Dial the number that follows
- **P** Dial in pulse mode.
- **T** Dial in tone mode.
- **,** Delay two seconds

E Local echo. Echo all commands.
- **E0** Local echo off.
- **E1** Local echo on.

F Type of transmission.
- **F0** Half duplex.
- **F1** Full duplex.

M Loudspeaker control
- **M0** Loudspeaker off.
- **M1** Loudspeaker on until carrier lock.
- **M2** Loudspeaker always on.

Q Result code echo.
- **Q0** Result messages sent.
- **Q1** Result messages suppressed.

S Set controlling register.
- **S0=n** Answer on nth ring.
- **S2=n** Set the escape code to n.
- **S7=n** Length of time to wait for carrier.

V Verbal or Numeric result mode.
- **V0** Numeric mode.
- **V1** Verbal mode.

X Result code set.
- **X0** Standard result codes sent.
- **X1** Extended result codes sent.

Z Reset

Table 1
S-100 Modem Command Set

Result Codes	
Verbal Mode	Numeric Mode
OK	0
CONNECT	1
RING	2
NO CARRIER	3
ERROR	4
CONNECT 1200	5

Table 2
S-100 Modem Result Codes

of dropping the DTR line.

When the escape sequence is seen the S-100 modem immediately disconnects from the phone line and returns to command mode. This action may not be desirable when you are communicating with systems that do not allow a return to local command mode while remaining on-line.

A complete list of all the commands recognized by the S-100 modem is contained in Table 1 (at left). The basic command form consists of the attention signal (AT) followed by one or more options and a single carriage-return character. For example, dialing a number and waiting for a carrier is accomplished with one command:

```
ATDT5551212
```

Here the leading AT is the attention sequence required of all commands. The D is the dial command followed by a T, which designates the tone dialing method. The rest of the command is the number to be dialed. The modem will dial the desired number without any further attention and will return the CONNECT or CONNECT 1200 result code if a connection is made. Otherwise, the NO CARRIER result code will be sent to indicate that no suitable connection was made in 17 seconds.

Acknowledgement of commands and the operating status of the modem are signaled by either full text messages or single digits sent following the execution of commands. Result codes are also sent to signal the controlling software of conditions such as the phone ringing and the presence or absence of a carrier. A complete list of the result codes is contained in Table 2 (at left).

The S-100 modem will also automatically switch between originate and answer modes. When answering the telephone, it will automatically choose either high- or low-speed communications. Result codes are then sent to the controlling software or terminal device to indicate whether a high- or low-speed connection has been established. The microcomputer can then adjust its own rate accordingly.

The S-100 modem comes with a speaker that reproduces the sound of the dialing tones and provides information about progress towards a connection. Once communication has been established, the speaker is automatically turned off if the appropriate option has been selected. Although the speaker is contained in the case of the computer, it is generally loud enough to be heard.

The Serial Interface

The serial interface, on the left side of the board, substitutes for the usual RS-232 connection between the computer and the modem. The 8251A USART provides the parallel S-100 bus interface and subsequent serialization of data and conversion of the baud rate to match that of the modem section. Communication with the USART thus takes place through two of the commonly used I/O channels of the 8080, Z80, and 8086 family of microprocessors.

The 8251A USART is a programmable serial communications device. Programming the 8251A requires the user to send commands to the 8251A. The 8251A is able to send information regarding the status of the communications to the microprocessor.

Installing the Modem

The first step in using the S-100 modem is to install it in the S-100 system. Because the S-100 modem uses the I/O port system of the microprocessor, the user must decide which ports to dedicate to the modem. One of the ports, the one with the lowest number, is used to transfer data to and from the serial interface. The other port is used to address the Mode/Command and Status registers of the 8251A. The S-100 modem is designed to use the two addresses at the top or bottom of any I/O port page. Thus, the modem may be addressed as 00,01; 0E,0F; 10,11; EE,EF and so on. This provides 32 different locations at which the modem's serial interface may be addressed.

Programming the S-100 Card Modem

Writing programs for the S-100 mo-

dem requires some knowledge of the 8251A USART and the command structures of that chip. Reading and writing data are fairly simple tasks for the 8251A. They merely require the use of the IN and OUT commands of the assembler along with the appropriate addresses. Setting up the USART for communications requires more skill.

The Mode/Command information is used to program the USART for the correct baud or bps rate and word length. Status information indicates whether the USART is ready to send or receive information. Errors are also indicated in the status register.

Listing One (at right) is a program fashioned after overlays used in the popular public-domain BYE or MBYE programs. It is a comprehensive example of how to implement all of the S-100 modem's features. A much simpler example could be written by following these steps:

1. Program the 8251A for the proper communications mode
2. Check the status for incoming characters
3. Check the status for errors
4. Get an incoming character
5. Process incoming character
6. Check the status for send ready
7. Send a character if status ready

Once the proper communications settings have been made, the program using the USART may send or receive information in any order desired. The status should always be checked before sending or receiving information.

Documentation

The manual accompanying the S-100 modem consists of 37 pages printed on one side. The first 20 pages provide information relevant only to the operation of the S-100 modem; the remainder of the manual is a reproduction of the 8251A data sheets. The documentation contains only enough information to get you started. The examples don't provide enough guidance on how to interact with the modem through commands or how to handle result codes. A large portion of the time I took writing the example found in Listing One was spent in front of the CRT as I used my debugger to solve the mysteries surrounding the workings of the command system.

User Impressions

I own two modems, the newest of which is the S-100 modem. During the past several months I have had the unique opportunity to switch between them when I met a communications problem. Never was I able to pinpoint a situation in which one performed better than the other, although I did find a few peculiarities in the S-100 modem that were bothersome. One particularly aggravating problem was that the S-100 had a nasty habit of answering the phone when no call had actually come in. The only other problem serious enough to deserve mention here is that, when dialing out at 300 baud, the originating carrier would be brought up immediately without waiting for the answer tone.

The Program Listing

I am placing this program in the public domain. If you find a problem or add an exciting feature, let me know so I can update the source code. You can download the program from my RCP/M (404-449-6588) or obtain a copy of it in one of a variety of formats (write me care of *DDJ*).

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**

CP/M Exchange Listing (Text begins on page 102)

```
;*****************************************************************
;*****************************************************************
;
;       This routine is in the form of an overlay for the
;       popular MBYE - BYE public domain RCP/M programs.
;       Although not entirely compatible with those programs
;       only minor modifications would be necessary to adapt it
;       for use with them.  It is written in 8080 mnemonic to
;       insure maximum compatibility with multiple standard
;       assemblers and CPUs.
;
;*****************************************************************
;*****************************************************************

;*****************************************************************
;*****************************************************************
;
;       Define the equates for this routine
;
;*****************************************************************
;*****************************************************************

false:          equ     0
true:           equ     not false

md_res_ok:      equ     '0'                     ;result - OK
md_res_con300:  equ     '1'                     ;result - CONNECT (300)
md_res_ring:    equ     '2'                     ;result - RING
md_res_nocar:   equ     '3'                     ;result - NO CARRIER
md_res_con1200: equ     '5'                     ;result - CONNECT (1200)

md_mhz:         equ     60                      ;set to processor speed X 10

md_cr:          equ     0dh                     ;carriage return
md_lf:          equ     0ah                     ;line feed
md_eom:         equ     080h                    ;end of message

md_base_port:   equ     00h                     ;modem ports start at zero
md_data_port:   equ     md_base_port            ;data port address
md_stat_port:   equ     md_data_port+1          ;status port address

md_tr_rdy:      equ     01h                     ;transmitter empty
md_rcv_rdy:     equ     02h                     ;receiver ready
md_par_err:     equ     08h                     ;parity error
md_ovf_err:     equ     010h                    ;overflow error
md_frm_err:     equ     020h                    ;framing error
md_error:       equ     md_par_err+md_ovf_err+md_frm_err
                                                ;general mask for all errors
md_carrier:     equ     080h                    ;data carrier

md_mode_syn:    equ     0                       ;mode byte for synchronous
md_mode_x1:     equ     1h                      ;mode for async. X 1 clock
```

(Continued on next page)

CP/M Exchange Listing

(Listing continued, text begins on page 102)

```
md_mode_x16:    equ     2h                      ;mode for async. X 16 clock
md_mode_x64:    equ     3h                      ;mode for async. X 64 clock
md_mode_c15:    equ     0h                      ;mode for character lnth of 5
md_mode_c16:    equ     4h                      ;mode for character lnth of 6
md_mode_c17:    equ     8h                      ;mode for character lnth of 7
md_mode_c18:    equ     0ch                     ;mode for character lnth of 8
md_mode_nop:    equ     0h                      ;mode for no parity
md_mode_pon:    equ     010h                    ;mode for parity
md_mode_oddp:   equ     0h                      ;mode for odd parity
md_mode_evnp:   equ     020h                    ;mode for even parity
md_mode_stpl:   equ     040h                    ;mode for 1 stop bit
md_mode_stpl5:  equ     080h                    ;mode for 1.5 stop bit
md_mode_stp2:   equ     0c0h                    ;mode for 2 stop bit

md_cmd_xmton:   equ     1h                      ;command for transmitter enable
md_cmd_xmtof:   equ     0h                      ;command for transmitter disable
md_cmd_dtron:   equ     2h                      ;command for DTR active
md_cmd_dtrof:   equ     0h                      ;command for DTR inactive
md_cmd_rcvon:   equ     4h                      ;command for receiver enable
md_cmd_rcvof:   equ     0h                      ;command for receiver disable
md_cmd_brkon:   equ     8h                      ;command for sending break
md_cmd_brkof:   equ     0h                      ;command for no break
md_cmd_eres:    equ     010h                    ;command for error reset
md_cmd_rtson:   equ     020h                    ;command for RTS active
md_cmd_rtsof:   equ     0h                      ;command for RTS inactive
md_cmd_ires:    equ     040h                    ;command for internal reset
md_cmd_hunt:    equ     080h                    ;command for hunt mode (not used)

outstr:         macro   string
                if      not nul string
                lxi     h,string                ;point at string to output
                endif
                call    md_out_msg              ;call subroutine
                mend

outchr:         macro   byte
                if      not nul byte
                mvi     c,byte                  ;load C with byte value
                endif
                call    md_out                  ;output one byte
                mend

tstngo:         macro   test,good,error
                mov     c,a                     ;save in C
                cpi     test                    ;is it an test
                if not nul good
                jz      good                    ;yes - take good exit
                endif
                if not nul error
                jnz     error                   ;no - take error exit
                mend

sloout:         macro   port
                if      md_mhz gt 40
                rept    ((md_mhz-40)/10)-1

                xchg                            ;allow 8251-A to catch up
                xchg                            ;*
                mend
                endif
                out     port
                mend

sloin:          macro   port
                if      md_mhz gt 40
                rept    ((md_mhz-40)/10)-1
                xchg                            ;allow 8251-A to catch up
                xchg                            ;*
                mend
                endif
                in      port
                mend

                page
;*****************************************************************
;*****************************************************************
;
;       This routine initializes the modem to the defaults
;       set after a hardware reset.
;
;*****************************************************************
;*****************************************************************

md_init:

                call    md_carck                ;is carrier still up
                cnz     md_hangup               ;yes - drop the line
                call    md_set1200              ;default to 1200 baud
                call    md_reset_modem          ;reset the MODEM

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
```

(Continued on page 108)

CP/M Exchange Listing

(Listing continued, text begins on page 102)

```
;       This subroutine exits with the phone on hook.
;
;*****************************************************************
;*****************************************************************

md_hangup:

                call    md_wait_1               ;wait one second
                lxi     h,md_dis_msg            ;point at disconnect message
                call    md_out_msg              ;write string to modem
                call    md_wait_1               ;wait one second
                call    md_carck                ;carrier should have dropped
                rz                              ;it did - return
                jp      md_hangup               ;error recovery may be improved
                                                ;for now just continue telling
                                                ;the modem to hangup

;*****************************************************************
;*****************************************************************
;
;       Dial a number
;
;*****************************************************************
;*****************************************************************

md_dial:

                outstr  md_attn_ldin            ;send AT
                outstr  md_dial_ldin            ;send D
                outstr  md_tone_stub            ;send T
                outstr  md_number               ;send number
                call    md_read_res             ;get the result code

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       The carrier check routine will return the Z condition
;       code if the carrier is not present.
;
;*****************************************************************
;*****************************************************************

md_carck:

                call    md_status               ;get modem status
                ani     md_carrier              ;strip all but carrier bit

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Output a control string to the modem.  The string must
;       end with a 080h.
;
;*****************************************************************
;*****************************************************************

md_out_msg:

                mov     c,m                     ;get a byte
                mov     a,c                     ;is this end of string?
                rlc                             ;*
                rc                              ;yes - return
                call    md_out                  ;output it
                inx     h                       ;point at next byte
                jmp     md_out_msg              ;loop until complete

;*****************************************************************

;*****************************************************************
;
;       Output one byte to the modem
;
;*****************************************************************
;*****************************************************************

md_out:

                call    md_status               ;get modem status
                ani     md_tr_rdy               ;is transmitter ready to
                                                ;receive a byte
                jz      md_out                  ;no - loop until ready
                mov     a,c                     ;put data byte into A
                out     md_data_port            ;output it
                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Input one byte from the modem
;
;*****************************************************************
;*****************************************************************

md_input:
```

```
                call    md_status               ;get USART status
                ani     md_rcv_rdy              ;is a byte available
                jz      md_input                ;no - loop until ready
                in      md_data_port            ;get data byte
                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Input a verbal or numeric response message from the
;       modem and convert to a numeric response in the
;       A register.
;
;*****************************************************************
;*****************************************************************

md_read_res:

                call    md_input                ;get the leading CR
                tstngo  md_cr,md_read_res_ver   ;if CR go to verbal routine

                sta     save_res_code   ;save result code for exit
                call    md_input                ;get the trailing CR
                tstngo  md_cr,md_read_res_ext2,md_read_res_err
                                                ;go to exit if CR otherwise
                                                ;go to error routine

md_read_res_ver:

                call    md_input                ;get the leading LF
                tstngo  md_lf,,md_read_res_err  ;if not LF go to error routine

                call    md_input                ;get the first response byte

                tstngo  'O',md_read_res_ok      ;is it OK
                tstngo  'C',md_read_res_con     ;is it CONNECT or CONNECT 1200
                tstngo  'R',md_read_res_ring    ;is it RING
                tstngo  'N',md_read_res_ncar    ;is it NO CARRIER

;
;       Some type of unexpected error has occurred.  An unknown result
;       message has been sent.  Simply empty the USART and return with
;       ERROR posted in the C register.
;

md_read_res_err:

                call    md_wait_1               ;delay one second
                call    md_status               ;has any further data been sent
                ani     md_rcv_rdy              ;*
                jnz     md_read_res_err         ;yes - loop until empty

                mvi     a,'4'                   ;set error code in A
                sta     save_res_code           ;save for exit
                jmp     md_read_res_ext2        ;return to caller

md_read_res_ok:

                call    md_input                ;get the K
                tstngo  'K',,md_read_res_err    ;is it really a K
                mvi     a,'0'                   ;set our numeric result code
                sta     save_res_code           ;*

md_read_res_ext:

                call    md_input                ;get the trailing CR
                tstngo  md_cr,,md_read_res_err  ;is it really a CR

md_read_res_ext1:

                call    md_input                ;get the trailing LF
                tstngo  md_lf,,md_read_res_err  ;is it really a LF

md_read_res_ext2:

                mvi     a,1                     ;wait time of .1 second
                call    md_wait_1a              ;allow modem to recover
                lda     save_res_code           ;put user result code into A
                ret                             ;return to caller

md_read_res_con:

                call    md_input                ;get the O
                tstngo  'O',,md_read_res_err    ;is it really a O
                call    md_input                ;get the N
                tstngo  'N',,md_read_res_err    ;is it really a N
                call    md_input                ;get the N
                tstngo  'N',,md_read_res_err    ;is it really a N
                call    md_input                ;get the E
                tstngo  'E',,md_read_res_err    ;is it really a E
                call    md_input                ;get the C
                tstngo  'C',,md_read_res_err    ;is it really a C
                call    md_input                ;get the T
                tstngo  'T',,md_read_res_err    ;is it really a T

                mvi     a,'1'                   ;set the 300 baud result
                sta     save_res_code           ;*

                call    md_input                ;get the trailing CR or blank
                tstngo  ' ',,md_read_res_ext1   ;is it really a BLANK
                call    md_input                ;get the 1
                tstngo  '1',,md_read_res_err    ;is it really a 1
                call    md_input                ;get the 2
                tstngo  '2',,md_read_res_err    ;is it really a 2
```

(Continued on next page)

CP/M Exchange Listing

(Listing continued, text begins on page 102)

```
                call    md_input                ;get the 0
                tstngo  '0',,md_read_res_err    ;is it really a 0
                call    md_input                ;get the 0
                tstngo  '0',,md_read_res_err    ;is it really a 0

                mvi     a,'5'                   ;set the 1200 baud result code
                sta     save_res_code           ;*
                jmp     md_read_res_ext         ;exit thru common routine

md_read_res_ring:

                call    md_input                ;get the I
                tstngo  'I',,md_read_res_err    ;is it really a I
                call    md_input                ;get the N
                tstngo  'N',,md_read_res_err    ;is it really a N
                call    md_input                ;get the G
                tstngo  'G',,md_read_res_err    ;is it really a G

                mvi     a,'2'                   ;set ring result code
                sta     save_res_code           ;*
                jmp     md_read_res_ext

md_read_res_ncar:

                call    md_input                ;get the O
                tstngo  'O',,md_read_res_err    ;is it really a O
                call    md_input                ;get the blank
                tstngo  ' ',,md_read_res_err    ;is it really a BLANK
                call    md_input                ;get the C
                tstngo  'C',,md_read_res_err    ;is it really a C
                call    md_input                ;get the A
                tstngo  'A',,md_read_res_err    ;is it really a A
                call    md_input                ;get the R
                tstngo  'R',,md_read_res_err    ;is it really a R
                call    md_input                ;get the R
                tstngo  'R',,md_read_res_err    ;is it really a R
                call    md_input                ;get the I
                tstngo  'I',,md_read_res_err    ;is it really a I
                call    md_input                ;get the E
                tstngo  'E',,md_read_res_err    ;is it really a E
                call    md_input                ;get the R
                tstngo  'R',,md_read_res_err    ;is it really a R

                mvi     a,'3'                   ;set no carrier result code
                sta     save_res_code           ;*
                jmp     md_read_res_ext         ;return to user

;*****************************************************************
;*****************************************************************
;
;       Get USART status and return in A
;
;*****************************************************************
;*****************************************************************

md_status:

                in      md_stat_port            ;get modem status
                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       This routine is a duplication of one that already exists
;       in the MBYE or BYE programs.  It is included here to make
;       this routine complete within itself.
;
;*****************************************************************
;*****************************************************************

md_wait_1:

                mvi     a,10                    ;constant to delay for 1 second

md_wait_1a:

                push    psw                     ;save loop counter
                call    md_delay                ;delay .1 second
                pop     psw                     ;restore loop counter
                dcr     a                       ;is loop complete
                jnz     md_wait_1a              ;no - loop until complete

                ret                             ;yes - return to caller

;*****************************************************************
;*****************************************************************
;
;       This subroutine will delay returning for .1 of a second
;       depending on the processor speed.
;
;*****************************************************************
;*****************************************************************

md_delay:
```

(Continued on page 112)

CP/M Exchange Listing

(Listing continued, text begins on page 102)

```
                lxi     b,(4167*(md_mhz/10)) + (417*(md_mhz mod 10))
                                                ;constant times mod 10 MHZ
md_delay_a:

                dcx     b                       ;decrement loop counter
                mov     a,b                     ;has counter elapsed
                ora     c                       ;*
                jnz     md_delay_a              ;*

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Ask the MODEM to respond
;
;*****************************************************************
;*****************************************************************

md_wake_up:

                outstr  md_attn_ldin            ;send AT
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get result code

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Reset the MODEM to it's default values
;
;*****************************************************************
;*****************************************************************

md_reset_modem:

                outstr  md_attn_ldin            ;send AT
                outstr  md_reset_stub           ;send Z
                outstr  md_eom_stub             ;send CR
                call    md_wait_1               ;*
                mvi     a,'0'                   ;set a result code of OK

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM to not answer the phone if it rings

;
;*****************************************************************
;*****************************************************************

md_set_noans:

                outstr  md_attn_ldin            ;send AT
                outstr  md_setreg_ldin          ;send S
                outstr  md_anscnt_stub          ;send 0=
                outchr  '0'                     ;send no rings
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM to answer the phone after 1 ring
;
;*****************************************************************
;*****************************************************************

md_set_ans1:

                outstr  md_attn_ldin            ;send AT
                outstr  md_setreg_ldin          ;send S
                outstr  md_anscnt_stub          ;send 0
                outchr  '1'                     ;send one ring
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM to ECHO all control characters
;
;*****************************************************************
;*****************************************************************

md_set_echo_on:

                outstr  md_attn_ldin            ;send AT
                outstr  md_echo_ldin            ;send E
```

```
                outstr  md_eon_stub             ;send 1
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM to not ECHO any control characters
;
;*****************************************************************
;*****************************************************************

md_set_echo_off:

                outstr  md_attn_ldin            ;send AT
                outstr  md_echo_ldin            ;send E
                outstr  md_eoff_stub            ;send 0
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM to FULL duplex operation
;
;*****************************************************************
;*****************************************************************

md_set_full_on:

                outstr  md_attn_ldin            ;send AT
                outstr  md_duplx_ldin           ;send F
                outstr  md_fdup_stub            ;send 1
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM to HALF duplex operation
;
;*****************************************************************
;*****************************************************************

md_set_half_on:

                outstr  md_attn_ldin            ;send AT
                outstr  md_duplx_ldin           ;send F
                outstr  md_hdup_stub            ;send 0
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM speaker on
;
;*****************************************************************
;*****************************************************************

md_set_spkr_on:

                outstr  md_attn_ldin            ;send AT
                outstr  md_spkr_ldin            ;send F
                outstr  md_spkon_stub           ;send 2
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM speaker off
;
;*****************************************************************
;*****************************************************************

md_set_spkr_off:

                outstr  md_attn_ldin            ;send AT
                outstr  md_spkr_ldin            ;send F
                outstr  md_spkoff_stub          ;send 0
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM speaker on during dialing
```

CP/M Exchange Listing

(Listing continued, text begins on page 102)

```
;
;*****************************************************************
;*****************************************************************

md_set_spkr_dial:

                outstr  md_attn_ldin            ;send AT
                outstr  md_spkr_ldin            ;send F
                outstr  md_spkdl_stub           ;send 1
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM result messages on
;
;*****************************************************************
;*****************************************************************

md_set_res_on:

                outstr  md_attn_ldin            ;send AT
                outstr  md_reslt_ldin           ;send Q
                outstr  md_reson_stub           ;send 0
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM result messages off
;
;*****************************************************************
;*****************************************************************

md_set_res_off:

                outstr  md_attn_ldin            ;send AT
                outstr  md_reslt_ldin           ;send Q
                outstr  md_resoff_stub          ;send 1
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM result messages to numeric
;
;*****************************************************************
;*****************************************************************

md_set_res_num:

                outstr  md_attn_ldin            ;send AT
                outstr  md_resmod_ldin          ;send V
                outstr  md_num_stub             ;send 0
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM result messages to verbal
;
;*****************************************************************
;*****************************************************************

md_set_res_ver:

                outstr  md_attn_ldin            ;send AT
                outstr  md_resmod_ldin          ;send V
                outstr  md_ver_stub             ;send 1
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM result messages to extended format
;
;*****************************************************************
;*****************************************************************
```

(Continued on page 116)

CP/M Exchange Listing

(Listing continued, text begins on page 102)

```
md_set_res_ext:

                outstr  md_attn_ldin            ;send AT
                outstr  md_res_ldin             ;send X
                outstr  md_extres_stub          ;send 1
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the MODEM result messages to standard format
;
;*****************************************************************
;*****************************************************************

md_set_res_std:

                outstr  md_attn_ldin            ;send AT
                outstr  md_res_ldin             ;send X
                outstr  md_stdres_stub          ;send 0
                outstr  md_eom_stub             ;send CR
                call    md_read_res             ;get the result code in A

                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Reset the USART.
;
;       This routine has been written for the worst case reset
;       condition.  Upon initial power on the 8251A may be in
;       an undetermined mode state.  For that reason the maximum
;       mode setup case of setting synchronous mode and two sync
;       characters is used.  Otherwise, there would be a
;       possibility of not consistently completing a reset.
;
;       When exit from this routine is taken the 8251A will be
;       ready for the initial mode and command - command bytes.
;
;*****************************************************************
;*****************************************************************

md_reset_usart:

                xra     a                       ;get a zero in A

                sloout  md_stat_port            ;output mode synchronous

                sloout  md_stat_port            ;output first sync byte

                sloout  md_stat_port            ;output second sync byte

                mvi     a,md_cmd_ires           ;8251A reset command
                sloout  md_stat_port            ;issue the reset command
                ret                             ;return to user

;*****************************************************************
;*****************************************************************
;
;       Reset the USART error status
;
;*****************************************************************
;*****************************************************************

md_reset_err:

                lda     save_cur_cmd            ;get error reset command
                sloout  md_stat_port            ;send to status port
                ret                             ;return to caller

;*****************************************************************
;*****************************************************************
;
;       Set the USART to 1200 baud
;
;*****************************************************************
;*****************************************************************

md_set1200:

                call    md_reset_usart          ;establish known initial mode

                mvi     a,md_mode_x16+md_mode_c18+md_mode_nop+md_mode_stp1
                                                ;set mode to:
                                                ;               16X clock
                                                ;               8 bits per characte
                                                ;               no parity
                                                ;               1 stop bit
                sta     save_cur_mode           ;save current mode byte
                sloout  md_stat_port            ;send the mode byte
```

```
                mvi     a,md_cmd_xmton+md_cmd_dtron+md_cmd_rcvon+md_cmd_rtson+md_cm
                                        ;set command to:
                                        ;               transmitter on
                                        ;               DTR on (not used)
                                        ;               receiver on
                                        ;               RTS on
                                        ;               RESET errors
                sta     save_cur_cmd    ;save current command byte
                sloout  md_stat_port    ;send the command byte
                ret                     ;return to caller

;*************************************************************
;*************************************************************
;
;       Set the USART to 600 baud
;
;*************************************************************
;*************************************************************

md_set600:

                call    md_reset_usart  ;establish known initial mode

                mvi     a,md_mode_x16+md_mode_c18+md_mode_nop+md_mode_stpl
                                        ;set mode to:
                                        ;               16X clock
                                        ;               8 bits per characte
                                        ;               no parity
                                        ;               1 stop bit
                sta     save_cur_mode   ;save current mode byte
                sloout  md_stat_port    ;send the mode byte

                mvi     a,md_cmd_xmton+md_cmd_dtron+md_cmd_rcvon+md_cmd_rtsof+md_cm
                                        ;set command to:
                                        ;               transmitter on
                                        ;               DTR on (not used)
                                        ;               receiver on
                                        ;               RTS off
                                        ;               RESET errors
                sta     save_cur_cmd    ;save current command byte
                sloout  md_stat_port    ;send the command byte
                ret                     ;return to caller

;*************************************************************
;*************************************************************
;
;       Set the USART to 300 baud
;
;*************************************************************
;*************************************************************


md_set300:

                call    md_reset_usart  ;establish known initial mode

                mvi     a,md_mode_x64+md_mode_c18+md_mode_nop+md_mode_stpl
                                        ;set mode to:
                                        ;               16X clock
                                        ;               8 bits per characte
                                        ;               no parity
                                        ;               1 stop bit
                sta     save_cur_mode   ;save current mode byte
                sloout  md_stat_port    ;send the mode byte

                mvi     a,md_cmd_xmton+md_cmd_dtron+md_cmd_rcvon+md_cmd_rtson+md_cm
                                        ;set command to:
                                        ;               transmitter on
                                        ;               DTR on (not used)
                                        ;               receiver on
                                        ;               RTS on
                                        ;               RESET errors
                sta     save_cur_cmd    ;save current command byte
                sloout  md_stat_port    ;send the command byte
                ret                     ;return to caller

;*************************************************************
;*************************************************************
;
;       Set the USART to 150 baud
;
;*************************************************************
;*************************************************************

md_set150:

                call    md_reset_usart  ;establish known initial mode
                mvi     a,md_mode_x64+md_mode_c18+md_mode_nop+md_mode_stpl
                                        ;set mode to:
                                        ;               16X clock
                                        ;               8 bits per characte
                                        ;               no parity
                                        ;               1 stop bit
                sta     save_cur_mode   ;save current mode byte
                sloout  md_stat_port    ;send the mode byte

                mvi     a,md_cmd_xmton+md_cmd_dtron+md_cmd_rcvon+md_cmd_rtsof+md_cm
                                        ;set command to:
                                        ;               transmitter on
                                        ;               DTR on (not used)
                                        ;               receiver on
                                        ;               RTS off
                                        ;               RESET errors
                sta     save_cur_cmd    ;save current command byte
                sloout  md_stat_port    ;send the command byte
                ret                     ;return to caller
```

(Continued on next page)

CP/M Exchange Listing

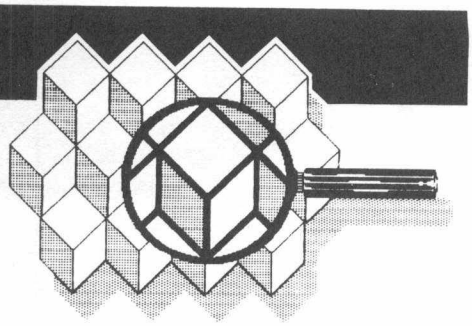
(Listing continued, text begins on page 102)

```
                page

;*****************************************************************
;*****************************************************************
;
;       This section of the program falls after all other
;       program code as required by the relocation routine used.
;
;*****************************************************************
;*****************************************************************

endobj:

md_dis_msg:     db      '+++',md_eom            ;disconnect from phone line
md_repeat_msg:  db      'A/',md_eom             ;repeat last command

md_attn_ldin:   db      'AT',md_eom             ;beginning of all messages

md_dial_ldin:   db      'D',md_eom              ;dial number lead-in
md_echo_ldin:   db      'E',md_eom              ;echo all commands lead-in
md_duplx_ldin:  db      'F',md_eom              ;duplex transmission
md_spkr_ldin:   db      'M',md_eom              ;speaker control
md_reslt_ldin:  db      'Q',md_eom              ;result message control
md_setreg_ldin: db      'S',md_eom              ;set registers
md_resmod_ldin: db      'V',md_eom              ;set numeric/verbal result mode
md_res_ldin:    db      'X',md_eom              ;set result mode

md_ans_stub:    db      'A',md_eom              ;answer phone immediately
md_pulse_stub:  db      'P',md_eom              ;dial via pulse method
md_tone_stub:   db      'T',md_eom              ;dial via tone method
md_pause_stub:  db      ',',md_eom              ;delay for 2 seconds during
dialing
md_eoff_stub:   db      '0',md_eom              ;turn local echo off
md_eon_stub:    db      '1',md_eom              ;turn local echo on
md_hdup_stub:   db      '0',md_eom              ;half duplex
md_fdup_stub:   db      '1',md_eom              ;full duplex
md_spkoff_stub: db      '0',md_eom              ;turn speaker off
md_spkdl_stub:  db      '1',md_eom              ;turn speaker on until connect
md_spkon_stub:  db      '2',md_eom              ;turn speaker on
md_reson_stub:  db      '0',md_eom              ;send result messages
md_resoff_stub: db      '1',md_eom              ;suppress result messages
md_anscnt_stub: db      '0=',md_eom             ;set answer count
md_escod_stub:  db      '2',md_eom              ;set escape code
md_carwat_stub: db      '7',md_eom              ;set wait for carrier time
md_num_stub:    db      '0',md_eom              ;result messages are numeric
md_ver_stub:    db      '1',md_eom              ;result messages are verbal
md_stdres_stub: db      '0',md_eom              ;standard result codes
md_extres_stub: db      '1',md_eom              ;extended result codes
md_reset_stub:  db      'Z',md_eom              ;reset command
md_eom_stub:    db      md_cr,md_eom            ;end of message

md_number:      ds      20                      ;place phone number here
                                                ;followed by md_eom_stub
save_res_code:  ds      1


save_cur_mode:  ds      1                       ;save area for current mode byte
save_cur_cmd:   ds      1                       ;save area for current command byte
```

Listing 1 - S-100 Modem Overlay Program

End Listing

OF INTEREST



by Alex Ragen

CP/M

Electronic Business, a magazine devoted to the dollars and cents issues of the computer business, recently published a long article analyzing the microprocessor market. It carried the surprising message that 8-bit processors make up by far the lion's share of that market, with 16-bit processors trailing far behind. Moreover, most industry observers expect this situation to continue for many years. The reason is that the now inexpensive 8-bit processors and their peripherals are perfectly adequate for most applications. Where 16-bit processors hold sway is in the prestige conscious world of the personal computer, where only the latest fashion is marketable. In my corner of the world, everyone talks only about BMWs and IBM PCs, but they drive Toyotas and, when they build a piece of equipment, they use Z80s whenever they can and develop the software for it with that workhorse, CP/M 80, the reports of whose death have been greatly exaggerated.

It's true that finding a place to buy CP/M 80 software is a challenge, but there is plenty of good software out there. Those who persevere will find that their blood, sweat and tears will be rewarded. One of the most nagging problems a dealer has in stocking CP/M 80 software is that he has to carry several dozen disk formats. As a result stores are not a good place to find software. The best source is the manufacturer, who can whip up a disk for your format using one of the disk emulation programs now widely available. A good source for advertisements is Kaypro's magazine, *PROFILES*, which is available at most Kaypro dealers.

Echelon Inc. is one company that continues to develop quality CP/M 80 software. Its *pièce de résistance* is ZCPR3, the CCP replacement which gives CP/M 80 a number of Unix like features. Also SYSLIB3, a macro library with 210 subroutines, DISCAT, a disk and file utility program, TERM 3, a telecommunications program, and many others. All programs come with extensive documentation. For a catalog and price list, contact the vendor at 101 First St., Los Altos, CA 94022, (415) 948-3820. **Reader Service Number 101.**

The KAMAS "Outline Processor" is now available for CP/M 80 2.2. KAMAS combines outline processing and information retrieval with word processing to provide a complete environment for developing and controlling text. KAMAS is an aid for writers, researchers and other professionals who must organize their ideas and categorize information. It allows users to classify information in a familiar outline structure, and then to alter and access the information based on that structure. Levels of the outline can be collapsed from the screen and hidden from view, and then expanded back into view for editing. KAMAS includes a full screen editor with document output controls as well as a Forth-like programming language. There is a growing body of public domain add-ons, and the vendor offers three volumes of these utilities. The price is $147 plus $4 shipping. Contact KAMASOFT Inc., 2525 S.W. 224th Avenue, Aloha, OR 97006, (503) 649-3765. **Reader Service Number 103.**

Alternative Languages

The Forth-83 Handy Reference Card is now available free from the Forth Interest Group (FIG). It functions as a pocket programming aid and reference for Forth programmers. Commands are grouped by function and include: stack manipulation, comparison, arithmetic and others. For additional information about the card and FIG, call the FIG hot line at (408) 277-0668 or write FIG, P.O. Box 8231, San Jose, CA 95155. **Reader Service Number 105.**

MacScheme is a modern implementation of Lisp for the 512K Macintosh. It offers an interpreter with full run time error detection and debugging and adheres to the standards for Scheme, an influential dialect of Lisp. It supports the most important features of Lisp: lexically scoped variables, first class procedures (closures), macros, and generic arithmetic, including floating point and infinite precision integers. The Smalltalk-like interface features multiple scrolling windows. The price is $125 without copy protection. Contact Semantic Microsystems, *1001* Bridgeway, Suite 543, Sausalito, CA 94965. **Reader Service Number 111.**

UO-LISP, which contains over 400 standard Lisp functions and comes with an optimizing compiler that generates native 8086 code, is now available for the IBM PC and compatibles. There is a multi-window screen editor that supports Lisp interactions and a complete program development package. The entire package is priced at $150. The LEARN LISP system, a subset with on line help, a special reference manual and tutorial guide, is priced at $85. UO-LISP2, a CP/M version, costs $125. Contact Northwest Computer Algorithms, P.O. Box 90995, Long Beach, CA 90809, (213) 426-1893. **Reader Service Number 113.**

CLISP is a Lisp interpreter written in C for the IBM PC. It has 40 functions for list manipulation, arithmetic, relational and Boolean opera-

tions. A modest library of Lisp functions is also included. The price is $150, which includes source code and documentation. Contact WESTCOMP Software Development Group, 517 North Mountain Avenue, Suite 229, Upland, CA 91786, (714) 982-1738. **Reader Service Number 115.**

Useful and practical expert systems can be designed and implemented by users with no prior knowledge of AI programming with the ES/P Advisor, according to the vendor. The product, written in Prolog 2 for the IBM PC, gives the user the benefit of general purpose (and Prolog 2) interfacing, fast execution, windows, garbage collection and other features. There are no limits in knowledge base or number of rules that can be handled. The price is $895. Contact Expert Systems International, 1150 First Avenue, King of Prussia, PA 19406, (215) 337-2300. **Reader Service Number 117.**

STOK PILOT-PC, a language for the development of advanced training and control applications, is now available for the IBM PC. This is the only high level language with integrated instructions for the storage and manipulation of audio on a hard disk. Contact Stok Software, 17 West 17th St., New York, NY 10011, (212) 243-1444. **Reader Service Number 119.**

Super Pascal, a complete development system for the Commodore 64 and 128 computers, is available for $59.95 from Abacus Software, 2201 Kalamazoo S.E., P.O. Box 7211, Grand Rapids, MI 49510, (616) 241-5510. **Reader Service Number 121.**

Apple

A transportable 300 baud modem which clips on to the back of the Apple IIc and solves the problem of sudden disconnect when a call comes in on a "call waiting" line, has been announced by Prometheus Products. The price is $199.95. Contact the vendor at P.O. Box 4156, Fremont, CA 94539, (415) 490-2370. **Reader Service Number 123.**

The Mac 3.5 card provides Apple II users with a 3.6 MHz 65C02 processor that speeds up their Apple in some cases by as much as 3.5 times, though the vendor is careful to point out that for programs that contain most of their code in the auxiliary memory of the Apple IIe, e.g. Pascal 1.2, the speed advantages will be less impressive. Contact A.P.P.L.E Co-op, 290 S.W. 43rd St., Renton, WA 98055, (206) 251-5222. **Reader Service Number 125.**

A RAM disk for the Macintosh is available from Symmetry Corporation. Quickdisk requires a 512K Mac or XL and costs $34. Contact Symmetry at 3900 East Camelback Road, Suite 103-S, Phoenix, Arizona 85018 (602) 224-5944. **Reader Service Number 127.**

Hippopotamus Software has announced the availability of level 2 of its Hippo-C for the Macintosh. Level 1, which has been available for several months, is designed for use by educational institutions and the general public. It incorporates the familiar Macintosh interface and includes an editor, full C compiler, linker, symbolic debugger, standard C library, online tutorial, and shell command processor. Level 2 features an optimizing compiler, 68000 assembler, linker, librarian, full floating-point support, header files and a Unix-like command shell, access to over 500 toolbox routines, the sound channels, and serial ports. Level 1 retails for $149.95, and level 2 retails for $399.95. An upgrade kit for level 1 costs $250. There are no licensing fees for software produced using Hippo-C, and nonprotected versions are available for $25 more. **Reader Service Number 129.**

The same vendor also has introduced Hippo-Lock, a "business Data Security System" for the Mac, which scrambles the contents of a file once the user creates a keyword phrase using the DES standard. Hippo-Lock works with microdisks, hard disks, and over the AppleTalk office network, and it can be used to scramble electronic mail on MCI mail, Easy-Link, and Compuserve. Contact the vendor at 985 University Avenue, Suite 12, Los Gatos, California 95030 (408) 395-3190. **Reader Service Number 131.**

DeskTop Software has announced a solution for the problem of incompatibility of copy protection schemes and hard disk usage. The new system, which transfers the copy protection information to the hard disk, is available for all three of DeskTop's products: 1stBASE, 1stPORT, and 1stMERGE. Present users can obtain the new system at no charge. Contact the vendor at 244 Wall Street, Princeton, New Jersey 08540 (609) 924-7111. **Reader Service Number 133.**

IBM-PC

The Disc Interchange Service converts files written in one format to a different format. They have added the 3½ inch drive to their format conversion capabilities, bringing the number of formats supported to nearly 200. The following formats are available: DG/One, Kaypro 2000 and the IBM PC 2 (when its details become known), as well as the Hewlett-Packard HP-150. Contact the vendor at 481 Great Road, Acton, MA 01720, (617) 263-6001. **Reader Service Number 109.**

A software package which is designed to ensure the security of data in a shared computer environment without the use of encryption has been announced by AST Research. The Knight Data Security Manager offers passwords, copy protection utilities and a variety of file management features for the IBM PC and compatibles. A complete menu driven DOS interface is also provided, though access to DOS is still possible in the normal way. Contact the vendor at 2121 Alton Ave., Irvine, CA 92714, (714) 863-1333. **Reader Service Number 135.**

X-VIEW 86, which allows hardware and software experts to observe the internal operations of DOS applications software, has been introduced by McGraw Hill. It is available for immediate shipment and is priced at $59.95 plus shipping. X-VIEW 86 is a software analyzer and consists of a diskette, documentation and technical support for registered users. the product is available on a special 10 day examination basis, and the price is fully refundable, provided the protective seal on the diskette is unbroken. Contact the vendor at 8111 LBJ Freeway, Dallas, TX 75251, (800) 221-8439, or (800) 233-8439 in Texas. **Reader Service Number 137.**

CP/EM makes CP/M 80 emulation in software available for the IBM PC and compatibles. It is priced at $79.95. Contact the ICU Group, P.O. Box 10118, Rochester, NY 14610, (716) 425-2519. **Reader Service Number 139.**

C and Unix

Computer Innovations has announced version 2.3 of their Optimizing C86 compiler, whose most important new feature is source level debugging support. This allows popular debuggers such as Periscope, Pfix Plus and Atron to be used. The documentation is completely new. It is priced at $395, or $35 for an upgrade for exsiting C86 users. Contact the vendor at 980 Shrewsbury Avenue, Tinton Falls, NJ 07724, (800) 922-0169. **Reader Service Number 141.**

HEXPOSE is a "visual" binary file editor for Unix which can be used for patching object modules, repairing damaged files and verifying the results of I/O operations. For further information, contact Specialized Systems Consultants, P.O. Box 55549, Seattle, WA 98125, (206) 367-UNIX. **Reader Service Number 143.**

National Semiconductor has announced the production release of Genix 4.2, a port of Berkeley 4.2 BSD Unix for National's Series 32000 32 bit microprocessor family. The package includes a C compiler and porting tools for a VAX/4.2 host. A Pascal compiler is available as well. National also offers System V/ Series 32000, based on AT&T's Unix System V. Contact the vendor at 2900 Semiconductor Drive, Santa Clara, CA 95051 (408) 721-5000. **Reader Service Number 145.**

Software Architecture and Engineering has announced a line of embeddable expert system tools implemented in C. Initial releases of the Knowledge Engineering System II support the IF . . . THEN production rules for inferencing, and future releases will add frame-based pattern matching and statistical inferencing methods. Contact the vendor at 1500 Wilson Boulevard, Suite 800, Arlington, Virginia 22209 (703) 276-7910. **Reader Service Number 147.**

Software Research Associates has introduced its Test Coverage Analysis Tool (TCAT) for C, which the vendor claims will improve the quality of unit and system level testing by a factor of five or more. It identifies which parts of a program have been tested and which have not. TCAT/C is available for VAX/Unix, VAX/VMS, and IBM PC and compatibles. Prices range from $975 to $9000 depending on the configuration. SRA is prepared to port TCAT/C to almost any operating system on a custom basis. For further information, contact the vendor at 580 Market Street, San Francisco, California 94104 (415) 957-1441. **Reader Service Number 149.**

A training package for C has been announced by Computer Innovations, the developers of C-86. "Introducing C," which consists of a self-paced training manual and C interpreter, runs on an IBM PC (DOS 2.0) with 192K. The price is $95. Contact the vendor at 980 Shrewsbury Avenue, Tinton Falls, New Jersey 07724. **Reader Service Number 151.**

Smart/C is an integrated precompilation development environment for C; it consists of a syntax-directed editor integrated with a screen-oriented interpreter and the Migrator, which allows existing C programs to become "amenable" to the Smart/C environment. It runs under Unix System V, Berkeley BSD 4.2, Xenix, and MSDOS. Contact AGS Computers, Advanced Products Division, 1139 Spruce Drive, Mountainside, New Jersey 07092 (201) 654-4321. **Reader Service Number 153.**

Interactive/C is a full K&R development system that includes a command processor, full screen editor, source level debugger, and execution profiler. Its multiwindow multi-user interface permits debugging of full screen graphics on one or more CRTs with simultaneous display of source code, program output, and system status. It is compatible with Lattice C and lists for $395. Contact the vendor, IMPACC Associates, at P.O. Box 93, Gwynedd Valley, Pennsylvania 19437. **Reader Service Number 155.**

C-LINK is an application generator that, working with the vendor's S-Tran BASIC to C translator, allows a programmer to develop programs in C by writing in BASIC. C-LINK runs under Unix or Xenix and costs $695. For further information, contact the vendor, SMI, at 20720 South Leapwood Avenue, Carson, California 90746 (213) 538-8174. **Reader Service Number 157.**

DDJ

Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

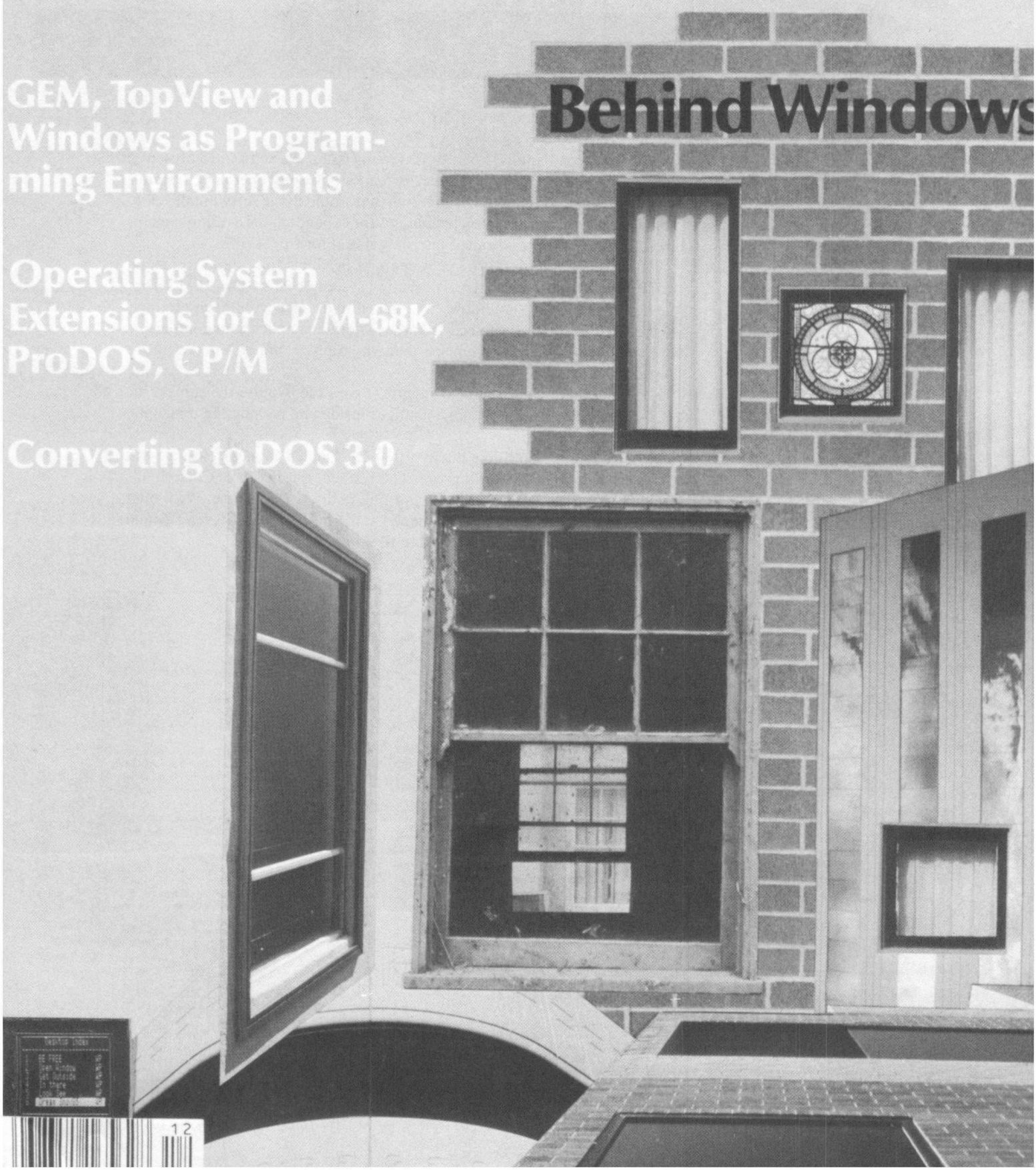
#110 DECEMBER 1985 $2.95 (3.95 CANAD

Behind Windows

GEM, TopView and Windows as Programming Environments

Operating System Extensions for CP/M-68K, ProDOS, CP/M

Converting to DOS 3.0



Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Managing Editor *Frank DeRose*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Vince Leone*
Typesetter *Jean Aring*

Production
Production Manager *Bob Wynne*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover Artist *Tom Upton*

Circulation
Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Kaminski*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Stephanie Barber*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*
Accountant *Marilyn Brown*
Adm. Coordinator *Kobi Morgan*

Advertising
Advertising Director
Shawn Horst (415) 424-0600
Advertising Sales
Walter Andrzejewski (617) 567-8361
Lisa Boudreau (415) 424-0600
Beth Dudas (714) 643-9439
Michele Beaty (317) 875-8093
Systems Manager *Ron Copeland*
Administrative Manager *Anna Kittleson*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0278-6508**

Customer Service: For subscription problems call: outside CA 800-321-3333; within CA 619-485-9623 or 566-6947. For book, back issue, or disk order problems call 415-424-1474.

Subscription Rates: $25 per year within the United States. Foreign subscription rates: $52.00 for airmail, $42.00 for surface mail. Foreign subscriptions must be pre-paid in U.S. Dollars, drawn on a U.S. Bank.

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 686-1520 TELEX: 620430 (WUI)

Entire contents copyright © 1985 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

ABC Audited Paid Circulation

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

December 1985
Volume 10, Issue 12

CONTENTS

In this Issue

This is our second annual operating systems issue. We've tried to present a variety of tools for extending the performance of several operating systems. We looked for techniques and programs that would make software development easier: a debugging tool, techniques for expanding the available memory for programs, a conditional instruction tool for CP/M-68K.

We also took a look at three commercial products that claim to extend the performance of the MS DOS operating system: DRI's GEM, IBM's TopView, and Microsoft's Windows. We talked to developers of each product, asking what programmers need to know to develop applications compatible with these operating environments and why a programmer might want to do such a thing.

Look closely at the issue next month; it won't look quite the same. Our art director has introduced some changes inside and outside the magazine that we think you'll like.

We don't think you'll like hearing that Dr. Dobb's Clinic is closing with this issue. We don't like telling you. The resident intern, Dave Cortesi, has gone off to pursue interests having nothing to do with software. We'll miss Dave's gentle wit. We'd like to reopen the Clinic, but we haven't found anyone yet to fill Dave's shoes, and until we do, we won't.

Next month: 68000 programming.

STATEMENT OF OWNERSHIP, MANAGEMENT, AND CIRCULATION

(Act of August 12, 1970, Section 3685, Title 39, United States Code)

1. Title of Publication: Dr. Dobb's Journal, Publication No. 307690.
2. Date of Filing: October 12, 1985.
3. Frequency of Issue: Monthly (12 issues, $25).
4. Location of Known Office of Publication: 2464 Embarcadero Way, Palo Alto, CA 94303.
5. Location of Headquarters of General Business Offices of the Publishers: 2464 Embarcadero Way, Palo Alto, CA 94303.
6. Names and Addresses of Publisher, Editor, and Managing Editor: Publisher, Laird Foshay, 2464 Embarcadero Way, Palo Alto, CA 94303. Editor, Michael Swaine, 2464 Embarcadero Way, Palo Alto, CA 94303. Managing Editor, Frank DeRose, 2464 Embarcadero Way, Palo Alto, CA 94303.
7. Owner: Otmar Weber, 2464 Embarcadero Way, Palo Alto, CA 94303; and C. F. Von Quadt, 2464 Embarcadero Way, Palo Alto, CA 94303.
8. Known Bondholders, Mortgages, and Other Security Holders Owning or Holding 1 Percent or More of Total Amount of Bonds, Mortgages or Other Securities: Markt & Technik, Hans-Pinsel-Strasse 2 8013 Haar bei Munich W. Germany.
9. Extent and Nature of Circulation:
   A. Total number of copies printed: Average number of copies each issue during the preceding 12 months: 41,693. Actual number of copies of single issue published nearest to filing date: 45,037.
   B. Paid Circulation. 1. Sales through dealers and carriers, street vendors, and counter sales. Average number of copies each issue during preceding 12 months: 8,707. Actual number of copies of single issue published nearest to filing date: 7,815. 2. Mail subscriptions. Average number of copies each issue during preceding 12 months: 26,568. Actual number of copies of single issue published nearest to filing date: 30,417.
   C. Total Paid Circulation. Average number of copies each issue during preceding 12 months: 35,275. Actual number of copies of single issue published nearest to filing date: 38,232.
   D. Free distribution by mail, carrier, or other means, samples, complimentary, and other free copies. Average number of copies each issue during preceding 12 months: 2,353. Actual number of copies of single issue published nearest to filing date: 3,305.
   E. Total distribution. Average number of copies each issue during preceding 12 months: 37,628. Actual number of copies of single issue published nearest to filing date: 41,537.
   F. Copies not distributed. 1. Office use, left over, unaccounted, spoiled after printing. Average number of copies each issue during preceding 12 months: 150. Actual number of copies of single issue published nearest to filing date: 150. 2. *Returns from news agents.* Average number of copies each issue during preceding 12 months: 3,915. Actual number copies of single issue published nearest to filing date: 3,350.
   G. Total. Average number of copies each issue during preceding 12 months: 41,693. Actual number of copies of single issue published nearest to filing date: 45,037.

I certify that the statements made by me above are correct and complete.

Laird Foshay, Publisher

Dr. Dobb's Journal

ARTICLES

Windowing Operating Environments — 28
by Michael Swaine
We talked to the developers behind GEM, TopView, and Windows to find out what these operating environments could mean to software developers. **Reader Ballot No. 192.**

BANKSWAP: A Banked Memory Debugging Tool for CP/M Plus — 34
by Albert S. Woodhull
CP/M Plus can address a megabyte, but debugging beyond the first 64K bank is a problem. **Reader Ballot No. 193.**

Adding a Copy Command to ProDOS — 54
by Shawn Day
COPY solves the problem of copying files between volumes without losing a BASIC program in memory. **Reader Ballot No. 194.**

CP/M-68K Conditional Batch Processing — 70
by Roger E. Donais
Efficient programming requires tools for automatically compiling, assembling, and linking with termination on error. IF.68K extends CP/M-68K in that direction. **Reader Ballot No. 195.**

ProDOS and the RAM Disk — 84
by Alfred Steele
Here's how one programmer developed a RAM-disk driver for ProDOS. **Reader Ballot No. 196.**

COLUMNS

Dr. Dobb's Clinic — 14
by D.E. Cortesi
The Clinic closes its doors. **Reader Ballot No. 190.**

C Chest — 16
by Allen Holub
Support routines for a new shell for MS DOS; An fgets that edits. **Reader Ballot No. 191.**

16-Bit Software Toolbox — 102
by Ray Duncan
Readers flame on Microsoft Assembler; notes on converting to MS DOS 3.0. **Reader Ballot No. 197.**

Mac Toolbox — 110
by John Bass
Concluding the MacSCSI interface series. **Reader Ballot No. 198.**

CP/M Exchange — 114
by W. E. Wilson
In honor of our focus on operating system extensions, our guest columnist tells of a product that makes CP/M a big operating system. **Reader Ballot No. 199.**

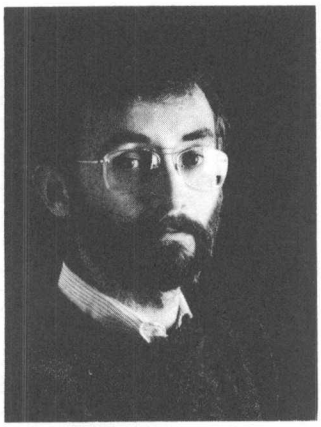
DEPARTMENTS

Editorial 6

Letters 8

Advertiser Index 128

EDITORIAL



It's been a litigious autumn in the software industry. At press time, Apple had forced DRI into changing the GEM Desktop; Apple lawyers were eyeing Atari and Microsoft; and Atari, anxious to get its software into ROM, was reportedly ready to challenge Apple. Apple's contention that its copyright protected not only the expression of algorithms in code but also the visual expression of the desktop (and implicitly that it had invented that desktop) had yet to be tested in court.

Adapso had been running ads attempting to educate software users about the criminal nature of unauthorized copying of software. Several software publishers had brought suit against customers to aid in the pedagogical process—all in reaction to a perceived loss of (potential) revenues and all very reminiscent of a campaign by the author of MITS BASIC a decade earlier. It wasn't a good tactic then to call your (potential) customers thieves and probably isn't now, but how are authors' and publishers' investments in software to be protected without alienating users?

Vault, a manufacturer of a copy-protection product, concluded that a Canadian company must have illegally disassembled Vault's code in order to produce its unlocking product; so Vault found a state in which it thought it could get a decision in its favor and sued. Vault lost, but one decision in a Louisiana court hardly dissipates the moral ambiguity that surrounds the entire copy-protection issue. Do we really want this escalating arms race between protectors and crackers?

The Vault case dealt with trade secrets. Copyright applies to works that are in some manner made public. No author of a book would object to someone analyzing his work in order to understand it—disassembling it, if you will. Trade secrets, on the other hand, are necessarily secret. Whom do trade secrets protect, and whom does copyright protect? Many software publishers treat the two protection methods as compatible, but does it really make sense that you can protect as a trade secret something protected under copyright law? As a book reviewer I could, under the fair-use interpretation of copyright law, excerpt and analyze a work of literature. Could I, as a software reviewer, disassemble and publish portions of commercial programs to show the workings of key algorithms?

These questions suggest, at the very least, that not everyone is of one mind regarding the protection of intellectual property. They also suggest something more alarming: that precedents are being set and opinions are being molded by bluster and threat, by cases that never come to court, and by venue-choosing ploys that take advantage of differing levels of understanding of the issues in different states. It shouldn't be that way.

You could do something about it. You probably know more about the issues than do most journalists, judges, or jurors. And I suspect you may have more empathy for the user than does the average software publisher, who often has stockholders to answer to. I urge you to make your expertise available to those making decisions with long-term implications. Educate lawyers and editors.

Become an expert witness for rational software law.

Michael Swaine

Michael Swaine



Small-C

Dear *DDJ*,

The Small-C problem reported by Gregor Owen (Letters, July 1985) has already been identified, fixed, and reported to Small-C users on my mailing list. It also appears in the "Small-C Update" article (*DDJ*, August 1985).

As it turns out, Owen's change does not address the real problem and introduces serious problems of its own. It is generally not a good idea simply to remove functions from a program even if they are referenced only once. Most programmers have reasons, other than the introduction of bugs, for placing even poorly documented functions in programs.

The function result( ) serves the vital role of causing the expression analyzer to note for binary operators whether or not the result is an address, and if so whether it is a character or integer address. Removing it will cause the analyzer always to assume the attributes of the left operand for the operator value. For example, "addr1 − addr2" should yield an integer result indicating the number of objects (bytes or words) between the two addresses. Without result( ), Small-C will think it has another address, and that has ramifications for the further process of expression evaluation.

The real problem was that the analyzer was losing track of the attributes of the lvalue for ?= operators when the right side evaluated to an address. The correction was to make hier1( ) (in CC31.C) remember two of the lvalue attributes for use at store time. This is done as follows:

1. define the local integer array
   lval3[2] after lval2[8].
2. between "if(k==0)..." and
   "if(lval[1])..." insert
   ```
   lval3[0] = lval[0];
   lval3[1] = lval[1];
   ```
3. change the call to store( ) to
   read
   ```
   store(lval3);
   ```

Owen's change appeared to work because in his test case the error he introduced caused the compiler to sidestep the original problem. The troublesome statement was:

```
lptr += r - bytes
```

where "r" was an integer and "bytes" was an array name (yields the array address). As mentioned above, the removal of result( ) would cause "r − bytes" to yield an integer result, thereby avoiding the original problem.

By the way, one might question the meaning of an integer minus an address. K&R point out on page 99 that such expressions are illegal.

Jim Hendrix
Rt. 1, Box 585
Oxford, MS 38655

Information Age Issues

Dear *DDJ*,

Please continue publishing articles like Dean Gengle's "Information Age Issues" (June 1985). Such ethical issues are too important to leave in the hands of a systems designer.

I must, however, take issue with Gengle's comment that "no one is talking about our rights to access information that is *not* private and that was collected legally at taxpayers' expense. Census data and Library of Congress information come to mind."

Your readers should know that both the American Library Association Government Documents Roundtable and the Special Libraries Association Government Relations Committee have indeed been talking about it. In addition, the Joint Committee on Printing of the Congress, which oversees the Government Printing Office, has been very concerned that material produced in machine-readable form by Federal agencies has not been made available to the GPO for distribution to the public through the depository libraries because it is not "printed."

Librarians protested strongly the distribution of census data in microfiche-only format because such data is not readily accessible to the poor or illiterate. Distribution on magnetic tape is a great service to those who can afford to have the tapes processed but may harm those who depend on such data for distribution of entitlement program benefits.

Write to your representatives and and urge them to support the efforts of the Joint Committee on Printing, the ALA, and SLA.

I'm not sure what Gengle is referring to by "Library of Congress information." He may be suffering from the misinformation provided by an episode of the television program "Whiz Kids" in which the elfin heroes broke into the Library of Congress computers and did a homework assignment. I doubt if there are any files at the LC, or anywhere else for that matter, that could produce the results obtained. In fact, the files at the LC are bibliographic, containing references to literature, but not full texts—at least not yet. Not only are these bibliographic files available to the public at the LC but they are available on magnetic tape to libraries and other institutions throughout the world. Many libraries located in this country access those records through OCLC, a bibliographic utility with over 11 million records. Some agencies do apparently try to hide

their information from the public, but the Library of Congress certainly does not.

Bruce B. Cox
Documents Librarian and
Automation Committee
Linda Hall Library
5109 Cherry Street
Kansas City, MO 64110

C Compiler Review

Dear *DDJ*,

I read with great interest your August 1985 review of MS DOS C compilers. I have two comments I would like to interject into the debate.

First, regarding the review of the documentation—this is always a touchy area, being rather subjective in impact—I am not convinced that a C compiler manual should include what is essentially a rehash of the standard K&R syntax chapters nor a rehash of the standard Unix manual's chapters 2 and 3. That is, unless a compiler and run-time library deviate from these accepted standards, I see no reason to repeat them. It only serves to confuse. As an example, I find the Lattice C documentation difficult to use because I generally expect it to be similar to the Unix manual. It isn't, and I always get lost trying to find something.

It is much better simply to include a "pointer" to the appropriate documentation with the caveat that any functions and/or syntactic notes included in the product's manual imply a deviation from the norm. Sort of like a "search path" if you will. One uses the Unix and K&R manuals unless the product's manual overrides them.

Second, the authors did not include as a criterion of judgement the number of other products that support a particular compiler. For instance, although Lattice C does not necessarily produce the tightest code nor run the quickest, it is important that one can obtain interfaces to major products. The Oracle DBMS includes a host-language interface (HLI) to Lattice C, and a HLI to dBASE II and III can be obtained from Lattice. Such interfaces are of great significance, and in these systems the choice of compiler will hinge entirely upon that single criterion. It is better, in other words, to put up with slightly slower execution than to have to generate in house an entire nontrivial HLI.

Otherwise, I was pleased with the review—it was obviously a strong effort. It is gratifying to this C and Unix nut to see the wide interest in these tools for programmers.

Brian Jay Wu
P.O. Box 203
Newbury Park, CA 91320

Dear *DDJ*,

When I heard that your magazine was going to do a review of C compilers and interpreters, I couldn't wait to read it.

The cover for the August 1985 issue claims that the review of C compilers is definitive. I think the review is marginally useful and generally misleading. Information important to the potential user was omitted. A list of considerations relating to the usefulness of the various C compilers follows:

1. Good luck to anyone who purchases the Microsoft compiler Version 3.0 without a hard disk! The four passes of the compiler total 290K, the CL.EXE driver takes 27K, the linker takes 41K, and the libraries needed total at least 120K. Add it up and you get a whopping 478K.

It is becoming common to run a compiler from inside your editor. COMMAND.COM needs about 18K and a programmer's editor may need 100K or more. We haven't even started to include a debugger, assembler, other libraries, object modules, or header files. In general, it would be nice to know the amount of space needed by the compiler, libraries, linker, and other assorted software the compiler needs to run.

2. The object-module format of the compilers was not mentioned. Microsoft compatibility may be an important issue for some people, especially those who buy libraries.
3. The assembler that comes with the Mark Williams compiler does not use Intel mnemonics.
4. The review mentions that the large model for Lattice allows disabling of pointer range checking, which avoids a wraparound on a 64K boundary when doing pointer arithmetic. Nowhere was it mentioned how perilous such an undertaking can be on a project of any size!
5. Several of the compilers support in-line assembly, which is extremely useful. I did not see any mention of the DeSmet or DRI compilers supporting that feature.
6. The DeSmet debugger, an excellent tool, wasn't even looked at!
7. The DeSmet compiler doesn't support the large memory model but does come with an overlay linker.
8. Although it can be useful to look at the size of the .EXE file produced by a compiler and linker, sometimes you want to know the sizes of the code and data produced by the compiler as well. Will the compiler tell you those statistics?
9. The reviewers didn't mention tradeoffs between compile/link time vs. speed of executable code (using an optimizer pass, for example).
10. The fact that Lattice and Computer Innovations automatically provide run-time stack-overflow checking was not mentioned.

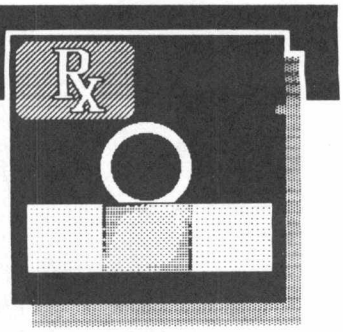
I would like to have seen more of an attempt to separate out evaluations of the compilers from the associated libraries. I couldn't care less about the performance of libraries, since for applications that require speed and efficiency I write my own routines.

I have a second reason for writing. It seems to me that people deciding which compiler to choose *might benefit* from feedback from programmers actually using C compilers. Since C compilers are used in a wide variety of applications and environments under MS DOS, readers would likely get different kinds of information than a reviewer would provide. Why not allocate space for a running column where readers can sound off about their favorite compiler and give reasons why they like it?

If you readers like that idea, write to *DDJ* about the compiler you like to work with, *mentioning* aspects that haven't been covered. *[We would welcome this input.—Ed.]* Include your work environment (network, single-user, etc.); machine; any idiosyncracies or work requirements on programming style; and why the compiler you work with is the greatest thing since *DDJ*.

Tom Hogan

DR. DOBB'S CLINIC



by D. E. Cortesi

Clinic to Close

The Clinic opened for business in May 1981. Now, four and one half years later, career changes compel me to close it down. The landlord will soon install new and interesting enterprises on the premises, but I thought I'd use our last meeting to look back and say thanks.

Two things come to mind when I reflect on the column. The first is trivial: that damned editorial We. I can't say why I hobbled myself with that foolish affectation in the first column nor why I didn't drop it in the second, but I did and then I didn't and from then on I felt compelled to continue an established style.

The strongest impression I retain is of the quality of the *DDJ* readership. As originally conceived, this column was to be written mostly by its readers with the Intern editing and supplying continuity, like a radio talk show in slow motion. It never quite worked out; you readers never provided enough original material to make a column.

But you responded when your interest was caught—my goodness how you responded! I posed Don Taylor's highly abstruse problem in the mathematics of graphics and a dozen of you sent detailed, multipage discussions heavy with matrix notation. I put up David Ross's BASIC program and 20 of you sent line-by-line discussions of what it did and how to do it better. Although the column was never written by readers in the way originally planned, you did supply a lot of material, correct a lot of my mistakes, and lend me your enthusiasm over the years.

Many of those who wrote were made into co-columnists and quoted in the various issues by name. They are named once more, with the issues they graced by their presence, in the following list. For every one of those mentioned there were often two, three, or a dozen more who had the same insight but wrote later or in a less quotable way. My personal thanks to all who wrote—it's been a privilege to associate with such a classy bunch. You folks keep up the good work, you hear?

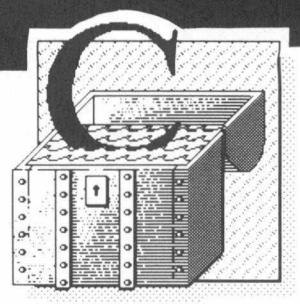
Allison, Dennis	8/81, 10/81
Amelang, Loren	2/83
Asay, Matt	10/81
Ashley, Allen	7/81, 10/81
Backus, Rex	4/85
Bacon, Phillip	7/81
Baenziger, Peter	8/84
Barker, L.	7/81, 10/81, 11/81, 10/82, 10/83
Barker, David	11/85
Barr, Michael	5/84, 8/85
Bates, Dan	9/81
Becus, Georges A.	9/81
Behler, Alan	5/83, 11/83
Bomberger, Alan	10/83
Brickell, Gavin	10/82
Briggs, R. C.	8/84
Brown, Russel	8/81
Butler, Edmund	7/81
Cage, Thomas W.	2/85, 6/85
Caldwell, Taylor	8/81
Canniff, Paul	4/85
Carr, Thomas E.	9/81
Carter, Everett	8/85
Castle, Pat	2/85
Chrapkiewicz, Thomas	8/84
Clark, David	2/84
Crowfoot, Norman	9/81
Dale, Gary	1/85
Davis, Charles	5/85
Davis, M. Patton	10/81
Dayhoff, Edward S.	11/81
Dempsey, J.	3/83
Dinsmore, Perry	11/84
Dodd, Michael S.	11/85
Driscoll, Michael	9/81
Dugan, Dan	4/82
Edgar, Gerald	7/81
English, Glenn	11/85
Enterline, Tom	8/84
Falk, Richard	3/84
Farmer, Wayne	8/81, 10/81
Fay, Jack	11/85
Feign, David	8/84
Floyd, Chet	9/84, 2/85
Fordham, Malcom	4/84
Fritz, Randolph	5/83, 6/83
Gardner, Calvin L.	8/84
Glassey, C. Roger	12/81
Goldman, Oscar,	5/84
Goodman, Bob	7/81
Grant, Steve	8/84
Greenlaw, Richard	10/81
Gunn, John Paul	2/85
Hahn, Harvey	5/83
Hammond, Nick	10/82, 6/83, 10/83, 10/85
Head, Gene	12/81
Hills, Terence	11/81
Hoats, David L.	12/81
Hoffman, Lee H.	2/85
Hole, Dr. William T.	2/85
Hoolko, Bob	7/81
Howell, Edgar	4/82
Howell, Jim	5/82, 2/84, 4/85
Hutchinson, Aubrey	2/83–6/83
Hutchinson, W. G.	4/82
Irving, Dwight	11/83
Jackson, Terry	2/85
Jensen, John Thayer	6/83
Johnson, Bruce R.	2/85
Johnston, Bill	12/81
Jones, James	4/85
Kellogg, David	6/83
Kendall, Bruce	8/82
King, David	2/84
Knipp, Ernest	10/82, 2/83
Koehler, W. B.	11/83
Komusin, Bruce	4/82, 5/82
Lane, Tom	7/82
Laurino, James	1/82, 7/83
Leonard, M. David	7/81
Levy, David	7/82

Lurie, Robert	10/81
McDaniel, C. T.	7/81
McDermott, Joseph	3/84
McKeon, Brian	4/85
McLanahan, David	1/84, 4/84, 8/84
McWorter, William A.	9/81, 5/83
Manay, Martin	10/81
Manns, Larry	11/85
Marchand, Fred	1/85
Marshall, Charles	3/84
Martin, Charles	2/85
Martin, John H.	8/81
Martin, Robert	10/81
Medlock, C. W.	5/84
Mesirov, Dick	3/84
Metzenthen, Bill	8/84
Meyer, Mike	10/84
Montgomery, Owen	5/84
Moore, Brian	3/83
Morgan, John	10/81
Neuhaus, Prof. H. J.	8/83
Newberry, Steve	2/84
Norling, Richard D.	11/84
Northrup, Mark	11/81
Novosielski, Gary P.	11/81
Nowell, Scott	11/81
Oakland, Richard L.	11/85
Oster, David	4/83
Palmer, John	10/82
Pasky, Robert	8/82
Pearson, Peter K.	8/85
Pierce, John	7/81
Pirko, Robert	3/83, 6/83, 7/83
Poli, Ray	4/82
Prince, Dr. Timothy C.	2/85
Pritchard, Steve	9/81
Pulver, Dusty	5/82
Purdum, Jack	2/84
Rivers, Wayne	1/85
Roberts, Glenn F.	7/84, 8/84
Rockwell, Charles	10/81
Ross, David	6/85, 8/85
Rouman, Jay S.	11/85
Rutiser, William A.	8/84
Saal, Harry	7/81
Sabin, Joseph	3/84, 2/85
Sackheim, Andrew	4/82
Safier, Orin	11/85
Salemma, A.	1/85
Samuel, Dan	4/82
Sand, Paul	1/85
Scardelis, Jim	4/84
Schafer, Dale	7/81
Schemm, John B.	2/85
Schuetz, Chuck	4/85
Schwab, Leonard	9/84
Semos, Steve	11/81
Shapin, Ted	4/84
Shaw, David A.	1/82, 4/82
Shaw, Jim	5/82
Shochat, David	11/85
Shuman, Dave	11/85
Smith, Billy	11/81
Smith, Burks	10/82, 2/83
Smith, Roger	8/85
Snow, J. R.	8/81
Stanek, Mitchell	9/81
Sullivan, Dave	4/84
Swearingen, Roger V.	2/85
Taylor, Bob	6/83
Taylor, Don	4/82, 8/82, 5/83
Thier, U.	7/84
Tilton, David S.	8/84, 10/84, 1/85, 4/85
Throckmorton, Jerry	10/81
Trout, Dana	10/82
Wagner, R. C.	7/84
Watzman, Barry	8/85
Wendel, J. G.	8/81
Wilkinson, Bill	7/82
Wilson, Ed	7/81
Wochnik, Michael	8/81
Wright, Don	9/82
Vieau, Dean	9/81

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

C CHEST



A New Shell for MS DOS, Part 1: IBM Cursor Control and an Fgets that Edits

by Allen Holub

Back in March we presented a new root module for the CP/M-80 version of the Aztec C compiler. Since then, several people have written to say that the proper place for wildcard expansion, pipes, redirection, and the like is in the shell (i.e., the command interpreter), not in the C root module. There's no point in bloating the size of your programs when you don't have to. I sympathize with this attitude, but frankly, I don't want to write a new CCP for CP/M. Digital Research just makes this process too hard.

MS DOS, however, is another matter. The command interpreter, command.com, is just a program and can be replaced with any other program. Moreover, the DOS interface library provides much more in the way of system level functions than does CP/M. Most important, DOS Version 2 and higher supports an exec function that allows one program to load and execute a second program.

Because most of my gripes about MS DOS are really aimed at the behavior of command.com, it seemed to me that writing a new shell for MS DOS would be more productive than complaining for the rest of my life about the features I didn't have. C is, after all, a language remarkably well suited for writing operating systems. So, over the next few months we will look at a *new shell for MS DOS*.

I've implemented most of what I like about the the Unix C and Bourne shells: command line wild card expansion, redirection, history, shell variables, a reasonable language for writing shell scripts (.bat files under MS DOS) and the lot. I've also taken the opportunity to add a couple of features that I've often wished were present in the Unix shells, such as an interactive command line editor.

String Routines

We'll start looking at the shell itself next month. This month we're going to look at several support routines. As usual, these are all designed to be useful in their own right. They make up five sets. The first three listings are routines for string manipulation. They are small, but useful. Next( ) (Listing One page 19) gets from a string the next object separated by a delimiter. Its calling syntax is:

```
char *next( linep, delim, esc )
char **linep;
int delim, esc;
```

Linep is the address of a pointer that points at the first character of the string to be processed. Delim is the delimiter that separates objects. Esc is an escape character. The routine skips leading white-space (a space or tab) and remembers the position of the first non-white character. It then looks for a delimiter not preceded by an escape character and replaces the delimiter with a '\0.' It returns a pointer to the remembered non-white character and it modifies *linep to point to the first character beyond where the delimiter used to be. It won't modify *linep to point past the end of the string. If delim is a white character, leading white space won't be skipped.

Listing Two (page 19) and Listing Three (page 19) are two routines for copying. Cpy(dest,src) is functionally the same as strcpy( ). It copies the source string (src) to the destination string (dest). There is, however, one difference: whereas strcpy returns its first argument, cpy returns a pointer to the NULL at the end of the destination string. This allows us to add onto the destination string without first having to scan through the string to find its end. The second copy routine is cptolower(dest,src). Cptolower works just like cpy( ), except all uppercase characters are mapped to lower case as they are copied.

Moving the Cursor and Writing Characters

Listing Four (page 20) is a collection of low-level routines for cursor manipulation and character output for the IBM PC. These routines use direct ROM BIOS calls, and so are much faster than the usual output routines, which use the normal DOS functions[1]. Moreover, they let you move the cursor around without having to install the ANSI.SYS driver, which is excruciatingly slow. IBM has been very good about maintaining the ROM BIOS interface over different versions of the operating system. As long as you access the BIOS via the correct interrupt mechanism, the routines are portable.

I've used the int86( ) function from Lattice to generate the system video interrupt. Microsoft C has an identical function, so these routines will port to Microsoft C without any problems. The dos( ) function printed in this column in July 1985 can also be modified quite easily to generate a video interrupt. Just replace the INT 21H instruction on line 68 with an INT 10H instruction.

Listing Four contains the following routines:

getpage( )

This routine gets the currently active video page. When you are in text mode, several pages are available for writing, though only one is displayed at any given time. Getpage( ) returns the number of the page now being displayed. This information is needed by some of the other routines.

cursize(top, bot)
This routine adjusts the cursor size. Every character is represented on the display by a fixed number of scan lines. *Cursize( )* causes the cursor to extend from the top scan line indicated to the bottom line indicated. For example, the standard monochrome display uses 14 lines numbered 0–13, the color adapter uses 8, numbered 0–7. So, on the color adapter, a normal underline cursor can be set up with the call cursize(6,7). You can make a large block cursor with the call cursize(0,7). Cursize(7,0) creates a two-part cursor, one line over the letter and a second line under the letter. Cursize(8,8) will make the cursor disappear.

posn = gcur(pagenum);
int pagenum;
short posn;
This routine returns the position of the cursor on the indicated video page. The position is returned in a single 16 bit short; the row number is in the top byte and the column number is in the bottom byte.

scur(posn, page)
short posn;
This modifies the cursor to rest at posn, which may be a value returned from gcur( ).

posn = getcur( );
setcur(posn);
These routines work just like scur( ) and gcur( ) except that they access the page currently being displayed.

wchar(c)
int c;
This routine writes a single character to the screen. It is dramatically faster than putchar( )—at least the one that comes with the Lattice compiler. Wchar( ) moves the cursor as it prints, just like putchar( ). However, the only control codes it recognizes are carriage return (\r), line feed (\n), bell (\007), and backspace (\b), so don't expect it to expand tabs. Also, \n is not interpreted as a "newline," but as a line feed. That is, it will get you to the current column on the next line. Use \r to get to the left edge of the current line.

wstr(s, move)
char *s;
int move;
This routine prints an entire string to the screen. If move is true, the cursor ends up positioned just past the end of the printed string. Otherwise, the cursor remains over the first character of the string. Wstr( ) uses wchar( ) as its output function.

An Fgets( ) that Edits

Listing Five (page 20) is an editing input function. Let me point out at the start that this function is general purpose in nature, even though it's written specifically for an IBM PC. It's a relatively simple matter to modify the routines for any terminal that has an addressable cursor. The primary access routine in this module is

```
char *efgets(buf, maxline, fp)
char *buf;
int maxline;
FILE *fp;
```

This is functionally similar to fgets( ), though there are several major differences. First, a pointer to the end (instead of to the start) of the loaded buffer is returned on success. A NULL is still returned on end of file. Second, line continuation is supported. If a line ends with a backslash (\), the backslash is deleted and the line is concatenated with the next line. The major differences from fgets( ) are apparent when fp is set to stdin. In this case several interactive editing functions are supported:

- LEFT-CURSOR (hit the left cursor key) moves the cursor to the left without erasing anything.
- RIGHT-CURSOR moves the cursor to the right.
- ^LEFT-CURSOR (hold down the CTRL key and hit the left cursor key) positions the cursor at the beginning of the previous word.
- ^RIGHT-CURSOR positions the cursor at the beginning of the next word.
- HOME positions the cursor at home position on the current line. Home is defined as the position of the cursor when the routine was entered. For example, if the cursor was in column 5 when efgets( ) was called, HOME puts it back in column 5. In fact, in this example there's no way to get the cursor into columns 1–4 from efgets. If a line is continued with a \<CR> combination, the cursor is moved to the HOME position of the next line, not to the left edge of the screen.
- END positions the cursor just after the rightmost character on the line.
- ESC is for an abort. The buffer is cleared, but the characters on the screen are not erased. The routine returns immediately. –1 is returned when an ESC is encountered.
- DEL deletes the character on which the cursor is resting and closes up the rest of the line to fill the gap. (i.e., if you delete the X in the string "aaaaXoooo," you end up with "aaaaoooo" on the screen, rather than "aaaa oooo").
- ^H is a destructive backspace. It moves the cursor left one space and then deletes that character, closing up the line to fill the gap.
- ^X erases the entire line and clears the buffer. However, efgets( ) doesn't return.

• ^M (CR) or Enter (LF) positions the cursor at the left edge (not the home position) of the next line and cause efgets( ) to return to the caller. A pointer to the end of the buffer (to the \0) is returned on success, NULL on end of file.

Typing any printing character causes that character to be printed at the current cursor position and the cursor to move right one space. Typing anything else has no effect. The cursor is never allowed to go past the end of the buffer, as specified by the parameter maxline in the efgets( ) call. However, when in editing mode, the cursor is not allowed to go past the end of the current line, even if maxline is longer (the bell will ring if you try). In this case, you can get to the next line with a \<CR>, but you can't edit anything on the previous line.

Conclusion

So that's the beginning. Next month we'll add some more routines to the pile and incorporate them into a simple MS DOS shell. In the following month we'll add various capabilities to that shell.

[*IBM PC readable versions of the listings for the entire shell and for MS DOS versions of various Unix utilities (e.g., grep, ls) will be made available through* DDJ *in the next couple of months. Watch this column for more details.—ed*]

Notes

[1] An excellent description of the IBM BIOS routines and how to use them is in *The Peter Norton Programmer's Guide to the IBM PC*, by Peter Norton (Bellevue: Microsoft Press: 1985).

DDJ

(Listing begins at right)

C Chest (Text begins on page 16)

Listing One

```
 1: /*      NEXT.C:         Skip to the next delimiter seperated object
 2:  *
 3:  */
 4:
 5: #define iswhite(c) ( (c) == ' ' || (c) == '\t' || (c) == '\n' )
 6:
 7:
 8: char    *next( linep, delim, esc )
 9: char    **linep;
10: {
11:      /* Linep is the address of a character pointer that points to
12:       * a string containing a series of delim seperated objects.
13:       * Next will return a pointer to the first non-white object in
14:       * *linep, replace the first delimiter it finds with a null, and
15:       * advance *linep to point past the null (provided that it's not
16:       * at end of string). Ø is returned when an empty string is passed
17:       * to next(). White space may be used as a delimiter but
18:       * in this case white space won't be skipped. A delimiter preceded
19:       * by "esc" is ignored. Quoted strings are copied verbatim.
20:       */
21:
22:          register char   *start, *end ;
23:          int             inquote = Ø;
24:
25:          if( !**linep )
26:                  return Ø;
27:
28:          start = *linep;
29:
30:          if( !iswhite(delim) )
31:                  for( ; iswhite(*start) ; start++ )
32:                          ;
33:
34:          for( end = start; *end && (*end != delim || inquote) ; end++ )
35:          {
36:                  if( *end == esc  &&  *(end+1) )
37:                          end++;
38:
39:                  else if( *end == '"' || *end == '\'' )
40:                          inquote = ~inquote;
41:          }
42:
43:          if( *end )
44:                  *end++ = '\Ø';
45:
46:          *linep = end;
47:          return start;
48: }
```

End Listing One

Listing Two

```
 1: char    cptolower( dest, src )
 2: char    *dest, *src;
 3: {
 4:          /*      Copy src to dest, mapping all upper case letters to lower
 5:           *      case. src and dest may be the same.
 6:           */
 7:
 8:          for(; *src ; src++ )
 9:                  *dest++ = ('A' <= *src && *src <= 'Z')
10:                                          ? *src + ('a'-'A') : *src ;
11:          *dest = '\Ø' ;
12:          return dest  ;
13: }
```

End Listing Two

Listing Three

```
 1: char    *cpy( dest, src )
 2: char    *dest, *src;
 3: {
 4:          /*      Works like strcpy but returns a pointer to the new end
 5:           *      of string (ie. to the null).
 6:           */
 7:
 8:          while( *src )
 9:                  *dest++ = *src++ ;
10:
11:          *dest = Ø;
12:          return dest;
13: }
```

End Listing Three

C Chest (Listing continued, text begins on page 16)

Listing Four

```
 1: #include <stdio.h>
 2: #include "/lc/dos.h"
 3:
 4: /*   VIDBIOS.C        Various cursor and i/o routine using
 5:  *                    the bios interrupts (see below for greater detail):
 6:  *
 7:  *          Copyright (C) 1985 Allen I. Holub. All rights reserved.
 8:  *
 9:  *   Externally accessable routines:
10:  *
11:  *    int   getpage ()                        Get active video page #
12:  *    void  cursize (top,bot)                 Set cursor size
13:  *    void  scur    (posn, page) short posn;  Set cursor position
14:  *    short gcur    (pagenum)                 Get cursor position
15:  *    void  setcur  (posn) short posn;        Set cur pos on current page
16:  *    short getcur  ()                        Get cur pos from cur page
17:  *    void  wchar   (c)                       write a single character
18:  *    wstr  wstr    (s, move) char *s;        write a string
19:  *
20:  */
21:
22: #define VIDEO_INT      0x10    /*      Video interrupt                 */
23:
24: #define CUR_SIZE       0x1     /*      Set cursor size                 */
25: #define SET_POSN       0x2     /*      Modify cursor posn              */
26: #define READ_POSN      0x3     /*      Read current cursor posn        */
27: #define WRITE_TTY      0xe     /*      Write character & move cursor    */
28: #define GET_VMODE      0xf     /*      Get current video mode & disp pg */
29:
30: /*-------------------------------------------------------------------------*/
31:
32: static union REGS       Regs;
33:
34: /*-------------------------------------------------------------------------*/
35:
36: int     getpage()
37: {
38:         /*      Returns the currently active display page number
39:          */
40:
41:         Regs.h.ah = GET_VMODE;
42:         int86( VIDEO_INT, &Regs, &Regs );
43:
44:         return (int) Regs.h.bh ;
45: }
46:
47: /*-------------------------------------------------------------------------*/
48:
49: cursize( top_line, bot_line )
50: {
51:         /*      Scan lines are numberd 0 at the top and 7 at the bottom.
52:          *      if the two are reversed you'll get a 2 part cursor.
53:          *      Top_line determines the position of the top scan line
54:          *      of the cursor, bot_line is the bottom. A normal cursor
55:          *      can be created with cursize(6,7). Cursize(0,7) will
56:          *      fill the entire area occupied by a character. Cursize(0,1
57:          *      will put a line over the character rather than under it.
58:          */
59:
60:         Regs.h.ch = top_line ;
61:         Regs.h.cl = bot_line ;
62:         Regs.h.ah = CUR_SIZE ;
63:         int86( VIDEO_INT, &Regs, &Regs );
64: }
65:
66: /*-------------------------------------------------------------------------*/
67:
68: scur( posn, pagenum )
69: short   posn;
70: {
71:         /*      Modify current cursor position. The top byte of "posn"
72:          *      value holds the row, the bottom by the column.
73:          *      Pagenum is the video page number.
74:          */
75:
76:         Regs.x.dx = posn     ;
77:         Regs.h.bh = pagenum ;
78:         Regs.h.ah = SET_POSN ;
79:         int86( VIDEO_INT, &Regs, &Regs );
80: }
81:
82: short   gcur( pagenum )
83: {
84:         /*      Get current cursor position. The top byte of the return
85:          *      value holds the row, the bottom by the column.
86:          *      Pagenum is the video page number.
87:          */
88:
89:         Regs.h.bh = pagenum  ;
90:         Regs.h.ah = READ_POSN ;
91:         int86( VIDEO_INT, &Regs, &Regs );
92:
93:         return( Regs.x.dx );
```

```
 94: }
 95:
 96: /*      Setcur and getcur work just like scur and gcur except that they
 97:  *      access the current video page.
 98:  */
 99:
100: setcur( posn )
101: short   posn;
102: {
103:         scur( posn, getpage() );
104: }
105: short   getcur()
106: {
107:         return gcur( getpage() );
108: }
109:
110: /*------------------------------------------------------------------------*/
111:
112: wchar( c )
113: {
114:         /*      Write a character to the screen in TTY mode. Only normal
115:          *      printing characters, BS, BEL, CR and LF are recognized.
116:          *      The cursor is automatically advanced and lines will wrap.
117:          */
118:
119:         Regs.h.bl = 0;          /* Use current color */
120:         Regs.h.al = c;
121:         Regs.h.ah = WRITE_TTY ;
122:         int86( VIDEO_INT, &Regs, &Regs );
123: }
124:
125: wstr( str, move_cur )
126: char    *str;
127: {
128:         /*      Write a string to the screen in TTY mode. If move_cur is
129:          *      true the cursor is left at the end of string. If not
130:          *      the cursor will be restored to its original position
131:          *      (before the write).
132:          */
133:
134:         register short  posn;
135:
136:         if( !move_cur )
137:                 posn = getcur();
138:
139:         while( *str )
140:         {
141:                 Regs.h.bl = 0 ;
142:                 Regs.h.al = *str++ ;
143:                 Regs.h.ah = WRITE_TTY ;
144:                 int86( VIDEO_INT, &Regs, &Regs );
145:         }
146:
147:         if( !move_cur )
148:                 setcur( posn );
149: }
150:
151: /*------------------------------------------------------------------------*/
152: #ifdef DEBUG
153:
154: main()
155: {
156:         cursize( 0, 7 );
157:         wstr( "The large cursor should be on the 'T'", 0 );
158:         getchar() ;
159:
160:         cursize( 6, 7 );
161: }
162:
163: #endif
```

End Listing Four

Listing Five

```
 1: #include <stdio.h>
 2:
 3: /*      EFGETS.C        An editing version of efgets. Recognizes \<CR>
 4:  *                      line termination and supports editing when input
 5:  *                      is from stdin.
 6:  *
 7:  *       Copyright (C) 1985 Allen I. Holub. All rights reserved.
 8:  *
 9:  *----------------------------------------------------------------------
10:  *
11:  * Externally accessable routines:
12:  *
13:  * void  ptail   (bp, end, move )     Print string from bp to end. Move
14:  *                                    cursor to end if move is true.
15:  * char  *egets  (start, bufsize)     Get a string from stdin w/ editing.
16:  *                                    get at most bufsize-1 chars.
17:  * char  *getl   (buf, maxline, fp)   like fgets but returns pointer to
18:  *                                    end of input string on success.
19:  * char  *efgets (buf, maxline, fp)   like getl but uses egets for
20:  *                                    standard input rather than getc.
```

(Continued on next page)

C Chest (Listing continued, text begins on page 16)

Listing Five

```
 *
 *              FILE *fp;
 *              char *buf, *bp, *end;
 *              int  move, bufsize, maxline;
 */

extern int   getpage ();        /* Sources in vidbios.c                 */
extern void  setcur  ();        /*              "                       */
extern short getcur  ();        /*              "                       */
extern void  wchar   ();        /*              "                       */
extern int   dos     ();        /* Part of the Lattice standard library*/
extern void  movemem ();        /*              "                       */

/*      Values returned from DOS when cursor keys etc. are hit:
 */

#define _LEFT           75
#define _RIGHT          77
#define _CTL_LEFT       115
#define _CTL_RIGHT      116
#define _INS            82
#define _DEL            83
#define _HOME           71
#define _END            79

/* The above are mapped as follows by getkey
 */

#define LEFT            0x80
#define RIGHT           0x81
#define CTL_LEFT        0x82
#define CTL_RIGHT       0x83
#define INS             0x84
#define DEL             0x85
#define HOME            0x86
#define END             0x87

#define BDOS_IN   8     /* raw (non echo) input function        */

#define CNTL_C  0x03            /*  ^C  */
#define CNTL_Z  0x1a            /*  ^Z  */
#define BEL     0x07            /*  ^G  */
#define ESC     0x1b            /*  ^[  */
#define CAN     0x18            /*  ^X  */


/*-----------------------------------------------------------------------*/

static int      getkey()
{
        /*      Return a key from the keyboard. Keys are gotten in raw
         *      input mode and mapped as specified above if necessary.
         */

        register int    c;
        static   int    ateof   = 0;

        if( ateof )
                return EOF;

        if( !(c = bdos(BDOS_IN)) )      /* Special function key */
        {
                switch( bdos(BDOS_IN) )
                {
                case _LEFT:             return( LEFT      );
                case _RIGHT:            return( RIGHT     );
                case _CTL_LEFT:         return( CTL_LEFT  );
                case _CTL_RIGHT:        return( CTL_RIGHT );
                case _INS:              return( INS       );
                case _DEL:              return( DEL       );
                case _HOME:             return( HOME      );
                case _END:              return( END       );
                default:                return( NULL      );
                }
        }
        else if( c == '\r' )                    /* map ENTER key to '\n' */
        {
                return( '\n' );
        }
        else if( c == CNTL_C || c == CNTL_Z )
        {
                ateof = 1;
                return EOF;
        }
        else
                return c;

}

/*-----------------------------------------------------------------------*/

ptail( bp, end, move )
```

```
113: register char   *bp, *end;
114: {
115:         /*      Print out all characters between bp and end (inclusive)
116:          *      without modifying the current cursor position. If move
117:          *      is zero the cursor will not change position, otherwise
118:          *      the cursor will be left pointing to the character
119:          *      referenced by end.
120:          */
121:
122:         register short  posn;
123:
124:         if( !move )
125:                 posn = getcur();
126:
127:         for( ;  *bp && bp <= end;   wchar(*bp++) )
128:                 ;
129:
130:         if( !move )
131:                 setcur( posn ); /* put cursor under first character */
132:         else
133:                 wchar('\b');    /* put cursor under last character  */
134: }
135:
136: /*---------------------------------------------------------------------*/
137:
138: char    *egets( start, bufsize )
139: char    *start;
140: {
141:         /*      Get a string with editing. If bufsize is wider than your
142:          *      screen, strange things will happen when you try to use
143:          *      the editing functions. If you access this function
144:          *      via fgetline() then it will input longer lines in 78
145:          *      character chunks and \<CR> can be used to extend a line.
146:          *
147:          *      ^H or BACKSPACE      Destructive backspace, close up
148:          *                           remainder of string to fill hole.
149:          *      LEFT   CURSOR        Non-destructive backspace
150:          *      RIGHT  CURSOR        Move right one character.
151:          *      ^LEFT  CURSOR        Left  to previous word or line start
152:          *      ^RIGHT CURSOR        Right to next word or line end
153:          *      HOME                 Left edge of line
154:          *      END                  Right edge of line
155:          *      CR or LF             Terminate line.
156:          *      ^X                   Erase entire line but don't return.
157:          *      ESC                  Return a null string immediatly.
158:          *      DEL                  Delete a current cursor position and
159:          *                              close up to fill hole.
160:          *      Any printing character  Enter that char at cursor posn
161:          *      Anything else           Ring the bell.
162:          *
163:          *      The bell will also ring if you try to move the cursor
164:          *      past either the left or right edges of the buffer.
165:          *      Return a pointer to the end of string normally, return 0
166:          *      on EOF and return -1 when ESC is encountered.
167:          */
168:
169:         register char  *bp;     /* Points at current cursor position    */
170:         register char  *end;    /* Points at largest possible cur pos.  */
171:         register char  *maxbp;  /* Points at rightmost char on line     */
172:         register int   c ;      /* Current character.                   */
173:         short          home;    /* place to remember the leftmost cursor
174:                                  * position.
175:                                  */
176:
177:         end     = start + (bufsize-1) ;
178:         *end--  = '\0' ;
179:         bp      = start;
180:         maxbp   = start;
181:
182:         /*      Fill the entire buffer with spaces
183:          */
184:
185:         home = getcur();                /* Get the current cursor   */
186:                                         /* position.                */
187:
188:         if( *bp )                       /* If the buffer isnt empty */
189:         {                               /* print out its contents   */
190:                 while( *bp )            /* and set maxbp to point   */
191:                     wchar( *bp++ );     /* at the previous end of   */
192:
193:                 maxbp = bp;             /* string.                  */
194:                 setcur( home );
195:         }
196:
197:         for( ; bp <= end ; *bp++ = ' ' )   /* and then fill the rest   */
198:                 ;                          /* of it with spaces.       */
199:
200:         bp = start;
201:
202:         /*      bp points into the bufer at the current cursor location.
203:          *      end points at the righmost place that the cursor movement
204:          *      commands can get us. There is actually one more place
205:          *      in the buffer. Get the line:
206:          */
207:
208:         while(  (c = getkey()) != '\n'  &&  c != EOF )
209:         {
210:                 switch( c )
```

(Continued on next page)

C Chest (Listing continued, text begins on page 16)

Listing Five

```
211:                {
212:                case LEFT:              /* Non-destructive backspace    */
213:
214:                        if( bp > start )
215:                        {
216:                                wchar( '\b' );
217:                                --bp;
218:                                break;
219:                        }
220:
221:                        wchar( BEL );
222:                        break;
223:
224:                case '\b':              /* Destructive backspace        */
225:
226:                        if( bp <= start )
227:                        {
228:                                wchar( BEL );
229:                                break;
230:                        }
231:
232:                        wchar( '\b' );
233:                        --bp;
234:
235:                        /* fall through to delete case */
236:
237:
238:                case DEL:           /* Delete character and close up    */
239:
240:                        if( bp >= maxbp )       /* nothing to delete    */
241:                                break;
242:
243:                        movmem( bp+1, bp, maxbp-bp );
244:                        *maxbp = ' ';
245:                        ptail( bp, maxbp-- , 0);
246:                        break;
247:
248:                case CTL_LEFT:      /* Cursor to start of previous word */
249:
250:                        if( bp > start )
251:                        {
252:                                do {
253:                                        --bp;
254:                                        wchar( '\b' );
255:
256:                                } while( bp > start && *bp == ' ' );
257:
258:                                while(   bp > start && *bp != ' ' )
259:                                {
260:                                        --bp;
261:                                        wchar( '\b' );
262:                                }
263:
264:                                if( *bp == ' ' )
265:                                        wchar( *bp++ );
266:                                break;
267:                        }
268:
269:                        wchar( BEL );
270:                        break;
271:
272:                case HOME:              /* Cursor to left extreme       */
273:                        bp = start ;
274:                        setcur( home );
275:                        break;
276:
277:                case ESC:               /* Erase entire line and return */
278:                        *start = 0;
279:                        wchar( '\r' );
280:                        wchar( '\n' );
281:                        return -1 ;
282:
283:                case CAN:               /* Erase entire line            */
284:                        setcur( home );
285:                        for( bp = start; bp < maxbp; *bp++ = ' ')
286:                                wchar(' ');
287:
288:                        setcur( home );
289:                        maxbp = bp = start;
290:                        break;
291:
292:                case RIGHT:             /* Cursor right one character   */
293:
294:                        wchar( ( bp < end ) ? *bp++ : BEL );
295:                        break;
296:
297:                case CTL_RIGHT:         /* Advance to next word         */
298:
299:                        while( bp < maxbp  &&  *bp != ' ' )
300:                                wchar( *bp++ );
301:
302:                        while(  bp < maxbp  &&  *bp == ' ' )
```

```
303:                                    wchar( *bp++ );
304:
305:                            break;
306:
307:                    case END:               /* Go to right extremity  */
308:
309:                            if( bp < maxbp )
310:                            {
311:                                    ptail( bp, maxbp, 1 );
312:                                    bp = maxbp ;
313:                            }
314:                            break;
315:
316:                    default:
317:                            /*      If we aren't at the right-most extreme
318:                             *      of the buffer, move the tail over to
319:                             *      make room for the current character,
320:                             *      else just print it.
321:                             */
322:
323:                            if( bp <= end  &&  (' ' <= c  &&  c < 0x7f) )
324:                            {
325:                                    if( bp < maxbp )
326:                                    {
327:                                            if( maxbp < end )
328:                                                    maxbp++ ;
329:
330:                                            movmem( bp, bp+1, end-bp );
331:                                            ptail ( bp, maxbp, 0 );
332:                                    }
333:
334:                                    wchar( c );
335:
336:                                    if( bp < end )
337:                                            *bp++ = c;
338:                                    else
339:                                    {                       /* we're at the  */
340:                                            wchar( '\b' );/* right margin  */
341:                                            wchar( BEL ); /* back up and   */
342:                                            *bp = c ;     /* ring the bell */
343:                                    }
344:
345:                                    if( bp > maxbp )
346:                                            maxbp = bp;
347:
348:                                    break;
349:                            }
350:
351:                            break;
352:                    }
353:            }
354:
355:            /*      Delete trailing whitespace, terminate the string, go
356:             *      to the next line, and return EOF if we're at end of
357:             *      file, the end pointer otherwise.
358:             */
359:
360:            for( ; *end == ' ' && end >= start ; --end )
361:                    ;
362:
363:            *++end = '\0';
364:            wchar( '\r' );
365:            wchar( '\n' );
366:
367:            return ( c == EOF  &&  start == end ) ? NULL : end ;
368: }
369:
370: /*---------------------------------------------------------------------*/
371:
372: char    *getl( buf, maxline, fp )
373: char    *buf;
374: FILE    *fp;
375: {
376:            /*      Works exactly like fgets but returns a pointer to the
377:             *      end of the string on success.
378:             */
379:
380:            register int    c;
381:            register char   *bp = buf;
382:
383:            while( (c = fgetc(fp)) != EOF  &&  c != '\n'  && --maxline > 0 )
384:                    *bp++ = c;
385:
386:            *bp = '\0' ;
387:
388:            return( (c == EOF && bp == buf) ? NULL : bp );
389: }
390:
391: /*---------------------------------------------------------------------*/
392:
393: char    *efgets( buf, maxline, fp )
394: char    *buf ;
395: FILE    *fp  ;
396: {
397:            /*    An editing version of fgets.
398:             *    Works like fgets but recognizes a back-slash at end of line
399:             *    if fp is stdin then raw i/o is used and various editing
400:             *    functions are enabled (see egets for details). A pointer
```

(Continued on next page)

C Chest (Listing continued, text begins on page 16)

Listing Five

```
401:          *    to the end of the input string is returned on success or
402:          *    NULL on end of file.
403:          */
404:
405:         register char   *bp, *start = buf ;
406:         int             linelen, col;
407:
408:         if( fp == stdin )
409:         {
410:                 /*  linelen is the amount of space left on the
411:                  *  current input line. Col is the column componant
412:                  *  of the current cursor position.
413:                  */
414:
415:                 linelen = 80 - (col = getcur() & 0xff);
416:         }
417:         while(1)
418:         {
419:                 bp = (fp == stdin) ? egets( buf, min(linelen, maxline) )
420:                                    :  getl( buf, maxline, fp )
421:                                    ;
422:
423:                 /*      If egets() found and ESC (bp == -1) or we've
424:                  *      hit end of file (!bp) or we've seen a
425:                  *      blank line (bp == buf) the last character on the
426:                  *      line isn't a \, break; Note that in the first
427:                  *      case we erase then entire buffer.
428:                  */
429:
430:                 if( bp == -1 )
431:                 {
432:                         *(bp = start) = '\0';
433:                         break;
434:                 }
435:                 else if( !bp  ||  bp <= buf  ||  *(bp-1) != '\\' )
436:                         break;
437:
438:
439:                 /*      Adjust maxline to compensate for the characters
440:                  *      already gotten and decrement bp so that we'll
441:                  *      overwrite the \ on the next pass. Then, if
442:                  *      we're getting input from stdin, position the
443:                  *      cursor in its original column but on the current
444:                  *      line.
445:                  */
446:
447:                 maxline -= (--bp - buf);
448:                 buf      = bp ;
449:
450:                 if( fp == stdin )
451:                         setcur( (getcur() & ~0xff) | col );
452:         }
453:
454:         return( bp );
455: }
456:
457: /*---------------------------------------------------------------------*/
458:
459: #ifdef DEBUG
460:
461: main()
462: {
463:         static char buf[80] ;
464:
465:         printf("              1         2         3         4\n");
466:         printf("     12345678901234567890123456789012345678901234567890\n");
467:         printf("     ");
468:
469:         while( efgets(buf, 40, stdin)  >  0 )
470:         {
471:                 printf("     %s<---\n\n", buf );
472:                 printf("              1         2         3         4\n");
473:                 printf("     12345678901234567890123456789012345678901234567890\n");
474:                 printf("     ");
475:                 *buf = 0;
476:         }
477: }
478:
479: #endif
```

End Listings

Windowing Operating Environments

TopView, GEM, and Windows

by Michael Swaine

What's really going on inside GEM, beneath TopView, behind Windows?

It's a good thing Bill Gates isn't thin-skinned. Otherwise he might take offense at the way so many programmers are spending so much time enhancing, augmenting, bypassing, subverting, taming, masking, and hiding MS DOS. And now Microsoft is getting into the act with Windows. This article examines Windows and two other "windowing, multitasking programming environments" for MS or PC DOS machines: IBM's TopView and DRI's GEM.

You could argue that the seeds were planted by no less a gardener than the Jolly Blue Giant when IBM released its PC with no operating system but with three operating system options: PC (a.k.a. MS) DOS; CP/M-86; and SofTech's UCSD p-System. Although IBM's subsequent pricing strategy strongly hinted that PC DOS was the PC's DOS, the other operating systems never entirely took the hint and went away. Nor did they even get humble: Consider NCI's recent ads for its p-System that announce a new release for the whole PC family (except for jr, whom the family never mentions). The ads, directed at programmers, suggest that with the p-System you can have compatibility with the operating system that your customers, in their ignorance, want (DOS), but can still work in a good programming environment (not DOS).

Although there were precursors, DRI should probably get the credit for bringing concurrent processing to personal computers with its concurrent version of CP/M. It soon became clear to DRI that to sell concurrency on IBM PCs it had to develop a concurrent PC DOS, acknowledging Microsoft's control of the PC operating system area. But DRI had raised the issue of concurrency on a single-user computer.

After DRI opened the concurrency window, there came the pop-ups, an undisciplined horde of dwarf programs that demonstrated even to ordinary mortals the benefits of some approximation of concurrency. They presented object lessons in the hazards of operating system anarchy as well. It was cute when Eastman Kodak put a package under the tree labeled "Open me first"; it was less appealing to confront a dozen software packages all clamoring "Load me first."

What allowed pop-ups to show off their concurrency was windows, an architectural feature opened by Xerox and polished by Apple. The success of the concept of windows is exemplified by its employment in dWindow, a non-operating system application. For years, Ashton-Tate's dBASE II set the standard for austerity in user interfaces—you can't get much simpler than a single period prompt. Liberty-Bell Software's dWindow does a dazzling cathedral window treatment on dBASE II and III that makes them look like entirely different products. But it is at the operating system level that the fenestration of the interface has become obligatory, and it was Apple's Macintosh that made it so.

Bill Gates and his Microsoft programmers were among the first developers to peek behind the blinds at Apple, and they were not slow to announce Microsoft's own window-oriented product, which they called Windows. Sometime between the announcement of Windows and its release, the Korean War ended, Alaska and Hawaii attained state-

hood, and Digital Research programmers conceived and brought out a windowing product called GEM, whose visual display, Apple recently noticed, looks rather like the Mac's. Meanwhile, IBM caught the drift, fumbled through its pockets, and pulled out a windowing, multitasking operating environment that it christened TopView.

With at least three windowing, multitasking operating environments to load on top of, stack next to, or wrap around DOS, the archetypal "fully loaded PC" was beginning to look like a pop-up on steroids. In fact, or at least in *PC Week*, Peter Norton described a dream he had in which he tried to run TopView, GEM, and Windows simultaneously. His hard disk flew to Poland.

Smart Money Talks a Lot

A programmer must face more serious questions of compatibility. How does each product work with DOS? How will existing and future programs run in each of these environments? How well will they run? What are the costs of laying another sheet of software between programmer and machine? We asked such questions of development-team members for each of the three products. But as the answers came back, industry analyses were circulating that questioned the viability of at least two of the products (the two that had actually been released). Were we wasting our time asking questions about these products?

Although some developers are producing TopView-compatible applications, the product has clearly not captured the imagination of users or developers. The recent agreement between IBM and Microsoft has fueled much speculation about the viability of TopView, and many observers seem to be betting on Microsoft. One popular argument is that IBM will create two systems: one using Windows and Microsoft's DOS, the other using a DOS-free descendant or unrelated successor to TopView—in any case, a proprietary operating system. Some, including IBM watcher Andy Seybold, speculate that the IBM-Microsoft agreement may have little to do with IBM's long-term goals and that IBM will use TopView as the centerpiece of its future operating systems. At least one analyst insists that IBM's history demonstrates that it will eventually have a proprietary operating system on the PC. But IBM has broken tradition several times in the short history of the PC, and the venerable wisdom about what IBM always does may not be as wise now as once it was. It does seem that it is getting harder to second-guess IBM, and that the benefits from succeeding therein may be even more dubious today than in mainframe days.

The smart money is writing off GEM in the IBM-compatible market, reasoning that if IBM owns the hardware and Microsoft owns DOS, where does that leave a company that tries to compete with them on their turf?

Perhaps DRI's success with GEM will have to come in the other end of the dumbbell. Lee Felsenstein, in proselytizing for his Hacker's Mac project, describes the personal computer market as an asymmetric dumbbell with one globe of IBM-compatibles and a smaller globe of Macalikes. He argues that the smaller globe can grow and prosper only with compatibility and proposes a radical strategy for forcing compatibility on unwilling Apple, Commodore, and Atari. Such a development would likely benefit DRI and GEM, but is it likely?

Perhaps not, given the fact that Apple has pressured DRI into changing GEM to decrease its similarity to the Macintosh's visual interface. GEM as originally released had to be terminated by November 15, that is, DRI had to stop supporting and advertising it. The new version will look less Macintoshish. Apple is also talking with Microsoft about Windows, but in softer tones.

Smart money and dream-machine designers aside, GEM provides MS DOS users and programmers with capabilities that Windows and TopView lack. But TopView and Windows have their own distinct, desirable features.

Three facts argue against accepting the judgment of the smart money too hastily. First, the three operating environments offer three different sets of capabilities to users and software developers. Second, no one needs any of these products; they are all frosting on the DOS, and users may decide among them—or against all of them—on grounds that smart money would consider dumb. Third, as Bob Frankston pointed out in *InfoWorld*, writing for TopView (or Windows or GEM) limits your market. In any case, we assume there is merit in understanding these products from a programmer's point of view.

Inside GEM

At one level, getting started developing an application that is compatible with GEM is simple. You get the Programmer's Toolkit and start writing. In terms of hardware, you need a PC with half a megabyte of memory and a color-graphics adapter. You should also have a hard disk and a mouse, although you can do development work without them.

Beyond these elementary requirements, one comes up against the fact that GEM is a message-passing program. Years of single-thread procedural programming experience will not prepare you adequately for the different programming model that GEM employs. According to one GEM programmer, the learning curve within DRI during GEM development was two months. Programmers who had never worked with any windowing system took two months to get up to speed in the GEM programming environment. Programmers who had had some windowing experience learned faster but had to unlearn some details that did not transfer.

GEM itself does not communicate directly with DOS; that is, the Virtual Device Interface and Window Services don't. The GEM Desktop does. File manipulation is handled through DOS calls. GEM supports a variety of devices and is expected to support the AST/Ashton-Tate/DRI/Quadram expanded-memory specification, which will become particularly significant when the multitasking version of GEM is released.

GEM compares more directly with Windows than with TopView; TopView is character-oriented and truly multitasking and GEM and Windows are neither. In comparing GEM and Windows as programming environments, Windows developers point to powerful features, and GEM de-

velopers talk about a clean programmer's interface. But the clearest advantages GEM has over Windows at the moment seem to be that GEM has been out long enough to land significant committed and producing OEMs, including Atari and Apricot, and the leverage that this gives the developer in porting an application to different environments.

What does the Apple-DRI settlement mean to programmers who have developed GEM-compatible software or who are thinking about doing so? Perhaps not much. The changes in GEM forced by Apple appear to be essentially cosmetic, and though the impact on the GEM Desktop will be significant, the programming impact may be minimal.

Beneath TopView

The chief difference between TopView on the one hand and GEM and Windows on the other is that TopView is truly multitasking. You can see multitasking in action if you bring up a visually active BASICA program in two TopView windows simultaneously. You'll see the program doing its thing in parallel with itself.

Because it is a multitasking operating environment, TopView permits the development of multitasking applications. The application developer can produce a task with its own subtasks. Then, in addition to running the application in multitasking mode with other tasks, TopView will multitask the application's subtasks. Tasks can communicate; if you get the object handle for a task you can send it a message.

Memory management under TopView is as simple as it is *under* DOS. When a program begins, it is assigned memory according to the demands of its fixed Program Information File. Henceforth it can get no more nor less. On a program's termination, its memory is freed.

TopView does not replace DOS. It sits atop DOS, handing off file I/O and other system functions to the system. It is at the character I/O level that TopView butts in, redirecting all character I/O. Among other things, this means that you can freely mix TopView and DOS calls. You can put the user in DOS under TopView and have the fact that TopView is active be unknown to them. (A Get-version call will show the user that TopView is active.)

Although TopView is character-oriented rather than graphics-oriented, it will work with a standard monochrome or color screen and anything up to the EGA. TopView will work with the EGA but won't take advantage of its extended graphics capabilities.

TopView developers are trying to woo other developers, arguing that you can bring a more powerful application to market more quickly if you write for the TopView environment—you can take advantage of a standardized display style, a toolkit of window design aids, and other development tools. You can also use what TopView designers call full-screen input. This wooing of developers has not been entirely unsuccessful; the Trio micro-to-mainframe product and the Lattice Topview Toolbasket are significant TopView-compatible products.

For existing programs, you can make use of as much of the windowing capability as makes sense. For some applications this may be useless, but the capability is there. The implementation of DOS services shows what IBM projects for future applications—making logical use of several windows rather than just throwing a frame around a full-screen display.

Behind Windows

Because Microsoft has good sources on what future versions of DOS will require, its programmers can do what they admonish independent developers not to do—work around DOS. Windows runs "side by side with DOS." When Windows comes up, its DOS executive replaces command.com, which is no longer needed.

Windows is made up of three pieces: two are the external pieces that everyone sees, the user interface and the graphics device interface (GDI). The third is the kernel, which interfaces to whatever kind of MS DOS is on the machine. This interface changes considerably with different versions of DOS and hides the operating environment from Windows.

The user interface differs from the Mac/GEM/TopView approach in tiling the screen with windows that don't overlap other windows.

At the GDI level is an interrogating interface: if a device says that it can only do bitblt, the interface will simulate everything else in the software using bitblt; if the device says it can do complex polygons with hash filling, GDI hands off to the device. The device manufacturer fills in capabilities up to some level and Windows simulates the rest. The goal at the GDI level was device independence—the ability to change one line of code and have output go to a different device.

At the level of the kernel, Windows has true compacting global memory management; it does not have true preemptive multitasking. It can allocate, reallocate, dynamically free and restore data. Its task handling is round robin nonpreemptive multitasking—the application must yield control. There are mechanisms for implicit yielding when the application is waiting for something.

The reason Microsoft gives for not implementing true preemptive multitasking is suggestive: DOS is not reentrant. If you preempt a task and it's in DOS at the time, you'd best get back to DOS quickly. So a good chunk of the time slicing would not be beneficial. That, Windows developers claim, is why TopView is slow. Rick Dill of the Windows design team says, "We think multitasking really belongs at the operating system level, and it will get there eventually." He means in MS DOS.

Windows started from the Smalltalk push model and changed to a more procedural approach only when Microsoft found that programmers didn't work well with its implementation of the push model. The general structure of a Windows application is: initialization; Windows-required initializations (because Windows does not require that an application be installed it does require that the application register itself with Windows when it starts); creating a window; and the main program while loop—Getmessage, Translatemessage, Dispatchmessage.

One of the hidden features is the ability for applications

written by different vendors to be integrated via cut and paste as well as the via the messaging mechanism. For example, Multiplan could register a message that says "cell changed." A presentation-graphics program could know that that message means it can look in the clipboard for the contents of the cell and automatically update the chart being displayed.

Windows is 85 percent written in C; the low-level part is written in assembly. Windows supports Pascal, FORTRAN, C, or Microsoft's macro assembler. Because of the new .exe format, programmers will have to use Microsoft's new linker.

According to Dill, Microsoft hopes to entice software developers with the knowledge that, in conforming with Windows, they're building a future DOS-conforming application. Windows is a sneak peek into DOS. (On the other hand, Dill admits there are software developers who have disassembled the .exe file and know more about the innards of DOS than he does.)

But does all this just get in the software developer's way? Most serious software developers find that they must work around DOS to produce a competitive product. Won't software developers find Windows just as encumbering?

"Our bitblt," Dill responds, "is faster than yours." He's serious: Microsoft's bitblt is a general-purpose source-pattern-destination function supporting four operators. The result is that there are 256 different operations available: others implement some subset, often 16, of those. The Windows graphics routines can move a graphics block around the screen almost as fast as the data can be moved through memory. Whether that answers the question is moot.

One significant change Microsoft is introducing is a new .exe file format. Future versions of DOS (of which Windows is a hint) will need more information in the .exe file. To maintain some compatibility, Microsoft has grafted the old .exe header, code, and data on top of the new .exe header. The old tells where the new begins, and the new has its own code and data. It also has something totally new called resources. Resources (like menus and dialogue boxes) can be changed without messing with the .exe file. Thus, one can have one binary that works worldwide, with the resources supplying the language-specific information.

What, then, of the .com files? In the next version of DOS, .com files go away. DOS 3 is the last Microsoft operating system in which memory-image programs will be supported under the operating system itself. "We just need the information," Dill explains, "to be able to do things like running that application up in high memory and being able to segment it correctly and deal with running an old style 8088 unprotected application under a 286." DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 192.**

BANKSWAP

A Banked Memory Debugging Tool for CP/M Plus

by Albert S. Woodhull

By making extended memory useful to 8-bit processors, CP/M Plus can help to keep 8080-, 8085-, and Z80-based microcomputers in competition with newer 16-bit systems. Although the 8-bit processors can directly address only 64K of memory, under CP/M Plus a megabyte can be efficiently used.

For the programmer there is a catch. Transient programs must reside in a single 64K bank. They have indirect access to alternate banks through the operating system. A user running purchased software will not be bothered by this. Developers and others who need to write or modify code that uses the alternate banks have a serious problem, however. CP/M Plus comes with development tools, such as the RMAC assembler and LINK, that enable one to write code for use in any bank of memory, but the standard CP/M Plus debugging tool, SID, is an ordinary transient program, incapable of examining or changing the alternate banks. The programmer needs new tools or a way to extend the power of the old ones.

My response to this need was BANKSWAP, an extension for SID or DDT that provides access to the alternate banks. I will describe this tool after a brief discussion of how CP/M Plus uses banked memory.

Apple's Filer is fine until you want to copy files between volumes without losing a BASIC program in memory.

Banked Memory in CP/M Plus

Additional banks of memory can improve the speed and memory availability for a user program even though the program resides entirely in the main bank. The speed increase comes from using the extra memory to buffer disk data and disk directories. The space increase comes from putting part of the operating system in an alternate memory bank, making almost all of the 64K of memory the processor can address available to the user's program. For instance, on an Apple with the Advanced Logic Systems CP/M Card the transient program area (TPA) is 60K, even though the CP/M Plus system itself (exclusive of the CCP) takes up 17K.

User programs cannot directly access data or executable code in alternate banks. The point can be argued, but I believe Digital Research made a wise choice in imposing these restrictions. Most programs that use a lot of memory use it to hold data, and disk buffering can provide an improvement in data access comparable to that of extended memory. Given the inherent limitations of 8-bit processors, it is not clear that allowing user programs to execute code resident in extended memory would be much more efficient than using code overlays swapped in from disk buffers.

Under CP/M Plus, programs must be written as if the extra memory is not there at all. As long as the minimum amount of memory needed for an application is present in the TPA bank, a program will be able to run on any CP/M system. This continues the CP/M tradition of providing portability for programs by using the operating system to make details of hardware irrelevant to user programs.

Albert S. Woodhull, Hampshire College, Amherst, MA 01002

The BANKSWAP Program

BANKSWAP is not a stand-alone program; it is an enhancement to SID or DDT that provides additional commands to copy blocks of memory from bank to bank. The normal functions of the debugger can be used on a copy of memory from another bank that has been brought to the TPA bank.

BANKSWAP is relocatable and is not necessarily loaded to the same location each time it is used. For ease of use, BANKSWAP installs a vector to its own entry point at the RST 5 location (28H) during installation. Typing G28 from the SID or DDT prompt brings up the BANKSWAP menu. The initialization process also displays a message to remind the user of the presence of BANKSWAP and the command to access it.

The BANKSWAP menu allows the user to choose the direction of the move, the memory addresses for the source and destination, and the length of the block to be moved. The menu also provides for easy return to DDT or SID and for the eventual removal of BANKSWAP. Copying is done in two steps through a buffer that is also in common memory. I chose to use a relatively small buffer and repeat the process several times in order to make the best use of memory space.

Listing One (page 38) for BANKSWAP.ASM contains comments that explain the operation of BANKSWAP, but I will emphasize a few points I found important in working with banked memory. Although I wrote this program for use on an Apple with the Advanced Logic Systems CP/M Card, there should be no problems in making BANKSWAP work on other implementations of CP/M Plus. The most critical point is to be sure that control is not lost while bank 1 is deselected. This means ensuring that BANKSWAP itself, the stack, and all data areas used are located in common memory. It is possible for a CP/M Plus system to be constructed so that interrupts and system calls can be handled while alternate banks are selected. With insurance in mind, I thought it best to disable interrupts and avoid calls to the standard BDOS entry point while bank 1 is deselected because the vectors on page 0 of bank 1 are then inaccessible.

I used a direct call to a BIOS routine to select the bank of memory. There are several points to mention in regard to that. First, the CP/M Plus documentation is emphatic in stating that direct BIOS calls must not be made by application programs; the reason for this is that under CP/M Plus some BIOS routines are always called from the bank 0 portion of the BDOS and will not return to a program located in bank 1. In fact, a separate BDOS function is provided for gaining direct access to the BIOS. The catch is that this BDOS function prohibits access to one BIOS routine—you guessed it—the one we need for BANKSWAP. Digital Research really doesn't want user programs to try to access other memory banks.

Having decided to take the law into your own hands and call the BIOS bank-select routine directly, you can't do it by accessing the BIOS vector in low memory just any old time for the same reason you can't use BDOS routines any old time—some of the time the low memory the program sees will be in bank 0. For this reason, I made the initialization portion of BANKSWAP fetch the BIOS vector so the necessary entry point could be calculated and stored locally. I did this only for the BIOS SELMEM routine; the program can be simplified if a BIOS XMOVE routine has also been implemented (this is not provided in the Advanced Logic Systems release of CP/M Plus).

Resident System Extensions

BANKSWAP must run in common memory. This is most easily accomplished by creating a relocatable file to be loaded to the top of the TPA. CP/M Plus has another feature, the Resident System Extension (RSX), to simplify this task.

An RSX is a page-relocatable program segment that CP/M Plus loads to the highest available address, just as DDT or SID are handled under older versions of CP/M. Each RSX module has a prefix, as shown in Listing One. The prefix includes a jump to the BDOS entry point in high memory (for the first RSX installed) or to the last previously installed RSX. Whenever an RSX is installed, the JMP instruction at location 5 is modified to point to it, so a chain of JMPs is traversed whenever a BDOS call is made. The address portion of the JMP at location 5 is also used by programs to detect the decreased size of the TPA, protecting the RSX from being overwritten.

Because all BDOS calls pass through the RSX JMP chain, RSXs can be written to intercept BDOS calls to customize BDOS performance. BDOS interception is not necessary, however. BANKSWAP intercepts only the very first BDOS call after it is loaded—this is a convenient way to force execution of the BANKSWAP initialization phase. The initialization code saves and restores whatever information is being passed to the BDOS and modifies the JMP chain to prevent reinitialization.

An RSX must be connected to a normal .COM program to be installed. CP/M Plus provides a utility, called GENCOM, to do this and also provides the RMAC and LINK programs needed to produce the relocatable program and its relocation bit map. The process is considerably more complicated than assembling and loading a program under CP/M 2, but it is easy to use the CP/M SUBMIT program to direct the process. An RSX can be attached to any .COM file. I have attached BANKSWAP to DDT.COM and SID.COM and it works with both. Listing Two (page 52) shows the RSXMAKER.SUB file that can create BANKSWAP.RSX and connect it to SID.COM.

When BANKSWAP is to be attached to SID or DDT it is assembled with a REMOVE flag set in the RSX prefix. This ensures that when a warm boot occurs upon exiting from SID the space occupied by the RSX will be freed. BANKSWAP can also be assembled to be loaded independently of SID by setting the ALONE equate true. In this case it is still necessary to attach the RSX code to a .COM file, which could be nothing more than a jump to location zero. As the Listing One shows, setting ALONE true resets the REMOVE flag in the RSX header, and it also adds an option to the BANKSWAP menu to allow later removal.

There is a potential problem with the version of BANKSWAP shown in Listing One, but I'm leaving it for someone else to fix. As noted above, BANKSWAP uses the RSX technique in order to be located at the highest available memory address. The actual amount of high memory that is common depends upon the hardware used, and I didn't figure out a way for a program to determine this. If too many RSXs are installed, the top of available memory can be below the common region. In this case BANKSWAP will probably cause a crash. On my Rev. A CP/M card the common-memory limit is at 8000H, which leaves room for a lot of RSXs, so I have never given the problem a high priority.

Using BANKSWAP

Once BANKSWAP has been attached to a copy of SID.COM it will be installed whenever SID is run. Listing Three (page 52) shows part of a typical session with enhanced SID. The initialization process tells the operator that BANKSWAP is available. A G28 command enters BANKSWAP, which displays a menu. I usually bring part of an alternate bank over to bank 1 first, then return to the debugger to examine, disassemble, or alter the copy. In fact, if executable code from some portion of bank 0 is brought over to the corresponding address range in bank 1 the debugger can be used to trace through it—providing that the code doesn't switch banks or access I/O or storage addresses in an alternate bank.

In the Apple environment there is one limitation: an Apple's I/O is all memory mapped in bank 0. If an attempt is made to access bank 0 in the range 6000H to 67FFH it may crash the system because some addresses in this range activate switches on the Apple main board or peripheral cards, including the CP/M Card itself.

Conclusion

One of my first reactions to CP/M Plus was a helpless feeling. I had written my own BIOS for CP/M 2.2 on my S-100 system, and I was accustomed to being able to explore how the system worked on any CP/M machine. With CP/M Plus, portions of the operating system were totally inaccessible to the tools I had on hand.

Most users don't need the kind of access I wanted. However, the initial version of CP/M Plus I received did not support my printer interface properly. I also had proposed to develop CP/M Plus driver software for companies that manufactured large-format disks and RAM-disk add-ons for Apples. The usability of my system as well as potential income depended upon my ability to patch various devices into CP/M Plus.

BANKSWAP solved my problems. It gave me a way to satisfy my curiosity about how the Advanced Logic Systems CP/M Plus BIOS worked, and I was able to figure out how to patch some of my peripheral drivers into CP/M Plus. Finally, I now had the tool I needed to begin to develop and debug enhancements to the supplied BIOS. In the process of developing BANKSWAP I also learned how to create and use RSXs, which are very useful CP/M Plus features.

DDJ

(Listing begins on next page)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 193.**

BANKSWAP (Text begins on page 34)

Listing One

```
                ; BANKSWAP.ASM
                ;
                ; A. S. Woodhull         28 June 83
                ; rev    1 July 85       -- minor editing
                ;       20 Oct 83
                ;
                ; This is designed to be run as a subprogram under DDT
                ; or SID, in a banked version of CP/M 3.0  The function
                ; of BANKSWAP is to copy blocks of memory from other
                ; banks to and from bank 1, using a buffer in the common
                ; area.  Normal DDT or SID functions can then be
                ; performed, using the copy of the code in bank 1.
                ; (Of course, you cannot trace through program segments
                ; that switch banks or access memory mapped I/O in
                ; another bank.)
                ;
                ; BANKSWAP must reside in the common area of memory, and
                ; a buffer area through which data can be copied must
                ; also be present in the common area.  BANKSWAP is to be
                ; assembled as a Resident System Extension (RSX), which
                ; will automatically be relocated to the top of the TPA.
                ; It is assumed that the common area is large enough to
                ; allow an RSX to fit--if this is not true BANKSWAP will
                ; not work in its present form.
                ;
                ; Because the location of BANKSWAP in memory is not fixed
                ; a jump through a fixed location is set up when the BANK-
                ; SWAP code is installed.  In this version the  RST 5 vector
                ; at 28H is used, but any convenient location on page zero
                ; may be used.
                ;
                ; Under CP/M 3 direct access of BIOS routines is generally
                ; to be avoided by user programs, since BIOS routines may
                ; have expectations about which bank is selected when they
                ; are called.  We will, however, do bank switching through
                ; the BIOS selmem routine.  For generality we will use the
                ; BIOS vector at location 1.
                ;
FFFF =          true:   equ     0ffffh
0000 =          false:  equ     not true
                ;
0000 =          alone:  equ     false   ;make false if attached to DDT/SID
                ;
0001 =          biosv:  equ     1       ;address of wboot in BIOS found here
004E =          selmem: equ     4eh     ;offset from wboot
                ;
0100 =          buflen: equ     100h    ;move block size
                ;
                ; default parameters
1000 =          movcnt: equ     1000h   ;to move 16 pages (4K) at a time
0100 =          b0dft:  equ     100h    ;start of block in bank 0
0100 =          bldft:  equ     100h    ;start of block in bank 1
                ;

                ; zero page addresses
0005 =          bdos:   equ     5       ;bdos entry point
0028 =          rstv:   equ     28h     ;RST 5 used as entry vector
                ;
                ; BDOS functions used
0001 =          conin:  equ     1       ;get a char
0009 =          printf: equ     9       ;print a string
000A =          rdbuf:  equ     10      ;read a line
                ;
                ; This is standard prefix for a Resident System Extension
                ;  see CP/M 3 Programmer's Guide, 1st ed., sec. 4.4, p.168
                ;
0000 0000000000serial: db       0,0,0,0,0,0
0006 C3A203     start:  jmp     install ;one-time routine
0009 C30000     next:   jmp     0       ;altered at installation
```

(Continued on page 40)

BANKSWAP (Listing continued, text begins on page 34)

Listing One

```
000C 0000       prev:   dw      0
                ;
                        if      alone
                rmvflg: db      0       ;keep this in memory
                        else
000E FF         rmvflg: db      0ffh    ;remove when main program ends
                        endif
                ;
000F 00         nonbnk: db      0       ;banked system
0010 42414E4B53         db      'BANKSWAP'
0018 00         loader: db      0
0019 0000               db      0,0
                ; *** note: If BANKSWAP is to be attached directly to SID.COM
                ; or DDT.COM then make rmvflg 0ffh to force removal
                ;
                bankexam:
001B 210000             lxi     h,0     ;get stack pointer
001E 39                 dad     sp      ;...and hold it for return
001F 220F04             shld    holdsp  ;...reset SP to a location
0022 310804             lxi     sp,locstk       ;...in common memory
                ;
                ; Main loop--exit by Quit or Remove command
                bankex2:
0025 CD2E00             call    prompt  ;this returns address of sub
0028 CD6D00             call    doit    ;do subroutine addressed by HL
002B C32500             jmp     bankex2
                ;
002E 111F02     prompt: lxi     d,menu  ;get ready for menu
0031 3A9103             lda     quietflag
0034 B7                 ora     a       ;suppress menu?
0035 CA3B00             jz      prmpt2
0038 11F402     prmptl: lxi     d,query ;set for prompt only
003B 0E09       prmpt2: mvi     c,printf
003D CD0500             call    bdos
                ; Get a character
0040 0E01               mvi     c,conin
0042 CD0500             call    bdos
0045 CD7001             call    crlf
                ; make upper case, reject non-alpha
0048 E65F               ani     5fh
004A FE41               cpi     'A'
004C DA3800             jc      prmptl  ;ask again if invalid
004F FE5B               cpi     'Z'+1
0051 D23800             jnc     prmptl  ;ask again if invalid
                ; find match in alphtbl
0054 010700             lxi     b,altblen       ;length of table
0057 211102             lxi     h,alphtbl+altblen       ;work back
005A BE         try:    cmp     m
005B CA6300             jz      match
005E 2B                 dcx     h
005F 0D                 dcr     c       ;count down
0060 C25A00             jnz     try
                ; if c= 0 no match found.  Now form address
0063 211102     match:  lxi     h,addrtbl
0066 09                 dad     b       ;add offset
0067 09                 dad     b       ;again, 2 bytes per table entry
                ; get the command address
0068 7E                 mov     a,m
0069 23                 inx     h
006A 66                 mov     h,m
006B 6F                 mov     l,a
006C C9                 ret             ;HL has action address
                ;
006D E9         doit:   pchl            ;call here to use action address
                ;
                getbank:
006E 2A9C03             lhld    b0start ;setup addresses
```

(Continued on page 42)

BANKSWAP (Listing continued, text begins on page 34)

Listing One

```
0071 220A04             shld    source
0074 2A9E03             lhld    blstart
0077 220C04             shld    dest
007A 3AA103             lda     length+1        ;low byte ignored
007D 320E04             sta     count
                ; page by page take a chunk of bank 0 to buffer, then move
                ; to bank 1 destination
                get2:
0080 F3                 di              ;be sure no interrupts
0081 3E00               mvi     a,0
0083 2A0804             lhld    selmv
0086 CD6D00             call    doit
0089 CDF400             call    getbuf
008C 3E01               mvi     a,1
008E 2A0804             lhld    selmv   ;point to BIOS selmem routine
0091 CD6D00             call    doit
0094 FB                 ei              ;interrupts safe again
0095 CDFD00             call    putbuf
0098 110001             lxi     d,buflen
009B 2A0A04             lhld    source
009E 19                 dad     d
009F 220A04             shld    source
00A2 2A0C04             lhld    dest
00A5 19                 dad     d
00A6 220C04             shld    dest
00A9 210E04             lxi     h,count ;repeat for required # of pages
00AC 35                 dcr     m
00AD C28000             jnz     get2
00B0 C9                 ret
                ;
                putbank:
00B1 2A9E03             lhld    blstart
00B4 220A04             shld    source
00B7 2A9C03             lhld    b0start
00BA 220C04             shld    dest
00BD 3AA103             lda     length+1        ;low byte ignored
00C0 320E04             sta     count
                ; page by page, move bank 1 data to buffer, then move it
                ; to bank 0
                put2:
00C3 CDF400             call    getbuf
00C6 F3                 di              ;be sure no interrupts
00C7 3E00               mvi     a,0
00C9 2A0804             lhld    selmv
00CC CD6D00             call    doit
00CF CDFD00             call    putbuf
00D2 3E01               mvi     a,1
00D4 2A0804             lhld    selmv
00D7 CD6D00             call    doit
00DA FB                 ei              ;interrupts safe again
00DB 110001             lxi     d,buflen
00DE 2A0A04             lhld    source
00E1 19                 dad     d
00E2 220A04             shld    source
00E5 2A0C04             lhld    dest
00E8 19                 dad     d
00E9 220C04             shld    dest
00EC 210E04             lxi     h,count
00EF 35                 dcr     m
00F0 C2C300             jnz     put2
00F3 C9                 ret
                ; source to buffer
                getbuf:
00F4 2A0A04             lhld    source
00F7 111104             lxi     d,buffer
00FA C30401             jmp     pb1
                ; buffer to dest
                putbuf:
```

(Continued on page 44)

Listing One

```
00FD 2A0C04             lhld    dest
0100 111104             lxi     d,buffer
0103 EB                 xchg
                ; common code for getbuf and putbuf
0104 010001     pbl:    lxi     b,buflen
                ; move (BC) bytes from (HL) to (DE) (could use BIOS move
                ; routine for this)
0107 7E         move:   mov     a,m
0108 12                 stax    d
0109 23                 inx     h
010A 13                 inx     d
010B 0B                 dcx     b
010C 78                 mov     a,b
010D B1                 ora     c
010E C20701             jnz     move
0111 C9                 ret
                ;
                ; Go back to SID/DDT
0112 116403     quit:   lxi     d,qmsg  ;say how to get back
0115 0E09               mvi     c,printf
0117 CD0500             call    bdos
011A 2A0F04             lhld    holdsp  ;restore stack pointer
011D F9                 sphl
011E FF                 rst     7       ;back to DDT or SID
                ;
                        if      alone
                ; Set for removal on termination of SID or DDT
                remove: lxi     h,rmvflg        ;set the remove flag
                        mvi     m,0ffh          ;...in the RSX prefix
                        lxi     h,rstv          ;then wipe out the entry
                        mvi     m,0ffh          ;...JMP with an RST 7
                        rst     7               ;leave via the debugger
                        endif   ;alone
                ;
                ; Set up addresses for move, also set up length
011F 111703     adset:  lxi     d,b0id  ;tell current bank 0 addr
0122 0E09               mvi     c,printf
0124 CD0500             call    bdos
0127 2A9C03             lhld    b0start ;get the address
012A CDDA01             call    addro   ;and print it
012D 219C03             lxi     h,b0start
0130 CD8101             call    update  ;enter hex to (HL)
0133 112503             lxi     d,blid  ;do it again for bank 1
0136 0E09               mvi     c,printf
0138 CD0500             call    bdos    ;tell
013B 2A9E03             lhld    blstart
013E CDDA01             call    addro
0141 219E03             lxi     h,blstart
0144 CD8101             call    update  ;get new address, if any
                ; Can fall through from adset or enter directly here to set
                ; length of block moved
0147 113303     lnset:  lxi     d,lnmsg ;tell current length
014A 0E09               mvi     c,printf
014C CD0500             call    bdos
014F 2AA003             lhld    length
0152 CDDA01             call    addro
0155 21A003             lxi     h,length
0158 CD8101             call    update  ;offer to change it
015B CD7001             call    crlf
015E C9                 ret
                ;
                ; Toggle menu off/on
015F 3A9103     xpert:  lda     quietflag
0162 2F                 cma                     ;toggle
0163 329103             sta     quietflag
0166 C9                 ret
                ;
                ; Get here on invalid command
```

(Continued on page 46)

Listing One

```
0167 11FA02     na:     lxi     d,namsg         ;say can't do it
016A 0E09               mvi     c,printf
016C CD0500             call    bdos
016F C9                 ret
                ;
0170 C5         crlf:   push    b
0171 D5                 push    d
0172 E5                 push    h
0173 F5                 push    psw
0174 111403             lxi     d,crlfstring
0177 0E09               mvi     c,printf
0179 CD0500             call    bdos
017C F1                 pop     psw
017D E1                 pop     h
017E D1                 pop     d
017F C1                 pop     b
0180 C9                 ret
                ;
                ; get string, do nothing if null, else convert, store at (HL)
                update:
0181 E5                 push    h               ;save the location
                ; loop back here if input is not valid
0182 210000     updl:   lxi     h,0             ;initial buffer
0185 229303             shld    inbuf+1
0188 114203             lxi     d,chquery       ;say what's up
018B 0E09               mvi     c,printf
018D CD0500             call    bdos
0190 119203             lxi     d,inbuf
0193 0E0A               mvi     c,rdbuf         ;read console until <ret>
0195 CD0500             call    bdos
0198 CD7001             call    crlf
019B 3A9303             lda     inbuf+1         ;get length of hex string
019E B7                 ora     a               ;check for 0 length input
019F C2A401             jnz     convert
                ; null string, go back
01A2 E1                 pop     h               ;retrieve value at entry
01A3 C9                 ret
                ;
                ; Convert the hex string in the buffer to binary
                convert:
01A4 210000             lxi     h,0     ;start with a zero
01A7 47                 mov     b,a     ;hold length in B
01A8 119403             lxi     d,inbuf+2
01AB 1A         conv2:  ldax    d       ;get first (or next) char
01AC 13                 inx     d
01AD FE60               cpi     60h
01AF DAB401             jc      conv3
01B2 E65F               ani     5fh     ;make lower case if necessary
01B4 D630       conv3:  sui     '0'
01B6 FA8201             jm      updl    ;must be valid hex, 0..9, A..F
01B9 FE0A               cpi     0ah
01BB DACA01             jc      num     ;jump if a good numeric
01BE D607               sui     7
01C0 FE0A               cpi     0ah
01C2 DA8201             jc      updl    ;error if not good alpha
01C5 FE10               cpi     10h
01C7 D28201             jnc     updl    ;error if not good alpha
01CA 29         num:    dad     h       ;multiply current val by 16
01CB 29                 dad     h
01CC 29                 dad     h
01CD 29                 dad     h
01CE 85                 add     l       ;add new least significant digit
01CF 6F                 mov     l,a
01D0 05                 dcr     b       ;countdown the digits
01D1 C2AB01             jnz     conv2
01D4 EB                 xchg            ;result to DE
01D5 E1                 pop     h       ;HL at entry says where to it
```

(Continued on page 48)

Listing One

```
01D6 73                 mov     m,e
01D7 23                 inx     h
01D8 72                 mov     m,d
01D9 C9                 ret
                ;
                ; Print HL as hex
                addro:
01DA D5                 push    d
01DB E5                 push    h
01DC EB                 xchg
01DD 215C03             lxi     h,hexstr        ;where to build string
01E0 7A                 mov     a,d
01E1 CDF301             call    byte    ;get A as 2 ASCII chars at (HL)
01E4 7B                 mov     a,e
01E5 CDF301             call    byte    ;again, low byte
01E8 115C03             lxi     d,hexstr
01EB 0E09               mvi     c,printf
01ED CD0500             call    bdos    ;print it
01F0 E1                 pop     h
01F1 D1                 pop     d
01F2 C9                 ret
                ;
                ; Convert byte to hex ASCII chars, put at (HL)
01F3 F5         byte:   push    psw
01F4 1F                 rar             ;get high nybble
01F5 1F                 rar
01F6 1F                 rar
01F7 1F                 rar
01F8 CDFC01             call    nybble
01FB F1                 pop     psw     ;fall through for low nybble
                ; nybble makes 1 char, advances output pointer
01FC E60F       nybble: ani     0fh
01FE C630               adi     '0'
0200 FE3A               cpi     3ah
0202 DA0702             jc      nput
0205 C607               adi     7
0207 77         nput:   mov     m,a
0208 23                 inx     h
0209 C9                 ret
                ;
                ; Acceptable command inputs go in this table
                alphtbl:
020A 00                 db      0       ;dummy for no match
020B 41                 db      'A'
020C 47                 db      'G'
020D 4C                 db      'L'
020E 50                 db      'P'
020F 51                 db      'Q'
                ;
                        if      alone
                        db      'R'
                        endif   ;alone
                ;
0210 58                 db      'X'
0007 =          altblen:        equ     $-alphtbl
                ; addresses of action routines, same order as alphtabl
                addrtbl:
0211 6701               dw      na      ;not available
0213 1F01               dw      adset   ;address set
0215 6E00               dw      getbank
0217 4701               dw      lnset   ;length set
0219 B100               dw      putbank
021B 1201               dw      quit
                ;
                        if      alone
                        dw      remove
                        endif
                ;
```

(Continued on page 50)

BANKSWAP (Listing continued, text begins on page 34)

Listing One

```
021D 5F01               dw      xpert   ;expert mode, no menu
                ;
021F 42616E6B73menu:    db      'Bankswap 1.0 by A. S. Woodhull
10/20/83',0dh,0ah
0248 66756E6374         db      'functions available:',0dh,0ah
025E 09412E2E2E         db      '       A...set move Addresses',0dh,0ah
0277 09472E2E2E         db      '       G...Get alternate bank',0dh,0ah
0290 094C2E2E2E         db      '       L...set move Length',0dh,0ah
02A6 09502E2E2E         db      '       P...Put alternate bank',0dh,0ah
02BF 09512E2E2E         db      '       Q...Quit to SID or DDT',0dh,0ah
                ;
                        if      alone
                        db      '       R...Remove BANKSWAP',0dh,0ah
                        endif
                ;
02D8 09582E2E2E         db      '       X...eXpert mode (no menu)',0dh,0ah
02F4 0D0A3F3F20query:   db      0dh,0ah,'?? $'
02FA 2E2E2E6675namsg:   db      '...function not available.'
                crlfstring:
0314 0D0A24             db      0dh,0ah,'$'
0317 42616E6B20b0id:    db      'Bank 0 addr: $'
0325 42616E6B20blid:    db      'Bank 1 addr: $'
0333 4C656E6774lnmsg:   db      'Length is now $'
                chquery:
0342 4368616E67         db      'Change to? (CR to keep): $'
035C            hexstr  ds      4
0360 480D0A24           db      'H',0dh,0ah,'$'
0364 52652D656Eqmsg:    db      'Re-enter BANKSWAP from DDT or SID by "G28"'
038E 0D0A24             db      0dh,0ah,'$'
                ;
                quietflag:
0391 00                 db      0       ;initialized off

0392 08         inbuf:  db      8       ;max length of buffer
0393                    ds      9
                ;
                ; default parameters: alter by Set and Length commands
039C 0001       b0start:        dw      b0dft
039E 0001       blstart:        dw      bldft
03A0 0010       length:         dw      movcnt
                ;
                ; One-time routine, on 1st BDOS call intercepted
                install:
03A2 C5                 push    b       ;keep everything as it was
03A3 D5                 push    d       ;...so BDOS function can be
03A4 E5                 push    h       ;...completed
03A5 F5                 push    psw
                ; set up restart vector for re-entry
03A6 3EC3               mvi     a,0c3h  ;a JMP instruction
03A8 322800             sta     rstv
03AB 211B00             lxi     h,bankexam
03AE 222900             shld    rstv+1
                ; set up address of BIOS routine accessed directly
03B1 2A0100             lhld    biosv           ;find where BIOS is
03B4 114E00             lxi     d,selmem        ;...and add offset
03B7 19                 dad     d
03B8 220804             shld    selmv
                ; then patch the RSX prefix to prevent reinstallation
03BB 2A0A00             lhld    next+1
03BE 220700             shld    start+1
                ; tell 'em we're ready
03C1 0E09               mvi     c,printf
03C3 11D003             lxi     d,imsg
03C6 CD0900             call    next
03C9 F1                 pop     psw     ;continue with the task that
03CA E1                 pop     h       ;...was so rudely interrupted
03CB D1                 pop     d
03CC C1                 pop     b
03CD C30900             jmp     next
```

(Continued on page 52)

BANKSWAP (Listing continued, text begins on page 34)

Listing One

```
                ;
03D0 42414E4B53imsg:    db      'BANKSWAP loaded. To access from DDT or '
03F7 5349442074         db      'SID type "G28"'
0405 0D0A24             db      0dh,0ah,'$'
                ;
                ; reuse installation code area for stack space
                locstk:
                ;
                ; uninitialized storage
0408            selmv:  ds      2       ;used for BIOS call to selmem
040A            source: ds      2       ;for moves to buffer
040C            dest:   ds      2       ;for moves from buffer
040E            count:  ds      1       ;blocks to move
040F            holdsp: ds      2       ;stack pointer from DDT or SID
                ;
                buffer:
0411                    ds      buflen
0511                    end
```

End Listing One

Listing Two

```
; RSXMAKER.SUB assemble RSX and attach to existing COM file
; ASW 21 Oct 83
; Usage: A>SUBMIT RSXMAKER <comfile> <asmfile>
;
rmac $2
link $2 [op]
ren $2.rsx=$2.prl
gencom $1 $2
```

End Listing Two

Listing Three

```
Listing Three. An example of the use of BANKSWAP to allow examination of a
        portion of a custom disk drive routine installed in bank 0.
        (Comments added in parentheses).

D>sid
BANKSWAP loaded. To access from DDT or SID type "G28"
CP/M 3 SID - Version 3.0
#g28                                    (go to BANKSWAP)
Bankswap 1.0 by A. S. Woodhull 10/20/83
functions available:
        A...set move Addresses
        G...Get alternate bank
        L...set move Length
        P...Put alternate bank
        Q...Quit to SID or DDT
        X...eXpert mode (no menu)
?? x                                    (remove the menu)
?? a                                    (set up range)
Bank 0 addr: 0100H
Change to? (CR to keep): 7000           (don't accept default)
Bank 1 addr: 0100H
Change to? (CR to keep): 7000
Length is now 1000H
Change to? (CR to keep): <CR>           (length is OK)
?? g                                    (get the data from bank 0)
?? q                                    (back to SID)
Re-enter BANKSWAP from DDT or SID by "G28"
*D61E                                   (we're back)
#d7c00 7c3f                             (examine the data from bank 0)
7C00: C3 09 7C C3 0F 7C C3 15 7C 21 94 0C C3 00 73 21 ..|..|..|!....s!
7C10: 74 0C C3 00 73 21 1B 0C C3 00 73 AE F1 0C A9 04 t...s!....s.....
7C20: D0 2D AE F1 0C AC DF 0C B9 BA 0C 9D 8E C0 AD DE .-..............
7C30: 0C 4A 48 A9 00 2A 0D EF 0C 0D F1 0C A8 B9 80 C0 .JH..*..........
#^C                                     (return to CP/M)
D>
```

End Listings

Adding a COPY Command to ProDOS

by Shawn Day

If you use ProDOS on an Apple or Apple-compatible system, you may have noticed that you cannot copy files from one volume to another without running FILER and losing any BASIC program you have in memory. I ran across this problem when I tried to copy files to the ProDOS RAM disk from within a BASIC program: I wanted my startup program to load all of the ProDOS editor and assembler files into the RAM disk so that I could have the maximum amount of storage space on my single disk drive and speed up the process of assembling long source files. To solve this problem, I wrote a program that adds a COPY command to ProDOS.

Additional banks of memory can buy you both speed and more usable memory even though your program must reside entirely in one bank.

Using the Program

To add the COPY command to ProDOS, simply type BRUN COPY or use the general-purpose "run" command, -COPY. The code will be installed in memory above the ProDOS file buffers and protected in the system bit map. To execute the command, use the following syntax:

```
COPY pathname1,pathname2
```

A new file (pathname2) will be created, and the contents of the file indicated by pathname1 will be copied into it. You can use this command to make backup copies of any file on the same volume or to copy files from one volume to another. If you are in the process of copying files from one volume to another and no ProDOS prefix has been set, you must specify the complete path name for both files, e.g.,

```
COPY /MYDISK/HELLO,
   /RAM/STARTUP
```

If a ProDOS prefix has been set, it will automatically be prefixed to either path name if required. For example, if the prefix is /RAM/, the command

```
COPY /MYDISK/HELLO,
   STARTUP
```

will copy the file /MYDISK/HELLO to the file /RAM/STARTUP. If the destination file name already exists on the destination volume, the command exits with the DUPLICATE FILENAME error. This prevents accidental erasure of files. If you want to copy a file over an already existing file, use the DELETE command to destroy the existing file and then use the COPY command.

COPY cannot be used to copy files directly from one disk to another if you have only one disk drive. However, if you have a ProDOS RAM disk available to you (that is, you are using an Apple IIc or a IIe with an extended 80-column card) you have the capability to copy the file into the RAM disk and then copy the file again from the RAM disk over to the destination disk.

Shawn Day, 724 Glenburn St., Kelowna, British Columbia V1Y 4G6 Canada

Entering the Program

The source code for COPY is shown in Listing One (page 58). To type in the program, you can either use an assembler or type the hexadecimal numbers in from the monitor. When you are finished, save the program with

```
BSAVE COPY,A$6000,L$2B6
```

How it Works

The COPY code is divided logically into three parts: installation, first-stage processing, and second-stage processing. I will describe the operation of each in turn.

Installation

When COPY is invoked, it relocates itself to reside between the ProDOS BASIC.SYSTEM and its file buffers. A call to GETBUFR with the accumulator containing the value $02 (Listing One, lines 69 and 70) reserves the required two pages of memory. The high byte of the starting address of the reserved space is returned in the accumulator. Line 71 checks to see if the call to GETBUFR was successful. In ProDOS Version 1.0.1 there is a bug in GETBUFR that causes it to return a successful error code whether or not an error occurs. Due to this bug, COPY will only work with ProDOS Version 1.1.1 and later. If you have an earlier version of ProDOS, see your dealer to get an update. If GETBUFR is unsuccessful, the program exits with an appropriate error message (line 72).

Assuming the GETBUFR call was successful, program control is transferred to GOTSPACE (lines 73–83), which protects the acquired memory space via the system bit map and updates RSHIMEM. The routine BITMAPS (lines 135–157) is called twice—once to protect each page. Then the value of RSHIMEM in the BASIC.SYSTEM's global page is decreased by 2 (lines 80–83). This protects the acquired memory from a call to FREBUFR. FREBUFR resets HIMEM to the value contained in RSHIMEM. Because COPY dynamically allocates buffer space during execution via GETBUFR and FREBUFR, it is necessary to perform this step. If you wish to have any other added ProDOS commands resident in memory during the execution of COPY, they must each be protected from FREBUFR by this method.

The next program segment (lines 87–95) hooks the COPY command into BASIC.SYSTEM's external command link. Any previously installed external commands are daisy chained by the method described in the *ProDOS Technical Reference Manual.* Lines (99–115) update all internal, nonrelocatable references so that the program will execute properly in its final location. They make use of the table of nonrelocatable references in lines 161–188.

After all the references are adjusted, the code is moved to its final location by the monitor MOVE routine (lines 119–131). This finishes the installation of COPY, and control is passed back to the system.

First-Stage Processing

When ProDOS runs across a command that it does not recognize, it passes control to EXTRNCMD in the BASIC.SYSTEM global page. With COPY installed, this location contains a jump to the relocated copy code (starting at line 203). Here the path-name buffer is checked to see if it contains the COPY command in upper or lower case. If not, the carry flag is set to indicate no match, and control passes to the next user-installed external command (lines 214-215 and 193). If there are no more user-installed commands, line 193 will contain a jump to an RTS instruction. Assuming the COPY command was found, execution continues at line 218, where the command-string length is decremented and stored in XLEN. Lines 220 and 221 store a 0 in XCNUM to indicate to BASIC.SYSTEM that the command is external, and lines 222–225 store the address where second-stage processing is to be resumed after the input string has been parsed by BASIC.SYSTEM. Next, PBITS and PBITS+1 are set to indicate that file creation is allowed, that the command requires two path names, and that the current prefix should be fetched if none is specified in the command string. Finally, the program exits with the carry flag cleared to tell BASIC.SYSTEM that the command has been identified.

Second-Stage Processing

After BASIC.SYSTEM has parsed the command string, control is passed to line 236. At this point, FBITS and FBITS+1 have been set by BASIC.SYSTEM to indicate which parameters were present in the command string. Lines 236–243 check that two path names were given and exit with a syntax error if not.

Assuming both path names were present, execution continues at PARMSOK (line 247), where a GET_FILE_INFO call is made to the MLI (machine-language interface) through the GOSYSTEM vector. This call gets the file information for the source file (the first path name specified in the command string). The file type is checked to make sure it is not a directory or bad block file. (Copying these file types would be meaningless.) If the file is either of these two types, the program exits with a FILE TYPE MISMATCH error.

Next, a new file is created with the file name specified by the second path name in the command line (lines 262–277). If an error occurs during the CREATE attempt, control passes to BADCALL (line 277), where the MLI error is translated to a BASIC.SYSTEM error, the carry flag is set, and an RTS instruction transfers control back to BASIC.SYSTEM. If the CREATE is successful, lines 281–289 attempt to obtain a 1K file buffer from the system for use by the destination file. If the attempt is unsuccessful, the NO BUFFERS AVAILABLE error is generated (lines 284–286).

Finally, the rest of free memory (excluding any resident BASIC programs and any variables that may have values assigned to them) is reserved for the file-transfer buffer (lines 294–316). It is desirable to have as large a buffer as possible in order to speed up the copy. If a very large BASIC program is in memory at the time the COPY command is executed, the file will be copied in several passes because only a small

amount of it can be read into memory at one time. If there is no BASIC program in memory, even very large files can be copied in one or two passes. If there is less than one free page of memory, all of the buffer space obtained so far is returned to the system, and the program exits with the NO BUFFERS AVAILABLE error (lines 302–305).

Now that all the buffers are set up, lines 320–331 open the destination file, and lines 335–337 open the source file. Any errors detected during this process cause the buffer space to be returned to the system and an appropriate error message to be generated. Lines 346–351 call NEWLINE to indicate that no NEWLINE character is to be recognized. The NEWLINE character is used by ProDOS when reading a file to signify the end of a line or record. For example, when reading lines from a text file, the NEWLINE character should be set to $8D (carriage return). This allows processing of the file line by line. Because it is desirable to read in as much of the file as possible during each pass, the NEWLINE character is disabled by setting the mask byte to 0 (lines 346 and 347).

Lines 361–378 perform the actual transfer. Lines 363 and 364 call the MLI READ routine to read in as much of the file as possible. An error during the read causes program control to pass to line 366, where the end-of-file condition is checked. If the end of file has been reached, lines 382–401 close both files, free the allocated buffer space, and return control to the system. Otherwise, lines 369–375 write the buffer contents out to the new file and then (assuming no errors were detected during the WRITE) go back for more (line 376). Lines 402–432 are the parameter lists used during the calls to the MLI.

Improvements and Modifications

The most obvious improvement is to allow the use of disk-to-disk copies with a single disk drive. You might also find it useful to allow COPYing a text file to the screen or a printer. Then you would have a general-purpose copy utility, such as CP/M's PIP command.

DDJ

ProDos Copy Listing

(Text begins on page 54)

```
0000:                1          LST   GEN,VSYM
0000:                2 ***********************************************
0000:                3 *                                             *
0000:                4 *     COPY Command for ProDOS BASIC.SYSTEM    *
0000:                5 *                by Shawn Day                 *
0000:                6 *                                             *
0000:                7 *       Syntax: COPY pathname1,pathname2      *
0000:                8 *                                             *
0000:                9 * This command copies the file specified by   *
0000:               10 * pathname1, to the file specified by pathname2. *
0000:               11 * If pathname2 does not exist, it is created.  *
0000:               12 * If pathname2 does exist, the command exits    *
0000:               13 * with a DUPLICATE FILENAME error.            *
0000:               14 *                                             *
0000:               15 *       Apple ProDOS Assembler version 1.0    *
0000:               16 *                                             *
0000:               17 ***********************************************
0000:        003C   19 A1L       EQU   $3C          ;Start address for MOVE
0000:        003D   20 A1H       EQU   $3D
0000:        003E   21 A2L       EQU   $3E          ;End address for MOVE
0000:        003F   22 A2H       EQU   $3F
0000:        0042   23 A4L       EQU   $42          ;Destination for MOVE
0000:        0043   24 A4H       EQU   $43
0000:        0073   25 HIMEM     EQU   $73          ;HIMEM pointer
0000:        00FC   26 TEMP      EQU   $FC          ;Temporary storage
0000:        00FD   27 OFFSET    EQU   $FD          ;Offset for relocation code
0000:        00FE   28 PTR       EQU   $FE          ;Temporary address pointer
0000:        00FE   29 COUNT     EQU   $FE          ;Temporary storage
0000:        0200   30 INBUF     EQU   $200         ;Keyboard input buffer
0000:        BE06   31 EXTRNCMD  EQU   $BE06        ;External cmd jmp vector
0000:        BE09   32 ERROUT    EQU   $BE09        ;Print error message
0000:        BE50   33 XTRNADDR  EQU   $BE50        ;External cmd address
0000:        BE52   34 XLEN      EQU   $BE52        ;Length of cmd string-1
0000:        BE53   35 XCNUM     EQU   $BE53        ;cmd no. (extrn. cmd = 0)
0000:        BE54   36 PBITS     EQU   $BE54        ;Command parameter bits
0000:        BE56   37 FBITS     EQU   $BE56        ;Command parameter bits found
0000:        BE6C   38 VPATH1    EQU   $BE6C        ;Pathname 1 buffer pointer
0000:        BE6E   39 VPATH2    EQU   $BE6E        ;Pathname 2 buffer
0000:        BE70   40 GOSYSTEM  EQU   $BE70        ;Perform MLI call
0000:        BE8B   41 BADCALL   EQU   $BE8B        ;Translate MLI error to BI error
0000:        BEB4   42 SGGINFO   EQU   $BEB4        ;GET_FILE_INFO MLI buffer
0000:        BEB7   43 FIACESS   EQU   $BEB7        ;Access used by lock/unlock
0000:        BEB8   44 FIFILID   EQU   $BEB8        ;File id of disk file
0000:        BEB9   45 FIAUXID   EQU   $BEB9        ;Aux id
0000:        BECE   46 OSYSBUF   EQU   $BECE        ;Buffer for MLI OPEN
0000:        BED0   47 OREFNUM   EQU   $BED0        ;Reference number of opened file
0000:        BED2   48 NEWLREF   EQU   $BED2        ;NEWLINE file reference number
0000:        BED3   49 NLINENMSK EQU   $BED3        ;NEWLINE enable mask and char
0000:        BED6   50 RWREFNUM  EQU   $BED6        ;Read file reference number
0000:        BED7   51 RWDATA    EQU   $BED7        ;Read buffer
0000:        BED9   52 RWCOUNT   EQU   $BED9        ;Read (Number of char requested)
0000:        BEDB   53 RWTRANS   EQU   $BEDB        ;Read (Number of char transferred)
0000:        BEDE   54 CFREFNUM  EQU   $BEDE        ;Close file reference number
0000:        BEF5   55 GETBUFR   EQU   $BEF5        ;Get buffer from system
0000:        BEF8   56 FREBUFR   EQU   $BEF8        ;Give buffer back to system
0000:        BEFB   57 RSHIMEM   EQU   $BEFB        ;HIMEM reset value for FREEBUFR
0000:        BF00   58 MLI       EQU   $BF00        ;MLI entry point
0000:        BF58   59 BITMAP    EQU   $BF58        ;System bit map table
0000:        FE2C   60 MOVE      EQU   $FE2C        ;Monitor MOVE routine
----- NEXT OBJECT FILE NAME IS COPY
6000:        6000   61          ORG   $6000
6000:               62 *-----------------------------------------------
6000:               63 * Installation
6000:               64 *-----------------------------------------------
6000:               65 *
6000:               66 *       Get some space for the COPY code
6000:               67 *       and protect it by updating the system bit map
6000:               68 *
6000:AD 03 61       69          LDA   PAGES        ;Number of pages needed
6003:20 F5 BE       70          JSR   GETBUFR      ;Ask the BI for some space
6006:90 03   600B   71          BCC   GOTSPACE     ;We got the space
6008:4C 09 BE       72          JMP   ERROUT       ;Couldn't find space
600B:48             73 GOTSPACE PHA                ;Save start address
600C:AE 03 61       74          LDX   PAGES        ;Number of pages to protect
600F:20 73 60       75 PROTECT  JSR   BITMAPS      ;Protect a page
6012:18             76          CLC
6013:69 01          77          ADC   #$01         ;Point to next page
6015:CA             78          DEX                ;Are we done?
6016:D0 F7   600F   79          BNE   PROTECT      ;If not
6018:AD FB BE       80          LDA   RSHIMEM      ;Protect code from FREEBUFR
601B:38             81          SEC
601C:ED 03 61       82          SBC   PAGES
601F:8D FB BE       83          STA   RSHIMEM
6022:               84 *
6022:               85 *       Hook routine into the ProDOS external command vector
6022:               86 *
6022:AE 07 BE       87          LDX   EXTRNCMD+1   ;Set link to next external command
6025:AC 08 BE       88          LDY   EXTRNCMD+2   ;or RTS (whichever was already there)
6028:8E 01 61       89          STX   LINK+1
602B:8C 02 61       90          STY   LINK+2
602E:A9 0E          91          LDA   #>START      ;Install the address of the
6030:8D 07 BE       92          STA   EXTRNCMD+1   ;COPY command handler in the
6033:68             93          PLA                ;external command jump vector
6034:48             94          PHA                ;Save start address for MOVE
6035:8D 08 BE       95          STA   EXTRNCMD+2
6038:               96 *
6038:               97 *       Update all non-relocatable references in the COPY code
6038:               98 *
6038:38             99          SEC                ;Find offset to be added to all
6039:E9 61         100          SBC   #<LINK       ;non-relocatable references
603B:85 FD         101          STA   OFFSET
603D:A2 00         102          LDX   #$00         ;Index start of table
603F:A0 00         103          LDY   #$00
6041:18            104          CLC
```

(Continued on page 60)

ProDos Copy Listing

(Listing continued, text begins on page 54)

```
6042:BD 91 60      105 RELOCATE  LDA   RELTABLE,X    ;Point to non-relocatable reference
6045:85 FE         106           STA   PTR
6047:BD 92 60      107           LDA   RELTABLE+1,X
604A:85 FF         108           STA   PTR+1
604C:B1 FE         109           LDA   (PTR),Y       ;Add offset to the reference
604E:65 FD         110           ADC   OFFSET
6050:91 FE         111           STA   (PTR),Y
6052:E8            112           INX                  ;Index next reference
6053:E8            113           INX
6054:EC C7 60      114           CPX   TABLESIZE     ;Number of references times 2
6057:90 E9   6042  115           BCC   RELOCATE
6059:              116 *
6059:              117 *         Move the code to its final home
6059:              118 *
6059:A2 00         119           LDX   #>LINK        ;Start
605B:A0 61         120           LDY   #<LINK
605D:86 3C         121           STX   A1L
605F:84 3D         122           STY   A1H
6061:A2 AB         123           LDX   #>END         ;End
6063:A0 62         124           LDY   #<END
6065:86 3E         125           STX   A2L
6067:84 3F         126           STY   A2H
6069:A0 00         127           LDY   #$00          ;Destination (Always on a page boundary)
606B:84 42         128           STY   A4L
606D:68            129           PLA                  ;We saved the high byte on the stack
606E:85 43         130           STA   A4H
6070:4C 2C FE      131           JMP   MOVE          ;Y-reg=$00
6073:              132 *
6073:              133 *         Subroutine to update system bit map
6073:              134 *
6073:A8            135 BITMAPS   TAY                  ;Save page address
6074:48            136           PHA
6075:8A            137           TXA                  ;Save X-reg
6076:48            138           PHA
6077:98            139           TYA                  ;Restore page address
6078:4A            140           LSR   A             ;Divide page address by 8
6079:4A            141           LSR   A
607A:4A            142           LSR   A
607B:AA            143           TAX                  ;Load X-reg with page address/8
607C:98            144           TYA                  ;Restore page address
607D:29 07         145           AND   #$07          ;Kill off the high order bits
607F:A8            146           TAY                  ;and put it in the Y-reg
6080:A9 00         147           LDA   #$00          ;Zero the mask
6082:38            148           SEC                  ;Prepare to put a 1 into mask
6083:6A            149 BITMAPS1  ROR   A             ;Rotate carry into mask
6084:88            150           DEY
6085:10 FC   6083  151           BPL   BITMAPS1      ;Rotate mask to proper position
6087:1D 58 BF      152           ORA   BITMAP,X      ;Update the actual bitmap
608A:9D 58 BF      153           STA   BITMAP,X
608D:68            154           PLA                  ;Restore X-reg and A-reg
608E:AA            155           TAX
608F:68            156           PLA
6090:60            157           RTS
6091:              158 *
6091:              159 *         Table of non-relocatable references
6091:              160 *
6091:1F 61         161 RELTABLE  DW    RELOC1+2
6093:24 61         162           DW    RELOC2+2
6095:2C 61         163           DW    NXTCHR2+2
6097:3E 61         164           DW    RELOC3+1
6099:78 61         165           DW    FILEOK+2
609B:7E 61         166           DW    RELOC4+2
609D:84 61         167           DW    RELOC5+2
609F:89 61         168           DW    RELOC6+2
60A1:8F 61         169           DW    RELOC7+2
60A3:95 61         170           DW    RELOC8+2
60A5:9B 61         171           DW    RELOC9+1
60A7:AE 61         172           DW    RELOC10+2
60A9:B3 61         173           DW    RELOC11+2
60AB:D6 61         174           DW    RELOC12+2
60AD:DF 61         175           DW    RELOC13+2
60AF:E7 61         176           DW    RELOC14+2
60B1:ED 61         177           DW    RELOC15+2
60B3:F3 61         178           DW    RELOC16+2
60B5:F9 61         179           DW    RELOC17+1
60B7:3B 62         180           DW    TRANSFER+2
60B9:3E 62         181           DW    RELOC18+2
60BB:51 62         182           DW    RELOC19+2
60BD:57 62         183           DW    RELOC20+2
60BF:5D 62         184           DW    RELOC21+1
60C1:74 62         185           DW    RELOC22+2
60C3:77 62         186           DW    RELOC23+2
60C5:7D 62         187           DW    RELOC24+1
60C7:36            188 TABLESIZE DFB   TABLESIZE-RELTABLE
60C8:        0038  189 FILLPAGE  DS    >0-FILLPAGE   ;Fill to next page boundary
6100:              190 *-------------------------------------------------
6100:              191 * The following is the COPY command code.
6100:              192 *-------------------------------------------------
6100:4C 00 00      193 LINK      JMP   $0000         ;Address planted to next command
6103:02            194 PAGES     DFB   <END-LINK+256 ;Number of pages occupied
6104:04 43 4F 50   195 CMDSTRU   STR   "COPY"        ;This is the command string
6108:59
6109:04 63 6F 70   196 CMDSTRL   STR   "copy"        ;Accept upper or lower case
610D:79
610E:              197 *-------------------------------------------------
610E:              198 * Check to see if the command is COPY. If not,
610E:              199 * then check for next user command, or tell ProDOS
610E:              200 * it is not a user command. If it is the COPY
610E:              201 * command, then set up for first stage processing.
610E:              202 *-------------------------------------------------
610E:AD 6C BE      203 START     LDA   VPATH1        ;Get command line pointer
6111:85 FE         204           STA   PTR
6113:AD 6D BE      205           LDA   VPATH1+1
6116:85 FF         206           STA   PTR+1
```

(Continued on page 62)

ProDos Copy Listing

(Listing continued, text begins on page 54)

```
6118:A0 00          207             LDY   #$00          ;Index command string
611A:C8             208 NXTCHR      INY                 ;Increment index
611B:B1 FE          209             LDA   (PTR),Y       ;Get character from input string
611D:D9 04 61       210 RELOC1      CMP   CMDSTRU,Y     ;Does it match upper case?
6120:F0 08   612A   211             BEQ   NXTCHR2       ;If yes
6122:D9 09 61       212 RELOC2      CMP   CMDSTRL,Y     ;Does it match lower case?
6125:F0 03   612A   213             BEQ   NXTCHR2       ;If yes
6127:38             214             SEC                 ;Flag match failure
6128:B0 D6   6100   215             BCS   LINK          ;Go to next command in the chain
612A:CC 04 61       216 NXTCHR2     CPY   CMDSTRU       ;Checked all characters?
612D:D0 EB   611A   217             BNE   NXTCHR        ;No, so check the next one
612F:88             218             DEY                 ;Put command length-1 in XLEN
6130:8C 52 BE       219             STY   XLEN
6133:A9 00          220             LDA   #$00          ;Mark command as external
6135:8D 53 BE       221             STA   XCNUM
6138:A9 4E          222             LDA   #>SECOND      ;Point to command handler
613A:8D 50 BE       223             STA   XTRNADDR
613D:A9 61          224 RELOC3      LDA   #<SECOND
613F:8D 51 BE       225             STA   XTRNADDR+1
6142:A9 0B          226             LDA   #$0B          ;Allow CREATE, require two pathnames
6144:8D 54 BE       227             STA   PBITS
6147:A9 04          228             LDA   #$04          ;Fetch prefix, if required
6149:8D 55 BE       229             STA   PBITS+1
614C:18             230             CLC                 ;No errors so far
614D:60             231             RTS                 ;Return to BASIC.SYSTEM
614E:               232 *-------------------------------------------
614E:               233 * COPY second stage processing. First check PBITS
614E:               234 * to see if the proper parameters were present.
614E:               235 *-------------------------------------------
614E:AD 56 BE       236 SECOND      LDA   PBITS         ;Parameters found
6151:4A             237             LSR   A             ;Filename found?
6152:90 03   6157   238             BCC   SYNERROR      ;No, so give error message
6154:4A             239             LSR   A             ;Second filename found?
6155:B0 04   615B   240             BCS   PARMSOK       ;Yes, so go do command
6157:A9 10          241 SYNERROR    LDA   #$10          ;Syntax error
6159:38             242 MLIERR      SEC                 ;Set carry to show error
615A:60             243 RETURN      RTS                 ;Return to system
615B:               244 *
615B:               245 *           Both filenames were given, so execute COPY
615B:               246 *
615B:A9 0A          247 PARMSOK     LDA   #$0A          ;Set up GET_FILE_INFO call
615D:8D B4 BE       248             STA   SSGINFO
6160:A9 C4          249             LDA   #$C4          ;MLI GET_FILE_INFO code
6162:20 70 BE       250             JSR   GOSYSTEM      ;Perform the MLI call
6165:B0 F3   615A   251             BCS   RETURN        ;If error, return carry set
6167:AD B8 BE       252             LDA   FIFILID       ;Get file type
616A:C9 0F          253             CMP   #$0F          ;Can't copy a directory file
616C:F0 04   6172   254             BEQ   MISMATCH
616E:C9 01          255             CMP   #$01          ;Can't copy a bad block file
6170:D0 04   6176   256             BNE   FILEOK
6172:A9 0D          257 MISMATCH    LDA   #13           ;FILE TYPE MISMATCH error
6174:D0 E3   6159   258             BNE   MLIERR        ;Always
6176:               259 *
6176:               260 *           File type is OK
6176:               261 *
6176:8D 93 62       262 FILEOK      STA   CRFILID       ;File type for CREATE
6179:AD B9 BE       263             LDA   FIAUXID       ;File aux type for CREATE
617C:8D 94 62       264 RELOC4      STA   CRAUXID
617F:AD BA BE       265             LDA   FIAUXID+1
6182:8D 95 62       266 RELOC5      STA   CRAUXID+1
6185:A9 C3          267             LDA   #$C3          ;File access (UNLOCKED)
6187:8D 92 62       268 RELOC6      STA   CRACCESS      ;We have to be able to write!
618A:AD 6E BE       269             LDA   VPATH2        ;Pathname for CREATE
618D:8D 90 62       270 RELOC7      STA   CRPATH
6190:AD 6F BE       271             LDA   VPATH2+1
6193:8D 91 62       272 RELOC8      STA   CRPATH+1
6196:20 00 BF       273 CREATE      JSR   MLI           ;Create the new file
6199:C0             274             DFB   $C0          ;MLI CREATE command code
619A:8F 62          275 RELOC9      DW    CRPARMS       ;CREATE parameter list
619C:90 03   61A1   276             BCC   SUCCESS       ;CREATE successful
619E:4C 8B BE       277             JMP   BADCALL       ;Translate to BI error code and exit
61A1:               278 *
61A1:               279 *           Create a file buffer for the destination file
61A1:               280 *
61A1:A9 04          281 SUCCESS     LDA   #$04          ;1K buffer required
61A3:20 F5 BE       282             JSR   GETBUFR       ;Get space
61A6:90 04   61AC   283             BCC   RELOC10
61A8:A9 0C          284             LDA   #$0C          ;NO BUFFERS AVAILABLE error
61AA:38             285             SEC
61AB:60             286             RTS
61AC:8D 9F 62       287 RELOC10     STA   OPEN2BUF+1
61AF:A9 00          288             LDA   #$00          ;Buffer always on page boundary
61B1:8D 9E 62       289 RELOC11     STA   OPEN2BUF
61B4:               290 *
61B4:               291 *           Create a transfer buffer for the file
61B4:               292 *           Use as much free memory as possible
61B4:               293 *
61B4:A9 FF          294 TRANSBUF    LDA   #$FF          ;Count how many pages we can get
61B6:85 FE          295             STA   COUNT         ;Initialize with -1
61B8:E6 FE          296 GETMORE     INC   COUNT         ;Count how many pages
61BA:A9 01          297             LDA   #$01          ;Get one page at a time
61BC:20 F5 BE       298             JSR   GETBUFR       ;Get space
61BF:90 F7   61B8   299             BCC   GETMORE       ;Keep going until we can't get more
61C1:A5 FE          300             LDA   COUNT         ;Check how many pages we got
61C3:D0 07   61CC   301             BNE   GOTENOUGH     ;Any non-zero number is okay
61C5:20 F8 BE       302             JSR   FREBUFR       ;Give back the allocated memory
61C8:A9 0C          303             LDA   #$0C          ;NO BUFFERS AVAILABLE error
61CA:38             304             SEC
61CB:60             305             RTS
61CC:8D DA BE       306 GOTENOUGH   STA   RWCOUNT+1     ;Save buffer size
61CF:A9 00          307             LDA   #$00          ;Length is always an even number of page
61D1:8D D9 BE       308             STA   RWCOUNT
61D4:AD 9F 62       309 RELOC12     LDA   OPEN2BUF+1    ;Get start address of buffer
61D7:38             310             SEC
```

(Continued on page 66)

ProDos Copy Listing

(Listing continued, text begins on page 54)

```
61D8:E5 FE          311           SBC   COUNT
61DA:8D D8 BE       312           STA   RWDATA+1      ;Save buffer start address
61DD:8D A4 62       313 RELOC13   STA   WR2BUF+1
61E0:A9 00          314           LDA   #$00          ;Buffer starts on a page boundary
61E2:8D D7 BE       315           STA   RWDATA
61E5:8D A3 62       316 RELOC14   STA   WR2BUF
61E8:               317 *
61E8:               318 *         Open the destination file
61E8:               319 *
61E8:AD 6E BE       320           LDA   VPATH2        ;Pathname pointer
61EB:8D 9C 62       321 RELOC15   STA   OPEN2PATH
61EE:AD 6F BE       322           LDA   VPATH2+1
61F1:8D 9D 62       323 RELOC16   STA   OPEN2PATH+1
61F4:20 00 BF       324           JSR   MLI           ;Perform MLI call
61F7:C8             325           DFB   $C8           ;MLI OPEN command code
61F8:9B 62          326 RELOC17   DW    OPEN2PARM
61FA:90 08   6204   327           BCC   OPENSRC       ;No errors, so open the source file
61FC:48             328           PHA                  ;Save error code
61FD:20 F8 BE       329           JSR   FREBUFR       ;Release buffer
6200:68             330           PLA                  ;Restore error code
6201:4C 8B BE       331           JMP   BADCALL       ;Translate error code and exit
6204:               332 *
6204:               333 *         Open source file
6204:               334 *
6204:A5 74          335 OPENSRC   LDA   HIMEM+1       ;Get buffer start
6206:8D CF BE       336           STA   OSYSBUF+1     ;(use general purpose buffer)
6209:A9 00          337           LDA   #$00          ;Buffer always on a page boundary
620B:8D CE BE       338           STA   OSYSBUF
620E:A9 C8          339           LDA   #$C8          ;MLI open code
6210:20 70 BE       340           JSR   GOSYSTEM      ;Perform the MLI call
6213:B0 1D   6232   341           BCS   ERRORET       ;Exit if error detected
6215:AD D0 BE       342           LDA   OREFNUM       ;Set up file reference number
6218:8D D2 BE       343           STA   NEWLREF       ;For NEWLINE call
621B:8D D6 BE       344           STA   RWREFNUM      ;For READ call
621E:8D DE BE       345           STA   CFREFNUM      ;For CLOSE call
6221:A9 00          346           LDA   #$00          ;Set up NEWLINE call
6223:8D D3 BE       347           STA   NLINENBL      ;to disable NEWLINE char
6226:A9 0D          348           LDA   #$0D          ;Non-zero
6228:8D D4 BE       349           STA   NLINENBL+1
622B:A9 C9          350           LDA   #$C9          ;MLI NEWLINE code
622D:20 70 BE       351           JSR   GOSYSTEM      ;Perform the MLI call
6230:90 07   6239   352           BCC   TRANSFER      ;If no errors detected
6232:48             353 ERRORET   PHA                  ;Save error code
6233:20 65 62       354           JSR   CLEANUP       ;Close files and free buffer
6236:68             355           PLA                  ;Restore error code
6237:38             356           SEC                  ;Flag error
6238:60             357           RTS                  ;Return to system
6239:               358 *
6239:               359 *         Both files are now open so perform transfer
6239:               360 *
6239:AD A0 62       361 TRANSFER  LDA   OPEN2REF      ;Set up WRITE file reference number
623C:8D A2 62       362 RELOC18   STA   WR2REF
623F:A9 CA          363 XFERLOOP  LDA   #$CA          ;MLI READ code
6241:20 70 BE       364           JSR   GOSYSTEM      ;Perform the MLI call
6244:90 06   624C   365           BCC   XFEROK        ;No errors
6246:C9 05          366           CMP   #$05          ;End of file?
6248:F0 1B   6265   367           BEQ   CLEANUP       ;Yes, so clean up and exit
624A:D0 E6   6232   368           BNE   ERRORET       ;Exit with appropriate error
624C:AD DB BE       369 XFEROK    LDA   RWTRANS       ;Number of bytes transferred
624F:8D A5 62       370 RELOC19   STA   WR2REQ        ;Request to write these bytes
6252:AD DC BE       371           LDA   RWTRANS+1
6255:8D A6 62       372 RELOC20   STA   WR2REQ+1
6258:20 00 BF       373           JSR   MLI           ;Write the data to the destination file
625B:CB             374           DFB   $CB           ;MLI WRITE code
625C:A1 62          375 RELOC21   DW    WR2PARMS
625E:90 DF   623F   376           BCC   XFERLOOP
6260:20 8B BE       377           JSR   BADCALL       ;Translate to BI error code
6263:B0 CD   6232   378           BCS   ERRORET       ;Always
6265:               379 *---------------------------------------------
6265:               380 * Close open files and free the allocated buffer space
6265:               381 *---------------------------------------------
6265:A9 00          382 CLEANUP   LDA   #$00          ;Default to no errors detected
6267:85 FC          383           STA   TEMP
6269:A9 CC          384           LDA   #$CC          ;MLI CLOSE code
626B:20 70 BE       385           JSR   GOSYSTEM      ;Close source file
626E:90 02   6272   386           BCC   RELOC22       ;If no error detected
6270:85 FC          387           STA   TEMP          ;Update error
6272:AD A0 62       388 RELOC22   LDA   OPEN2REF      ;Close destination file
6275:8D AA 62       389 RELOC23   STA   CLOSE2REF
6278:20 00 BF       390           JSR   MLI           ;Perform the MLI call
627B:CC             391           DFB   $CC           ;MLI CLOSE code
627C:A9 62          392 RELOC24   DW    CL2PARMS      ;CLOSE parameter list
627E:90 05   6285   393           BCC   FREE          ;If no error detected
6280:20 8B BE       394           JSR   BADCALL       ;Translate to BI error
6283:85 FC          395           STA   TEMP          ;Update error
6285:20 F8 BE       396 FREE      JSR   FREBUFR       ;Deallocate file buffer
6288:18             397           CLC                  ;Default to no errors detected
6289:A5 FC          398           LDA   TEMP          ;Error code in A
628B:F0 01   628E   399           BEQ   RET           ;Zero if no errors
628D:38             400           SEC                  ;Flag error
628E:60             401 RET       RTS
628F:               402 *---------------------------------------------
628F:               403 * Parameter list for CREATE
628F:               404 *---------------------------------------------
628F:07             405 CRPARMS   DFB   $07           ;Parameter count
6290:       0002    406 CRPATH    DS    2             ;Pointer to pathname
6292:       0001    407 CRACCESS  DS    1             ;Access permitted byte
6293:       0001    408 CRFILID   DS    1             ;File type
6294:       0002    409 CRAUXID   DS    2             ;File aux type
6296:01             410           DFB   $01           ;Standard file (Not directory)
6297:00 00          411 CRDATE    DFB   $00,$00       ;Create date
6299:00 00          412 CRTIME    DFB   $00,$00       ;Create time
629B:               413 *---------------------------------------------
629B:               414 * Parameter list for OPEN
629B:               415 *---------------------------------------------
```

(Continued on page 68)

ProDos Copy Listing

(Listing continued, text begins on page 54)

```
629B:03          416 OPEN2PARM DFB   $03          ;Parameter count
629C:      0002  417 OPEN2PATH DS    2            ;Pointer to pathname
629E:      0002  418 OPEN2BUF  DS    2            ;Pointer to I/O buffer
62A0:      0001  419 OPEN2REF  DS    1            ;File reference number
62A1:            420 *---------------------------------------------
62A1:            421 * Parameter list for WRITE
62A1:            422 *---------------------------------------------
62A1:04          423 WR2PARMS  DFB   $04          ;Parameter count
62A2:      0001  424 WR2REF    DS    1            ;File reference number
62A3:      0002  425 WR2BUF    DS    2            ;File transfer buffer
62A5:      0002  426 WR2REQ    DS    2            ;Number of bytes to transfer
62A7:      0002  427 WR2TRANS  DS    2            ;Number of bytes transferred
62A9:            428 *---------------------------------------------
62A9:            429 * Parameter list for CLOSE
62A9:            430 *---------------------------------------------
62A9:01          431 CL2PARMS  DFB   $01          ;Parameter count
62AA:      0001  432 CLOSE2REF DS    1            ;File reference number
62AB:      62AB  433 END       EQU   *
```

```
   3D A1H           3C A1L           3F A2H           3E A2L
   43 A4H           42 A4L         BE8B BADCALL     6083 BITMAPS1
 6073 BITMAPS     BF58 BITMAP      BEDE CFREFNUM     62A9 CL2PARMS
 6265 CLEANUP     62AA CLOSE2REF   6109 CMDSTRL      6104 CMDSTRU
   FE COUNT       6292 CRACCESS    6294 CRAUXID     ?6297 CRDATE
?6196 CREATE      6293 CRFILID     628F CRPARMS      6290 CRPATH
?6299 CRTIME      62AB END         6232 ERRORET      BE09 ERROUT
 BE06 EXTRNCMD    BE56 FBITS      ?BEB7 FIACESS      BEB9 FIAUXID
 BEB8 FIFILID     6176 FILEOK      60C8 FILLPAGE     BEF8 FREBUFR
 6285 FREE        BEF5 GETBUFR     61B8 GETMORE      BE70 GOSYSTEM
 61CC GOTENOUGH   600B GOTSPACE       73 HIMEM     ?0200 INBUF
 6100 LINK        6172 MISMATCH    BF00 MLI          6159 MLIERR
 FE2C MOVE        BED2 NEWLREF     BED3 NLINENBL     611A NXTCHR
 612A NXTCHR2       FD OFFSET\     629E OPEN2BUF     629B OPEN2PARM
 629C OPEN2PATH   62A0 OPEN2REF    6204 OPENSRC      BED0 OREFNUM
 BECE OSYSBUF     6103 PAGES       615B PARMSOK      BE54 PBITS
 600F PROTECT       FE PTR         61AC RELOC10      61B1 RELOC11
 61D4 RELOC12     61E5 RELOC14     611D RELOC1       61DD RELOC13
 61EB RELOC15     61F1 RELOC16     61F8 RELOC17      623C RELOC18
 624F RELOC19     6255 RELOC20     625C RELOC21      6272 RELOC22
 6275 RELOC23     6122 RELOC2      627C RELOC24      613D RELOC3
 617C RELOC4      6182 RELOC5      6187 RELOC6       618D RELOC7
 6193 RELOC8      619A RELOC9      6042 RELOCATE     6091 RELTABLE
 628E RET         615A RETURN      BEFB RSHIMEM      BED9 RWCOUNT
 BED7 RWDATA      BED6 RWREFNUM    BEDB RWTRANS      614E SECOND
 BEB4 SSGINFO     610E START       61A1 SUCCESS      6157 SYNERROR
 60C7 TABLESIZE     FC TEMP        ?61B4 TRANSBUF     6239 TRANSFER
 BE6C VPATH1      BE6E VPATH2      62A3 WR2BUF       62A1 WR2PARMS
 62A2 WR2REF      62A5 WR2REQ      ?62A7 WR2TRANS    BE53 XCNUM
 623F XFERLOOP    624C XFEROK       BE52 XLEN        BE50 XTRNADDR
```

```
   3C A1L           3D A1H           3E A2L           3F A2H
   42 A4L           43 A4H           73 HIMEM         FC TEMP
   FD OFFSET          FE PTR         FE COUNT       ?0200 INBUF
 600B GOTSPACE    600F PROTECT     6042 RELOCATE     6073 BITMAPS
 6083 BITMAPS1    6091 RELTABLE    60C7 TABLESIZE    60C8 FILLPAGE
 6100 LINK        6103 PAGES       6104 CMDSTRU      6109 CMDSTRL
 610E START       611A NXTCHR      611D RELOC1       6122 RELOC2
 612A NXTCHR2     613D RELOC3      614E SECOND       6157 SYNERROR
 6159 MLIERR      615A RETURN      615B PARMSOK      6172 MISMATCH
 6176 FILEOK      617C RELOC4      6182 RELOC5       6187 RELOC6
 618D RELOC7      6193 RELOC8      619A RELOC9      ?6196 CREATE
 61A1 SUCCESS     61AC RELOC10     61B1 RELOC11     ?61B4 TRANSBUF
 61B8 GETMORE     61CC GOTENOUGH   61D4 RELOC12      61DD RELOC13
 61E5 RELOC14     61EB RELOC15     61F1 RELOC16      61F8 RELOC17
 6204 OPENSRC     6232 ERRORET     6239 TRANSFER     623C RELOC18
 623F XFERLOOP    624C XFEROK      624F RELOC19      6255 RELOC20
 625C RELOC21     6265 CLEANUP     6272 RELOC22      6275 RELOC23
 627C RELOC24     6285 FREE        628E RET          628F CRPARMS
 6290 CRPATH      6292 CRACCESS    6293 CRFILID      6294 CRAUXID
?6297 CRDATE     ?6299 CRTIME      629B OPEN2PARM    629C OPEN2PATH
 629E OPEN2BUF    62A0 OPEN2REF    62A1 WR2PARMS     62A2 WR2REF
 62A3 WR2BUF      62A5 WR2REQ     ?62A7 WR2TRANS     62A9 CL2PARMS
 62AA CLOSE2REF   62AB END         BE06 EXTRNCMD     BE09 ERROUT
 BE50 XTRNADDR    BE52 XLEN        BE53 XCNUM        BE54 PBITS
 BE56 FBITS       BE6C VPATH1      BE6E VPATH2       BE70 GOSYSTEM
 BE8B BADCALL     BEB4 SSGINFO    ?BEB7 FIACESS      BEB8 FIFILID
 BEB9 FIAUXID     BECE OSYSBUF     BED0 OREFNUM      BED2 NEWLREF
 BED3 NLINENBL    BED6 RWREFNUM    BED7 RWDATA       BED9 RWCOUNT
 BEDB RWTRANS     BEDE CFREFNUM    BEF5 GETBUFR      BEF8 FREBUFR
 BEFB RSHIMEM     BF00 MLI         BF58 BITMAP       FE2C MOVE
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 15-JAN-84 21:28
** TOTAL LINES ASSEMBLED   433
** FREE SPACE PAGE COUNT   79
```

End Listing

CP/M-68K Conditional Batch Processing

by Roger E. Donais

Anyone who uses batch processing is certain to recognize the value of conditional instructions. A few basic submit statements, such as SKIPIF, PAUSEIF, and ABORTIF have been added to CP/M-80 in the past. These instructions make it possible for compiling, assembling, and linking to be handled by a single submit file that is smart enough to stop the process if any phase should fail.

CP/M-68K and CP/M-80 are similar in many ways. In fact, files and programs can share the same disk without adverse consequences. They are, however, as different as the processors they support. Because CP/M-80 creates a working copy of a submit file, that file becomes the natural target for implementing batch control functions. On the other hand, CP/M-68K stores the expanded file in memory. The fact is that a direct approach would require more knowledge about the undocumented aspects of CP/M-68K than can be acquired in an evening!

A simple "if" can greatly speed the software development process.

IF.68K (Listing One, page 72) began as two outlines. The first listed what was wanted and the other described how it would be accomplished. One pencil later the following compromise was reached:

Syntax: The conditional statement is structured so as to minimize programming effort by taking advantage of the parsing done by the CCP, and yet present a recognizable form of conditional statement.

IF <parameter> <condition>
.<action> /$$$$/ <statement>

Condition: The first four characters taken from the file-name field of the file control block created by the CCP are used to identify the condition to be tested. An asterisk may be appended to indicate that the complementary condition is to be used.

AMBIG tests for an ambiguous parameter (wildcards).
EXISTS tests for an existing file.
EMPTY tests for an empty file or nonexistent file.
NULL tests for an undefined parameter.

Actions: The first character taken from the file-type field of the same file control block is used to identify the desired action. An asterisk may again be appended to indicate that the complementary action is to be taken.

S(kip) ignores the statement following /$$$$/.
Q(uit) aborts submit file processing.
P(ause) prompts for operator assistance.

The double-slash quad-dollar-sign token separates the conditional statement from the executable CP/M com-

Roger E. Donais, 7506 Republic Ct., Alexandria, VA 22306

mand tail. When the submit file is expanded, a dollar sign is used as the leading character to signal formal parameter replacement. If the next character is not an ASCII digit, the character is written to the expanded output and no replacement is performed. This simply means that we use four dollar signs in the submit file but have the program search for only two.

One disadvantage with this compromise approach is that CP/M-68K will pass only one command. This limits the range of the conditional statement to one CP/M-68K command. It's bad enough that the same conditional statement has to be entered over and over again in order to control a range of commands, but the aggravation of watching it reload each time is just too much!

With a little camouflage, CP/M can be tricked into passing more than one command. The best character seems to be a semicolon. Very few programs dare use this character because it marks the beginning of a comment. This makes it a prime candidate for an alternate command separator. It affects very few programs and conceals its true intent from CP/M, and our program need only replace these bogus separators with the exclamation marks that CP/M-68K expects to find.

The problem with this wonderful logic is that CP/M-68K seems to stop building the command tail after it finds a semicolon. Because the basic technique could use any character, a backslash became the next choice. Although the entire command tail is passed, the chain function does not protect the referenced line but leaves it where it is. It doesn't take much imagination to realize what happens when the default buffer is overwritten. So we are limited to using it with built-in commands and the aggravation of using only one external command.

Let's look at the resulting kludge. The program starts by looking for an action character in the second file control block. If it fails to find a match, as would be the case with a null parameter, it flips a switch before testing the first file control block. Control is then transferred according to the indicated action.

Each action begins by making a call to evaluate the condition. This evaluation ends with a jump that complements the original condition according to a NOT condition and a NOT action request. This is definitely a kludge that capitalizes upon the parsing and setup done by the CCP. The file-name and file-type fields are normally padded with spaces (an even $20). The presence of an asterisk (CP/M wildcard) changes the padding character to a question mark (an odd $3F). The code uses this even/odd fill character to blindly obtain the desired state.

The two complementary labels, TRUE and FALSE, are truly illusionary. What began as a simple matter of going to one or the other, based upon the resulting condition, turned into quite a nightmare. At times it seemed that neither one was right. Had they been ZIP and ZAP they might not have confused the issue so much. Debugging eventually reached a point where I would no longer look at them or at the remarks. It became, "Well, it's BEQ now, so let's try BNE" or "target is TRUE, so try FALSE!" It became worse regardless of what I tried. Finally, everything was wrong. That made it easy. TRUE became FALSE and FALSE became TRUE. I no longer care how it looks nor if it makes sense. It works and that is the end of it!

A skip is performed by merely ending the program and returning to the CCP. The command is executed by invoking the CP/M-68K chain function after replacing all semicolons with exclamation marks.

There may be a more direct way to abort than the method used. Programs, however, evolve according to a programmer's knowledge and imagination. My 68000 and CP/M-68K knowledge is very limited, so that left only imagination. The *CP/M User's Guide* says that a submit statement contained in a submit file would transfer control if the referenced file exists. Otherwise the statement would be ignored and processing would resume with the next statement in the current file.

The solution is obvious: invoke an existing, empty submit file—*et voilà*! The actual code goes one step more. It creates a submit file with a single instruction that subsequently erases itself. The result may not be fast, but it does abort without having to keep more junk on the disk.

Yes, the whole thing has the potential to be fooled and foiled. But if you remain within the guidelines and don't wander beyond its incomplete parsing and blind processing, it performs exactly as intended. The crazy little submit file in Listing Two (page 82) may well be the world's slowest ERAQ, but it does provide an example to demonstrate the performance of IF.68K. The additional example in Listing Three (page 82) not only conditionally assembles and links the modules of an assembly project but also copies the changed source files from a RAM disk to floppy. Naturally, execution speed is improved by loading IF.68K from a RAM disk rather than floppy.

DDJ

Listing One

```
*****************************************************************
*                                                               *
* PROGRAM:  IF.68K                  LAST UPDATE:  10 Mar 85     *
*                                                               *
* PURPOSE:  PROVIDE BASIC CONTROL FOR CP/M-68k SUBMIT FACILITY  *
*                                                               *
* AUTHOR:   Roger E. Donais         TEL: (703) 765-0615         *
*           7506 Republic Ct.                                   *
*           Alexandria, VA 22306                                *
*                                                               *
* UPDATE LOG:                                                   *
*                                                               *
* 1.2   10 Mar 85 --- Replaced non-working semi-colon with      *
*                        back-slash as command separator and    *
*                        added ^Z check for empty file  ...red  *
* 1.1   14 Nov 84 --- Added semi-colon as muliple command      *
*                        separator                       ...red *
* 1.0   11 Jul 84                                              *
*                                                               *
*****************************************************************
```

```
*                                                                   *
* SYNTAX:                                                           *
*                                                                   *
*     IF <Filename> <Condition>.<Action> /$$$$/ <statement>          *
*                                                                   *
*     CONDITION IS: AMBIG - TO TEST FOR AMBIGIOUS FILESPEC          *
*                   EXIST - TO TEST FOR AN EXISTING FILE            *
*                   EMPTY - TO TEST FOR AN EMPTY FILE               *
*                   NULL  - TO TEST FOR A NULL PARAMETER            *
*                                                                   *
*     ACTION IS:    S - TO SKIP STATMENT PART OF COMMAND LINE       *
*                   Q - TO QUIT SUBMIT FILE PROCESS                 *
*                   P - TO PAUSE FOR OPERATOR ASSISTANCE            *
*                                                                   *
*     STATEMENT IS ANY VALID CP/M COMMAND                           *
*                                                                   *
*     NOTE:  AN ASTERISK (*) MAY BE APPENDED TO THE CONDITION       *
*            AND/OR THE ACTION TO INDICATE NEGATION.  AMBIG*        *
*            WOULD THUS TRANSLATE "NOT AMBIGIOUS" AND Q* WOULD      *
*            TRANSLATE "DO NOT QUIT".                               *
*                                                                   *
*        MULTIPLE COMMANDS MAY BE SEPARATED BY SEMI-COLONS, AN      *
*        EXCLAIMATION MARK (!) OR END OF LINE ENDS STATEMENT.       *
*                                                                   *
*********************************************************************

* CP/M FUNCTION DEFINITIONS
bdos            equ  2
fnABORT         equ  0
fnCINP          equ  1
fnCOUT          equ  2
fnPRINT         equ  9
fnRESET         equ 13
fnOPEN          equ 15
fnCLOSE         equ 16
fnDELETE        equ 19
fnREAD          equ 20
fnWRITE         equ 21
fnCREATE        equ 22
fnDMA           equ 26
fnCHAIN         equ 47

.page
* MISC CONSTANTS
FCB1            equ $5C         1st CP/M File Control Block
FCB2            equ $38         2nd CP/M File Control Block
NAM1            equ FCB1+1      1st FCB File Name field
NAM2            equ FCB2+1      2nd FCB File Name field
TYP1            equ NAM1+8      1st FCB File Type field
TYP2            equ NAM2+8      2nd FCB File Type field
BUFF            equ $80         Default CP/M Buffer / Command Line
SIZE            equ $0F+FCB1    CP/M File Size (Sector Count) FOR FCB1


*****************************************************************
* PHASE-1 --- PROCESS (P)AUSE), (S)KIP  and (Q)UIT PARAMETERS

        MOVE.L  4(A7),A6        A6 Contains Base Page
        LEA     128(A6),A3      DEFAULT BUFFER
        MOVE.B  TYP2(A6),D0     Get Action Character - 2nd FCB
RETRY:
        CMP.B   #'S',D0
        BEQ     SKIP
        CMP.B   #'Q',D0
        BEQ     QUIT
        CMP.B   #'P',D0
        BEQ     PAUSE
        MOVE.B  TYP1(A6),D0     Get Action Character - 1st FCB
        EOR #1,NULLFLG
        BNE     RETRY
ERROR:
        MOVE.L #MESSAGE,A1
ERROR1:
        MOVE.B (A1)+,D1
        BEQ  ERROR2
        EXT  D1
        MOVE #fnCOUT,D0
        TRAP #bdos
        BRA  ERROR1
ERROR2:
        MOVE #fnABORT,D0
        TRAP #bdos
```

CP/M-68K (Listing continued, text begins on page 70)

Listing One

```
NULLFLG:
        DC.W 0000

.page
*****************************************************************
* PHASE-2 --- PROCESS AMBIG, EXISTS, EMPTY and NULL PARAMETERS
COMPARE:
        MOVE #4-1,d0            4 Character token count
        LEA  NAM2(A6),A0        Get Token address
        TST NULLFLG
        BEQ L1
        LEA NAM1(A6),A0
L1:     LSL.L #8,D2             Make room for next character
        MOVE.B (A0)+,D2           and load it
        DBRA D0,L1              Repeat for all 4-characters

        MOVE.L #TABLE,A0        Point to Function Table
        MOVE   #4-1,D0          Number of Entries
L2:     MOVE.L (A0)+,D1         Get Current Entry
        CMP.L   D2,D1
        BEQ  JUMP               Exit on Match
        ADDA #4,A0              Else Step to Next Entry
        DBRA D0,L2                and Repeat for ALL Entries
L3:     ADDA #4,A7              Pop callers from stack
        BRA  ERROR                and abort
JUMP:
        TST NULLFLG
        BEQ JMP1                EVERYTHING GOOD ON 2ND FCB
        TST D0
        BNE L3                  AND ONLY NULL ON 1ST FCB
JMP1:   MOVE.L (A0),A0          JUMP TO FUNCTION
        JMP (A0)
.page
*****************************************************************
SKIP:   * SKIP COMMAND TAIL IF CONDITION TRUE
        BSR COMPARE
        BNE CHAIN               Execute Command Tail if FALSE
EXIT:   RTS                     and Ignore if TRUE

*****************************************************************
QUIT:   * QUIT (ABORT) SUBMIT IF CONDITION TRUE
        BSR COMPARE
        BNE CHAIN               Ignore Command Tail if FALSE
ABORT:
        MOVE.L  #ABORTMSG,D1
        MOVE    #fnPRINT,D0
        TRAP    #bdos
        MOVE    #fnRESET,D0
        TRAP    #bdos           Make Certain Drive A: is R/W
        MOVE.L  #ABORTREC,D1
        MOVE    #fnDMA,D0       Setup for File Output
        TRAP    #bdos
        MOVE.L  #ABORTFCB,D1
        MOVE    #fnDELETE,D0    Delete any '$$$ABORT.SUB' File
        TRAP    #bdos
        MOVE    #fnCREATE,D0    Then make a new one
        TRAP    #bdos
        MOVE    #fnWRITE,D0     Fill it
        TRAP    #bdos
        MOVE    #fnCLOSE,D0     Close it
        TRAP    #bdos
        MOVE.L  #ABORTSUB,D1    and Chain to it
        BRA     CHAIN1


*****************************************************************
PAUSE:  * PAUSE IF CONDITION TRUE
        BSR COMPARE
        BNE CHAIN               EXECUTE W/O STOPPING IF FALSE
   X9:
        MOVE.L #PROMPT,D1
        MOVE #fnPRINT,D0
        TRAP #bdos
        MOVE #fnCINP,D0
```

Listing One

```
        TRAP #bdos
        AND  #$5F,D0            Force upper case
        CMP.B   #'Q',D0
        BEQ     ABORT           Quit on "Q"
        CMP.B   #'S',D0
        BEQ     EXIT            Skip on "S"
        CMP.B   #'C',D0
        BNE     X9              And continue on "C"
.page
CHAIN:  * CHAIN TO COMMAND TAIL

        ADD     #BUFF,A6        Point to command line
        MOVE.B  (A6)+,D0        Get command length
        EXT     D0

X5:     DBRA D0,X6              Ignore Statement if Command
        RTS                       Tail cannot be found

X6:     CMP.B #'/',(A6)+
        BNE X5                  Look for leading slash

        CMP.B #' ',-2(A6)       Insure that the previous
        BEQ X7                    character was either
        CMP.B #9,-2(A6)           a space or tab
        BNE X5
X7:
        CMP.B #'$',(A6)+        The Original four dollar
        BNE X5                    signs will appear
        CMP.B #'$',(A6)+          as only two, so
        BNE X5                    check for 2 $'s


        CMP.B #'/',(A6)+        Check for trailing slash
        BNE X5

        CMP.B #' ',(A6)+        And finally delimiting
        BEQ X8                    space or tab
        CMP.B #9,-1(A6)
        BNE X5
X8:
* So we got the command tail. Let's take a quick break
* and repalce all semicolons with exclaimation marks.
        move.l A6,A1            copy pointer and
        move   D0,D1            character count,
        bra Y10                 then begin search
Y9:
          move.b #'!',-1(A1)
Y10:
            cmp.b #'\',(A1)+
          dbeq D1,Y10           search until eol or '\'
        beq Y9                  substitute & continue if '\'

        SUB #1,A6               Step back to new len position
        ADD #1,D0                 adjust remaining command length
        MOVE.B D0,(A6)            and punch it
        MOVE.L A6,D1            Put the command tail in D1
CHAIN1:
        MOVE #fnDMA,D0          Make addr in D1
        TRAP #bdos              the Default Buffer
        MOVE #fnCHAIN,D0
        TRAP #bdos              and Chain it

*****************************************************************
OPEN:
        MOVE.L  A6,D1           Base page to D1
        ADD.L   #FCB1,D1          and step to fcb
        MOVE    #fnOPEN,D0
        TRAP #bdos
        CMP.B   #$FF,D0         Set flags on open file
        RTS
.page
NULL1:  CMP.B #$20,NAM1(A6)     Test for no file name
        BNE TRUE
```

Listing One

```
FALSE:  MOVE #0,D0              Start with FALSE
        BRA TOGGLE

TRUE:   MOVE #1,D0              Start with TRUE
TOGGLE: EOR.B D0,TYP2-1(A6)     TOGGLE "NOT CONDITION"
        MOVE.B TYP2-1(A6),D0
        EOR.B D0,TYP2+2(A6)     TOGGLE "NOT ACTION"
        BTST #0,TYP2+2(A6)      FLAG RESULT
        RTS

*****************************************************************
NULL:   * TEST FIRST FCB FOR NULL PARAMETER

        TST NULLFLG             If FCB2 is in use the
        BEQ NULL1               Condition is FCB1 Filename
        EOR.B  D0,TYP1-1(A6)    else TOGGLE
        MOVE.B TYP1-1(A6),D0    using FCB1 Pad Characters
        EOR.B  D0,TYP1+2(A6)
        BTST #0,TYP1+2(A6)
        RTS

*****************************************************************

EMPTY:  * TEST FIRST FCB FOR NULL, NON-EXISTANT OR EMPTY FILE

        CMP.B #20,NAM1(A6)      NO FILE NAME IS EMPTY FILE
        BEQ FALSE
        BSR OPEN                NON-EXISTANT IS ALSO EMPTY
        BEQ FALSE
        MOVEQ #fnREAD,D0
        TRAP  #2
        TST   D0

        BNE FALSE               NO RECORD == EMPTY FILE
        CMP.B #$1A,(A3)
        BEQ FALSE               ^Z MUST BE EMPTY ASCII FILE
        BRA TRUE

*****************************************************************
EXIST:  * TEST FIRST FCB FOR EXISTING FILE

        BSR OPEN
        BNE FALSE               File not Found Therefore Non-Existant
        BRA TRUE                otherwise it's naturally there

*****************************************************************
AMBIG:  * TEST FIRST FCB FOR AMBIGIOUS NAME OR TYPE

        MOVE #11-1,D0           Test 8-Char NAME + 3-Char TYPE
      X4:
          CMP.B #'?',NAM1(A6,D0)
        DBEQ D0,X4
        BNE TRUE                Contains "?" Therefore Ambigious
        BRA FALSE
.page
*****************************************************************
******************** DATA STORAGE AREA ************************
TABLE   DC.B 'AMBI'
        DC.L AMBIG              *********************************
        DC.B 'EMPT'             *              NOTE               *
        DC.L EMPTY              *                                 *
        DC.B 'EXIS'             * NULL MUST BE THE LAST TABLE     *
        DC.L EXIST              * ENTRY. SEE COMPARE SUBROUTINE   *
        DC.B 'NULL'             *                                 *
        DC.L NULL               *********************************

ABORTSUB:
        DC.B 10,'A:$$$ABORT',0
ABORTREC:
        DC.B 'ERA A:$$$$$$ABORT.SUB',$D,$A
```

 (Listing continued, text begins on page 70)

Listing One

```
        DC.L ';SUBMIT FILE PROCESSING ABORTED',$D,$A,$1A
ABORTFCB:
        DC.B 1,'$$$ABORTSUB'
        DC.B 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
ABORTMSG
        DC.B $D,$A,'ABORTING ...$'

PROMPT:
  DC.B $7,$D,$A,$D,$A,'SUBMIT FILE PROCESSING PAUSED.',$D,$A,$D,$A

  DC.B $7,'    PRESS: (Q) TO QUIT, (S) TO SKIP  -or-  (C) TO CONTINUE $'

MESSAGE:
  DC.B 7,$D,$A,$A
  DC.B 'IF.68K Ver 1.2 CONDITIONAL SUBMIT COMMAND FORMAT:',$D,$A,$A
  DC.B '  IF <filespec>  AMBIGx.yx /$$$$/ <command tail>',$D,$A
  DC.B '  IF <filespec>  EXISTx.yx /$$$$/ <command tail>',$D,$A
  DC.B '  IF <filespec>  EMPTYx.yx /$$$$/ <command tail>',$D,$A
  DC.B '  IF <parameter>  NULLx.yx /$$$$/ <command tail>',$D,$A,$A

  DC.B '  WHERE: x is an optional "*" to indicate negation',$D,$A
  DC.B '     and y is (Q)uit, (S)kip or (P)ause',$D,$A,$A

  DC.B '  NOTE:  Command tail may consist of multiple commands',$D,$A
  DC.B '         separated by backslashes (\) and ends at an',$D,$A
  DC.B '         exclamation mark (!) or physical end of line.',$D,$A,$A

  DC.B '  EXAMPLE:  IF A:JUNK.TYP  /$$$$/ EMPTY.S CMD1\CMD2!CMD3',$D,$A,$A
  DC.B '    Neither CMD1 nor CMD2 will execute if A:JUNK.TYP',$D,$A
  DC.B '    is empty.  Regardless, CMD3 will always execute.'

  DC.B $D,$A,$A,0

        END
```

End Listing One

Listing Two

```
             ERAQ.SUB  -  File Deletion with Query

DIR $1
;-------------------------------------------------------
IF $1  EXISTS.P /$$$$/ ERA $1
IF $2  NULL.Q   /$$$$/
SUBMIT ERAQ $2 $3 $4 $5 $6 $7 $8 $9
```

End Listing Two

Listing Three

```
             ASM.SUB  -  Simple 68K Assemble and Link

IF $1 NULL.S /$$$$/ ERA $1.O
IF FILE1.BAK EXISTS.S* /$$$$/ ERA FILE1.O\PIP B:=FILE1.S
IF FILE2.BAK EXISTS.S* /$$$$/ ERA FILE2.O\PIP B:=FILE2.S
IF FILE3.BAK EXISTS.S* /$$$$/ ERA FILE3.O\PIP B:=FILE3.S
IF FILE4.BAK EXISTS.S* /$$$$/ ERA FILE4.O\PIP B:=FILE4.S
ERA *.BAK
IF FILE1.O   EXISTS.S  /$$$$/ A:AS68 -S A: -L FILE1.S
IF FILE2.O   EXISTS.S  /$$$$/ A:AS68 -S A: -L FILE1.S
IF FILE3.O   EXISTS.S  /$$$$/ A:AS68 -S A: -L FILE1.S
IF FILE4.O   EXISTS.S  /$$$$/ A:AS68 -S A: -L FILE1.S
A:LO68 -R FILE1.O FILE2.O FILE3.O FILE4.O
```

End Listings

ProDOS and the RAM Disk

by Alfred Steele

I have a Synetix Flashcard with 288K of RAM that I use with Apple's CP/M, Pascal, and DOS 3.3. When I got my card more than two years ago, there was no ProDOS. After using ProDOS several times, I knew that I could not live without my RAM disk, so I decided to roll my own driver. I did several things to speed up the process of writing the driver. I used only 256K of the 288K memory on the Flashcard. The Flashcard uses six banks of memory. Four banks have 64K and the other two have only 16K each. I chose to use only the four 64K banks of memory to save development time and reduce code size. The loss of 32K of storage space is not a problem at this time. With the 256K of RAM in use, I get a RAM disk with 512 blocks of storage space.

Using a RAM disk speeds hi-res display by a factor of four.

This is Version 1.3 of the RAM-disk driver (see Listing, page 85). Version 1.0 was larger and faster but not relocatable. It had to be loaded in page 3 of memory. Version 1.1 fixed bugs in the installation program. Version 1.2 is the first version inich the RAM-disk driver code is small enough to fit inside of a free space in ProDOS. The program is completely relocatable anywhere in memory. One side effect of making the driver smaller and relocatable is that it is slower than in the other versions.

Version 1.3 fixes some errors in the installation code and adds more user prompts. This version of the RAM-disk driver can be installed anywhere in memory, but only three options are given at run time. You may place the driver inside a free space in ProDOS at $FF00. This space may only be free in ProDOS Version 1.01. Later versions of ProDOS may not support the driver at that address. You may also place the driver under the BASIC interpreter or at $300 hex page 3. The reason I advise you to install the driver inside of ProDOS is to support MousePaint, which uses page 3 of memory. Information on how to use the interpreter can be found in the *ProDOS Technical Reference Manual* and *Beneath Apple ProDOS*. Both books are really necessary.

The code for the RAM-disk driver is not hard to follow. The RAM-disk driver code that will be relocated to any address starts at label SSDSTART and ends at ENDSSD. The routine is only 8E hex (142 decimal) bytes long. The main part of the installation code starts at label MAIN. This is a series of subroutines to check if the RAM disk has already been formatted. If this is a first time installation, the program title and the volume name of the RAM disk are displayed. The driver code is relocated to the address of your choice. The ProDOS active devices are listed and the count is updated. The address of the device driver is placed in the vector table for Slot 5, Drive 1. The RAM-disk volume directory information is written to the RAM disk and the volume bit map is written to disk. Information on the diskette volume can be found in Chapter 4 of *Beneath Apple ProDOS* and Appendix B of the *ProDOS Technical*

Alfred Steele, Box 6296, APO NY 09012

Reference Manual.

The program has been written to be easy to use. If you try to install the driver more than once, the program will display a message that a driver has already been installed and abort. If the driver finds a formatted volume directory header on the RAM disk it will ask you if you want to clear it or not. If you type N, the program will install the driver but won't clear the RAM-disk directory. If you type Y, the program will install the driver and reformat the RAM disk. If any other driver, such as a hard disk driver, is already installed in the active device vector for Slot 5, Drive 1, the program will abort with an error message.

Once the RAM-disk driver is installed you can access the RAM disk using standard ProDOS commands and utilities. You can catalog the *RAM disk using* either the command CAT /RAM or CAT,S5,D1. If you want to copy files to the RAM disk use your copy of ProDOS FILER [*or see the article "Adding a COPY Command to ProDOS" on page 54 of this issue—ed*]. The system intergradation is clean and efficient and provides super fast loading and storing of any type of file.

One benchmark that I ran using the RAM disk consisted of loading high-resolution pictures to the screen. I made up a ProDOS test disk with a short BASIC program to load 16 different hi-res pictures as fast as possible. Using Apple's Disk II it took about 35 seconds to display all 16 pictures. Using the same program and pictures loaded to the RAM disk it only took 8.3 seconds to display all 16 pictures. The program runs about 4.2 times faster using the RAM disk for an effective I/O transfer speed of 15.9K per second vs. the Disk II transfer speed of 3.8K per second.

For those who don't want to type in more than 900 lines of code and comments, I will send you a copy of the source and object code on one Apple ProDOS disk for $15 postpaid anywhere in the United States. Please send a money order because a personal check is difficult to process in West Germany. To assemble the code, you need a copy of the ProDOS assembler/editor, or you can assemble the code under a DOS 3.3 assembler and move the object code to a ProDOS disk. My address is:

Alfred Steele
Box 6296
APO NY 09012

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 196**.

Ram Disk Listing (Text begins on page 84)

```
SOURCE    FILE #01 =)/RAMDISK.SOURCE/SSD3
0000:                1 ********************************
0000:                2 *                              *
0000:                3 *    FLASHCARD RAMDISK DRIVER  *
0000:                4 *                              *
0000:                5 *         VERSION 1.3          *
0000:                6 *          JULY 1985           *
0000:                7 *                              *
0000:                8 *       COPYRIGHT 1985         *
0000:                9 *       ALFRED STEELE          *
0000:               10 *                              *
0000:               11 * THIS PROGRAM MAY BE COPIED    *
0000:               12 * FOR PERSONAL, NON-PROFIT USE  *
0000:               13 *                              *
0000:               14 ********************************
0000:               15 *

0000:               16             LST   ON
----- NEXT OBJECT FILE NAME IS /RAMDISK.SOURCE/SSD3.0
2000:          2000 17             ORG   $2000
2000:               18 *
2000:               19 * TEXT STUFF
2000:               20 *
2000:          0000 21 EOL         EQU   $00
2000:          008D 22 CR          EQU   $8D
2000:               23 *
2000:               24 * ADDRESS OF SSD DRIVER
2000:               25 *
2000:          0000 26 TEMPZERO    EQU   $00
2000:          0002 27 SSDCODE     EQU   $02
2000:               28 *
2000:               29 * MLI DISK DRIVE ZERO PAGE
2000:               30 *
2000:          0001 31 READ        EQU   $01
2000:          0002 32 WRITE       EQU   $02
2000:          0006 33 DIRBLOCKS   EQU   $06
2000:          0008 34 SAVEBYTES   EQU   $08
2000:               35 *
2000:               36 * DEVICE DRIVER PARAMETERS
2000:               37 *
2000:          0042 38 COMMAND     EQU   $42
2000:          0043 39 UNITNUMB    EQU   $43
2000:          0044 40 BUFFADR     EQU   $44
2000:          0046 41 BLOCKNUM    EQU   $46
2000:          0048 42 RAMBANK     EQU   $48
2000:          004A 43 TEMP        EQU   $4A
2000:               44 *
2000:               45 * LOW BYTE ADDRESS OF BLOCK
2000:               46 * NUMBERS IN SSD RAM DISK.
2000:               47 *
2000:          00D2 48 BLOCK1      EQU   $D2
2000:          00D3 49 BLOCK2      EQU   $D3
2000:          00D4 50 BLOCK3      EQU   $D4
2000:          00D5 51 BLOCK4      EQU   $D5
2000:          00D6 52 BLOCK5      EQU   $D6
2000:          00D7 53 BLOCK6      EQU   $D7
2000:               54 *
2000:               55 * SSD ADDRESS FOR CARD IN SLOT 5
2000:               56 *
2000:          C0D0 57 SSDLOW      EQU   $C0D0
2000:          C0D1 58 SSDHI       EQU   $C0D1
2000:               59 *
2000:          BEF5 60 GETBUFF     EQU   $BEF5
2000:          BF00 61 MLI         EQU   $BF00
2000:          BF31 62 DEVCNT      EQU   $BF31
2000:          BF32 63 DEVLST      EQU   $BF32
2000:          BF1A 64 DEVADR51    EQU   $BF1A
2000:               65 *
2000:               66 * VALUE OF DEVICE DRIVER POINTER
2000:               67 * IF THE SLOT AND DRIVE IS NOT
2000:               68 * BEING USED.
2000:               69 *
2000:          D0A2 70 NODEV       EQU   $D0A2
2000:               71 *
2000:               72 * BI GLOBAL PAGE ADDRESS
2000:               73 *
2000:          BE00 74 WARMDOS     EQU   $BE00
2000:               75 *
2000:               76 * HARDWARE LOCATIONS
2000:               77 *
2000:          C000 78 KEY         EQU   $C000
2000:          C010 79 STROBE      EQU   $C010
2000:               80 *
2000:               81 * MONITOR SUBROUTINES
2000:               82 *
2000:          FC58 83 HOME        EQU   $FC58
2000:          FD8E 84 CROUT       EQU   $FD8E
2000:          FDED 85 COUT        EQU   $FDED
2000:          FF3A 86 BELL        EQU   $FF3A
2000:               87 *
2000:               88 ********************************
2000:               89 *
2000: 4C 90 21      90             JMP   MAIN
2003:               91 *
2003:               92 ********************************
2003:               93 *
2003:               94 * THIS IS THE SSD RAMDISK DRIVER
2003:               95 * CODE THAT IS MOVED TO SSDCODE
2003:               96 * AT THE START OF THIS PROGRAM.
2003:               97 * THIS ROUTINE MUST START ON A
2003:               98 * PAGE BOUNDARY.  I USE THE DUMMY
2003:               99 * DS STATEMENT BELOW SO THAT THE
2003:              100 * ROUTINE WILL START ON A PAGE
2003:              101 * BOUNDARY.  IF YOU CHANGE ANY
2003:              102 * CODE ABOVE BE SURE TO ADJUST
2003:              103 * THE DS STATEMENT BEFORE YOU
2003:              104 * ASSEMBLE AND RUN THE PROGRAM.
2003:              105 *
2003:              106 *
2003:          00FD 107            DS    $FD
2100:              108 *
2100:              109 *
2100: D8           110 SSDSTART    CLD
2101:              111 *
2101:              112 * SAVE ALL MLI ZERO PAGE LOC
2101:              113 * PLUS TWO NEEDED BY DRIVER
2101:              114 *
2101: AD D3 C0     115             LDA   $C0D3
2104: A2 08        116             LDX   #SAVEBYTES
2106: B5 42        117 S1          LDA   COMMAND,X
2108: 48           118             PHA
2109: CA           119             DEX
210A: 10 FA  2106  120             BPL   S1
210C:              121 *
210C:              122 * COMPUTE WHICH 64K BLOCK WE
210C:              123 * NEED TO READ OR WRITE TO.
210C:              124 * EACH 64K BLOCK OF RAM HAS 128
210C:              125 * PRODOS 512 BYTES BLOCKS IN IT.
210C:              126 * WE ARE ONLY USING THE 64K
210C:              127 * BANKS SO ONLY BLOCKS:
210C:              128 *
210C:              129 *       BLOCK1     BLOCKS   0-127
210C:              130 *       BLOCK2     BLOCKS 128-255
210C:              131 *       BLOCK4     BLOCKS 256-383
210C:              132 *       BLOCK5     BLOCKS 384-511
```

(Continued on next page)

Ram Disk Listing (Listing continued, text begins on page 84)

```
210C:                   133 *
210C:                   134 * ARE BEING USED. WHEN WE KNOW
210C:                   135 * WHAT BANK WE WILL USE WE MOVE
210C:                   136 * THAT BANKS LOW BYTE FOR THE
210C:                   137 * SSD RAM CARD BANK ADDRESS INTO
210C:                   138 * THE LOW-BYTE OF THE ZERO PAGE
210C:                   139 * POINTER RAMBANK. THE HI-BYTE
210C:                   140 * IS SET TO #$C0. IF BLOCKNUM+1
210C:                   141 * IS NOT ZERO WE NEED TO ACCESS
210C:                   142 * BLOCKS 256-383 OR 384-511.
210C:                   143 *
210C:A5 47              144 COMPUTE   LDA   BLOCKNUM+1
210E:D0 0C    211C      145           BNE   C2
2110:                   146 *
2110:                   147 * BLOCKS 0-127 OR 128-255
2110:                   148 *
2110:A5 46              149           LDA   BLOCKNUM
2112:30 04    2118      150           BMI   C1
2114:                   151 *
2114:                   152 * BLOCKS 0-127
2114:                   153 *
2114:A9 D2              154           LDA   #BLOCK1
2116:D0 0E    2126      155           BNE   C4
2118:                   156 *
2118:                   157 * BLOCKS 128-255
2118:                   158 *
2118:A9 D3              159 C1        LDA   #BLOCK2
211A:D0 0A    2126      160           BNE   C4
211C:                   161 *
211C:                   162 * BLOCKS 256-383 OR 384-511
211C:                   163 *
211C:A5 46              164 C2        LDA   BLOCKNUM
211E:30 04    2124      165           BMI   C3
2120:                   166 *
2120:                   167 * BLOCKS 256-383
2120:                   168 *
2120:A9 D5              169           LDA   #BLOCK4
2122:D0 02    2126      170           BNE   C4
2124:                   171 *
2124:                   172 * BLOCKS 384-511
2124:                   173 *
2124:A9 D6              174 C3        LDA   #BLOCK5
2126:85 48              175 C4        STA   RAMBANK
2128:                   176 *
2128:                   177 * SET UP HI-BYTE OF SSD CARD
2128:                   178 *
2128:A9 C0              179           LDA   #$C0
212A:85 49              180           STA   RAMBANK+1
212C:                   181 *
212C:                   182 * DECODE TYPE OF COMMAND FOR
212C:                   183 * RAM DISK DRIVER TO DO.

212C:                   184 *
212C:A5 42              185           LDA   COMMAND
212E:85 4A              186           STA   TEMP
2130:F0 48    217A      187           BEQ   EXIT
2132:                   188 *
2132:                   189 * SAVE COMMAND IN Y-REG AND
2132:                   190 * MASK OUT BITS 1 AND 0.
2132:                   191 * IF A-REG = ZERO AFTER MASK
2132:                   192 * THEN THIS IS A VALID READ
2132:                   193 * OR WRITE COMMAND.
2132:                   194 *
2132:29 FC              195           AND   #$FC
2134:F0 06    213C      196           BEQ   SETBLOCK
2136:                   197 *
2136:                   198 * ERROR COMMAND HANDLER
2136:                   199 * INVALID COMMAND SENT
2136:                   200 *
2136:A9 27              201           LDA   #$27
2138:85 4A              202           STA   TEMP
213A:D0 3E    217A      203           BNE   EXIT
213C:                   204 *
213C:                   205 * READ OR WRITE COMMAND. SET UP
213C:                   206 * BLOCK NUMBER AND BRANCH TO
213C:                   207 * READBYTE OR WRITEBYTE.
213C:                   208 *
213C:                   209 * EACH 64K BANK HOLDS 128 PRODOS
213C:                   210 * 512 BYTE BLOCKS. GET THE LOWER
213C:                   211 * 7 BITS AND SHIFT ONCE LEFT.
213C:                   212 * THIS WILL MULTPLY THE NUMBER
213C:                   213 * 0-127 TO 0-254 EVEN VALUES
213C:                   214 * ONLY. THIS ADJUSTED BLOCK
213C:                   215 * NUMBER IS USED AS THE HI-BYTE
213C:                   216 * OF THE ADDRESS IN THE 64K BANK
213C:                   217 * THE LOW-BYTE IS SET TO ZERO
213C:                   218 * AND INC FROM 0 TO $FF.
213C:                   219 *
213C:A5 46              220 SETBLOCK  LDA   BLOCKNUM
213E:29 7F              221           AND   #$7F
2140:0A                 222           ASL
2141:85 47              223           STA   BLOCKNUM+1
2143:8D D1 C0           224           STA   SSDHI
2146:                   225 *
2146:                   226 * MOVE COMMAND TO A-REG AND
2146:                   227 * SET UP LOOP COUNT TO RUN
2146:                   228 * THRU READ OR WRITE TWICE.
2146:                   229 * EACH LOOP THRU READ OR WRITE
2146:                   230 * WILL MOVE 256 BYTES.
2146:                   231 *
2146:A0 00              232           LDY   #0
2148:84 46              233           STY   BLOCKNUM

214A:A2 02              234           LDX   #2
214C:                   235 *
214C:                   236 * READ OR WRITE ONE 512 BYTE
214C:                   237 * PRODOS BLOCK.
214C:                   238 *
214C:8C D0 C0           239 SSDIO     STY   SSDLOW
214F:84 4A              240           STY   TEMP
2151:                   241 *
2151:                   242 * CHECK A-REG TO SEE IF WE NEED
2151:                   243 * TO DO A READ OR WRITE.
2151:                   244 *
2151:                   245 *    A-REG = $01 FOR READ
2151:                   246 *    A-REG = $02 FOR WRITE
2151:                   247 *
2151:                   248 * ROTATE A-REG THRU CARRY AND
2151:                   249 * CHECK IF CARRY SET OR CLEAR.
2151:                   250 * IF CARRY SET THEN READ. IF
2151:                   251 * CARRY CLEAR THEN WRITE.
2151:                   252 *
2151:A5 42              253           LDA   COMMAND
2153:4A                 254           LSR
2154:90 0B    2161      255           BCC   WRITEBYTE
2156:                   256 *
2156:A0 00              257 READBYTE  LDY   #0
2158:B1 48              258           LDA   (RAMBANK),Y
215A:A4 4A              259           LDY   TEMP
215C:91 44              260           STA   (BUFFADR),Y
215E:18                 261           CLC
215F:90 08    2169      262           BCC   CONT
2161:                   263 *
2161:                   264 * WRITE BYTE TO RAMDISK
2161:                   265 *
2161:B1 44              266 WRITEBYTE LDA   (BUFFADR),Y
2163:A0 00              267           LDY   #0
2165:91 48              268           STA   (RAMBANK),Y
2167:A4 4A              269           LDY   TEMP
2169:                   270 *
2169:                   271 * COMMON CODE TO INC TO NEXT
2169:                   272 * BYTE TO READ OR WRITE
2169:                   273 *
2169:C8                 274 CONT      INY
216A:D0 E0    214C      275           BNE   SSDIO
216C:                   276 *
216C:E6 45              277           INC   BUFFADR+1
216E:E6 47              278           INC   BLOCKNUM+1
2170:A5 47              279           LDA   BLOCKNUM+1
2172:8D D1 C0           280           STA   SSDHI
2175:CA                 281           DEX
2176:D0 D4    214C      282           BNE   SSDIO
2178:                   283 *
2178:84 4A              284           STY   TEMP
217A:                   285 *
217A:                   286 * THIS PART OF THE ROUTINE IS
217A:                   287 * ENTERED IF THE STATUS, READ
217A:                   288 * OR WRITE I/O IS COMPLETE.
217A:                   289 *
217A:                   290 * SSD EXIT RETURN
217A:                   291 *
217A:A4 4A              292 EXIT      LDY   TEMP
217C:A2 00              293           LDX   #0
217E:68                 294 EX1       PLA
217F:95 42              295           STA   COMMAND,X
2181:E8                 296           INX

2182:E0 09              297           CPX   #SAVEBYTES+1
2184:D0 F8    217E      298           BNE   EX1
2186:                   299 *
2186:AD D2 C0           300           LDA   $C0D2
2189:18                 301           CLC
218A:98                 302           TYA
218B:F0 01    218E      303           BEQ   EX2
218D:38                 304           SEC
218E:                   305 *
218E:60                 306 EX2       RTS
218F:EA                 307 ENDSSD    NOP
2190:                   308 *
2190:                   309 ********************************
2190:                   310 *
2190:                   311 * THIS IS THE MAIN SET OF JSR
2190:                   312 * STATEMENTS TO INSTALL AND
2190:                   313 * INIT THE RAMDISK.
2190:                   314 *
2190:20 50 22           315 MAIN      JSR   CHECKDEV
2193:20 97 23           316           JSR   WHERETO
2196:20 4F 25           317           JSR   TITLE
2199:20 64 25           318           JSR   DISPVOL
219C:                   319 *
219C:20 B4 21           320           JSR   RELOCATE
219F:20 C1 21           321           JSR   SETDEV
21A2:20 D7 21           322           JSR   SETVECTOR
21A5:                   323 *
21A5:                   324 * CHECK CLEAR DIRECTORY FLAG
21A5:                   325 * IF CLEARDIR = 1 THEN SKIP
21A5:                   326 * THE ROUTINES TO WRITE A NEW
21A5:                   327 * DIRECTORY TO THE RAMDISK AND
21A5:                   328 * SKIP THE ROUTINE TO WRITE A
21A5:                   329 * NEW BITMAP TO THE RAMDISK.
21A5:                   330 *
21A5:AD 4A 26           331           LDA   CLEARDIR
21A8:D0 09    21B3      332           BNE   MN1
21AA:                   333 *
21AA:                   334 * DO A NORMAL FIRST TIME INIT
21AA:                   335 *
21AA:20 E2 21           336           JSR   CLEAREND
21AD:20 EF 21           337           JSR   WRITEDIR
21B0:20 35 22           338           JSR   BITMAP
21B3:60                 339 MN1       RTS
21B4:                   340 *
21B4:                   341 ********************************
21B4:                   342 *
21B4:                   343 * THIS ROUTINE IS USED TO MOVE
21B4:                   344 * THE SSD RAM DISK DRIVER TO
21B4:                   345 * THE ADDRESS OF SSDCODE.
21B4:                   346 * THE ROUTINE IS COMPLETELY
21B4:                   347 * RELOCATABLE AND DOES NOT HAVE
21B4:                   348 * TO BE ON A PAGE BOUNDARY.
21B4:                   349 *
21B4:A0 00              350 RELOCATE  LDY   #0
21B6:B9 00 21           351 R1        LDA   SSDSTART,Y
21B9:91 02              352           STA   (SSDCODE),Y
21BB:C8                 353           INY
21BC:C0 8F              354           CPY   #ENDSSD-SSDSTART
21BE:D0 F6    21B6      355           BNE   R1
21C0:60                 356           RTS
21C1:                   357 *
```

Ram Disk Listing (Listing continued, text begins on page 84)

```
21C1:                   358 ********************************
21C1:                   359 *
21C1:                   360 * THIS ROUTINE IS USED TO MAKE
21C1:                   361 * ROOM IN THE DEVICE LIST OF
21C1:                   362 * ACTIVE DEVICES FOR THE RAM
21C1:                   363 * DISK DRIVE IDENTIFICATION.
21C1:                   364 *
21C1:AE 31 BF           365 SETDEV      LDX     DEVCNT
21C4:E8                 366             INX
21C5:8E 31 BF           367             STX     DEVCNT
21C8:                   368 *
21C8:BD 31 BF           369 SD1         LDA     DEVLST-1,X
21CB:9D 32 BF           370             STA     DEVLST,X
21CE:CA                 371             DEX
21CF:D0 F7    21C8      372             BNE     SD1
21D1:                   373 *
21D1:                   374 * STORE THE RAM DISK IN LIST
21D1:                   375 *
21D1:A9 50              376             LDA     #$50
21D3:8D 32 BF           377             STA     DEVLST
21D6:60                 378             RTS
21D7:                   379 *
21D7:                   380 ********************************
21D7:                   381 *
21D7:                   382 * INSTALL RAMDISK DRIVER VECTOR
21D7:                   383 *
21D7:A5 02              384 SETVECTOR   LDA     SSDCODE
21D9:8D 1A BF           385             STA     DEVADR51
21DC:A5 03              386             LDA     SSDCODE+1
21DE:8D 1B BF           387             STA     DEVADR51+1
21E1:60                 388             RTS
21E2:                   389 *
21E2:                   390 ********************************
21E2:                   391 *
21E2:                   392 * CLEAR BUFFER FROM END OF DISK
21E2:                   393 * DATA TO END OF DISK BUFFER.
21E2:                   394 *
21E2:A0 00              395 CLEAREND    LDY     #0
21E4:98                 396             TYA
21E5:99 82 26           397 CE          STA     MIDBUFF,Y
21E8:99 82 27           398             STA     MIDBUFF+$100,Y
21EB:C8                 399             INY
21EC:D0 F7    21E5      400             BNE     CE
21EE:60                 401             RTS
21EF:                   402 *
21EF:                   403 ********************************
21EF:                   404 *
21EF:                   405 * CALL MLI TO WRITE OUT THE
21EF:                   406 * SSD RAM DISK VOLUME DIRECTORY
21EF:                   407 *
21EF:A9 06              408 WRITEDIR    LDA     #DIRBLOCKS

21F1:8D 7E 26           409             STA     BUFFER+39
21F4:A9 02              410             LDA     #$02
21F6:8D 4F 26           411             STA     WRITEBLOCK
21F9:                   412 *
21F9:20 00 BF           413             JSR     MLI
21FC:81                 414             DB      $81
21FD:4B 26              415             DW      MLIWRITE
21FF:                   416 *
21FF:                   417 * CLEAR AND SET BLOCKS
21FF:                   418 * 3 THRU DIRBLOCKS OF
21FF:                   419 * VOLUME DIRECTORY. WE HAVE TO
21FF:                   420 * SET UP THE LAST/NEXT BLOCK
21FF:                   421 * NUMBERS IN THE FIRST FOUR (4)
21FF:                   422 * BYTES OF EACH BLOCK.
21FF:                   423 *
21FF:A2 03              424             LDX     #3
2201:20 28 22           425 CB1         JSR     CLEARBUFF
2204:                   426 *
2204:CA                 427             DEX
2205:8E 57 26           428             STX     BUFFER
2208:E8                 429             INX
2209:E8                 430             INX
220A:                   431 *
220A:E0 06              432             CPX     #DIRBLOCKS
220C:D0 07    2215      433             BNE     CB2
220E:                   434 *
220E:A9 00              435             LDA     #0
2210:8D 59 26           436             STA     BUFFER+2
2213:F0 03    2218      437             BEQ     CB3
2215:                   438 *
2215:8E 59 26           439 CB2         STX     BUFFER+2
2218:                   440 *
2218:                   441 * WRITE VOLUME DIRECTORY BLOCK
2218:                   442 * TO SSD RAM DISK.
2218:                   443 *
2218:CA                 444 CB3         DEX
2219:8E 4F 26           445             STX     WRITEBLOCK
221C:                   446 *
221C:20 00 BF           447             JSR     MLI
221F:81                 448             DB      $81
2220:4B 26              449             DW      MLIWRITE
2222:                   450 *
2222:E8                 451             INX
2223:E0 06              452             CPX     #DIRBLOCKS
2225:D0 DA    2201      453             BNE     CB1
2227:60                 454             RTS
2228:                   455 *
2228:                   456 ********************************
2228:                   457 *
2228:                   458 * CLEAR ENTIRE FILE BUFFER
2228:                   459 *

2228:A0 00              460 CLEARBUFF   LDY     #0
222A:98                 461             TYA
222B:99 57 26           462 CL1         STA     BUFFER,Y
222E:99 57 27           463             STA     BUFFER+$100,Y
2231:C8                 464             INY
2232:D0 F7    222B      465             BNE     CL1
2234:60                 466             RTS
2235:                   467 *
2235:                   468 ********************************
2235:                   469 *
2235:                   470 * SET ALL BLOCKS NOT USED IN
2235:                   471 * RAM DISK TO HEX $FF TO SHOW
2235:                   472 * THAT THOSE BLOCKS ARE FREE.
2235:                   473 * SET FIRST BYTE TO SHOW THAT
2235:                   474 * BLOCKS 0 THRU DIRBLOCKS ARE
2235:                   475 * IN USE.
2235:                   476 *
2235:A9 01              477 BITMAP      LDA     #$01
2237:8D 57 26           478             STA     BUFFER
223A:A0 3F              479             LDY     #63
223C:A9 FF              480             LDA     #$FF
223E:                   481 *
223E:99 57 26           482 BM2         STA     BUFFER,Y
2241:88                 483             DEY
2242:D0 FA    223E      484             BNE     BM2
2244:                   485 *
2244:A9 06              486             LDA     #DIRBLOCKS
2246:8D 4F 26           487             STA     WRITEBLOCK
2249:                   488 *
2249:                   489 * WRITE OUT BIT MAP TO SSD
2249:                   490 * RAM DISK.
2249:                   491 *
2249:20 00 BF           492             JSR     MLI
224C:81                 493             DB      $81
224D:4B 26              494             DW      MLIWRITE
224F:                   495 *
224F:60                 496             RTS
2250:                   497 *
2250:                   498 ********************************
2250:                   499 *
2250:                   500 * THIS ROUTINE IS USED TO CHECK
2250:                   501 * FOR A DEVICE DRIVER ALREADY
2250:                   502 * INSTALLED FOR SLOT AND DRIVE
2250:                   503 * WE WANT TO USE. I USE SLOT 5
2250:                   504 * DRIVE 1 FOR THE RAMDISK DRIVER
2250:                   505 * IF THE ADDRESS AT SLOT 5 DRIVE
2250:                   506 * 1 DOES NOT POINT TO $D0A2 THE
2250:                   507 * PROGRAM WILL ABORT WITH AN
2250:                   508 * ERROR MESSAGE.  IF WE FIND NO
2250:                   509 * DRIVER INSTALL I READ BLOCK 2
2250:                   510 * OF THE RAMDISK AND COMPARE IT
2250:                   511 * WITH THE VOLUME DIRECTORY
2250:                   512 * HEADER INFORMATION AT BUFFER.
2250:                   513 * IF THE INFORMATION IS THE SAME
2250:                   514 * A MESSAGE IS PRINTED ASKING
2250:                   515 * THE USER IF HE WANTS TO ZERO
2250:                   516 * THE DIRECTORY ON THE RAMDISK
2250:                   517 * OR NOT.  IF WE WANT TO CLEAR
2250:                   518 * THE DIRECTORY I SET CLEARDIR
2250:                   519 * TO ZERO.  IF WE DON'T WANT TO

2250:                   520 * CLEAR THE DIRECTORY I SET
2250:                   521 * CLEARDIR TO ONE.
2250:                   522 *
2250:AD 1A BF           523 CHECKDEV    LDA     DEVADR51
2253:C9 A2              524             CMP     #>NODEV
2255:D0 7A    22D1      525             BNE     DEVABORT
2257:                   526 *
2257:AD 1B BF           527             LDA     DEVADR51+1
225A:C9 D0              528             CMP     #<NODEV
225C:D0 73    22D1      529             BNE     DEVABORT
225E:                   530 *
225E:                   531 * LETS TRY AND READ THE RAM DISK
225E:                   532 * DIRECTORY AT BLOCK 2 AND
225E:                   533 * COMP IT TO SOME OF THE BYTES
225E:                   534 * AT ADDR BUFFER TO SEE IF THE
225E:                   535 * RAMDISK MIGHT HAVE A FORMATTED
225E:                   536 * DIRECTORY ON IT.
225E:                   537 *
225E:A9 01              538             LDA     #READ
2260:85 42              539             STA     COMMAND
2262:                   540 *
2262:                   541 * ADDRESS OF READ BUFFER
2262:                   542 *
2262:A9 82              543             LDA     #>MIDBUFF
2264:85 44              544             STA     BUFFADR
2266:A9 26              545             LDA     #<MIDBUFF
2268:85 45              546             STA     BUFFADR+1
226A:                   547 *
226A:                   548 * BLOCK NUMBER TO READ
226A:                   549 *
226A:A9 02              550             LDA     #$02
226C:85 46              551             STA     BLOCKNUM
226E:A9 00              552             LDA     #$00
2270:85 47              553             STA     BLOCKNUM+1
2272:                   554 *
2272:                   555 * READ THE RAMDISK USING DRIVER
2272:                   556 * AT SSDSTART IN THIS PROGRAM
2272:                   557 *
2272:20 00 21           558             JSR     SSDSTART
2275:                   559 *
2275:                   560 * WE WILL NOW ASSUME AT THERE
2275:                   561 * IS THE INFORMATION IN BLOCK 2
2275:                   562 * OF THE RAMDISK AT ADDRESS
2275:                   563 * MIDBUFF. COMPARE THREE SETS OF
2275:                   564 * BYTES TO SEE IF IT MATCHS A
2275:                   565 * VOLUMN DIRECTORY HEADER BLOCK.
2275:                   566 * IF ANY BYTE DOES NOT MATCH
2275:                   567 * THE ROUTINE JUMPS TO NODIR
2275:                   568 * IF ALL THREE SETS OF BYTES
2275:                   569 * MATCH I ASK USER IF HE WANTS
2275:                   570 * TO CLEAR THE DIR OR NOT.
2275:                   571 *
2275:A2 00              572             LDX     #$00
2277:BD 57 26           573 CD0         LDA     BUFFER,X
227A:DD 82 26           574             CMP     MIDBUFF,X
227D:D0 4C    22CB      575             BNE     NODIR
```

Ram Disk Listing (Listing continued, text begins on page 84)

```
227F:                    576 *
227F:E8                  577             INX
2280:E0 04               578             CPX   #$04
2282:D0 F3    2277       579             BNE   CD0
2284:                    580 *
2284:A2 14               581             LDX   #$14
2286:BD 57 26            582 CD1         LDA   BUFFER,X
2289:DD 82 26            583             CMP   MIDBUFF,X
228C:D0 3D    22CB       584             BNE   NODIR
228E:                    585 *
228E:E8                  586             INX
228F:E0 25               587             CPX   #$25
2291:D0 F3    2286       588             BNE   CD1
2293:                    589 *
2293:A2 27               590             LDX   #$27
2295:BD 57 26            591 CD2         LDA   BUFFER,X
2298:DD 82 26            592             CMP   MIDBUFF,X
229B:D0 2E    22CB       593             BNE   NODIR
229D:                    594 *
229D:E8                  595             INX
229E:E0 2B               596             CPX   #$2B
22A0:D0 F3    2295       597             BNE   CD2
22A2:                    598 *
22A2:                    599 * ASK USER IF HE WANT TO ZERO
22A2:                    600 * DIRECTORY OR NOT.  IF THE USER
22A2:                    601 * WANTS TO ZERO THE DIRECTORY
22A2:                    602 * JUST CONTINUE WITH THE INSTALL
22A2:                    603 * PROGRAM.  IF THE USER DOES NOT
22A2:                    604 * WANT TO ZERO THE DIRECTORY
22A2:                    605 * DO A JUMP TO THE DOS WARMSTART
22A2:                    606 *
22A2:20 58 FC            607             JSR   HOME
22A5:20 3A FF            608             JSR   BELL
22A8:                    609 *
22A8:A0 00               610             LDY   #0
22AA:B9 EF 22            611 CD3         LDA   DIRFOUND,Y
22AD:C9 00               612             CMP   #EOL
22AF:F0 09    22BA       613             BEQ   CD4
22B1:                    614 *
22B1:09 80               615             ORA   #$80
22B3:20 ED FD            616             JSR   COUT
22B6:C8                  617             INY
22B7:4C AA 22            618             JMP   CD3
22BA:                    619 *
22BA:20 41 25            620 CD4         JSR   GETKEY
22BD:C9 59               621             CMP   #'Y'
22BF:F0 0A    22CB       622             BEQ   NODIR
22C1:                    623 *
22C1:C9 4E               624             CMP   #'N'
22C3:D0 F5    22BA       625             BNE   CD4
22C5:                    626 *
22C5:                    627 * THE USER DOES NOT WANT TO
22C5:                    628 * CLEAR THE DIRECTORY.
22C5:                    629 *
22C5:A9 01               630             LDA   #$01

22C7:8D 4A 26            631             STA   CLEARDIR
22CA:60                  632             RTS
22CB:                    633 *
22CB:                    634 * WE WANT TO CLEAR THE DIRECTORY
22CB:                    635 * ON THE RAMDISK SO SET FLAG FOR
22CB:                    636 * CLEAR AND RETURN TO ROUTINE.
22CB:                    637 *
22CB:A9 00               638 NODIR       LDA   #$00
22CD:8D 4A 26            639             STA   CLEARDIR
22D0:60                  640             RTS
22D1:                    641 *
22D1:                    642 * THIS PART OF THE ROUTINE IS
22D1:                    643 * ENTERED IF WE FIND A DEVICE
22D1:                    644 * DRIVER ADDRESS THAT DOES NOT
22D1:                    645 * POINT TO THE NO DEVICE DRIVER
22D1:                    646 * ACTIVE ADDRESS OR TO THE
22D1:                    647 * ADDRESS WHERE WE WANT TO PUT
22D1:                    648 * OUR CODE.  THERE COULD BE A
22D1:                    649 * HARD DISK OR OTHER STORAGE
22D1:                    650 * DEVICE ALREADY INSTALL FOR
22D1:                    651 * THAT SLOT/DRIVE.
22D1:                    652 *
22D1:20 58 FC            653 DEVABORT    JSR   HOME
22D4:20 3A FF            654             JSR   BELL
22D7:                    655 *
22D7:A0 00               656             LDY   #0
22D9:B9 4B 23            657 DA1         LDA   ABORTMSG,Y
22DC:C9 00               658             CMP   #EOL
22DE:F0 09    22E9       659             BEQ   DA2
22E0:                    660 *
22E0:09 80               661             ORA   #$80
22E2:20 ED FD            662             JSR   COUT
22E5:C8                  663             INY
22E6:4C D9 22            664             JMP   DA1
22E9:                    665 *
22E9:20 3A FF            666 DA2         JSR   BELL
22EC:4C 00 BE            667             JMP   WARMDOS
22EF:                    668 *
22EF:                    669 *********************************
22EF:                    670 *
22EF:                    671 * DIRECTORY FOUND MESSAGE
22EF:                    672 *
22EF:8D 8D 8D 8D         673 DIRFOUND    DFB   CR,CR,CR,CR
22F3:8D 8D 8D 8D         674             DFB   CR,CR,CR,CR
22F7:20 20 20 20         675             ASC   '
2303:44 49 52 45         676             ASC   'DIRECTORY     FOUND'
2312:8D 8D               677             DFB   CR,CR
2314:20 20 20 20         678             ASC   '

231A:43 4C 45 41         679             ASC   'CLEAR         DIRECTORY "Y" OR "N"'
2334:8D 8D               680             DFB   CR,CR
2336:20 20 20 20         681             ASC   '
234A:00                  682             DFB   EOL
234B:                    683 *
234B:                    684 * ABORT MESSAGE
234B:                    685 *
234B:8D 8D 8D 8D         686 ABORTMSG    DFB   CR,CR,CR,CR
234F:8D 8D 8D 8D         687             DFB   CR,CR,CR,CR
2353:20 20 20 20         688             ASC   '
2357:44 45 56 49         689             ASC   'DEVICE        DRIVER ALREADY INSTALLED'
2376:8D 8D               690             DFB   CR,CR
2378:20 20 20 20         691             ASC   '
2385:50 52 4F 47         692             ASC   'PROGRAM       ABORT'
2392:8D 8D 8D 8D         693             DFB   CR,CR,CR,CR
2396:00                  694             DFB   EOL
2397:                    695 *
2397:                    696 *********************************
2397:                    697 *
2397:                    698 * THIS ROUTINE IS USED TO ASK
2397:                    699 * THE USER WHERE HE/SHE WOULD
2397:                    700 * LIKE TO MOVE THE RAMDISK
2397:                    701 * DRIVER CODE TO.  THERE ARE
2397:                    702 * THREE LOCATIONS TO MOVE THE
2397:                    703 * CODE TO OR THE USER CAN QUIT
2397:                    704 *
2397:                    705 * LOC 1:  INSIDE PRODOS AT $F000
2397:                    706 * LOC 2:  BETWEEN THE BASIC
2397:                    707 *         INTERPRETER AND THE
2397:                    708 *         FILE BUFFERS.
2397:                    709 * LOC 3:  AT $300
2397:                    710 *
2397:20 58 FC            711 WHERETO:    JSR   HOME
239A:A0 00               712             LDY   #0
239C:B9 0F 24            713 WT0         LDA   WHEREMSG,Y
239F:C9 00               714             CMP   #EOL
23A1:F0 09    23AC       715             BEQ   WT1
23A3:                    716 *
23A3:09 80               717             ORA   #$80
23A5:20 ED FD            718             JSR   COUT
23A8:C8                  719             INY
23A9:4C 9C 23            720             JMP   WT0
23AC:                    721 *
23AC:20 3A FF            722 WT1         JSR   BELL
23AF:20 41 25            723             JSR   GETKEY
23B2:C9 31               724             CMP   #'1'
23B4:F0 15    23CB       725             BEQ   WT2
23B6:                    726 *
23B6:C9 32               727             CMP   #'2'
23B8:F0 20    23DA       728             BEQ   WT3
23BA:                    729 *
23BA:C9 33               730             CMP   #'3'
23BC:F0 2A    23E8       731             BEQ   WT4
23BE:                    732 *

23BE:C9 51               733             CMP   #'Q'
23C0:D0 EA    23AC       734             BNE   WT1
23C2:                    735 *
23C2:20 58 FC            736             JSR   HOME
23C5:20 3A FF            737             JSR   BELL
23C8:4C 00 BE            738             JMP   WARMDOS
23CB:                    739 *
23CB:                    740 * MOVE CODE INSIDE PRODOS
23CB:                    741 *
23CB:A9 00               742 WT2         LDA   #$00
23CD:85 02               743             STA   SSDCODE
23CF:A9 FF               744             LDA   #$FF
23D1:85 03               745             STA   SSDCODE+1
23D3:AD 81 C0            746             LDA   $C081
23D6:AD 81 C0            747             LDA   $C081
23D9:60                  748             RTS
23DA:                    749 *
23DA:                    750 * MOVE CODE UNDER BI
23DA:                    751 *
23DA:A9 01               752 WT3         LDA   #1
23DC:20 F5 BE            753             JSR   GETBUFF
23DF:B0 10    23F1       754             BCS   BUFFERROR
23E1:                    755 *
23E1:85 03               756             STA   SSDCODE+1
23E3:A9 00               757             LDA   #$00
23E5:85 02               758             STA   SSDCODE
23E7:60                  759             RTS
23E8:                    760 *
23E8:                    761 * MOVE CODE TO $300
23E8:                    762 *
23E8:A9 00               763 WT4         LDA   #$00
23EA:85 02               764             STA   SSDCODE
23EC:A9 03               765             LDA   #$03
23EE:85 03               766             STA   SSDCODE+1
23F0:60                  767             RTS
23F1:                    768 *
23F1:                    769 * ERROR GETTING BUFFER FROM SYS
23F1:                    770 *
23F1:20 3A FF            771 BUFFERROR   JSR   BELL
23F4:20 58 FC            772             JSR   HOME
23F7:A0 00               773             LDY   #0
23F9:B9 B3 24            774 BE0         LDA   NOBUFF,Y
23FC:C9 00               775             CMP   #EOL
23FE:F0 09    2409       776             BEQ   BE1
2400:                    777 *
2400:09 80               778             ORA   #$80
2402:20 ED FD            779             JSR   COUT
2405:C8                  780             INY
2406:4C F9 23            781             JMP   BE0
2409:                    782 *
2409:20 3A FF            783 BE1         JSR   BELL
240C:4C 00 BE            784             JMP   WARMDOS
240F:                    785 *
240F:                    786 *********************************
```

```
240F:                   787 *
240F:                   788 * MESSAGE ASKING USER WHERE
240F:                   789 * TO PUT RAMDISK DRIVER CODE.
240F:                   790 *
240F:8D 8D 8D 8D        791 WHEREMSG  DFB   CR,CR,CR,CR
2413:20 20 20 20        792           ASC   '
2418:52 41 4D 44        793           ASC   'RAMDISK        DRIVER ADDRESS
242F:4F 50 54 49        794           ASC   'OPTIONS'
2436:8D 8D 8D           795           DFB   CR,CR,CR
2439:20 20 20 20        796           ASC   '
243F:5B 31 5D 20        797           ASC   '[1]            INSIDE PRODOS AT $FF00'
2459:8D 8D              798           DFB   CR,CR
245B:20 20 20 20        799           ASC   '
2461:5B 32 5D 20        800           ASC   '[2]            UNDER BASIC INTERPRETER'
247C:8D 8D              801           DFB   CR,CR
247E:20 20 20 20        802           ASC   '
2484:5B 33 5D 20        803           ASC   '[3]            AT $300
248F:8D 8D              804           DFB   CR,CR
2491:20 20 20 20        805           ASC   '
2497:5B 51 5D 55        806           ASC   '[Q]UIT'
249D:8D 8D              807           DFB   CR,CR
249F:20 20 20 20        808           ASC   '
24A9:20 20 20 20        809           ASC   '
24B2:00                 810           DFB   EOL
24B3:                   811 *
24B3:                   812 ********************************
24B3:                   813 *
24B3:                   814 * NO DOS BUFFER FOUND UNDER BI
24B3:                   815 *
24B3:8D 8D 8D 8D        816 NOBUFF    DFB   CR,CR,CR,CR
24B7:8D 8D 8D 8D        817           DFB   CR,CR,CR,CR
24BB:20 20 20 20        818           ASC   '
24C9:46 41 54 41        819           ASC   'FATAL          ERROR'
24D4:8D 8D              820           DFB   CR,CR
24D6:20 20 20           821           ASC   '
24D9:43 4F 55 4C        822           ASC   'COULD          NOT ALLOCATE BUFFER BETWEEN
'
24FA:8D 8D              823           DFB   CR,CR
24FC:20 20 20 20        824           ASC   '
2506:42 49 20 41        825           ASC   'BI             AND FILE BUFFERS'
2519:8D 8D              826           DFB   CR,CR
251B:20 20 20 20        827           ASC   '
2520:46 4C 41 53        828           ASC   'FLASHCARD      DRIVER NOT INSTALLED'
253E:8D 8D              829           DFB   CR,CR
2540:00                 830           DFB   EOL
2541:                   831 *
2541:                   832 ********************************
2541:                   833 *
2541:                   834 * THIS ROUTINE IS USED TO GET
2541:                   835 * A SINGLE KEYPRESS FROM THE
2541:                   836 * KEYBOARD AND RETURN IT IN
2541:                   837 * THE A-REG WITH THE MSB OFF.
2541:                   838 *
2541:8D 10 C0           839 GETKEY    STA   STROBE
2544:AD 00 C0           840 GK1       LDA   KEY
2547:10 FB    2544      841           BPL   GK1
2549:8D 10 C0           842           STA   STROBE
254C:29 7F              843           AND   #$7F
254E:60                 844           RTS
254F:                   845 *
254F:                   846 ********************************
254F:                   847 *
254F:                   848 * PRINT PROGRAM TITLE
254F:                   849 *
254F:20 58 FC           850 TITLE     JSR   HOME
2552:A0 00              851           LDY   #0
2554:B9 9B 25           852 TI1       LDA   MSG,Y
2557:C9 00              853           CMP   #EOL
2559:F0 08    2563      854           BEQ   TI2
255B:                   855 *
255B:09 80              856           ORA   #$80
255D:20 ED FD           857           JSR   COUT
2560:C8                 858           INY
2561:D0 F1    2554      859           BNE   TI1
2563:60                 860 TI2       RTS
2564:                   861 *
2564:                   862 ********************************
2564:                   863 *
2564:                   864 * THIS ROUTINE WILL DISPLAY THE
2564:                   865 * RAM DISK VOLUMN NAME IF THE
2564:                   866 * INSTALL WAS SUCCESSFULL.
2564:                   867 *
2564:A0 00              868 DISPVOL   LDY   #0
2566:B9 31 26           869 DV0       LDA   VOLNAME,Y
2569:C9 00              870           CMP   #EOL
256B:F0 09    2576      871           BEQ   DV1
256D:09 80              872           ORA   #$80
256F:20 ED FD           873           JSR   COUT
2572:C8                 874           INY
2573:4C 66 25           875           JMP   DV0
2576:                   876 *
2576:A0 00              877 DV1       LDY   #0
2578:B9 5C 26           878 DV2       LDA   VOLUME,Y
257B:C9 00              879           CMP   #EOL
257D:F0 0C    258B      880           BEQ   DV3
257F:C0 0E              881           CPY   #14
2581:F0 08    258B      882           BEQ   DV3
2583:                   883 *
2583:09 80              884           ORA   #$80
2585:20 ED FD           885           JSR   COUT
2588:C8                 886           INY
2589:D0 ED    2578      887           BNE   DV2
258B:                   888 *
258B:20 8E FD           889 DV3       JSR   CROUT
258E:20 8E FD           890           JSR   CROUT
2591:20 8E FD           891           JSR   CROUT
2594:20 8E FD           892           JSR   CROUT
2597:20 3A FF           893           JSR   BELL
259A:60                 894           RTS
259B:                   895 *
259B:                   896 ********************************
259B:                   897 *
259B:                   898 * PROGRAM TITLE
259B:                   899 *
259B:8D 8D 8D           900 MSG       DFB   CR,CR,CR
259E:8D 8D 8D           901           DFB   CR,CR,CR
25A1:20 20 20 20        902           ASC   '
25AA:50 52 4F 44        903           ASC   'PRODOS         RAMDISK DRIVER'
25BF:8D 8D              904           DFB   CR,CR
25C1:20 20 20 20        905           ASC   '
25CE:46 4F 52 20        906           ASC   'FOR            FLASHCARD'
25DB:8D 8D              907           DFB   CR,CR
25DD:20 20 20 20        908           ASC   '
25EB:56 45 52 53        909           ASC   'VERSION        1.3'
25F6:8D 8D              910           DFB   CR,CR
25F8:20 20 20 20        911           ASC   '
2605:41 4C 46 52        912           ASC   'ALFRED         STEELE'
2612:8D 8D              913           DFB   CR,CR
2614:20 20 20 20        914           ASC   '
2623:4A 55 4C 59        915           ASC   'JULY           1985'
262C:8D 8D 8D 8D        916           DFB   CR,CR,CR,CR
2630:00                 917           DFB   EOL
2631:                   918 *
2631:                   919 *
2631:                   920 *
2631:20 20 20 52        921 VOLNAME   ASC   '               RAMDISK VOLUME NAME /'
2649:00                 922           DFB   EOL
264A:                   923 *
264A:                   924 ********************************
```

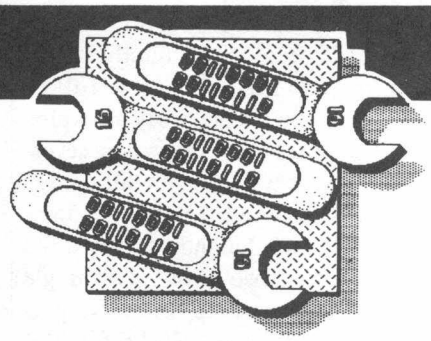
Ram Disk Listing (Listing continued, text begins on page 84)

```
264A:                   925 *
264A:                   926 * CLEAR DIRECTORY FLAG
264A:                   927 * SET FLAG FOR NO CLEAR
264A:                   928 *
264A:01                 929 CLEARDIR  DB     $01
264B:                   930 *
264B:                   931 ********************************
264B:                   932 *
264B:                   933 * MLI WRITE PARAMETER LIST
264B:                   934 *
264B:03                 935 MLIWRITE  DB     $03
264C:50                 936           DB     $50
264D:57 26              937           DW     BUFFER
264F:02 00              938 WRITEBLOCK DW    $0002
2651:                   939 *
2651:                   940 ********************************
2651:                   941 *
2651:                   942 * MLI READ PARAMETER LIST
2651:                   943 *
2651:03                 944 MLIREAD   DB     $03
2652:50                 945           DB     $50
2653:00 20              946 READBUFF  DW     $2000
2655:02 00              947 READBLOCK DW     $0002
2657:                   948 *
2657:                   949 ********************************
2657:                   950 *
2657:                   951 * MLI READ BUFFER
2657:                   952 *
2657:                   953 * THE BYTES FROM BUFFER TO
2657:                   954 * MIDBUFF IS THE RAMDISK VOLUME
2657:                   955 * DIRECTORY HEADER. THESE BYTES
2657:                   956 * HAVE TO BE WRITTEN TO THE
2657:                   957 * RAMDISK AT BLOCK 2 BEFORE THE
2657:                   958 * RAMDISK CAN BE USED BY PRODOS
2657:                   959 *
2657:                   960 * WE WILL ALSO USE PARTS OF
2657:                   961 * THIS VOLUME DIRECTORY HEADER
2657:                   962 * AS A CHECK TO SEE IF THE
2657:                   963 * RAMDISK IS ALREADY FORMATTED
2657:                   964 * IF SO THE USER CAN REFORMAT
2657:                   965 * THE RAMDISK OR LEAVE IT ALONE
2657:                   966 *
2657:00 00              967 BUFFER    DB     $00,$00
2659:                   968 * LAST BLOCK
2659:03 00              969           DB     $03,$00
265B:                   970 * STORAGE TYPE/NAME LENGTH
265B:F3                 971           DB     $F3
265C:                   972 * VOLUME NAME
265C:52 41 4D           973 VOLUME    ASC    'RAM'
265F:00 00 00 00        974           DB     $00,$00,$00,$00
2663:00 00 00 00        975           DB     $00,$00,$00,$00
2667:00 00 00 00        976           DB     $00,$00,$00,$00
266B:                   977 * RESERVED
266B:00 00 00 00        978           DB     $00,$00,$00,$00
266F:00 00 00 00        979           DB     $00,$00,$00,$00
2673:                   980 * CREATION DATE/TIME
2673:00 00 00 00        981           DB     $00,$00,$00,$00
2677:                   982 * VERSION/MIN VERSION
2677:00 00              983           DB     $00,$00
2679:                   984 * ACCESS BYTE
2679:43                 985           DB     $43
267A:                   986 * ENTERY LENGTH
267A:27                 987           DB     $27
267B:                   988 * ENTERIES PER BLOCK
267B:0D                 989           DB     $0D
267C:                   990 * FILE COUNT ON DISK
267C:00 00              991           DW     $0000
267E:                   992 * BLOCK NUMBER OF BIT MAP
267E:06 00              993           DW     $0006
2680:                   994 * TOTAL NUMBER BLOCKS ON DISK
2680:00 02              995           DW     $0200


2682:                   996 * REST OF BUFF MORE THEN NEEDED
2682:         0200      997 MIDBUFF   DS     512
```

```
234B ABORTMSG        23F9 BE0             2409 BE1             FF3A BELL
2235 BITMAP            D2 BLOCK1            D3 BLOCK2        ?   D4 BLOCK3
  D5 BLOCK4            D6 BLOCK5          ?   D7 BLOCK6        46 BLOCKNUM
223E BM2               44 BUFFADR         23F1 BUFFERROR      2657 BUFFER
2118 C1              211C C2              2124 C3             2126 C4
2201 CB1             2215 CB2             2218 CB3            2277 CD0
2286 CD1             2295 CD2             22AA CD3            22BA CD4
21E5 CE              2250 CHECKDEV        222B CL1            2228 CLEARBUFF
264A CLEARDIR        21E2 CLEAREND          42 COMMAND      ?210C COMPUTE
2169 CONT            FDED COUT            FD8E CROUT            8D CR
22D9 DA1             22E9 DA2             22D1 DEVABORT       BF1A DEVADR51
BF31 DEVCNT          BF32 DEVLST            06 DIRBLOCKS      22EF DIRFOUND
2564 DISPVOL         2566 DV0             2576 DV1            2578 DV2
258B DV3             218F ENDSSD            00 EOL            217E EX1
218E EX2             217A EXIT            BEF5 GETBUFF        2541 GETKEY
2544 GK1             FC58 HOME            C000 KEY            2190 MAIN
2482 MIDBUFF         264B MLIWRITE        BF00 MLI          ?2651 MLIREAD
21B3 MN1             259B MSG             24B3 NOBUFF         D0A2 NODEV
22CB NODIR           21B6 R1                48 RAMBANK      ?2655 READBLOCK
?2653 READBUFF       ?2156 READBYTE         01 READ           21B4 RELOCATE
 2106 S1               08 SAVEBYTES      21C8 SD1            213C SETBLOCK
 21C1 SETDEV         21D7 SETVECTOR         02 SSDCODE        C0D1 SSDHI
 214C SSDIO          C0D0 SSDLOW          2100 SSDSTART       C010 STROBE
   4A TEMP          ?   00 TEMPZERO       2554 TI1            2563 TI2
 254F TITLE         ?   43 UNITNUMB       2631 VOLNAME        265C VOLUME
 BE00 WARMDOS         240F WHEREMSG       2397 WHERETO      ?   02 WRITE
 264F WRITEBLOCK      2161 WRITEBYTE      21EF WRITEDIR       239C WT0
 23AC WT1             23CB WT2            23DA WT3            23E8 WT4
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 14-JUN-83 09:55
** TOTAL LINES ASSEMBLED   997
** FREE SPACE PAGE COUNT   87
```

End Listing

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

The programs published in this month's column are available for downloading from the Laboratory Microsystems RBBS at (213) 306-3530 (300 baud or 1200 bps).

Microsoft Assembler

I never cease to be astonished at the flurry of letters that results every time we publish something about the Microsoft Macro Assembler.

J. C. Hoisington of Morgan Computing (publisher of Trace86 and Professional Basic) writes: "We switched from IBM to Microsoft at Version 2.0. That means we didn't get SALUT but we did get to upgrade to Version 3.0 early this year for $75.00. Version 3.0 finally corrects many of the errors (and ridiculous restrictions) of earlier versions. The most useful to us is that they finally allow

```
lods byte ptr es:[si+0]
```

"which assembles as 26AC. I still found many bugs in Version 2.0, but 3.0 seems to have very few glitches."

David L. Rabbers, of Seattle, Washington, writes: "Regarding the errors [published in previous columns] two of them are yours, not the assembler's. Specifically, any constant ending in b or B is always treated as binary, and any constant ending in d or D is always treated as decimal. This is true regardless of the current radix setting. So, if you set the radix to 16, you must see to it that hex constants ending in b, B, d, or D are followed by an h or H. A little inconvenient, but how else could you specify binary or decimal constants in the midst of radix 16?

"One feature of Microsoft MASM Version 3.0 is a new linker that supposedly supports simple overlays. Unfortunately, there is no hint that I can find on how to use this feature. If you know how, or discover how, *please* publish it." Good point. My manuals don't breath a word about the construction of overlays. Anyone out there have some inside information on this subject?

Robert A. Blair, of Walnut, California, made the same point about radix overrides and went on to say: "This assembler should be used in Computer Science courses in college as an example of the wrong way to design a compiler; it is the worst I have ever seen. If you thought that Version 1 had a zillion bugs, wait until you use Version 2 for a while.

"I have to define my data before I use it (I thought I only had to do that in Pascal) or explicitly tell the compiler where it is to avoid phase errors. A multi-pass compiler should be able to figure out where the data is located without my having to tell it. The accompanying assembly listing shows a few errors that I know about. The 80286 PUSH immediate instruction generates the wrong length for some values. It will generate an instruction that requires two bytes of immediate data and only inserts one byte, or generates an instruction that requires one byte of data and inserts two bytes. The OFFSET operator causes link errors occasionally. The MASK used with HIGH does not work ... These are just the tip of the iceberg." Robert's contribution accompanies the column as the listing (page 106).

Converting to DOS 3

Dan Daetwyler is a regular correspondent to *Doctor Dobb's* and always has something interesting to say. This month he shared some of his experiences with the latest version of DOS: "When DOS 2.0 introduced the concept of a file handle, I looked it over fairly carefully, and then decided we were being shown 'the way to go.' I went. It was a nice concept, and effectively the number of file handles that could be used at one time was unlimited. They made it so you could not use more than 99 handles at once—a good, typical effort by programming to cope with the concept of infinity on a finite machine. This was all well and good, since no one was going to have more than 99 files going on a PC.

"Then along came the new Version 3.x DOS. Gee, it just *had* to be better (and bigger), and all that good stuff. I read the new features section, and there *were* a couple of goodies. I devoured them carefully, and put them to immediate use. I bought Version 3.0, even though each time I swear I'll wait for the x.1 release—I get so tired of fighting MS bugs. Still, I bought it, and the Technical Reference, and really thought I read the documentation thoroughly. Along came Version 3.1, for which I breathed a large sigh of relief, since I could now stop fighting DOS bugs and worry about my own. And then I hit the thing with one of my moderately large complex systems. And bounced like a rubber ball.

"Buried in the details of the FILE statement in the CONFIG dissertation is a neat little statement that they've arbitrarily reduced the number of active handles to 20! The constraint is for a process, which is interestingly undefined, but you can assume it's a DOS job or task. The horrible part is, they meant it!

"So 20 isn't too bad a constraint. There won't be too many people who want to exceed that limit. Well, on the same page there's a note that five of the handles are grabbed by DOS.

That's very true. So now you're down to 15. Still a reasonable number? Not in my estimation. Let's follow through on a reasonably complex data base problem. We've got several—let's say five—master files, each of which is indexed. The three primary files have double indices, while the less important two have only a single index. Five plus three times two plus two gives us thirteen. Still safe? Not really. As you're aware, the only way you can force a buffer purge in DOS is to pseudo-close a file. Doing this with a duplicate handle permits you to avoid the killing performance of a re-open. Let's be stingy. We'll only use one handle for this purpose. We've still got one left. However, I've yet to write a system that doesn't have at least three and sometimes half a dozen supporting files for archives, output, etc. The net result is that you end up short about five handles, even on a relatively small system."

Dan's idea of a "relatively small system" would give many of us nightmares. Note how Dan offhandedly supplies the reason for the existence of that mysterious DOS function 45H called DUP, whose "explanation" in the DOS Technical Manual is "Purpose: returns a new file handle for an open file that refers to the same file at the same position." I had always been perplexed by this function, and the Unix manuals weren't any more helpful than the DOS manual. By the way, the 5 handles preassigned by DOS to the Standard Devices aren't really lost. You can always close them, and then reuse them for some other file or device.

Dan writes on: "My current effort involves a minimum of 33 handles, and even that makes me uncomfortable. Goodbye to DOS Version 3.x. Except I can't! I can't really lock a customer into the old DOS. Especially *with* multi-tasking 'right around the corner.'

"If you think this tale has a happy ending, you're wrong. Let me tell you some of the things I've tried. You can set up a resident task, split the data base, and let the resident task own part of it. The catch is that if you try to communicate with the resident task, you're running under the main task control structure, and DOS will tell you that the access isn't authorized. I haven't figured out how DOS knows, as yet, so I can't figure out how to switch task structures. The traditional use of a user interrupt vector doesn't work. DOS senses this and tells you again that the access is not authorized. Setting the file up with the proper attributes for a shared resource, and loading SHARE, doesn't help. I guess I could fight through a disassembly of DOS and find out what they're doing, but I don't want to spend the rest of the summer on this.

"I started to drop back to my own FCBs, and drop the handle concept. There's no limit to the number of files I can open, then, but then I'm back to the problems of the pre-handle DOS, and I can't see any way to purge the buffers. I can't really see exposing my customer to loss of data base when the power drops unexpectedly. I can't actually give him total integrity, but using the pseudo-close purge gets the risk factor down to the noise level. I've three of the complex database systems out now, and in use on about fifty machines. We're semi-rural, and the power fluctuation during thunderstorms has to be seen to be believed, yet I've never (knock wood) lost a database from that source, since I started using the pseudo-close.

"So right now, I'm stuck. I can't move two of my systems to DOS 3.x, and the new one isn't even in the ball park . . . So this tale ends with questions. Do you know anything of help in this area?"

Actually, even the DOS 2.1 manual (pages 4-10) said that the maximum number of handles that a process could have concurrently open was 20, but that the maximum for the system was 99. This apparently includes background tasks such as the print spooler. Perhaps they just didn't enforce the process maximum. The manual for DOS 3.1 specifies the maximum for the system can be in the range 8-255, but again the maximum per process is 20. For those of you who are still using FCBs for whatever reason, be warned that DOS 3.1 puts some new and fairly strict restrictions on the use of FCBs when the file sharing module is also loaded. The default maximum number of FCBs that can be simultaneously open is 4. If you try to open more than your maximum, DOS unilaterally closes the least recently used FCB. You can increase the maximum allowed number of simultaneously open FCBs by a command FCBS=nn in the CONFIG.SYS file, just as FILES=nn works for handles. This is an omen, I believe, that FCBs are as doomed as the dinosaurs.

Microsoft and IBM have issued particular warnings about use of FCBs in the networking environment. If you use an FCB to access a file and fail to close it (a very common thing to do when you are using the file for read only), the system can bog down due to excessive connections accumulating across the network. Also, they remark that moving FCBs around in memory after they are opened can cause problems.

Dan proceeds: "Tale two is slightly more amusing. When I got my AT, and then started hearing all the terror stories about the hard disk, I considered myself very fortunate when I didn't encounter any . . . for a while. Then I had to load a 5 Mbyte database, and my trouble started. After several days of frustration, I nailed my friendly dealer to the wall, and IBM emergency shipped a replacement hard disk, which made life a lot better. In the interim, I did several interesting things. I put a trap in the program so that any time the I/O failure occurred, I'd get control. I was running under DOS 3.0, of course. The first thing I discovered was that when the disk glitched, the error code that DOS gave me back had no relation to disk I/O at all. In fact, DOS kept insisting that I was 'out of memory.' On a half megabyte machine with one small application?—not bloody likely.

"The next thing I discovered was that if I simply tried the operation again, by looping back to the interrupt, 99% of the time the thing worked happily on the second try. Very interesting! Then I discovered by accident that I was running with VERIFY=OFF. So I changed to

VERIFY=ON, and got a whole new set of behavior patterns. Different error codes, and the fail ratio went way down. Running in this mode, the problem seemed to be occurring when DOS encountered something it thought was a bad track, so I offloaded my 20 Mbyte disk, and reformatted. Oh boy. The plot thickens like glue. I could run FORMAT repetitively, and every time I ran it I would find a different set of 'bad tracks.' There didn't even seem to be a consistent pattern. I believe it did have a relationship to something, but as near as I could detect, it was the phases of the moon, or the position of Jupiter.

"I won't bore you with more of the experiments that cost me about ten days of time, but I will tell you the punch line. We all heard about the hard disk problems in all of the trade journals. IBM replaced hard disks left and right, and the disk vendor was the bad guy. Very interesting, since, when I got Version 3.1, all the problems disappeared. Now DOS gives back the correct error codes. Now FORMAT is at least consistent."

Dan's second tale is the latest of a raft of mutually exclusive explanations of the mysterious AT hard disk problems. We have heard that the original hard disk was no good because the vendor tried to ramp up production too fast (and a recent review of the CMI hard disk showed it to be much more shock-sensitive and error-prone than its competitors); that the FORMAT program had a bug that prevented it from actually checking the second half of the 20 Mbyte hard disk; that the disk controller had design or component problems; and so on. Throughout most of this time, the official IBM press flacks insisted that everything was wonderful and there was no AT disk problem at all. It certainly makes one apprehensive about the position we'll all be in if IBM manages to knock off all its competitors in the PC market.

DDJ

16-Bit Listing (Text begins on page 102)

```
1                                       page    55,132
2                                       title   Microsoft Assembler Bugs
3                                       .286C
4
5                                   ; Some IBM PC Macro Assembler 2.0 "features"
6                                   ; contributed by Robert Blair
7
8       0000                        foo     segment para public 'CODE'
9                                           assume  cs:foo
10      0000                        foos    proc    far
11      0000  E9 010B R                     jmp     foobar
12                                                          ; this bug causes link errors if
13                                                          ; the offset is larger than 256.
14                                                          ; the error is
15                                                          ; 'fixup offset exceeds field width'
16      0003     02 [                       dw      2 dup (offset foobar)
17                    010B R
18                          ]
19
20
21                                                          ; this will work ok
22      0007     02 [                       dw      2 dup (foobar)
23                    010B R
24                          ]
25
26
27      000B  0100 [                       db      256 dup (0)
28                    00
29                       ]
30
31
32      010B  CB                    foobar: ret
33
34                                  ; it appears that if any calculations are required, the compiler
35                                  ; truncates after the first two bytes.  The assembler reference
36                                  ; manual p. 2-15 states this is done for the small assembler but it
37                                  ; does it for both assemblers.
38
39      010C  80 51 00 00                   dd      60*60*24        ; incorrect
40      0110  80 51 01 00                   dd      86400           ; correct
41
42
43                                  ; Now try HIGH and MASK operators together
44
45                                  fooi    record  f1:1,f2:1,f3:3,f4:1,f5:1,f6:1,f7:8
46      = 0200                      fc1     equ     0200h
47      = 4000                      fc2     equ     4000h
48
```

```
                                                        ; immediate data should be
0114  25 0200                          and     ax,mask f5                      ; 0200h
0117  25 4000                          and     ax,mask f2                      ; 4000h
011A  25 0040                          and     ax,high (mask f2)               ; 0040h
011D  25 0042                          and     ax,mask f5+high(mask f2)        ; 0240h
0120  25 4200                          and     ax,(high(mask f2))+(mask f5)    ; 0240h
0123  25 0200                          and     ax,fc1                          ; 0200h
0126  25 4000                          and     ax,fc2                          ; 4000h
0129  25 0040                          and     ax,high fc2                     ; 0040h
012C  25 0042                          and     ax,fc1+high fc2                 ; 0240h
012F  25 4200                          and     ax,(high(fc2))+(fc1)            ; 0240h
0132  25 4200                          and     ax,fc1 or high fc2              ; 0240h


                                ; 80286 PUSH IMMEDIATE instruction bugs
                                ; an op code of 6A requires one byte of immediate data
                                ; an op code of 68 requires two bytes of immediate data

0135  6A 00                            push    0               ; 6a00 correct
0137  6A 7F                            push    127             ; 6a7f correct
0139  6A FF                            push    -1              ; 6aff correct
013B  68 80                            push    128             ; should be 680080
013D  68 0080                          push    word ptr 128    ; generates correct code
0140  68 FF                            push    255             ; should be 6800FF
0142  68 00FF                          push    word ptr 255    ; generates correct code

0145  6A FFFF                          push    0ffffh          ; should be 6aFF or 68FFFF
0148  6A 34                            push    byte ptr 1234h  ; lost data with no error


                                ; I have to define my data before use or tell the assembler
                                ; where it is.  Multi-pass compilers should be smarter
                                ; than this

014A  2E: D7                           xlat    fred            ; causes (spurious) phase error
                                                               ; message but generates right code

014C  2E: D7                           xlat    cs:fred         ; works ok

              FF [              fred    db      255 dup (0)
 E r r o r   ---       6:Phase error between passes
                       00
                          ]


                                ; This bug is not very serious because the only thing wrong is
                                ; the assembly listing.  The generated code is correct but the
                                ; listing of the generated code is not very neat.

024C  0D 3D [D 0A                      db      13,10,13,10,61 dup ('-'),13,10
                       2D
                          ]
                         0D 0A


028F                            foos    endp
028F                            foo     ends
                                        end     foos                                    End Listing
```

Introduction to Using Your MacSCSI Host Adapter

by John Bass

The MacSCSI host adapter was designed to work with most SASI- and SCSI-compatible controller cards. It is also a reasonable starting point for other custom Mac interfaces. Source software written in Aztec C is provided for custom configuring your system. If your controller is Xebec S1410 compatible you should be up and running with little trouble.

Changing the Parameters for the Drive and Controller

For controllers other than the Xebec S1410, you may need to modify the last byte of the format, read, and write commands to reflect the vendor-specific requirements. The Xebec S1410 uses this byte to encode the step rate for the drive—as shipped, the code is set up to use a Seagate ST506 drive with the half-step strap enabled. If you are using a Xebec S1410 with another drive, consult your controller manual for the correct drive code to use. Then change the cmd.sc_vendor assignment from 1 to the correct drive-type code.

Because the Xebec defaults to an ST506-drive configuration we didn't need to issue any additional commands to define the drive configuration to the controller. This is an area that is very vendor specific—read your controller documentation very carefully. You will want to read the descriptions for the format, read, and write commands closely. Also check the error-code definitions carefully because they vary greatly between some controllers.

You may need to issue a Set Media or Set Mode command that defines the drive parameters for number of heads, cylinders, write precomp, write reduce, and so on. Some drives, such as the Xebec Owl, have an embedded controller and don't need such a setup parameter. Some controllers encode the parameters at format time and write them on the disk for future use. Other controllers require a Mode Select or other command to set the drive parameters prior to the first format, read, or write command.

The long evolution from early SASI controllers to the current proposed SCSI specification has left a lot of nonconforming controllers around the marketplace, and many more nonconforming controllers are likely to be produced before the standard is finished and adopted. Until then, the software drivers for each host adapter will need to be customized per controller.

A copy of the current SCSI (ANSC X3T9.2) specification is available from the American National Standard publications office. For NCR-5380 information contact your local NCR Microelectronics representative or write to the NCR Microelectronics Division, 1635 Aeroplaza Dr., Colorado Springs, CO 80916; (800) 525-2252.

Mounting Drives Inside or Outside the Mac

Several 5-, 10-, and 20-megabyte 3½-inch drives are now on the market, and controllers in the same form factor are available as well. The easiest way to mount a 3½-inch drive, controller, and power supply internally is to fabricate a sheet metal and/or Plexiglass mount and screw it inside the plastic back of the Mac. To get AC for your power supply, install a power cable to the video board after the switch and AC filters. Be sure to leave a long enough service loop on both the power and SCSI cables to make getting the back off easy.

Other internal mounting configurations require more attention to sheet metal design and choice of controller, drive, and power supply. Convection cooling can be enhanced by mounting everything on the right side and using the sheet metal and Plexiglass to form a chimney extending from the bottom to the vents at the top of the case. EMI fields from the power supplies, flyback, and yoke coils present some problems in getting the disk to operate without errors—proper selection of shielding material and position is critical.

The easiest way to interface drives to the MacSCSI is to mount them externally in a separate box, so that physical mounting, heat, and EMI problems are less severe. Several manufacturers have standard line enclosures. Some of these enclosures are complete with power supply, cables, fans, and maybe a drive of your choice.

External drives require an external cable. The SCSI standard specifies an EMI shielded cable, plug, and header assembly. We use AMP part number 1-499977-0 for the panel-mounted connector and AMP part number 102793-4 for the ground plane (shield assembly) that mates with the connector. Use the ground plane only if you plan to use a SCSI standard shielded cable—it will interfere with a standard nonshielded IDC ribbon-cable connector.

Installing the MacSCSI Board

Getting the back off a Mac requires a Torx 5 screwdriver with a long shaft. There are five screws: two are located at the top in holes under the handle, two are on the back toward the bottom above the connectors, and one is located behind the battery cover. Carefully use a wooden ruler or the battery cover to ease the crack along

the front seam apart—do not use a screwdriver as a pry because the case plastic is very soft and will show pry marks.

Remove the power and floppy cables from the digital board, then *gently slide* it out of the sheet metal frame. Place the board on a table with the keyboard connector toward the back and the other connector near you. Lay the MacSCSI board near the ROMs with the SCSI header near you. Remove and transfer both of the Mac ROMs onto the MacSCSI board, then plug the MacSCSI board back into the ROM sockets.

Now clip the miniplug onto pin 8 of the TSM PAL located at D1 on the right edge of the board. Lay the clip flat against the board alongside the 15.667200 crystal and secure it with a piece of masking tape across the crystal and over the edge of the board. Route the wire between the switches and toroid coil and continue alongside the metal shield to the MacSCSI board. Replace the digital board in the frame, flexing slightly to pass the MacSCSI header past the edges of the frame. Be careful not to pull the board out of the ROM sockets. Cable the SCSI port as required, reconnect the power and floppy cables, and reassemble the case.

Bringing Up the System for the First Time

Once you have the system cabled up and have made any required software changes, try formatting the drive. Watch for the drive to be selected. On some drives you can see the positioner step from track to track, and on others you may be able to feel the drive stepping. Formatting takes a while—for 5- and 10-megabyte drives it may take several minutes, and 100-megabyte drives may need a half hour. The format routine initializes the Mac file system by writing a volume header and clearing all other blocks and then rereads all blocks to check for errors.

If you have problems with Aztec C you can add printfs to the C I/O routines and follow the progress. Once you have the format routine and I/O routines running, compile and link the driver with the I/O routines.

If you are connecting to something other than a disk you can simply include the I/O routines in your application and drive the SCSI bus directly.

Miscellaneous Comments

We have used MacSCSI under Finder 4.1 and Aztec C Shell 1.06D and F with a nonpartitioned 5-megabyte drive. Aztec C Shell and environment seem to run well even with a very large number of files. Finder has some limits you should keep in mind. On large volumes it gets pretty slow after 150 files. Finder also crashes sometimes when too many files get created. Deleting some files or the desktop under Aztec C sometimes makes it happy; at other times it seems that you need to back up the disk under Aztec C Shell and reformat the volume. We are working with Apple to find the solutions to these problems.

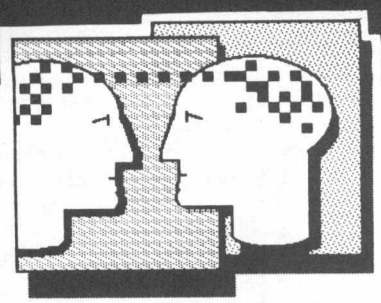
We have used the driver with and without a cache with some interesting results. For small cache sizes the overhead of maintaining the cache slows the overall operation down. For the Aztec C environment the minimum size is roughly the sum of all the programs, libraries, and include files in use—it is easier to start up the RAM disk and copy everything to it. For large cache sizes it is a pain to run large programs or the switcher because you need to restart the driver with a smaller cache size. The overall conclusion for our working environment is that, although the cache can create RAM-disk speeds, it sometimes creates floppy speeds. The MacSCSI hard disk is nearly as fast as the RAM disk anyway, and maintaining the cache just becomes a headache when switching environments.

Another minor annoyance is that you can't remount the MacSCSI volume if you eject it from an open box—never do so. We are working on adding the code to trap the ejects and ignore them. The eject solution, multiple volumes via partitions, and other improvements will be available in our 2.0 software release within a few weeks.

We are collecting changes for other controller-and-drive combinations to be available on our update disk. If you would like to share your changes, send us a disk with your modifications and we will return it with the current updates. Mail your changes to Fastime, P.O. Box 12508, San Luis Obispo, CA 93406.

DDJ

CP/M EXCHANGE



by W. E. Wilson

Software running on a Z80-based system under CP/M-80 will frequently perform better than comparable software running on an IBM PC under MS DOS. For example, Microsoft BASIC-86 is actually slower on the IBM PC than BASIC-80 on a Z80-based system. Programs running under CP/M-80 also are frequently smaller than their counterparts on MS DOS. Translate a Z80 assembly-language program into 8086 assembly language and watch the code expand.

Sixteen-bit systems do have a number of advantages over their 8-bit predecessors. You can address more memory and can have better, even color, graphics. The PC can communicate easily with mainframe systems and will probably be more compatible with future developments than the older 8-bit systems. On the other hand, there are still many more software packages available for CP/M than there are for MS DOS, some quite good and very reasonably priced.

When comparing Z80-based systems to the IBM PC, there are some basic facts that you should keep in mind. First of all, the limited addressable memory space of 8-bit systems used to force programmers to write tight, rapidly executing code. With the advent of 16-bit systems and more addressable memory space, software developers were no longer compelled to keep code size small, and they shifted to less efficient programming techniques, including writing in system-independent, high-level languages. Second, most popular 16-bit programs were originally written for 8-bit systems. Developers have often moved these programs to the PC by translating the 8-bit code line by line into the equivalent 16-bit code. The result is that much of the potential efficiency of the 16-bit systems has been lost. For example, many of the spreadsheets that run without any problems on a 64K 8-bit system require more than 128K on a 16-bit system. Third, the memory overhead for the MS DOS operating system is so much greater than the overhead for CP/M-80 that a 128K PC is approximately equivalent to a 64K CP/M-80 system. Finally, the 5¼-inch floppy disk drives on a PC are as slow as molasses in January when compared to the typical 8-inch drives on the older CP/M-80 systems.

W. E. Wilson, Washington State University, Nuclear Radiation Center, Pullman, WA 99164

Mini System Software for a Micro

Back in the days before the invention of the processor on a chip and the development of modern micro systems, memory was very expensive. Minicomputer companies such as Digital Equipment Company provided overlaying linkers for use with their computers. In the early '70's the institution where I worked had a DEC PDP-15 with 64K of memory. DEC provided a software package called Chain and Execute that allowed you to run a very large program on the PDP-15 by segmenting it into a series of overlays. I used Chain and Execute to run some huge programs written in FORTRAN for the IBM 360 on the little DEC system. Not very much is written on the subject of overlays[1], but this technique is used extensively in mainframe systems and in multiuser operating systems.

The Linking Process

The creation of an executable program from a number of source files involves several steps. First, the source files are individually read by the compiler and converted into one or more object modules. Some compilers produce relocatable object code directly. Small-C and its derivatives generate assembly-language output that must be assembled. The following discussion applies to compilers of the latter type only.

It is not practical to have the compiler generate all the code needed for each module. Instead, code is generated only for operations specific to the particular module. General-purpose functions and subroutines from the run-time library are merely referenced by the compiler and only later linked into the program. These external references are "place holders" into which absolute addresses are eventually placed. For functions and variables defined within the module, the compiler generates references relative to the beginning of the module. This means that if the module is eventually loaded at address L, then all references will be correct if L is added to them.

An assembler then reads each individual module created by the compiler and generates for each one a relocatable object module. This module is called relocatable because it can be relocated anywhere in memory.

The final stage is linking. The linker searches the run-time library for any referenced subroutines and functions. It loads these subroutines and functions along with the relocatable object modules into their final positions in contiguous blocks of memory. Finally, it fills in the absolute addresses for externals and adjusts the relative addresses for internals.

A standard linker builds programs in memory. Thus, the maximum program size is equal to the total memo-

```
0100–0102 JP   ENTRY
0103–0107 Identification Code
0108–0109 End + 1 of Last Overlay or first free memory location above overlay
          system
010A–010B End + 1 of Root Segment or the start of first overlay
010C–0114 Eight Character Root File Name


Byte 0–1    Overlay Transfer Address (Load Address)
Byte 2–3    End + 1 of Segment
Byte 4–11   Eight Character Segment File Name
```

Figure 1
Root and Overlay Segment Headers

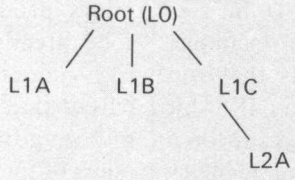


Figure 2
Structure of the test overlay system

```
ROOT(MAIN);OVINIT,OVRLAY,LO;
END;
SYMFIL(CTEST);
OVL(1,L1A);
INCL(L1A);
END;
OVL(1,L1B);
INCL(L1B);
END;
OVL(1,L1C);
INCL(L1C);
END;
OVL(2,L2A);
INCL(L2A);
END;
DONE;
```

Figure 3
ZLKO Command File Used to Create Overlay System

```
                    ZLKO Command Summary
CMND   = Take Command Input From Specified File
DCOM   = Display Common Block Information
DMAP   = Display Global and External Symbols
DONE   = Complete Current Segment and Quit
DOVL   = Display Overlay Structure
DSEG   = Display Information On Current Overlay Segment
DUND   = Display Undefined Symbols
END    = Complete Current Segment, forces a SRCH
HELP   = Display Command List
INCL   = Include The Listed REL Files in The Current Segment
OVL    = Build Overlay (.OVL) File
PBSE   = Set Program Code Base Location
QUIT   = Quit and Return to CP/M
RELDMP = Dump Relocatable (.REL) File
ROOT   = Build ROOT (.COM) File
SRCH   = Search Specified (.REL) Library
STOP   = Quit Building Current Segment and Save The Link Table
SYMFIL = Generate Symbol File For Symbolic Debugger
?SYM   = Suppress Symbols Starting with ?
TCSW   = Turn Off Module Counting
```

Figure 4
ZLKO Help Command Output

ry space minus the space used by the operating system and the linker. An overlaying linking loader links the program on disk rather than in memory and thus can link a program that will fill the entire available memory space or more. There is, however a penalty paid in speed. An overlaying linking loader must read every file twice—once to locate the globals and determine the size of the module and once to do the actual linking.

ZLKO

ZLKO from ZEE MICROWARE is an overlaying linking loader for Z80-based systems. The most important feature of ZLKO comes into play when a program is too big to fit into available memory space. The standard solution to this problem is to use the call-next-program technique. This technique involves dividing the program into a selected number of segments, each small enough to fit individually into memory. Each sequential segment except the last must load and execute the next segment. This method is only supported by a few compilers and has a number of disadvantages:

- It is not possible to return to a calling segment
- All segment intercommunication must be via disk files
- Selective segment execution is not possible

All of these weaknesses can be overcome by using the overlaying capabilities of ZLKO.

The user segments the program into a small root (main) module and one or more overlays, each of which is composed of a number of subroutines and functions. The subroutines are grouped into overlays according to criteria selected by the user. ZLKO then constructs a multilevel "tree" of overlays in which the overlays at any given level can call overlays at the next lower level. The execution of the segmented program begins when the root module is executed. The root will then activate other overlays by calling the subroutine "ovrlay" that has been linked with the root segment. The root segment will stay in memory

the entire time. The overlays will be loaded into memory and control passed to each as program execution proceeds. When an overlay returns control to the higher level that activated it, the higher level may then activate another overlay beneath itself or pass control down one level. Each overlay on the same level in the tree is loaded into memory starting at the same location.

Routines in an overlay can reference any symbol defined within that overlay and all global symbols located in any active higher-level overlay up to and including the root segment. Global symbols and routines in the root segment may be referenced from any segment. Frequently needed routines and functions should be included in the root segment. Routines in an overlay cannot, however, reference symbols or routines defined in an overlay on the same level or in a segment lower down the tree.

ZLKO Usage

ZLKO places a 21-byte header on the root-segment file and a 12-byte header on each overlay-segment file (see Figure 1, page 116). The headers include the primary CP/M file name of the segment as well as address information. In the case of an overlay segment, the overlay name and transfer address are used by the ovrlay subroutine to load the overlay. The execution process starts at the root segment and proceeds by descending down and climbing back up each branch of the overlay tree structure one branch at a time.

Using ZLKO with C offers one significant advantage over using it with a language like FORTRAN. FORTRAN employs the call-by-reference technique. Hence, arguments cannot be passed to a called overlay except by means of the COMMON block in the root segment. On the other hand, C uses the call-by-value technique. Thus, values can easily be passed to a function in an overlay. C passes the arguments by pushing them on the stack, and the function in the called overlay simply pops them off the stack.

ZLKO does not automatically insert code into the root segment to set the stack pointer because the required call to main( ) in the first routine in the root segment will take care of setting the stack pointer. That is, this operation is handled by the C compiler, which expands the call to main to include setting the stack pointer.

Most Small-C compilers use the symbol _memptr to designate the first free memory location above the program code space. The Q/C compiler I used to test ZLKO uses the symbol $MEMRY for compatibility with the L80 linking loader. Under ZLKO, the value to be used for $MEMRY or _memptr is stored in the 8th and 9th bytes of the root-segment header. This value must be extracted from the header and stored in location $MEMRY or _memptr in the compiled C program before program execution commences.

This operation is readily accomplished by using the ovinit function shown in Listing One (page 119). Ovinit must be linked into the root segment and must be the first function called from within main( ). The ovrlay function supplied with ZLKO must be linked into the root segment following ovinit. Each time an overlay is to be loaded, the user program must make a call to the ovrlay function using a pointer to the overlay name as the last argument of the ovrlay function call. The overlay test (see next section) demonstrates the use of the ovrlay function. Ovrlay will load the specified overlay at the transfer address specified in the header and then execute the overlay.

ZLKO Test

I tested the ZLKO overlaying linking loader with a simple overlay system written in C. The test program was compiled with the Q/C compiler and assembled with the CWA assembler distributed by The Code Works. The structure of the test overlay system is shown in Figure 2 (page 116) and the command file used to create the overlay system in Figure 3 (page 116). The code for the root segment and each overlay is presented in Listing Two (page 119). Listing Three (page 125) shows the output that ZLKO sent to the console during the test. The linking process took 126 seconds and the resulting overlay system executes in 11 seconds.

Using ZLKO

ZLKO is relatively easy to use once you understand how to pass to ovrlay a pointer to the overlay to be loaded by the root segment. There are, however, a couple of things you should keep in mind. First of all, because the input information required by ZLKO is extensive, the potential for keyboard errors is quite high. This problem can be solved by using a command file instead of direct input. For example, entering the command:

```
ZLKO CMND(CTEST);
```

causes ZLKO to take its input from the command file CTEST.ZLK. Second, you may have trouble getting accustomed to the fact that all commands require a combination of semicolon and carriage return as a terminator. Finally, although the manual adequately describes the proper use of ZLKO, it will not make you an expert at using overlays. All of the commands accepted by ZLKO are shown in Figure 4 (page 116) in the form of a printed copy of the output of the ZLKO Help command. ZLKO is quite verbose and lets you know what is going on at all times. Part of the verbosity is because of the fact that ZLKO displays each component of a command string as it is being parsed.

Conclusion

ZLKO is an easy-to-use, relatively efficient overlaying linking loader that is particularly useful to anyone using a Small-C compiler. ZLKO will allow the user to run very large programs under CP/M-80 and to construct fancy software in which a menu-selection operation determines which overlay (program) is executed next. There are at least two other overlaying linkers available for use with CP/M-80, but they are much more expensive than ZLKO and only provide a few additional capabilities. ZLKO only supports a multiple-file overlay file system, whereas some others also support a single-file overlay file system that uses the random-access capabilities of CP/M-80. A single-file

overlay system will execute slightly faster but is more difficult to construct and debug.

Do not discard your older 8-bit system if you can perform the tasks you need to accomplish by simply purchasing an overlaying linking loader such as ZLKO. ZLKO is available from Elliam Associates, 2400 Bessemer St., Woodland Hills, CA 91367.

Notes

[1] A good discussion of overlays is provided by S. H. Kaisler, *The Design of Operating Systems for Small Computer Systems*, John Wiley, 1983.

DDJ

CP/M Exchange

(Text begins on page 114)

Listing One

```
------------------------------------------------------------

/* Get addess of first free memory location from ROOT header
and put it in location $MEMRY. This version is for use with the
Q/C compiler.  Change $MEMRY to _memptr for use with small-c */

ovinit()
      {
# asm
        EXT     $MEMRY
        LD      HL,(0108H)      ;Get lst free mmry addr
        LD      ($MEMRY),HL     ;Put in $MEMRY
# endasm
     }
------------------------------------------------------------
```

End Listing One

Listing Two

```
------------------------------------------------------------------------

/*  Root program L0.C for testing ZLKO with Q/C */

#include <qstdio.h>

main()
    {
char *ovla, *ovlb, *ovlc;
ovla = "L1A     ";
ovlb = "L1B     ";
ovlc = "L1C     ";
/* initialize Root Segment For Overlays */
      ovinit();
        printf("ZLKO Overlaying Linker Test with Q/C\n\n");
        printf("Starting at level zero\n");
        printf("Call Overlay L1A\n");

 /* Load Overlay L1A */
          ovrlay(&ovla);        /* Pass pointer to ovlname */
            printf("Returned to Level zero from Level one pgm L1A\n");
            printf("Next Call Overlay L1B\n");
 /* Load Overlay L1B */
          ovrlay(&ovlb);        /* Pass pointer to ovlname */
            printf("Returned to Level zero from Level one pgm L1B\n");
            printf("Next Call Overlay L1C\n");
/* Load Overlay L1C */
        ovrlay(&ovlc);          /* Pass pointer to ovlname */
          printf("Returned to level zero, Overlay Test Done\n");
        }

------------------------------------------------------------------------

/* overlay L1A */
levella()
        {
        printf("At Level one, Overlay L1A\n");
        }


------------------------------------------------------------------------

/* overlay L1B */
levellb()
        {
        printf("At Level one, Overlay L1B\n");
        }

------------------------------------------------------------------------

/* overlay L1C */
levellc()
        {

char *ovld;
int value1, value2, sum;
ovld = "L2A     ";
        printf("At Level one, Overlay L1C\n");
        printf("Call Level two, Overlay L2A 5 times\n");
        /* Load Overlay 2A */
          value1 = 0, value2 =9;
          while (++value1 < 6)
        { ++value2;
  printf("At Level one Calling L2A, value1 = %d, value2 =
```

Listing Two

```
%d\n",valuel,value2);
        /* The pointer to the overlay name must be the last parameter */
        /* The ZLKO "ovrlay" routine uses the value in the HL register- */
        /* -to find the overlay name but does not alter the stack */
        /* The called routine must have a dummy last parameter */
          ovrlay(valuel,value2,&ovld); }
          /* Pass values and pointer to ovlname */
        }
int sum;
retval(sum)
        {
        printf("Back at Level one, Returned sum valuel + value2 =%d\n\n",sum);
        }

--------------------------------------------------------------------------------

/* overlay L2A */
level2a(argl,arg2,arg3)
        /* argl is valuel, arg2 is value2, and arg3 is ovlname pointer */
        /* dummy arg3 is required to match ovrlay function call in L1C */
int argl, arg2, arg3;
        {
        printf("At Level two, Overlay L2A\n");
        printf("Received parameter valuel =%d, value2 =%d\n",argl,arg2);
/* Pass sum argl + arg2 back to level one*/
        retval(argl + arg2);
        }
```

End Listing Two

Listing Three

```
        OUTPUT FROM ZLKO TO CONSOLE DURING TEST OVERLAY CONSTRUCTION

->CMND(CTEST);
  CMND(
  CTEST)<--Command File Input
  ROOT(
  MAIN)
  ;
  OVINIT,<--Implied Include List
  OVRLAY,
  L0;
  END;<--Searching For Required Modules

-- Building File on Disk --

Loading     (0:OVINIT .REL)
Loading     (0:OVRLAY  .REL)
Loading     (0:L0      .REL)
Loading from(0:CRUNLIB .REL)

  -- Filename --                    -- File Information --
  0:MAIN     .COM        P.Base: 0100    P.Size: 0EA7    Seg.End+1: 1182
                         D.Base: 0FBB    D.Size: 01C7
                         C.Base: 1182    C.Size: 0000

  SYMFIL(
  CTEST)<--Creating Symbol File
  ;
  OVL(
  1,
  L1A)
  ;
  INCL(
  L1A)
  OLV(
  1,
  L1B)
  ;
  INCL(
  L1B)
  ;
  END;<--Searching For Required Modules
  ;
-- Building File on Disk --

Loading     (0:L1B     .REL)
  -- Filename --                    -- File Information --
  0:L1B      .OVL        P.Base: 1182    P.Size: 000E    Seg.End+1: 11B7
                         D.Base: 119C    D.Size: 001B
                         C.Base: 11B7    C.Size: 0000
```

 (Listing continued, text begins on page 114)

Listing Three

```
  OVL(
  1,
  L1C)
  ;
  INCL(
  L1C)
  ;
  END;<--Searching For Required Modules

-- Building File on Disk --

Loading      (0:L1C      .REL)

  -- Filename --                    -- File Information --
  0:L1C      .OVL        P.Base: 1182    P.Size: 00B2    Seg.End+1: 12F5
                         D.Base: 1240    D.Size: 00B5
                         C.Base: 12F5    C.Size: 0000

  OVL(
  2,
  L2A)
  ;
  INCL(
  L2A)
  ;
  END;<--Searching For Required Modules

-- Building File on Disk --

Loading      (0:L2A      .REL)

  -- Filename --                    -- File Information --
  0:L2A      .OVL        P.Base: 12F5    P.Size: 0045    Seg.End+1: 138C
                         D.Base: 1346    D.Size: 0046
                         C.Base: 138C    C.Size: 0000

  DOVL;
--- Overlay Structure ---

Name     Number  P.Base  End     Parent
----     ------  ------  ---     ------

L1A      0000    1182    11B6
L1B      0001    1182    11B6
L1C      0002    1182    12F4
L2A      0003    12F5    138B    L1C

DONE;
No Segment Under Construction
```

End Listings

Index

16-Bit Software Toolbox, 75–84, 234–240, 323–327, 413–418, 481–486, 560–571, 648–651, 723–729, 849–854, 925–928

2651 USART, 458–462

68000
  binary search routine, 849–854
  pseudo-random-number generator, 849–854
  square root routine, 413-418, 849-854
  Tiny BASIC interpreter, 117–120

8086
  binary-to-ASCII conversion, 234–240
  unsigned divide subroutines, 234–240

Ada, An Advanced Introduction..., 164–165
Adding a COPY Command to ProDOS, 904–910
Aho-Corasick algorithm, 680–696
Albrecht, Bob, 106–109, 258–259, 559
Algorithms
  Aho-Corasick, 680–696
  Bose-Nelson, 698–703
  Knuth's hyphenation, 742–758
  pattern matching, 680–696
  search 678–679, 680–696, 697
  sort 678–679, 697, 698–703
Allison, Dennis, conversation with, 155
AMPRO Bookshelf Computer, 405–407
AMPRO Little Board, 405–407, 721–722
  real-time clock for, 794–799
Anderson, Brian R., 822–826
ANSI X3.28, 629–632
APL*PLUS/PC, Version 3.1, 59–69
Apple II
  adding COPY command to ProDOS, 904–910
  RAM-disk driver for ProDOS, 918–924
APL*PLUS/PC, Version 3.1, 59–69
Archiving Files with CP/M-80 and CP/M-86, 24–41
ASCOM, 629–632
Ashdown, Ian, 24–41, 73–74, 680–696
Assembler, Small-Mac, 623–628
Asynchronous Protocols, 629–632
Aztec C, 594–608
  root module for, 174–186

BANKSWAP: A Banked Memory Debugging Tool for CP/M Plus, 893–903
BASIC, 407–412
  logarithms, 808–815
Bass, John L., 711–720, 791–793, 929–930
Batch procedures, CP/M, 328–332
Batch processing, CP/M-68K, 911–917
Benchmarks for C compilers, 594–608
BetterBASIC, Version 1.1, 407–412
Binary-to-ASCII conversion, 8086 assembly language, 234–240
Bit Manipulation in Modula-2, 822–826
Bit-map manipulation, C, 436–445
Blum, Robert, 156–163, 241–246, 328–332, 489–494, 652–653, 721–722, 794–799, 855–867
Bose-Nelson Sort, 698–703
Bowhill, Sidney A., 765–784
Brandly, Gordon, *117–120*

BRIEF, 833–848
Brodie, Leo, 759–764
BU
  backup utility for CP/M-80, CP/M-86, 24–41
  patch to CP/M 2.2 BDOS, 426–427
Bubble sort, 678–679, 697
Build a Custom PC or Clone, 524–526
Burton, C. E., 121–139
Buvel, Raymond L., 94

C
  bit-map manipulation, 436–445
  book review, 73–74
  Christensen protocols, 463–478
  command-line processing, 353–359
  compilers, 594–608, 664–670
  croot.c, 174–186
  cursor manipulation, 879–888
  decision variables in graphics primitives, 360–366
  EFGETS, 879–888
  fgrep, 680-696
  getargs(), 353-359
  grep, 436-445
  Knuth's hyphenation algorithm, 742–758
  ls, 510–523, 742–758
  make, 633–647
  object-allocation module, 428–435
  pattern matching, 680–696
  pipes, 174–186
  preprocessor use and abuse, 446–451
  printf(), 103–105
  queue management, 436–445
  Quicksort, 316–322
  quoted arguments, 174–186
  Shell sort, 316–322
  shell support, 879–888
  style, 103–105
  wildcard expansion, 174–186
C Chest, 174–186, 316–322, 353–359, 436–445, 510–523, 633–647, 664–670, 742–758, 879–888
C Compilers for MS-DOS, 594–608
C Programming Tutor, The, 73–74
C-systems 8088 Software Development Package, 594–608
C-Terp, 594–608
C UART Controller, 458–462
C/Unix Programmer's Notebook, 103–105
Carroll, David W., 479–480, 629–632, 816–821,
Celko, Joe, 698–703
Children and computers, 106–109
Christensen Protocols in C, 463–478
CICFIG recompiler, 765–784
Cipher system, 121–139
Clark, David D., 6–7, 70–72, 273–292
Clock, real-time for S-100 bus, 534–608
Clute, Alan, 787–790
Coer d'Alene, printing characters on MX-100, 293–304
Command-line processing, C, 353–359
Commodore 64
  MBOOT and MODEM7 for CP/M, 42–56
  solid-shape graphics package, 367–388
Communications
  asynchronous protocols, 629–632

Christensen protocols in C, 463–478
C UART controller, 458–462
Envoy review, 479–480
freedom and security, 452–455
increasing modem speed, 456–457
MBOOT, MODEM7 for C-64, 42–56
specialized protocol software, 629–632
Compilers
C reviews, 594–608
design of, 660–663
Forth recompiler, 765–784
how they work, 664–670
peephole optimizer for, 609–622
Prolog, 389–398
recursive descent, 664–670
Small-C update, 623–628
Compiler Written in Prolog, A, 389–398
CompuServe A, B, 629–632
Computer Calisthenics, 572–577
Computer Innovations CI-86, 594–608
Conn, Richard, 405–407
Control C CC-86, 594–608
Controller, C UART, 458–462
Cortesi, D. E., 8–10, 98–102, 196–209, 260–263, 345–349, 350–352, 428–435, 507–509, 590–593, 660–663, 738–742, 808–815, 877–878
CP/M
batch procedures, 328–332
buffered disk I/O, 156–163
debugging console, 489–494
file-backup utility, 24–41
MBOOT, MODEM7 for C-64, 42–56
ZLKO overlaying linking loader, 931–936
CP/M 2.2, BU, patch to BDOS, 426–427
CP/M 2.2 Goes PUBlic
errors, 257
Heath users, 341–343
updates, 341–343
CP/M-68K Conditional Batch Processing, 911–917
CP/M-80, CP/M-86, file-backup utility for, 24–41
CP/M Plus
BANKSWAP, a banked-memory debugging tool, 893–903
PIP patch, 241–246
RSX for problem with, 241–246
CP/M Exchange, 156–163, 241–246, 328–332, 489–494, 652–653, 721–722, 794–799, 855–867, 931–936
CR-LF, 629–632
Croot.c, wildcard expansion and pipes for Aztec C, 174–186
Crosstalk, 629–632

Datalight C, 594–608
Day, Shawn, 904–910
DBASE II, changing default file type, 140–142
Debugging console for SID, ZSID, and DDT, 489–494
Debugging tools
CP/M Plus, 893–903
Turbo Pascal, 95–97
Decision variables in graphics primitives, 360–366
Design of a Forth Target Compiler, 765–784
Designing a Real-Time Clock for the S-100 Bus, 534–558
DeSmet C, 594–608
Device driver, template for MS-DOS, 560–571
DDJ editor, experience as, (Suzanne Rodriguez), 106–109
Diana, An Intermediate Language for Ada, 164–165
Disk Maker I, 400–405
Donais, Roger E., 911–917
Dr. Dobb is a Subversive, 350–352
Dr. Dobb's Clinic, 8–10, 98–102, 260–263, 345–349, 428–435, 507–509, 590–593, 660–663, 738–742, 808–815, 877–878
DRI C, 594–608
DUMP filter, 849–854
Duncan, Ray, 75–84, 234–240, 323–327, 413–418, 481–486, 560–571, 648–651, 723–729, 849–854, 925–928

EC, 833–848
EcoSoft C, 594–608
Edgar, G. A., 389–398
Editors
character, for FX-80, 273–292
reviews, 833–848
EDIX, 833–848
Edwards, Mark U., 833–848
EFGETS, 879–888
Emacs, 833–848
Encryption/decryption, 121–139
Enhanced ADFGVX Cipher System, An, 121–139
Envoy, Version 1.1, 479–480
Epsilon, 833–848
Epson FX-80, character editor for, 273–292
Epson MX-100, creating fonts on, 293–304
Evslin, Tom, interview with, 85–86

Fancy Font System, The, Version 2.0, 70–72
Fatten Your Mac, 2, 11–16, 170
Festschrift for Doctor Dobb, 106–109
Fgrep, file-search utility, 680–696
File transfer, packet protocol program, 463–478
Filters
DUMP, 849–854
TEE, 323–327, 648–651
TRIM, 481–486
Financial & Statistical Library, Version 1.0, 59–69
Fire in the Valley, 110–116
Fixpath() function, 738–742
Fonts, creating on MX-100, 293–304
Forth
MacFORTH review, 787–790
NC4000 processor, 759–764
Neon review, 785–787
recompiler, 765–784
unstructures programming in, 57–58
Forth compiler for MC68000, errors, 94
Fox, David L., 609–622
Freiberger, Paul, 110–116
Functions
EFGETS, editing input, 879–888
fixpath(), processing DOS path names, 738–742
Furuta, Richard, 704–710
Fx80char: A Character Editor for Epson FX-80 Printers, 273–292

Gay, Don, 458–462
GEM, 889–892
Gengle, Dean, 452–455
Getargs(), 353–359
GNU Manifesto, The, 187–190
Grammars, 664–670
Graphics package, C-64, 367–388
Graphics primitives, using decision variables in, 360–366
Graphics routines, IBM PC, 75–84
Greek and math symbols, producing on Prowriter, 269–272
Grep.c
adding bit maps, 436–445
compiling with C/80, 257, 263
error, 5
GW BASIC, comparison with BetterBASIC, 407–412

Hackers, 106–109
Hash-table search, 833–848
Head, Gene, 140–142
Heckel, Paul, 827–832
Hendrix, J. E., 623–628, 875
HEX file-conversion protocol, 629–632
Hogan, Tom, 360–366
Holub, Allen, 174–186, 316–322, 353–359, 436–445, 510–523, 633–647, 664–670, 742–758, 879–888

Homebrew Computer Club, 110–116
Horn, Bruce, 785–787
HP LaserJet, printing utility for, 723–729
Human factors
  designing software, 314–315
  engineering, 264–268
Hyphenation, using Knuth's algorithm, 742–758

IBM PC
  editors, 833–848
  graphics routines, 75–84
  sketching program for, 305–309
ImageWriter, producing Lisa QuickDraw images on, 17–23
Income tax, analyzing, 210–233
Information Age Issues, 452–455
Instant C, 594–608
Interpreters
  C, 594–608
  Tiny BASIC for 68000, 117–120
Introduction to Using Your MacSCSI Host Adapter, 929–930

Jennings, Earle, 95–97
Johnson, John R., 164–165

Kachigan, Karl R., 310–312
KERMIT, 629–632
King, Steve, 247–249
Krantz, Donald, 463–478
Kronman, Jim, 400–405, 524–526

Lafleur, Tom, 11–16
Lattice C, 594–608
  Control-Break handler for, 723–729
Let's Mouse Around, 305–309
Lisa, printing QuickDraw images on ImageWriter, 17–23
Logarithms in BASIC, Pascal, 808–815
Logic programming, applications and features, 190–195
Ls directory utility, 510–523, 742–758

MacFORTH, 787–790
Macintosh
  Open and Save dialogs for, 234–240
  SCSI hard-disk interface, 711–720
  upgrading to 512K, 11–16
  using MacSCSI host adapter, 929–930
MacKay, Pierre, 704–710
MacSCSI, 711–720
  driver routine, 791–793
  installation and uses, 929–930
Mac Toolbox, 711–720, 791–793, 929–930
Magic Mushroom for the Epson Alice, A, 293–304
Make utility, 633–647
Malpas, John, 190–195
Mayer, Thom, 17–23
MBOOT and MODEM7 for the C-64's CP/M, 42–56
MBOOT for C-64, 42–56
McKay, Derek, 257, 341–343
MDM730 for C-64, 42–56
Micro-PROLOG, 196–209
  compiler written in, 389–398
Microsoft C, 594–608
Microsoft Windows, 889–892
MicroTEX, Version 1.4A1, 704–710
Mitchell, Bridger, 257, 341–343
MITE, 629–632
MIX, 833–848
Mk utility, 633–647
MNP (Microcom Networking Protocol), 629–632
MODEM4 for C-64, 42–56
MODEM7, 629–632
  for C-64's CP/M, 42–56
  program compatible with, 463–478
Modems
  increasing speed of, 456–457
  packet protocol file-transfer program, 463–478
  S-100 review, 855–867
Modems: 2400 Bit/Sec and Beyond, 456–457
Modula-2
  bit manipulation in, 822–826
  compared with Pascal, 816–821
Modula-2 versus Pascal for Microcomputers: An Update, 816–821
Modula-2/86, Version 1.04, 143–154
MONTY, 264–268
MONTY Plays Scrabble, 264–268
More dBASE Tips and Techniques, 140–142
Morse, Philip D., II, 765–784
Mouse, using with Turbo Pascal sketching program, 305–309
Move-It, 629–632
MS-DOS
  chart of DOS services, 345–349
  cursor-manipulation, character-output routines, 879–888
  device drivers, program to display chain, 413–418
  DUMP filter for, 849–854
  editing input function, EFGETS, 879–888
  ls directory utility, 510–523
  make utilities, 633–647
  processing path names, 738–742
  requesting disk input, 660–663
  shell support routines, 879–888
  TEE filter for, 323–327
  template for installable device driver, 560–571
  TRIM filter for, 481–486
  windowing programs for, 889–892
MWC C Programming System, 594–608

NC4000, 759–764
Nelson, Amer W., 269–272
Neon, Version 1.0, 785–787
New Shell for MS-DOS, A, Part 1, 879–888

Object allocation, C, 428–435
Object-oriented programming, history of, 785–787
Optimizer for assembly-language source code, 609–622
Osborne 1, characters not on keyboard, 269–272

Palmer, Gary B., 293–304
Parallel Pattern Matching and Fgrep, 680–696
Parsers, recursive descent, 664–670
Parsons, Ronald G., 69–70, 305–309
Parsons, Vincent, 305–309
Pascal,
  QuickDraw routines in, 17–23
  sketching program in, 305–309
  Turbo vs. standard, 507–509
  using mouse in programs, 305–309
PC-DOS, Version 3.0, 75–84
PCTEX, Version 1.0, 704–710
PC/XT, building a work-alike, 524–526
Peephole Optimizer for Assembly Language Source Code, A, 609–622
Phonetic characters, printing on Epson MX-100, 293–304
Piotrowski, Walt, 42–56
PIP, patch for CP/M Plus, 241–246
Pipes, assembly-language support routines for, 174–186
Pittman, Tom, 106–109
Pmate, 833–848
Polar, 6–7
Polymake utility, 633–647
Printers
  building a spooler, 527–533
  character editor for FX-80, 273–292
  phonetic characters on MX-100, 293–304
ProDOS
  adding a copy command, 904–910
  RAM-disk driver for 918–924

ProDOS and the RAM Disk, 918–924
Programming
C preprocessor, use and abuse, 446–451
grammars, 664–670
object-oriented, 785–787
parsing, 664–670
recursive descent, 664–670
sorting, 678–679, 697
unstructured Forth, 57–58
using NC4000, 759–764
Programming in Logic, 190–195
Programs
adding copy command to ProDOS, 904–910
BANKSWAP debugging tool, 893–903
Bose-Nelson sort, 698–703
character editor for FX-80, 273–292
CICFIG Forth recompiler, 765–784
conditional batch processing with CP/M-68K, 911–917
CP/M benchmarks for buffered disk I/O, 156–163
debugging console for SID, 489–494
Greek and math symbols on Prowriter, 269–272
implementing S-100 modem features, 855–867
Knuth's hyphenation algorithm, 742–758
modifying USART parameters, 458–462
MS-DOS device drivers, 413–418
peephole optimizer for assembly-language source code, 609–622
phonetic characters on MX-100, 293–304
Prolog compiler, 389–398
Quicksort, 316–322
RAM-disk driver for ProDOS, 918–924
Shell sort, 316–322
sketching, using mouse, 305–309
solid shape drawing, C-64, 367–388
TA, Tax Advisor, 210–233
template for MS-DOS device driver, 560–571
XFR, file transfer, 463–478
Prolog
compiler in, 389–398
features, 190–195
income-tax analysis program in, 210–233
introduction to, 196–209
Micro-PROLOG, 196–209
syntax, 196–209
PROM-based machines, Forth recompiler for, 765–784
Protocols
ANSI X3.28, 629–632
ASCOM,629–632
asynchronous, 629–632
Christensen, in C, 463–478
CompuServe A, B, 629–632
CR-LF, 629–632
Crosstalk, 629–632
HEX file conversion, 629–632
KERMIT, 629–632
MITE, 629–632
MNP (Microcom Networking Protocol), 629–632
MODEM7, 629–632
Move-It, 629–632
Smartcom, 629–632
specialized software, 629–632
Telink, 629–632
XMODEM, 629–632
XON-XOFF, 629–632
Prowriter, producing Greek and math symbols with, 269–272
Pseudo-random-number generator, 98–102
Ps-make utility, 633–647
PUBLIC.ASM utility
error, 257
Heath users, 341–343

Quasi-random-number generator, 590–593
Queue-management routines, 436–445

QuickDraw Meets Imagewriter, 17–23
QuickDraw, routine to produce images on ImageWriter, 17–23
Quicksort, 316–322, 678–679, 697
Quoted arguments, 174–186

Raab, Susan, 11–16
RAM-disk driver for ProDOS, 918–924
Realizable Fantasies, 106–109, 187–190, 258–259, 350–352, 559
Relph, Richard, 594–608
Reviews
Ada, An Advanced Introduction . . ., 164–165
AMPRO Bookshelf Computer, 405–407
AMPRO Little Board, 405–407, 721–722
APL*PLUS/PC, Version 3.1, 59–69
Aztec C68, 594–608
BetterBASIC, Version 1.1, 407–412
BRIEF, 833–848
Computer Innovations CI-86, 594–608
Control C CC–86, 594–608
C Programming Tutor, The, 73–74
c-systems 8088 Software Development Package, 594–608
C-Terp, 594–608
Datalight C, 594–608
DeSmet C, 594–608
Diana, An Intermediate Language for Ada, 164–165
Disk Maker I, 400–405
DRI C, 594–608
EC, 833–848
EcoSoft C, 594–608
EDIX, 833–848
Emacs, 833–848
Envoy, Version 1.1, 479–480
Epsilon, 833–848
Fancy Font System, The, Version 2.0, 70–72
Financial & Statistical Library, Version 1.0, 59–69
GEM, 889–892
GW BASIC, 407–412
Instant C, 594–608
Lattice C, 594–608
MacFORTH Levels 1, 2, and 3, 787–790
Microsoft C, 594–608
Microsoft Windows, 889–892
MicroTEX, Version 1.4A1, 704–710
MIX, 833–848
Modula-2/86, Version 1.04, 143–154
MONTY, 264–268
MWC C Programming System, 594–608
NC4000 processor, 759–764
Neon, Version 1.0, 785–787
PC-DOS, Version 3.0, 75–84
PCTEX, Version 1.0, 704–710
Pmate, 833–848
Run/C, 594–608
SAVVY PC, 247–249
Software Toolworks C, 594–608
SPSS/PC, 312–313, 319
TOOLS, Volume 1, 59–69
Toolworks C/80, Version 3.1, Mathpack, 399–400
TopView, 889–892
Turbo Toolbox, Version 1.0, 310–312
U.S. Robotics Modem, 855–867
VEDIT PLUS, 833–848
VSI—Virtual Screen Interface, Version 2.09, 69–70
Wizard C, 594–608
XTC, 833–848
XyWrite II Plus, 833–848
ZLKO, 931–936
Rindsberg, Don, 527–533
Robinson, Howard H., 765–784
Rodriguez, Suzanne, 106–109
Routines
2^k in BASIC, Pascal, 808–815

68000 square root, 413–418
binary search, 849–854
bit-map manipulation, 436–445
character output, 879–888
CP/M file I/O, 156–163
cursor manipulation, 879–888
graphics, for IBM PC, 75–84
integer square root, 849–854
MacSCSI driver, 791–793
object allocation, 428–435
pipe support, 174–186
printing QuickDraw images on ImageWriter, 17–23
pseudo-random-number generator, 98–102, 849–854
quasi-random-number generator (8086, Z80), 590–593
queue management, 436–445
real-time clock, AMPRO Little Board, 794–799
shell support, 879–888
Rylander, Richard L., 367–388
Run/C, 594–608

S-100 bus, real-time clock for, 534–558
S-100 modem, 855–867
SAVVY PC, Version 4.0, 247–249
Schlobohm, Dean, 210–233
Schmidt, Michael, 143–154
Schreiner, Axel, 446–451, 673–677
SCISTAR for Prowriter users, 269–272
Scrabble, a computer variation of, 264–268
SCSI Your Mac, 711–720
Shell sort, 316–322, 678–679, 697
Shoemaker, D. C., 399–400
Shortcut to SCISTAR: For Prowriter Users, 269–272
SID2TTY debugging console, 489–494
Signals (Unix), 673–677
Skjellum, A., 103–105
Small-C
update, 623–628
bug, 503
fix, 875
Small-Mac, 623–628
Smartcom, 629–632
Software Designer, The, 85–86, 155, 314–315, 678–679, 827–832
Software Toolworks C, 598–608
Sola, Steven Anthony, 312–313, 319
Solid Shape Drawing on the Commodore 64, 367–388
Spooler for parallel printer, 527–533
SPSS/PC, 312–313, 319
Square root routine, 68000, 413–418
Stallman, Richard, 187–190
Steele, Alfred, 918–924
Swaine, Michael, 85–86, 110–116, 155, 258–259, 314–315, 559, 678–679, 889–892
System throughput, 98–102

TA, Tax Advisor Program, 210–233
Tax Advisor: A Prolog Program Analyzing Income Tax Issues, 210–233
TEE filter, 323–327
bug, 648–651
Telink, 629–632
Threaded code, Forth, 759–764, 765–784
Threaded Code Microprocessor Bursts Forth, A, 759–764
Tiny BASIC for the 68000, 117–120
TOOLS, Volume 1, 59–69
Toolworks C/80
Mathpack, 399–400
Version 3.1, 399–400
TopView, 889–892
Tour of Prolog, A, 196–209
Trask, Matthew, 407–412
TRIM filter, 481–486
Turbo Pascal
compared with Modula-2, 816–821
compared with standard, 507–509
debugging tool in, 95–97
sketching program in 305–309
Turbo Toolbox, Version 1.0, 310–312
Two TEX Implementations for the IBM PC, 704–710
Typesetting technical text, 704–710

UART, 458–462
Ultimate Parallel Printer Spooler, The, 527–533
Unger, Stefan H., 59–69
Unix
compatible software system, 187–190
signals, 673–677
Unix Exchange, 446–451, 673–677
Unsigned divide subroutines, 8086 assembly-language, 234–240
Unstructured Forth Programming, 57–58
USART
C UART controller, 458–462
in S-100 modem, 855–867
Using Decision Variables in Graphics Primitives, 360–366
U.S. Robotics Modem Review, 855–867
Utilities
Control-Break handler for Lattice C, 723–729
fgrep, file search, 680–696
file backup, CP/M-80, CP/M-86, 24–41
landscape printing with HP LaserJet, 723–729
ls, 510–523
make, 633–647
mk, 633–647
Polymake, 633–647
Ps-make, 633–647
PUBLIC.ASM, 257, 341–343

VALGOL, compiler for, 389–398
VEDIT PLUS, 833–848
VSI—Virtual Screen Interface, Version 2.09, 69–70

Walsh, Dale, 456–457
Ward, Terry A., 264–268
Wiesenberg, Michael, 572–577
Wilcox, Alan D., 534–558
Wildcard expansion, 174–186
Wilson, W. E., 931–936
Wilton, Richard, 57–58
Windowing Operating Environments: TopView, GEM, and Windows, 889–892
Windowing programs, 889–892
Wizard C, 594–608
Woodhull, Albert S., 893–903
WordStar files, incorporating Greek and math symbols in, 269–272

XFR, file-transfer program, 463–478
XMODEM, 463–478, 629–632
XON-XOFF, 629–632
XTC, 833–848
XyWrite II Plus, 833–848

Z80, ZLKO overlaying linking loader, 931–936
ZLKO overlaying linking loader, 931–936
Zoomracks: Designing a New Software Metaphor, 827–832

Other Software Tools from *Dr. Dobb's Journal* and M&T Publishing

Dr. Dobb's Toolbook for C Item #005 $29.95

From *Dr. Dobb's* and Brady Communications, this book includes a comprehensive library of valuable C code. *Dr. Dobb's* most popular articles on C are updated and reprinted here, along with new C programming tools. This book includes a complete C compiler, an assembler, text processing programs, and more!

The Small-C Handbook by James Hendrix Item #006 $17.95
The Small-C Handbook with MS/PC-DOS Addendum Item #006A $22.95

Also from *Dr. Dobb's* and Brady Communications, the handbook is a valuable companion to the Small-C compiler, described below. The book explains the language and the compiler and contains entire source listings of the compiler and its library of arithmetic and logical routines.

Small-C Compiler Item #007 $19.95

Like a home study course in compiler design, the Small-C Compiler and *The Small-C Handbook* (above) provide all you need to learn how compilers are contructed, as well as teaching the C language at its most fundamental level. Full source code is included on disk in both CP/M and MS/PC-DOS versions.

Small Tools: Programs or Text Processing Item #010A $29.95

This package of text processing programs written in Small-C is designed to perform specific, modular *functions* on text files. Source code only is included. Small Tools is available in both CP/M and MS/PC-DOS versions and includes complete documentation.

Small-Mac: An Assembler for Small-C Item #012A $29.95

Small-Mac is a macro assembler designed to stress simplicity, portability, adaptability, and educational value. Small-Mac is available for CP/M systems only and includes source code on disk with complete documentation.

SH: A Unix-like Utility Package for MS-DOS Item #160 $29.95

This MS-DOS shell written in C replaces COMMAND.COM and implements the most-used parts of the Unix C shell. The package includes a disk with full C source code and documentation in a Unix-style manual.

/util: A Unix-like Utility package for MS-DOS Item #161 $29.95

This collection of utilities is intended to be accessed through SH but can be used separately. It contains programs and subroutines that, when coupled with SH, create a fully functional Unix-like environment. The package includes a disk with full C source code and documentation in a Unix-style manual.

Dr. Dobb's Toolbook for Z80 by David Cortesi Item # 022 $25
Z80 Toolbook with software on disk Item # 022A $40

This book contains everything users need to write their own Z80 assembly language programs, including a method of designing programs and coding them in assembly language and a complete, integrated toolkit of subroutines. All the software in the book is available on disk.

Dr. Dobb's Toolbook of Forth Item # 030 $22.95
Toolbook of Forth with disk Item #031 $39.95

This comprehensive collection of useful Forth programs and tutorials contains expanded versions of *DDJ*'s best Forth articles and other material, including practical code and in-depth discussions of advanced Forth topics. The screens in the book are also available on disk as ASCII files in the following formats: MS/PC-DOS, Apple II, Macintosh, CP/M.

Dr. Dobb's Toolbook of 68000 Programming Item # 040 $29.95

From *Dr. Dobb's* and Brady Communications, this collection of practical programming tips and tools for the 68000 family contains the best 68000 articles reprinted from *Dr. Dobb's* along with much new material. The book contains many useful applications and examples. The software in the book is also available on disk in the following formats: MS/PC-DOS, Macintosh, CP/M 8", Osborne, Amiga, Atari 520ST.

Dr. Dobb's Listing Disk #1/86 Item #170 $14.95

As a useful adjunct to the magazine, *Dr. Dobb's* offers the convenience of selected listings on disk. Disk #1/86 contains a collection of listings from *Dr. Dobb's* January through April 1986 issues. Formats: MS-DOS, Macintosh, CP/M.

Dr. Dobb's Listing Disk #2/86 Item #171 $14.95

Disk #2/86 contains selected listings from *Dr. Dobb's* May through August 1986 issues. Formats: MS-DOS, Macintosh, CP/M.

Dr. Dobb's Journal of Software Tools, 1-Year Subscription Item #001 $25

To order any of these products, send your payment along with $2.25 per item for shipping to M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063. When ordering disks, please indicate format.

The Dr. Dobb's Series

Volume 1

Chronicles the advent of the microcomputer era in 1976. Always pertinent for bit crunching and byte saving, home-brew computer projects, and the technical history of home computing. Topics include: Tiny BASIC, the first words on CP/M, Speech Synthesis, Floating Point Routines, 6502 Disassembler for the Apple, and much more.

Volume 2

The small computer emerges as a powerful tool for the modern age in these issues from 1977. Topics include: Lawrence Livermore Lab's BASIC, Dr. Starkweather's PILOT, Using a Modem, String Handling Techniques, Ciphers, Turtle Graphics, micro utilities.

Volume 3

Explores the foundations of the mass market computer industry in 1978. Topics include: Programming in BASIC, PILOT, and Pascal, RAM Memory Testers, Apple utility programs, S-100 Bus Standard, STRUBAL: A Structured BASIC Compiler.

Volume 4

Heralds the widespread acceptance of the microcomputer in America. Innovative ideas and articles guide the reader through the age of discovery in 1979. Topics include: Selecting Business Software, Microcomputer Speech and Music, Information Networks, interfacing techniques.

Volume 5

Focuses on the technological promise of the modern microcomputer and the creative challenge facing programmers in 1980. Topics include: The revolutionary impact of CP/M, C programming and the UNIX operating systems, survey of computer networks, software portability, introduction to FORTH, compiler writing.

Volume 6

The microcomputer enters the mainstream—these issues assess the revolutionary impact of microcomputers on the individual and on our society in 1981. Topics include: Computer Conferencing, The Power of FORTH, 8- and 16-Bit Technology, Rubik's Cube Simulator, An Adventure Game Development System.

Volume 7

Examines the potential of powerful 16-bit micros including the birth of the IBM PC in 1982. Topics include: In-depth coverage of FORTH, 68000, 8088 programming, C software tools, utilities for CP/M including a spelling checker, using bulletin boards, and more.

Volume 8

DDJ turns pro. Some of the most powerful, professional programmer's tools ever published in a magazine are in this volume, including Small-C, the RED editor, and an Ada subset.

Volume 9

Shaping things to come. In 1984 DDJ examined new programming environments: Prolog, expert systems, Modula-2, and Pascal. Other topics include GREP, Unix internals, and two encryption systems.

Volume 10

The year of living dangerously. In 1985 DDJ added more memory, a SCSI port, and a hard disk to the Macintosh; challenged the Unix establishment with plans for a free Unix; and asked hard questions about privacy and control in the Information Age. Includes software tools in C, Modula-2, Forth, Pascal, assembly language, and Prolog.



M&T BOOKS