MICRO/Apple
MICRO/Apple 1

Ford Cavallari, Editor

MICRO INK
Incorporated
P.O. Box 6502
Chelmsford, Massachusetts 01824
Notice

Apple is a registered trademark of Apple Computer, Inc.
MICRO is a trademark of Micro Ink, Inc.

Cover Design and Graphics by Gary Fish

Every effort has been made to supply complete and accurate information. However, Micro Ink, Inc. assumes no responsibility for its use, nor for infringements of patents or other rights of third parties which would result.

Copyright© 1981 by Micro Ink, Inc.
P.O. Box 6502 (34 Chelmsford Street)
Chelmsford, Massachusetts 01824

All rights reserved. With the exception noted below, no part of this book or the accompanying floppy disk may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without prior agreement and written permission of the publisher.

To the extent that the contents of this book is replicated on the floppy disk enclosed with the book, it may be stored for retrieval in an Apple Computer. The original retail purchaser is permitted to make one [1] copy of the disk solely for his own back-up purposes.

MICRO/Apple Series ISSN: 0275-3537
MICRO/Apple Volume 1 ISBN: 0-938222-05-8
Printed in the United States of America
Printing 10 9 8 7 6 5 4 3 2 1
Floppy disk produced in the United States of America
To My Parents

Acknowledgements

I wish to thank the entire MICRO staff, and especially the editorial board which worked (almost) as hard as I did on this project. Special thanks go to Marjorie Morse, to whom this project really belongs and without whom there would be no book; to Richard Rettig for providing the insight and wisdom that only an experienced book publisher can; to Mary Ann Curtis, for providing editorial advice and moral support—both in abundance; to Gary Fish, for truly inspired artistic advice; to Editor/Publisher Bob Tripp, without whom there would be no MICRO and hence no MICRO/Apple; to the technical services staff at MICRO; and finally to Ski, for being there. Since the compilation of this book required extensive re-typesetting of text and equally major re-generation of program listings, I'd also like to express thanks and appreciation to all those hearty souls at MICRO involved in these processes. I especially wish to thank Emmalyn H. Bentley and Joanne McQueen for their endless hours of typesetting; Loren Wright, Darryl Wright, Paul Geffen, and Bill Francis for programming and technical services; Linda Gould for layout work; and L. Cathi Bland for editorial assistance. At Dartmouth, where I edited much of the work, I'd like to thank David [Taz] Townsend, Nick [Nuke] Armington, Jim [Boj] Pearson, and Heather [Mustafa] MacLeod, all of whom contributed more than they might imagine to this book's completion.

F.C.
# Contents

## INTRODUCTION

### 1 BASIC AIDS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applesoft Renumbering</td>
<td>5</td>
</tr>
<tr>
<td>Better Utilization of Apple Computer Renumber and Merge Program</td>
<td>9</td>
</tr>
<tr>
<td>SEARCH/CHANGE in Applesoft</td>
<td>12</td>
</tr>
<tr>
<td>An Apple II Program Edit Aid</td>
<td>17</td>
</tr>
</tbody>
</table>

## I/O ENHANCEMENTS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Little Plus for Your Apple II</td>
<td>25</td>
</tr>
<tr>
<td>Zoom and Squeeze</td>
<td>29</td>
</tr>
<tr>
<td>A Slow List for Apple BASIC</td>
<td>33</td>
</tr>
<tr>
<td>Alarming Apple</td>
<td>37</td>
</tr>
</tbody>
</table>

## RUNTIME UTILITIES

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Statement Generator</td>
<td>43</td>
</tr>
<tr>
<td>An EDIT Mask Routine in Applesoft BASIC</td>
<td>47</td>
</tr>
<tr>
<td>Business Dollars and Sense in Applesoft</td>
<td>55</td>
</tr>
<tr>
<td>Lower Case and Punctuation in Applesoft</td>
<td>62</td>
</tr>
</tbody>
</table>

## GRAPHICS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphing Rational Functions</td>
<td>69</td>
</tr>
<tr>
<td>A Hi-Res Graph Plotting Subroutine in Integer BASIC for the Apple II</td>
<td>75</td>
</tr>
</tbody>
</table>
How to Do a Shape Table Easily and Correctly  
*John Figueras*

Define Hi-Res Characters for the Apple II  
*Robert F. Zant*

Apple II High Resolution Graphics Memory Organization  
*Andrew H. Eliason*

## EDUCATION

Apple Pi  
*Robert J. Bishop*

Sorting Revealed  
*Richard C. Vile, Jr.*

Solar System Simulation with or without an Apple II  
*David A. Partyka*

Programming with Pascal  
*John P. Mulligan*

## GAMES

Spelunker  
*Thomas R. Mimlitch*

Life for Your Apple  
*Richard F. Suitor*

Apple II Speed Typing Test with Input Time Clock  
*John Broderick, CPA*

Ludwig Von Apple II  
*Marc Schwartz and Chuck Carpenter*

## REFERENCE

An Apple II Programmer's Guide  
*Rick Auricchio*

Exploring the Apple II DOS  
*Andy Hertzfeld*

Applesoft II Shorthand  
*Allen J. Lacy*

The Integer BASIC Token System in the Apple II  
*Frank D. Kirschner*

Creating an Applesoft BASIC Subroutine Library  
*N.R. McBurney*

LANGUAGE INDEX  

AUTHOR INDEX — Biographies included  

DISK INFORMATION
Introduction

Over the past four years, MICRO magazine, the 6502 Journal, has covered the Apple more comprehensively than any of the other 6502-based microprocessors. Apple-related articles in MICRO have outnumbered all others combined. This trend, of course, does not detract from the quality of the other 6502 machines; it merely serves to underscore the tremendous popularity of the Apple. Among businesspeople, educators, scientists, and computer hobbyists, the Apple has firmly established itself as the preferred machine.

MICRO published its first issue in 1977, the year that the Apple II was first available commercially. MICRO's first issue, a rather thin and unsophisticated publication, bore on its cover a large picture of the then unknown Apple II. Since then, Apples have multiplied enormously. Right now, some 150,000 Apples around the world perform an astonishing variety of tasks. And, thousands of their users read MICRO regularly.

While Apples found their way to many corners of the earth, so has MICRO, for Apple users have been eager to take MICRO's word on how best to use their machines. As Apples multiplied, the number of pages in MICRO has more than quadrupled; the quality of the magazine has been greatly enhanced; and the magazine's coverage of the Apple has expanded accordingly. As Apple and MICRO grew up together—and both continue to prosper—it seems natural to offer Apple users an improved version of some of the best and most useful Apple-related articles published in MICRO. With this volume, we therefore begin a new series dedicated to Apple users the world over, a series entitled MICRO/Apple.

This volume, MICRO/Apple 1, contains some of the more valuable, general interest articles and programs published in the magazine since 1977. For the first time, they have been brought together, placed into chapters, and updated and corrected by the authors and the MICRO staff. All the material—even that which appeared back in 1977—is now up to date. And all the programs appearing with these articles—Integer BASIC, Applesoft BASIC, and assembly language programs alike—have been keyed into our MICRO-lab Apples, tested, and printed out in a standard format.

Subsequent volumes of MICRO/Apple will contain comprehensive reference materials, more advanced machine language routines, educational primers, and even games. MICRO magazine will of course continue—in fact expand—its coverage of the Apple. MICRO will continue to be the source for new and innovative programs, product release information, Apple, and 6502 news. MICRO/Apple will be its reference partner—the book you keep next to your Apple along with the magazine. Future MICRO/Apple volumes will contain some of the best MICRO articles, plus other original material, some of it too lengthy to fit into the magazine format.
MICRO/Apple has been designed with the user in mind. The book will lie flat on a desk, next to your Apple. A diskette (13 sector DOS 3.2 format), which comes with each book, was created to free you from the tedium of typing in hundreds of lines of code.

We hope that this new approach will encourage use of those routines that you might have seen or heard of, but were unable to type in. And we especially hope that our MICRO/Apple books will encourage you to experiment with your Apple, to learn more of its capabilities—for that is what MICRO has always been about.

The microcomputer/microprocessor revolution is definitely here. And the Apple II is clearly one of the computers at the head of this revolution. The Apple is not merely an extremely useful tool—it can accomplish more than many room-size computers of just ten years ago. So, congratulations for owning or having access to an Apple, one of the finest micros in the world! And good luck...because, with an Apple at your side and this volume of MICRO/Apple in hand, you are helping to lead the revolution!

Ford Cavallari, Editor
March 1981
# 1 BASIC AIDS

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Applesoft Renumbering</td>
<td>5</td>
</tr>
<tr>
<td>James D. Childress</td>
<td></td>
</tr>
<tr>
<td>Better Utilization of Apple Computer Renumber and Merge Program</td>
<td>9</td>
</tr>
<tr>
<td>Frank D. Chipchase</td>
<td></td>
</tr>
<tr>
<td>SEARCH/CHANGE in Applesoft</td>
<td>12</td>
</tr>
<tr>
<td>James D. Childress</td>
<td></td>
</tr>
<tr>
<td>An Apple II Program Edit Aid</td>
<td>17</td>
</tr>
<tr>
<td>Alan G. Hill</td>
<td></td>
</tr>
</tbody>
</table>
Introduction

This chapter contains a group of utility programs designed to make programming in BASIC easier and less time consuming. Among the Applesoft aids, you'll find a number of programs and supporting articles which should not only be useful in practice, but also will help to enrich your understanding of the way Applesoft works. The "Applesoft Renumbering" article by J.D. Childress contains an informative discussion on Applesoft; this renumbering program provides an efficient and easily understood method to renumber a program. "Better Utilization of Apple Renumber and Merge" by Frank D. Chipchase, takes a different approach to renumbering—it suggests small modifications to the canned renumbering routine which comes with DOS 3.2 master diskette. These modifications make the program easier to use. Lastly, "SEARCH/CHANGE" by J.D. Childress presents a superb text editing system clearly and succinctly.

Moving to Integer BASIC, Alan Hill's "Program Edit Aid" provides a quick and easy way to search an Integer BASIC program for any string. This machine language program is fast and gives considerable insight into the structure of Integer BASIC.
Applesoft Renumbering

*by James D. Childress*

This reliable utility program enables the Apple programmer to renumber any Applesoft program. However, unlike most renumbering programs presently available, this one is written in straightforward Applesoft BASIC, and this provides an easily understandable insight into the way programs are stored in memory.

The need for a program written in Applesoft to renumber Applesoft programs is moot now that Apple has made available the 3.2 version of its disk operating system, that is, if you have a disk system. I wrote the present renumbering program while my disk drive was out of action, before the release of the 3.2 version.

**Comparison**

This Applesoft renumbering program (hereafter called RENUMB) is dreadfully slow; it took 7.9 minutes to renumber an 8.5K program. In comparison, the 3.2 disk renumber program did the job in 7.8 seconds.

RENUMB cannot change the line number after a GOTO, a GOSUB, or a THEN equivalent of a GOTO when the new line number has more digits than the old one. The program prints a list of these changes which must be made by hand. If there is not enough space, RENUMB inserts only the least significant digits. For example, the line

```
100 ON L GOTO 180, 190
```

with a line number shift upwards by 1005 would be given as

```
1105 ON L GOTO 185, 195
```

With the manual change instructions shown here:

```
LINE 1105: INSERT 1185 AFTER GOTO.
LINE 1105: INSERT 1195 AFTER COMMA.
```
If there is more space than needed, RENUMB inserts leading zeros. [Note that the Applesoft interpreter preserves such leading zeros whereas the 3.2 disk renumber program does not.]

RENUMB has one useful feature in common with the 3.2 disk renumber program, namely the capability of renumbering only a specified portion of a program. This feature must be used with care since you can renumber a part of a program with lines equal to, or in between, some of the line numbers of the remaining part of the program.

Unlike the 3.2 disk program, RENUMB does not order such lines into the proper sequence. If you really want that, you must run RENUMB first, then use the screen/cursor editing controls to copy the out-of-sequence lines through the Applesoft interpreter. You're left with the nontrivial problem of getting rid of the still remaining out-of-sequence lines.

Operation

To use RENUMB, you need to append RENUMB to the program to be renumbered. After the two programs are properly loaded, renumbering is accomplished by a RUN 63000 command. Give the requested information, then be patient; remember that RENUMB is numbingly slow.

Copy carefully all the manual changes listed. If you want to see them again, you can do so by a GOTO 63360 command provided you have done nothing to clear the variables, i.e., have not given any RUN commands or changed any line of the program.

You may use the SPEED command to slow up the display and the CTRL-C command to interrupt the display without clearing the variables. Once the variables have been cleared, there is nothing you can do except start from the beginning, that is, load the programs again.

At the beginning of the program run, you are asked for a rough estimate of the number of program lines (numbered lines) to be renumbered. Be generous, within limits of available memory. If your estimate is too small, you will get a

?BAD SUBSCRIPT ERROR IN 630X0

where \( x \) = 6, 7, or 8 since your estimate is used for array dimensioning. Unless your program is especially rich in branches, an estimate, say, about 50% greater than the number of line numbers will suffice.

Program Design

The design of RENUMB is quite simple. First RENUMB searches the program being renumbered for line numbers (and their memory locations) and the line numbers (and memory locations) after GOTOs, GOSUBs, THENs, and COMMAs in multiple branches. This search is done by lines 63040-63090 and for branches,
the subroutine at 63250. Lines 63130 and 63140 make the changes at the branches and line 63180 at the labels. The routine beginning at 63350 prints out those changes that must be made by hand.

All else is bookkeeping. Note: In line 63030, START is the address in memory of the beginning of the program. Finally, if you write very GOTOy and GOSUBy programs, you may want to change the definition of DD in line 63030.

**Applesoft**

Let your Apple be your textbook and teacher. For example, starting fresh with Applesoft in the computer, enter

```
1 PRINT: GOTO 521
521 PRINT "FREE":LIST 521
```

While this program runs without error, that is not necessary. You can enter anything you want to see how Applesoft handles it.

Now go to the monitor and look at

```
801 - OC 08 01 00 BA 3A
    AB 35 32 31 00

80C - 10 08 09 02 BA 22 46 52 45 45 22
    3A
    BC 35 32 31 00

810 - 00 00
```

for ROM Applesoft (1001 for RAM Applesoft). In the above lines, arranged here for clarity, OC, 08, 10 08, and the final 00 00 point to the next instruction in memory, the 00 00 pointer labelling the end of the program. 01 00 and 09 02 are the line numbers, 1 and 521 respectively. BA is the token for PRINT; 3A is the ASCII code for the colon; AB is the token for GOTO; 35 32 31 gives the line number for the GOTO; and 00 indicates the line ending. 22 46 52 45 45 22 is a direct ASCII code rendition of "FREE". Finally BC is the token for LIST and 35 32 31 is the line number 521 after LIST.

Study of the previous paragraph shows that Applesoft puts things into memory almost exactly the way you type them on the keyboard, except that the interpreter removes spaces, puts in instruction addresses, translates its command words into tokens, and uses ASCII code and hexadecimal, low-order bit first notation.
RENTER

```basic
62987 REM ***************
62988 REM *
62989 REM *APPLESOFT RENUMBERING *
62990 REM *JAMES D. CHILDRESS *
62991 REM *
62992 REM *RENEMBER *
62993 REM *
62994 REM *COPYRIGHT (C) 1981 *
62995 REM *MICRO INK, INC. *
62996 REM *CHELMSFORD, MA 01824 *
62997 REM *ALL RIGHTS RESERVED *
62998 REM *
62999 REM ***************
63000 END
63005 HOME: VTAB (3): PRINT "RENUMBERING PROGRAM": PRINT
63010 PRINT "LINES TO BE RENUMBERED": INPUT "BEGINNING LINE--":INPUT "ENDING LINE--":TRM: INPUT "TOTAL NUMBER OF LINES (ROUGHLY)---";
63020 INPUT "RENUMBERED BEGINNING LINE--":SK: INPUT "INCREMENT--":ADD
63030 PRINT "RENUMBERING BEGINNING LINE--":SK: INPUT "INCREMENT--":ADD
63030 START = 256 * PEEK (104) + PEEK (103):M = START + 2:DD = INT (D / 4): DIM LS(D),LM(DD),LM(DD),LOC(DD),NAS(DD),ND(DD),INS(DD),INS(DD)
63040 L = L + 1:LS(L) = M:LC = 256 * PEEK (M + 1) + PEEK (M): IF LC > 6
62900 THEN 63100
63050 FCRI J = M + 2 TO M + 255:TST = PEEK (J): IF TST = 0 THEN M = J + 3: GOTO 63040
63060 IF TST = 171 THEN NAS(K + 1) = "GOTO": GOSUB 63250
63070 IF TST = 176 THEN NAS(K + 1) = "GOSUB": GOSUB 63250
63080 IF TST = 196 AND PEEK (J + 1) > 47 AND PEEK (J + 1) < 58 THEN NA $(K + 1) = "THEN": GOSUB 63250
63090 NEXT
63100 FOR J = 1 TO L:LNU = 256 * PEEK (LS(J) + 1) + PEEK (LS(J)): IF L
63100 REM OR LNU > 62900 THEN PRINT: PRINT "RENUMBERING COMPLETED THROUGH LINE ":;LUN;":": GOTO 63350
63110 IF LNU < BGN THEN 63190
63120 SK$ = "0000" + STRS (SK):SKS = RIGHTS (SKS,5)
63130 FOR I = 1 TO K: IF LNU < > INS(I) THEN NEXT: GOTO 63100
63140 FOR KA = 1 TO ND(ID):POKE LOC(I) + 1 + ND(I) - KA, VAL ( MIDS (SKS
63150 +6 - KA,1,1)) + 48: NEXT
63160 IF LNU = INS(I) THEN INS(I) = SK
63170 IF LEN (STRS (SK)) > ND(I) THEN FCRI 1 = 1
63180 NEXT
63190 FOR I = 1 TO K: IF LNU = LN(I) THEN LM(I) = SK: IF LNU < BGN THEN
63200 LM(I) = LNU
63210 NEXT
63210 SK = SK + ADD: LUN = LNU
63220 NEXT
63250 K = K + 1:LN(K) = LC:SU = PEEK (J + 1) - 48
63260 FOR KA = J + 2 TO J + 6:FCRI = PEEK (KA): IF CPR = 0 OR CPR = 58 OR
63270 CPR = 44 THEN GOTO 63290
63270 SU = 10 * SU + CPR - 48
63280 NEXT
63290 LOC(K) = J:ND(K) = KA - 1 - J:INS(K) = SU:J = KA - 1: IF CPR = 44 THEN
63300 NAS(K + 1) = "COMMA":J = KA: GOTO 63250
63300 RETURN
63310 END
63330 IF PCP < > 1 THEN END
63360 PRINT :PRINT "NOTE: YOU MUST MAKE THE FOLLOWING CHANGES": PRINT "HERE"
63360 S MANUALLY": PRINT
63370 FOR I = 1 TO K: IF LEN (STRS (INS(I))) < = ND(I) THEN NEXT : END
63380 PRINT "LINE ":;LM(I);": INSERT ";INS(I);" AFTER ";NAS(I);".
63390 NEXT : END
```
Better Utilization of Apple Computer Renumber and Merge Program

Frank D. Chipchase

The renumber and merge program provided with the DOS 3.2 (and 3.3) is one of the most powerful utilities available for the Apple. The technique presented below adds even more power and versatility to this highly useful program, by making the routine more quickly and easily available from the disk.

I consider a utility program excellent when it can be utilized at any time under any condition. This brings me to that marvelous Applesoft Renumber and Merge program which comes with DOS 3.2.

Many times, during programming or editing, the need arises to move chunks of your program to different locations, to renumber portions of your program, or to merge in some of your favorite routines. Now comes the test of using a good utility program.

You did not load and run the A/S - R/N & M Program prior to starting work on your program. Now what?

Save your program, load and run the A/S - R/N & M program, now load back in the program you were working on and you are ready to go again. Meanwhile, your train of concentration has been broken on what you were originally doing.

There is a better way; at least I think there is. If we plan ahead a little bit.

If the A/S - R/N & M program is set up as a 'B' file then when it is needed it can be 'BLOADED' into memory while our program that is being worked on stays in memory and is undisturbed.

Here's the procedure in setting up an A/S - R/N & M 'B' file. The next time you boot a disk check to see what HIMEM: is set for right after the disk is booted. This is found by doing the following from the keyboard.

Print PEEK (115) + 256 * Peek (116) (C/R).
(On a 48K HIMEM: 38400 — on a 32K HIMEM: 22016.) The next thing to do after recording your system HIMEM: is to load and run that outstanding renumber and merge program that Apple Computer gave you on your master DOS 3.2 diskette. When the A/S prompt character returns it means that the Renumber program has been put into a little corner someplace in your computer's memory, ready for your beck and call.

Actually where it resides in memory is right under your system's previous HIMEM: which was set when you first booted (this is the number you first recorded).

HIMEM: has now been changed by the renumber program. Let's record the new HIMEM: again, from the keyboard.

Print PEEK (115) + 256*PEEK (116) (C/R)

(On a 48K HIMEM: 36352 — on a 32K HIMEM: 19968.)

We now have two numbers which we recorded. Subtract the smaller from the larger, this should equal 2048.

O.K., let's put the renumber program into a 'B' file on disk. From the keyboard:

BSAVE A/S-R/N-M, A [your 2nd HIMEM: number you recorded], L 2048. For a 48K this would look like BSAVE A/S-R/N-M, A36352, L2048. For a 32K BSAVE A/S-R/N-M, A36352, L2048. O.K., the 'B' file for the renumber program is all set.

Now, let's assume you are merrily programming away and the renumber program is not in memory.

The need occurs for renumbering, a merge or a hold. The newly created A/S-R/N-M 'B' file can now be 'BLOAD'ed in without disturbing your existing program. From the keyboard-BLOAD A/S-R/N-M (C/R). Once the 'B' file is loaded in, there are a few instructions that must be issued to your computer so that it knows the A/S-R/N-M program is in memory and where it is when it is needed. From the keyboard enter the following instructions:

For A 48K System:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIMEM:</td>
<td>36352</td>
</tr>
<tr>
<td>POKE 1013</td>
<td>76</td>
</tr>
<tr>
<td>POKE 1014</td>
<td>0</td>
</tr>
<tr>
<td>POKE 1015</td>
<td>142</td>
</tr>
</tbody>
</table>

For A 32K System:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIMEM:</td>
<td>19968</td>
</tr>
<tr>
<td>POKE 1013</td>
<td>76</td>
</tr>
<tr>
<td>POKE 1014</td>
<td>0</td>
</tr>
<tr>
<td>POKE 1015</td>
<td>78</td>
</tr>
</tbody>
</table>
O.K., that's it. You are all set to use the Renumber program. As you will note, your existing program is still in memory and is undisturbed. The first instruction reset your system's HIMEM: below the A/S-R/N-M program that you just BLOADED in. This is required for when you use the hold feature of the program. The last three POKE instructions tell the ampersand character "&", which you use when using the renumber program, where to find the A/S-R/N-M program in your system. (See Applesoft manual p.123.)

All the operating commands and formats that are used for the renumber program are valid and are used in the same manner. To free up the 2K of memory the A/S-R/N-M program is occupying, do a HIMEM: 38400 for a 48K system or a HIMEM: 22016 for a 32K system.

Now that you have come this far the ideal thing to do is set up a 'T' [text] file and let your disk 'exec' the whole procedure into your APPLE.

The program to write a text file would look like the following:

```
]LIST
10D$=CHR$(4): REM CTRL D
20 PRINT D$: "OPEN RENUMBER-MERGE"
30 PRINT D$: "WRITE RENUMBER-MERGE"
40 PRINT "BLOAD A/S-R/N-M"
50 PRINT "HIMEM:36352:" REM FOR 32K SYSTEM USE 19968
60 PRINT "POKE 1013,76"
70 PRINT "POKE 1014,0"
80 PRINT "POKE 1015,142": REM FOR 32K SYSTEM USE 78 IN PLACE OF 142
90 PRINT D$: "CLOSE RENUMBER-MERGE"
100 END
```

After the above program is run, a text file, named Renumber-Merge, will be created. Make sure this 'T' file is on the same diskette as your 'B' file A/S-R/N-M.

Now, whenever the renumber program is required all you have to do is type in EXEC Renumber-Merge.
SEARCH/CHANGE
In Applesoft

by J.D. Childress

Many larger computer systems feature text editors which simplify local and global editing of program lines. This SEARCH/CHANGE routine equips your Apple with the same capabilities. By directly interacting with your Applesoft program, SEARCH/CHANGE can massively reduce editing time!

Although slow, the program works well and reliably. But one warning, SEARCH/CHANGE is like the girl with the curl — when it’s good, it’s very, very good, but when it’s bad, it’s horrid. If it misfires, your program likely will be lost. The wise will always keep a current backup.

Preliminaries

Insertion of a CHANGE item longer or shorter than the SEARCH item requires that spaces either be added or deleted. This is accomplished by a shift of the program in memory and corresponding changes of all the next-address pointers. Needless to say, the part of memory space being used by SEARCH/CHANGE must not be jiggled or else its operation will be clobbered. So that the SEARCH/CHANGE program can remain fixed in memory and all the Applesoft operational pointers functional, spacers (colons) are added in line 62999.

The heart of this memory move is the call from Applesoft (see lines 63370 and 63380 in the program listing). The memory move call given in CONTACT 5, 5 (June 1979) works only for Integer BASIC; a call to the Apple HOTLINE produced the information that the move call had to be routed through a short machine language routine. The routine supplied by Apple is the following: POKE 768, 160: POKE 769, 0: POKE 770, 76: POKE 771, 44: POKE 772, 254. The corresponding call is then CALL 768. This location is $300-$304. I use that space for my SLOW LIST utility. Changing to location $342-$346 will cause no ill effects (see line 63010).

One block of memory cannot be moved into a second block overlapped by the first because one byte would be moved into another before the latter’s content had been read. Thus a two-step procedure is required. Line 63370 moves the memory block to the top of memory just below HIMEM and line 63380 moves it back to the desired end location.
Recall how Applesoft stores BASIC in memory. The end of each line is indicated by a zero byte. The next two bytes contain a pointer, low byte first, to the next line's first byte, the low byte of its next-address pointer. When a branch is executed, the program skips along these pointers from the first until the indicated address is found. If any next-address pointer points to a wrong address, all gang agley. So until all these pointers are put aright, the program being searched and changed is simply hidden from the operating program: Line 63160 POKEs the next-address pointer for line 63000 into the pointer location of line 1; after the dust settles, line 63230 restores the original pointer.

Design

A few additional comments on the design of the SEARCH/CHANGE Program are offered here in lieu of remark statements in the program itself.

A search is made, lines 63020-63030, from the end of program memory to find two things: the location of the LIST in line 63310 and the location of the beginning of line 62999. Two numbers—540 and 1730—are set for SEARCH/CHANGE exactly as written (lines 63000 and following) in the listing. The first number causes a skip from the end of the program to the immediate neighborhood of LIST, the second, a skip from LIST to near the end of line 62999. If any changes are made to SEARCH/CHANGE or if anything is put higher in memory, these numbers will have to be adjusted.

One effect of the above search is a delay after the return following RUN 63000. A too-short delay should alert the user that line 62999 might not have enough colons for substantial changes. If the colons are depleted, line 63350 halts the change operation and prints a message to that effect.

The perceptive reader will note a number of small numbers in various lines. These are finagle factors adjusted (but probably not optimized) to make the program run. For example, the 10 in line 63040 prevents the appearance of multiple line 62999's for a change item shorter than the search item. If changes are made in the program as given, these numbers also may have to be adjusted.

The program identifies the search item, FOR loop lines 63090-63110; then identifies the change item, if any, FOR loop line 63130.

The actual search is carried out by FOR loop lines 63180-63220. To get the best operating speed, we close the FOR loop within a single line (line 63180) if no byte of significance is found. Even so, the testing for up to three conditions takes time. If one of these conditions is not met, then the following lines either pass to subroutine line 63330 to complete the item identification test and make the item change (if one is entered), or set the strings search flag, or start the search of the next program line, whichever is indicated.

If a change is to be made, the afore described memory moving is done and line 63400 then POKEs in the replacement. After return from the subroutine, line 63210 updates the address pointers for the program line.
Line 63120 determines that the search is over when line 62999 is reached and passes to output after unmasking the hidden program. The routine, lines 63280-63320, accomplishes the line listing feature. Note that the line number has to be POKEd in so that there should always be five digits following LIST. After use of the program, the actual number that appears here when line 63310 is listed is the last number POKEd in. There should be leading zeros if that number had less than five digits. The Applesoft interpreter preserves these leading zeros whereas the Apple renumbering program does not. The colons allow space in the latter case.

Operation

Merge/append SEARCH/CHANGE to the program to be searched. Note that the program being searched lies below SEARCH/CHANGE in memory. Enter the search item as line 1 and the replacement item as line 2. Note that anything that will list as line 1 (or line 2) can be entered. Execute with a RUN 63000.

Experience has shown no need to search both inside and outside strings at the same time; it's an either/or situation. The SEARCH/CHANGE is made outside strings only unless the first character of line 1 is the quotation mark; in that case, the SEARCH/CHANGE is made inside strings only. For example,

1 TOTAL

would search for TOTAL outside strings but

1 "TOTAL

would search for TOTAL inside strings.

A quotation mark can be used with line 2 in a similar way to sneak "forbidden" words past the interpreter. To sneak things into line 1 (i.e. a number), preface the entry with @. This sneaking should be used with care in changes outside strings; the interpreter has a way of exacting its revenge on sneaky things.

Your attention is directed to the fact that the item replacement is made as each item is found. This fact taken with the slow speed of the program dictates patience on the part of the user. Don't hit Ctrl-C to see what's going on. Stopping things in the midst of a memory move or a pointer updating can be disastrous.

As mentioned earlier, things can go wrong sometimes, even without help. If the worst happens, try entering a new line or deleting a line or both; that sometimes will save part or almost all the program. A sensible precaution is to check line 62999 often and keep it well stocked with colons. And again, keep a current backup copy.

A wise idea is to use a SLOW LIST utility with SEARCH/CHANGE (I recommend the one supplied with S-C ASSEMBLER II). Then if a LIST command produces endless junk, the listing can be aborted without the additional hazard of a RESET.
Another good idea is to know your HIMEM. If something goes wrong, it's possible for the HIMEM setting to be ratcheted down to a low value.

Addendum

Since writing SEARCH/CHANGE, I have acquired APPLE-DOC by Roger Wagner. APPLE-DOC, an excellent set of programs, works well and is much faster than the present program. I use APPLE-DOC often but find some changes that APPLE-DOC can't make; for example, replacing something like B(IJ) by B(X%(I JJ).
[Listing of BASIC code]
An Apple II Program Edit Aid

by Alan G. Hill

Oftentimes it would be very useful to search an Integer BASIC program for occurrences of a certain string—especially when editing long programs. The Apple's built-in editor does not offer this feature, but this program edit aid does. The BASIC edit program drives a machine language routine, making the search capability here exceptionally quick.

When editing an Apple Integer BASIC program, you often want to locate all occurrences of a variable name, character string, or BASIC statements. This is usually the case when you are changing a variable name, moving a subroutine, etc., and you want to be sure you have located all references. This BASIC Edit program should aid your editing.

Load the BASIC program into high memory and append the program to be edited to it. The Edit program uses a machine language routine at hex 300 to 39F to search BASIC statements for the requested string and return the BASIC line number in memory locations 17 and 18. The routine is re-entered at 846 to find the line number of the next occurrence. This process is continued until no further occurrences can be found. The high order byte of the line number (location 18) is set to hex FF to indicate that the search is finished.

BASIC Edit Program

Note in line 32680 of the BASIC program that LIST LINE is an invalid BASIC statement. You will have to resort to a little chicanery to get the statement in. First code line 32680 as PRINT LINE. Then enter the monitor and change the PRINT token ($62) to a LIST token ($74). This is easiest done if you code line 32680 first and then search for the token in high memory ($3FFA when HIMEM is 16384). (Ed. Note: The author's assumption is correct.)

After coding the BASIC program and the machine language routine, you will then need to append the program to be edited. Note that the program must have line numbers less than 32600. To append a program, you must first 'hide' the Edit program. This is done by moving the HIMEM pointer (202) and (203) down below the Edit program. Then load the edited program and reset HIMEM. i.e.:
LOAD (EDIT PROGRAM)
POKE 76, PEEK (202)
POKE 77, PEEK (203)
LOAD (PROGRAM TO BE EDITED)
POKE 76,0 HIMEM MOD 256
POKE 77,64 HIMEM/256

You can then RUN 32600 the Edit program. Enter the character string or variable name to be searched when prompted by "'FIND?'". To search for a hex string (e.g. all occurrences of COLOR=), enter an @ character followed by the desired hex character pair (@ 66 for the COLOR= example).

EXAMPLES

To find all occurrences of: Input
SCORE SCORE
XYZ XYZ
RETURN @ 5B
DIM A @ 4EC1
All references to 1000 @ E803

The Edit program will end if the screen is full ( > 18 lines). To continue the search for more occurrences, a RUN 32720 will return another page. Happy Editing!

Find Routine
Page Zero Memory Map

$3-4 Address of search limit. Set to HIMEM by routine, but could be set lower to avoid searching Edit program.

$6-7 Address of BASIC Token compared. Incremented until it exceeds Limit Address.

$8-9 Ending address — 1 of current statement being scanned.

$A-B Address of string being searched. Set up by Edit program.

$C Length — 1 of string being searched. Set up by Edit program.

$11-12 Line number of statement containing the requested string. $12 is set to $FF if no more occurrences.

Here is an addition to make the program run smoother: If you add the following lines the Apple will pause and display an "'®'" in the lower right-hand corner when the screen fills up with text. This will prompt you to hit any key and the Apple will clear the screen and continue where it left off. This process will continue until there are no more occurrences of the search item.
Change: 32690

IF PEEK(37) 18 THEN 32700

Add:

32692 VTAB 23:TAB 39:PRINT"@"
32695 KEY = PEEK(-16384): IF KEY 127 THEN 32695
32697 POKE -16368,0: call -936

32580 REM ****************************
32581 REM * PROGRAM EDIT AID *
32582 REM * ALAN G. HILL *
32583 REM *
32584 REM * EDIT *
32585 REM *
32586 REM *
32587 REM * COPYRIGHT (C) 1981 *
32588 REM *
32589 REM * MICRO INK, INC. *
32590 REM * CHELMSFORD, MA 01824 *
32591 REM *
32592 REM * ALL RIGHTS RESERVED *
32593 REM *
32594 REM ****************************
32595 REM
32596 DIM A$(30)
32597 INPUT "FIND?",A$: CALL -936: IF A$(1,1)="@" THEN 32630:KK = LEN(A$):
32598 FOR I = 1 TO KK: POKE 911+I, ASC(A$(I,I)): NEXT I
32600 POKE 12,KK-1: GOTO 32650
32601 A$=A$(2, LEN(A$)):KK = LEN(A$): FOR I = 1 TO KK STEP 2: J = ASC(A$(I,I))
32602 -176:JJ = ASC(A$(I+1,I+1))-176
32603 IF J>9 THEN J=J-7: IF JJ>9 THEN JJ=JJ-7:
32604 POKE 912+I/2,J*16+JJ: NEXT I:
32605 POKE 10,912 MOD 256: POKE 11,912/256
32606 CALL 768
32607 IF PEEK (18)>127 THEN 32730:LINE= PEEK (17)+ PEEK (18)*256
32608 LIST LINE
32609 IF PEEK (37)>18 THEN 32730
32610 CALL 846
32611 GOTO 32670
32612 CALL -936: GOTO 32700
32613 END
0800 1 ;**************************
0800 2 ;* FIND ROUTINE *
0800 3 ;* BY *
0800 4 ;* ALAN G. HILL *
0800 5 ;* Copyright (C) 1979 *
0800 6 ;* ALL RIGHTS RESERVED *
0800 7 ;* BASIC AIDS ~
0800 8 :********************
0800 9 ;* ROUTINE TO SEARCH *
0800 10 ;* INTEGER BASIC PROGRAM *
0800 11 ;* GRAMS FOR ALL OCCURRENCES OF SPECIFIED ITEMS *
0800 12 ;* ******************
0800 13 ;* HILO EPZ $03 HIMEM LO BYTE *
0800 14 ;* HIHI EPZ $04 HIMEM HI BYTE *
0800 15 ;* BSL EPZ $06 BASIC STATEMENT LO *
0800 16 ;* BSH EPZ $07 BASIC STATEMENT HI *
0800 17 ;* SEAL EPZ $08 STATEMENT ENDING ADDRESS LO *
0800 18 ;* SEAH EPZ $09 STATEMENT ENDING ADDRESS HI *
0800 19 ;* LNL EPZ $11 LINE NUMBER LO *
0800 20 ;* LNH EPZ $12 LINE NUMBER HI *
0800 21 ;* MAIN PROGRAM *
0800 22 ;**********~*****
0800 23 ;********************
0800 24 ;********************
0800 25 ;********************
0800 26 ;********************
0800 27 ;********************
0800 28 ;********************
0800 29 ;********************
0800 30 ;********************
0800 31 ;********************
0800 32 ;********************
0800 33 ;********************
0800 34 ;********************
0800 35 ;********************
0800 36 ;********************
0800 37 ;********************
0800 38 ;********************
0800 39 ;********************
0800 40 ;********************
0800 41 ;********************
0800 42 ;********************
0800 43 ;********************
0800 44 ;********************
0800 45 ;********************
0800 46 ;********************
0800 47 ;********************
0800 48 ;********************
0800 49 ;********************
0800 50 ;********************
0800 51 ;********************
0800 52 ;********************
0800 53 ;********************
0800 54 ;********************
0800 55 ;********************
0800 56 ;********************
0800 57 ;********************
0800 58 ;********************
0800 59 ;********************
0800 60 ;********************
0800 61 ;********************
0800 62 ;********************
0800 63 ;********************
0800 64 ;********************
0800 65 ;********************
0800 66 ;********************
0800 67 ;********************
0800 68 ;********************
0800 69 ;********************
0800 70 ;********************
0800 71 ;********************
0800 72 ;********************
0800 73 ;********************
Hill Program Edit Aid 21

0350 8506 74 STA BSL ; TO NEXT STATEMENT
0352 A509 75 LDA SEAL
0354 8507 76 STA BSH
0356 A200 77 LDX $00 ; POINT TO LENGTH OF
0358 A902 78 LDA $02 ; STATEMENT BYTE
035A 2054 79 JSR INCPTN
035D DO81 80 BNE LENGTH ; ALWAYS BRANCH
035F A3FF 81 LIMIT LDA $SPF ; SET UP LARGE LINE NUMBER
0361 8512 82 STA LNH ; TO INDICATE AT END OF SEARCH
0363 60 83 RTS ; RETURN TO BASIC
0364 84 ;
0364 85 ;***********************************
0364 86 ;* POINTER INCREMENT ROUTINE*
0364 87 ;***********************************
0364 88 ;
0364 18 89 INCPTN CLC ; ROUTINE TO INCREMENT
0365 7506 90 ADC BSL,X ; POINTERS. ENTER WITH
0367 9506 91 STA BSL,X ; XREG = DISPLACEMENT
0369 8507 92 LDA BSH,X ; FROM
036B 6900 93 ADC $00 ; BSL,BSH
036D 9507 94 STA BSH,X ; ACC = INCREMENT AMOUNT
036F 60 95 RTS
0370 96 ;
0370 97 ;***********************************
0370 98 ; TOKEN ADDR INCREMENT ROUTINE*
0370 99 ;***********************************
0370 100 ;
0370 A506 101 INCTOK LDA BSL ; ROUTINE TO INCREMENT
0372 C508 102 CMP SEAL ; THE TOKEN ADDRESS BY 1
0374 A507 103 LDA BSH ; SET CARRY IF AT END
0376 E509 104 SBC SEAL ; OF STATEMENT
0378 E606 105 INC BSL
037A D002 106 BNE REXIT
037C E607 107 INC BSH
037E 60 108 REXIT RTS
037F 109 ;
037F 110 ;***********************************
037F 111 ;* COMPARISON ROUTINE *
037F 112 ;***********************************
037F 113 ;
037F A40C 114 COMPAR LDY $0C ; ROUTINE TO COMPARE
0381 B10A 115 COMPY LDA (STRL),Y ; REMAINING CHARACTERS
0383 D106 116 CMP (BSL),Y ; (C) LENGTH OF CHARACTER
0385 F003 117 BEQ COMPX ; STRING -1
0387 A000 118 LDY $00 ; RESET YREG
0389 60 119 RTS
038A 68 120 COMPX DEY
038B 10F4 121 BPL COMPY ; FOUND A MATCH! POP STACK ADDRESS
038D 68 122 PLA ; AND RETURN TO BASIC. POP ADDRESS
038E 68 123 PLA ; IS ALREADY IN LNL AND LNH.
038F 60 124 RTS
125 END

*******************************************
*  * SYMBOL TABLE -- V 1.5  *
*  *
*******************************************

LABEL. LOC.  LABEL. LOC.  LABEL. LOC.

** ZERO PAGE VARIABLES:
HILO 0003 HIII 0004 BSL 0006 BSH 0007 SEAL 0008 SEAL 0009
STRL 000A LNL 0011 LNH 0012

** ABSOLUTE VARIABLES/LABELS:
START 0300 LENGTH 0310 TTOKEN 0336
INCTOK 033F LIMIT 035F INCPTN 0364 INCTOK 0370 REXIT 037E COMPAR 037F
COMPY 0381 COMPX 038A
SYMBOL TABLE STARTING ADDRESS:6000
SYMBOL TABLE LENGTH:0002
I/O ENHANCEMENTS

Introduction

A Little Plus for Your Apple II
Craig Peterson

Zoom and Squeeze
Gary B. Little

A Slow List for Apple BASIC
R.B. Sander-Cederlof

Alarming Apple
Paul Irwin
Introduction

The keyboard and video display of the Apple represent the heart of the user interface. Through this interface, most communication between the programmer and the computer takes place. The following chapter offers some enhancements to the interface which should make working with your Apple a little bit easier.

"Edit-Plus" by Craig Peterson, gives all the advantages of the Apple II Plus screen editing system to Apple II standard machines. "Zoom and Squeeze" by Gary Little provides some further enhancements to the screen editing system which should speed up program modification sessions considerably.

"Slow-List" by Bob Sander-Cederlof, lets you control the speed at which integer BASIC output is sent to the screen. And "Alarming Apple" by Paul Irwin transfers the Apple's speaker into an audio error-alarm from Applesoft BASIC. The four programs—all in machine language—each demonstrate good interface techniques and should be of particular interest to machine language programmers.
A Little Plus for Your Apple II

by Craig Peterson

The Apple II Plus with its autostart ROM provides several useful editing features not found on the standard Apple II. If you own an Apple II and are envious of these new features, try EDITPLUS; it provides you with these features—and more—at no cost!

A while back, Apple Computer, Inc., came out with a new version of their Apple II computer called the Apple II Plus. In this new machine comes the now famous Auto-Start ROM, and one of its neat features is a very much improved editing capability. In particular, for the standard Apple II owner, the 'non-copy' movement of the cursor requires two keystrokes for each column or row moved. (e.g. 'ESC', 'D', 'ESC', 'D', etc., etc.) Very tedious, and sometimes a bit unreliable.

On an Apple II Plus you just press 'ESC' and then use the I, J, K, or M key for cursor control up, left, right, and down respectively. And for really great action, you can use the repeat key along with the I, J, K, M to speed the cursor 'non-copying' to any position on the screen. To get out of this editing mode, you just press any key other than the I, J, K or M key. This last key will be handled like a normal escape function and then you will be out of the special editor. Really nice, huh?

A second feature of the Apple II Plus is the ability to stop program listings. By pressing 'Ctrl-S' during a listing, that blur of characters will be stopped so you can read the program. Pressing any key will begin the listing again right where it stopped. This works in both Integer BASIC and Applesoft. It even works in the Monitor to stop a trace if you wish. In Applesoft, if the second key pressed is a 'Ctrl-C', the listing will be aborted—just as you would expect normally.

Screen Editing

If you would like to be able to do this on your standard Apple II, you can either purchase the Auto-Start ROM, which has this and other features, or you can use the EDITPLUS program.
The EDITPLUS program is not very large and the way it works is fairly simple. Typing 'Call 768' revises the input and output hooks so that any I/O will be sent through EDITPLUS. The editing portion of the program, through the input hook, just looks for an 'ESC' character. If found, the program then checks the next character to see if it is an I, J, K, or M. If it is, the proper cursor action is performed and the next character is checked to see if it is an I, J, K, or M, and so on. The first non-IJKM character causes the program to do a normal escape function and then exit this mode. To totally disengage from this feature of EDITPLUS, just type 'IN#0', which restores the normal input hook address.

Control-S Feature

The control S feature of EDITPLUS uses the output hook. During any output, the program checks the keyboard strobe and if a 'Ctrl-S' has been pressed, the output is stopped after the next carriage return. The EDITPLUS waits until a key is pressed again and at that time the output continues. If the second key is a 'Ctrl-C', the keyboard strobe is left on so that Applesoft will see the 'Ctrl-C' and abort the listing. To totally disengage from this feature of EDITPLUS, just type 'PR#0', which restores the normal output hook address.

An additional feature which I've added to all of this is escape L. By typing 'ESC' 'L', you leave whatever BASIC you are in and jump to the Monitor, which is much quicker and easier than typing Call-151 all the time.

Program Explanation

The assembly program listing for EDITPLUS is fairly self-explanatory. This example is assembled at good old page 3, hex address $300, but it could be anywhere you want. Also, this example is set up for use with 3.2 DOS on a 48K system. If you have 3.1 DOS and 48K memory, use DOS addresses $A7AD and $A99E in place of $A851 and $AASB in lines 200, 210, 400, 640, and 690. If you have less than 48K, adjust these addresses downward a commensurate amount. Also, 3.1 DOS is peculiar in that it won't allow you to BRUN EDITPLUS right off the disk. You must BLOAD it, and then Call 768. If you don't have a disk system, simply change line 400 to RTS and delete lines 640, 680, and 690. If this change is made, it will be necessary to reassemble the program, or pad the revised lines with NOPs ($EA), because the branch addresses will change.

So there you have it—a nice edit program for your Apple II. No longer do you need to be jealous of those folks that have an Apple II Plus. You too can have fun editing (and TRACE and STEP too, heh! heh!).

Update

Since the original publication of the EDITPLUS program, I've added a few more features. Here is the disassembled listing of a slightly more enhanced "EDITPLUS2." All of the original features are the same and I've included the following additions:

1) 'ESC' 'H' will clear and home the screen.
2) 'ESC' 'P' will perform a POKE 33,33 to change the screen width to 33 columns for easier editing of literals (string values inside of quote marks).

3) 'ESC' 'N' returns the screen width to a normal 40 columns.

The program is set up for use with 48K RAM and 3.2 DOS. To revise it for other configurations, consult the article for necessary changes.

It sure makes editing easier, and it works in Integer BASIC, Applesoft and in the Monitor. Also, the enhancements make it valuable for owners of Apple II Plus computers.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>;***************</td>
<td>EDIT PLUS</td>
</tr>
<tr>
<td>0800</td>
<td>2 ;*</td>
<td>CRAIG PETERSON</td>
</tr>
<tr>
<td>0800</td>
<td>3 ;*</td>
<td>EDIT PLUS</td>
</tr>
<tr>
<td>0800</td>
<td>4 ;*</td>
<td>COPYRIGHT (C) 1981</td>
</tr>
<tr>
<td>0800</td>
<td>5 ;*</td>
<td>MICRO INK, INC.</td>
</tr>
<tr>
<td>0800</td>
<td>6 ;*</td>
<td>ALL RIGHTS RESERVED</td>
</tr>
<tr>
<td>0800</td>
<td>7 ;*</td>
<td></td>
</tr>
<tr>
<td>0800</td>
<td>8 ;*</td>
<td>EDIT PLUS</td>
</tr>
<tr>
<td>0800</td>
<td>9 ;*</td>
<td></td>
</tr>
<tr>
<td>0800</td>
<td>10 ;*</td>
<td></td>
</tr>
<tr>
<td>0800</td>
<td>11 ;*</td>
<td></td>
</tr>
<tr>
<td>0800</td>
<td>12 ;*</td>
<td>A PROGRAM TO GIVE THE STANDARD</td>
</tr>
<tr>
<td>0800</td>
<td>13 ;*</td>
<td>APPLE II THE ENHANCED CURSOR</td>
</tr>
<tr>
<td>0800</td>
<td>14 ;*</td>
<td>EDITING CAPABILITIES OF THE</td>
</tr>
<tr>
<td>0800</td>
<td>15 ;*</td>
<td>APPLE II PLUS</td>
</tr>
<tr>
<td>0800</td>
<td>16 ;*</td>
<td></td>
</tr>
<tr>
<td>0800</td>
<td>17 ;*</td>
<td></td>
</tr>
<tr>
<td>0800</td>
<td>18 ;*</td>
<td></td>
</tr>
<tr>
<td>0800</td>
<td>19</td>
<td>;***************</td>
</tr>
<tr>
<td>0800</td>
<td>20</td>
<td>EPZ</td>
</tr>
<tr>
<td>0800</td>
<td>21</td>
<td>EPZ</td>
</tr>
<tr>
<td>0800</td>
<td>22</td>
<td>EPZ</td>
</tr>
<tr>
<td>0800</td>
<td>23</td>
<td>EPZ</td>
</tr>
<tr>
<td>0800</td>
<td>24</td>
<td>EPZ</td>
</tr>
<tr>
<td>0800</td>
<td>25</td>
<td>EPZ</td>
</tr>
<tr>
<td>0800</td>
<td>26</td>
<td>EPZ</td>
</tr>
<tr>
<td>0800</td>
<td>27</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>28</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>29</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>30</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>31</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>32</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>33</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>34</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>35</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>36</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>37</td>
<td>EQU</td>
</tr>
<tr>
<td>0800</td>
<td>38</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>39</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>40</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>41</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>42</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>43</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>44</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>45</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>46</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>47</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>48</td>
<td>OBJ</td>
</tr>
<tr>
<td>0800</td>
<td>49</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>50</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>51</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>52</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>53</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>54</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>55</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>56</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>57</td>
<td>JMP</td>
</tr>
<tr>
<td>0800</td>
<td>58</td>
<td>JMP</td>
</tr>
</tbody>
</table>

;CHANGE DOS POINTERS & RETURN
;GET NEXT CHARACTER
;IS CHARACTER = 'ESC'?
;IF SO, GO TO ESC2
;IF NOT, RETURN
;PREPARE A POINTER AND
;TURN I,J,K, AND M
;INTO A,B,C, AND D,
;RESPECTIVELY...
;DO STANDARD ESCAPE
0325 A24 59 ESC2 LDY CH ;GET THE NEXT INPUT
0327 B128 60 LDA (BASL),Y ; CHARACTER.....
0329 46 PHA
032A 293F 62 AND $0F
032C 0940 63 ORA $40
032E 912A 64 STA (BASL),Y
0330 6B 65 PLA
0331 201BF D JSR 201BFD
0334 C9C8 67 CMP $C8 ;IS CHAR = 'H'? 0
0336 D002 68 BNE SKIP ;IF NOT=H, SKIP
0338 A9C0 69 HOME LDA $C0 ;IF IS, LOAD 'E'
033A C9D0 70 SKIP CMP $D0 ;IS CHAR = '$'? 1
033C F038 71 BEQ POKE ;IF IS, --> POKE
033E C9CE 72 CMP $CE ;IS CHARACTER > 'N'? 2
0340 F010 73 BEQ NRML ;IF='N', GOTO NRML
0342 B019 74 BCS RTRN ;IF>'N', GOTO RTRN
0344 C9C9 75 CMP $C9 ;IS CHARACTER < '1'? 3
0346 9015 76 BCC RTRN ; IF SO, RETURN.
0348 C9CC 77 CMP $CC ;IS CHARACTER = 'L'? 4
034A D0CF 78 BNE SPECL ; IF <> 'L', DO SPECL.... 5
034C 2051A8 79 JSR DOS ; IF = 'L', RESET DOS
034F 4656FF 80 JMP MNTR ; POINTERS AND JUMP TO MNTR.
0352 A92D 81 NRM LDA #28 ;NORMAL SCREEN
0354 D005 82 BNE COUNT ; WIDTH=$28 (40)
0356 201FDC ; POKE JSR CEOP & ;CLEAR TO EOP &
0359 A921 84 LDA #21 ; POKE 13,13
035B 8521 85 CONT STA WNWD ;STORE->WINWIDTH
035D 38 86 RTRN SEC ;CHAR. IS NOT I,J,K, OR M,
035E 202FC8 87 JSR ESC1 ; SO DO A STANDARD ESC.
0361 A244 88 LDY CH ;CORRECT YSAVE REGISTER IN
0363 BCSBAA 89 STY YDOS ; DOS AND RETURN...
0366 400FFD 90 JMP RKEY
0369 8435 91 SVID STY YSAV ;SAVE Y.
036B C9BD 92 CMP $BD ;IS CHARACTER A CR?
036D D018 93 BNE RETN ;IF NOT, RETURN.
036F AC0C0 94 LDY BRKD ;GET KEYBOARD CHARACTER.
0372 1013 95 BPL RETN ;NO STROBE, RETURN.
0374 C093 96 CPY $93 ;IS IT CTRL 'S'?
0376 D00F 97 BNE RETN ;IF NOT, RETURN.
0378 2C1OC0 98 BIT STRB ;CLEAR KEYBOARD STROBE
037A AC0C0 99 AGIN LDY BRKD ;IS KEY PRESSED?
037C 0F6 100 BPL AGIN ;IF NOT, TRY AGAIN!
0380 C0B 101 CPY $B3 ;IS IT CTRL 'C'?
0382 F003 102 BEQ RETN ;IF SO, LEAVE STROBE
0384 2C1OC0 103 BIT STRB ;CLEAR KEYBOARD STROBE
0387 A435 104 RETN LDY YSAV ;RESTORE Y AND REJOIN
0389 40FFD 105 JMP OUT1 ; OUTPUT....
038C 106 ;
038C C4C2C1 ;107 TABL HEX C4C2C1FFC3
038F FFC3 108 END

******************************************************************************
* *  SYMBOL TABLE -- V 1.5  *
* ******************************************************************************

LABEL. LOC. LABEL. LOC. LABEL. LOC.

** ZERO PAGE VARIABLES:

WNWD 0021 CH 0024 BASL 0028 YSAV 0035 CSWL 0036 CSWH 0037
KSWL 0038 KSWH 0039

** ABSOLUTE VARIABLES/LABELS

DOS A851 YDOS AA5B BRKD C000 STRB C010
ESCl FC2C CEOP FC42 KEY FD0C OUT1 FDF0 KEYN FD1B MNTR FF65
BGIN 0300 SKEY 0313 SPCL 031B ESC2 0325 HOME 0338 SKIP 033A
NRML 0352 POKE 0356 CONT 035B RTRN 035D SVID 0369 AGIN 037B
RETN 0387 TABL 038C

SYMBOL TABLE STARTING ADDRESS:6000
SYMBOL TABLE LENGTH:0112
Zoom And Squeeze

by Gary B. Little

A short program for the Apple II that makes it easier to edit BASIC programs. ZOOM provides a fast way to copy over a program line; SQUEEZE changes the screen width to 33 characters and eliminates embedded blanks.

ZOOM and SQUEEZE is a short machine-language routine written for the Apple microcomputer to facilitate the editing of BASIC programs. It recognizes two commands: CTRL-Q and CTRL-Z. The CTRL-Q command causes the screen window width to be automatically set to 33 and the CTRL-Z command causes the cursor to quickly copy over all text from its current position to the end of the line.

The ZOOM Feature

In order to edit a program line on the Apple it is necessary to do more than simply move the cursor directly to the area to be changed, make the changes, and then press RETURN. The required procedure is to position the cursor at the beginning of the line number, copy down to the area to be changed (by using the right-arrow and repeat keys), make the changes, enter the edited line. If the line is a very long one, the copying-over part of this procedure takes up an enormous amount of time, which can be better used for other purposes.

The 'ZOOM' part of the ZOOM and SQUEEZE routine can be used to speed up this copying tremendously. By simply pressing CTRL-Z the cursor can be moved virtually instantaneously from its current position to the right edge of the current line, while automatically copying over all the text on the screen in between. For example, copying over a program line that takes up three lines on the video screen takes only six quick steps after the cursor has been positioned at the beginning of the line number: CTRL-Z, right-arrow, CTRL-Z, right-arrow, CTRL-Z, RETURN. This takes approximately 2 seconds to accomplish. By the way of contrast, copying over the line in the ordinary way by using the right-arrow key in conjunction with the repeat key takes approximately 13 seconds (see the NOTE).

It is clear, then, that this feature could save hours of debugging time for a busy programmer.
The SQUEEZE Feature

When a line of a BASIC program is listed on the video screen with the window width set at its default value of 40 columns, the output is carefully formatted by the Apple by embedding blanks on the left and right side of the listing. That is to say, there is not a continuous 'wraparound' display of the information that you typed in to create the line. For example, if you enter the line

```
100 PRINT "THIS IS AN EXAMPLE OF A FORMATTED LISTING"
```

and then LIST it, the Apple will respond with

```
100 PRINT "THIS IS AN EXAMPLE OF A FORMATTED LISTING"
```

where a '*' indicates an embedded blank. This formatting technique makes it very easy to read a LISTed line, but it can create a minor problem when it becomes necessary to edit the line.

The problem arises when, as in the example, the blanks are embedded between the quotation marks associated with a PRINT statement. If this line is to be edited without retyping it from scratch, the right-arrow key (in conjunction with the repeat key) must be used to copy over substantial portions of the line and by so doing all 6 of the embedded blanks between 'F' and 'FORMATTED' will mysteriously appear in the argument of the PRINT statement unless they are skipped over by performing pure-cursor movements, that is, repeated ESC-A commands or, for AUTOSTART ROM users, repeated K commands after ESC has been pressed. The need to perform these pure-cursor movements is annoying and inconvenient, to say the least.

This problem can be avoided if the window width is 'squeezed' to 33 columns before LISTing the line and editing it. If this is done, the embedded blanks disappear and the line can be edited without worrying about the need to perform pure-cursor movements.

The window width can be changed to 33 by entering the command POKE 33,33 from BASIC immediate-execution mode. However, with the ZOOM and SQUEEZE routine in effect, all you need to do is press CTRL-Q. The width can be returned to its default value of 40 by simply entering the command TEXT from immediate-execution mode.

How ZOOM and SQUEEZE Works

ZOOM and SQUEEZE can be activated by BRUNning it from disk or by loading it, entering the command 300G from the monitor, and then returning to BASIC. The routine resides from $300 to $33A.

After it has been activated, the Apple's input hook at $38 (low), $39 (high) is set equal to the ZOOM and SQUEEZE entry point at $309. Thereafter, all keyboard input is checked to see whether CTRL-Q or CTRL-Z has been pressed; if not, then nothing special happens.
If CTRL-Q is pressed, the short subroutine beginning at $310 and ending at $316 is executed. All this subroutine does is store $21 (decimal 33) at location $21. This is the location in the monitor that contains the current window width. A blank is then displayed on the screen to indicate that this has occurred.

If CTRL-Z is pressed, the subroutine beginning at $317 is executed. What happens then is that the characters displayed on the screen from the current cursor position to the end of the line are placed in the input buffer one-by-one. If the buffer is overflowing, the program line will be backslashed and cancelled in the ordinary way.

Details of the programming algorithms involved can be easily deduced by inspecting the accompanying source listing for ZOOM and SQUEEZE.

NOTE: It is possible to speed up the repeat-key function by soldering a 100K resistor in parallel to the resistor at position R4 on the Apple keyboard unit. For details, see the article "Repeat Key Speed-Up" by V.R. Little in the February 1980 edition of APPLEGRAM, the newsletter of the Apples British Columbia Computer Society, Vancouver, B.C.
**I/O Enhancements**

```
0309 35 ;*******************
0309 36 ;ZOOM/SQUEEZE SUBS*
0309 37 ;*******************
0309 38 ;
0309 201BFD 39 INHK JSR KEYIN ;GET A CHARACTER
030C C991 40 CMP #$91 ;CTRL-Q PRESSED?
030E D007 41 BNE CTRLZ ;IF NOT, CHECK FOR CTRL-Z
0310 A921 42 LDA #$21 ;CHANGE WINDOW WIDTH
0312 8521 43 STA WIDTH ;TO 31
0314 A9A0 44 LDA #$A0 ;OUTPUT A SPACE
0316 60 45 RTS
0317 46 ;
0317 C99A 47 CTRLZ CMP #$9A ;CTRL-Z PRESSED?
0319 D01F 48 BNE RTS1 ;IF NOT, RETURN
031B A424 49 LOOP LDY CH ;TAKE A CHARACTER
031D B128 50 LDA (BASL),Y ;OFF VIDEO SCREEN
031F 48 51 PHA
0320 E624 52 INC CH
0322 E624 53 INC CH
0324 A524 54 LDA CH ;IF CURSOR POSITION IS
0326 C521 55 CMP WIDTH ;AT FAR RIGHT,
0328 B00B 56 BCS FIN ;THEN FINISHED
032A C624 57 DEC CH
032C 68 58 PLA ;STORE CHARACTER
032D 9D0002 59 STA IN,X ;IN INPUT BUFFER
0330 68 60 INX
0331 D0E8 61 BNE LOOP ;GET ANOTHER CHARACTER OFF SCREEN
0333 CA 62 DEX ;BUFFER FULL
0334 60 63 RTS ;SO RETURN
0335 68 64 FIN PLA
0336 C624 65 DEC CH ;SET PROPER CHARACTER
0338 C624 66 DEC CH ;POSITION AND
033A 60 67 RTS RTS ;RETURN
68 END

***** END OF ASSEMBLY

**************************
*\* SYMBOL TABLE -- V 1.5 *\*
*\*"
A Slow List
for Apple BASIC

by R.B. Sander-Cederlof

The speed at which listings are produced can be controlled in Applesoft via the speed command. But unfortunately no similar control exists for Integer BASIC. Slow list takes care of this problem, and provides several other useful control features.

One of the nicest things about Apple BASIC is its speed. It runs circles around most other hobby systems! Yet there are times when I honestly wish it were a little slower.

Have you ever typed in a huge program, and then wanted to review it for errors? You type "LIST", and the whole thing flashes past your eyes in a few seconds! That's no good, so you list in piecemeal—painfully typing in a long series like:

    List 0,99
    List 100,250
    .
    .
    LIST 21250,21399

As the reviewing and editing process continues, you have to type these over and over and over... Ouch!

In a meeting of the Dallas area "Apple Corps" several members expressed the desire to be able to list long programs slowly enough to read, without the extra effort of typing separate commands for each screen-full. One member suggested appending the series of LIST commands to the program itself, with a subroutine to wait for a carriage return before proceeding from one screen-full to the next. For example:
34 I/O Enhancements

9000 LIST 0,99:GOSUB 9500
9010 LIST 100,250: GOSUB 9500
.
.
9250 LIST 21250,21399:GOSUB 9500
9260 END
9500 INPUT A$:RETURN

While this method will indeed work, it is time-consuming to figure out what line ranges to use in each LIST command. It is also necessary to keep them up-to-date after adding new lines or deleting old ones.

The Slow List Program

But there is a better way! I wrote a small machine language program which solves the problem. After this little 64-byte routine is loaded and activated the LIST command has all the features needed:

1. The listing proceeds at a more leisurely pace, allowing you to see what is going by.
2. The listing can be stopped temporarily, by merely pressing the space bar. When you are ready, pressing the space bar a second time will cause the listing to resume.
3. The listing can be aborted before it is finished by typing a carriage return.

The routine as it is now coded resides in page three of memory, from $0340 to $037F. It is loaded in the usual way (BLOAD Slow List).

After the routine is loaded, you return to BASIC. The slow-list features are activated by typing "CALL 887". They may be de-activated by typing "CALL 878" or by hitting the RESET key.

How It Works

The commented assembly listing should be self-explanatory, with the exception of the tie-in to the Apple firmware. All character output in the Apple funnels through the same subroutine: COUT, at location $FDED. The instruction at $FDED is JMP [$0036]. This means that the address which is stored in locations $0036 and $0037 indicates where the character output subroutine really is.

Every time you hit the RESET key, the firmware monitor sets up those two locations to point to $FDF0, which is where the rest of the COUT subroutine is located. If characters are supposed to go to some other peripheral device, you would patch in the address of your device handler at these same two locations. In the case of the slow-list program, the activation routine merely patches locations $0036 and $0037 to point to $0340. The de-activation routine makes them point to $FDF0 again.
Every time slow-list detects a carriage return being output, it calls a delay subroutine in the firmware at $FCA8. This slows down the listing. Slow-list also keeps looking at the keyboard strobe, to see if you have typed a space or a carriage return. If you have typed a carriage return, slow-list stops the listing and jumps back into BASIC at the soft entry point ($E003). If you have typed a space, slow-list goes into a loop waiting for you to type another character before resuming the listing.

That is all there is to it! Now go turn on your Apple, type in the slow-list program, and list to your heart's content!

```
0800 1 ;******************************
0800 2 ;
0800 3 ;* SLOW LIST *
0800 4 ;* B.SANDER-CEDERLOF *
0800 5 ;*
0800 6 ;* SLOW LIST *
0800 7 ;*
0800 8 ;* COPYRIGHT(C) 1978 *
0800 9 ;* MICRO INK, INC. *
0800 10 ;*ALL RIGHTS RESERVED*
0800 11 ;*
0800 12 ;* ROUTINE TO SLOW *
0800 13 ;* DOWN INT. BASIC *
0800 14 ;* LISTINGS......... *
0800 15 ;*
0800 16 ;******************************
0800 17 ;
0800 18 ;
0800 19 ;
0800 C9BD 20 ORG $340 ;MAIN PROGRAM
0800 C00 21 OBJ $800
0800 C04 22 ;
0800 C98D 23 SLOW CMP $8D ;CHECK IF CHAR IS CARRIAGE RETURN
0800 23A1 24 BNE CHROUT ;NO, SO GO BACK TO COUT
0800 48 25 PHA ;SAVE CHARACTER ON STACK
0800 25F C0C0 26 BIT $CO00 ;TEST KEYBOARD STROBE
0800 100E 27 BPL WAIT ;NOTHING TYPED YET
0800 ADD0C0 28 LDA $C000 ;GET CHARACTER FROM KEYBOARD
0800 2C10C0 29 BIT $C010 ;CLEAR KEYBOARD STROBE
0800 28A0 30 CMP $S00 ;CHECK IF CHAR IS A SPACE
0800 2F01 31 BEQ STOP ;YES - STOP LISTING
0800 C98D 32 CMP $8D ;CHECK IF CHAR IS CARRIAGE RETURN
0800 F009 33 BEQ ABORT ;YES - ABORT LISTING
0800 A900 34 WAIT LDA $500 ;MAKE A LONG DELAY
0800 20A8FC 35 JSR $FCA8 ;CALL MONITOR DELAY SUBROUTINE
0800 68 36 PLA ;GET CHARACTER FROM STACK
0800 4CF0FD 37 CHROUT JMP $FDF0 ;REJOIN COUT SUBROUTINE
0800 4C03E0 38 ABORT JMP $E003 ;SHORT ENTRY INTO APPLE BASIC
0800 2C00C0 39 STOP BIT $C000 ;WAIT UNTIL KEYBOARD STROBE
0800 10FB 40 BPL STOP ;APPEARS ON THE SCENE
0800 8D10C0 41 STA $C010 ;CLEAR THE STROBE
0800 30EA 42 BMI WAIT ;UNCONDITIONAL BRANCH
0800 43 ;
0800 44 ;******************************
0800 45 ;*
0800 46 ;* SUBROUTINE TO DE- *
0800 47 ;* ACTIVATE SLOW LIST *
0800 48 ;******************************
0800 A9F0 49 OFF LDA $F0 ;RESTORE $FDF0 TO
0800 B53E 50 STA $36 ;LOCATIONS $36 AND $37
0800 A9FD 51 LDA $FDF
0800 B537 52 STA $37
0800 60 53 RTS
0800 54 ;
```
0377  55  ;***********************
0377  56  ;* SUBROUTINE TO *
0377  57  ;* ACTIVATE SLOW LIST *
0377  58  ;***********************
0377  59  ;
0377  A940  60 ON  LDA $540  ;SET $340 INTO
0378  8536  61 STA $36  ;LOCATIONS $36 AND $37
0378  A903  62 LDA $503
0379  8537  63 STA $37
037F  60  64 RTS
0380  65  66 END

***** END OF ASSEMBLY

*************************************************************
*     SYMBOL TABLE -- V 1.5 *
*************************************************************

** ZERO PAGE VARIABLES:

** ABSOLUTE VARIABLES/LABELS

SLOW  0340  WAIT  0358  CHROUT  035E  ABORT  0361  STOP  0364  OFF  036E
      0377

SYMBOL TABLE STARTING ADDRESS:6000
SYMBOL TABLE LENGTH:004A

Ed. note: To make compatible with DOS, change locations 36 and 37 (COUT Hooks) to locations AA53 and AA54 (DOSCOUT Hooks). This has been done on your disk.
Alarming Apple

by Paul Irwin

Have you ever wanted to trap your errors in a way so complete and so foolproof that it would be absolutely impossible to miss them? Well here it is—the Alarming Apple routine! Using this routine, the Apple will respond to any program error with a keyboard lockout and a two-tone alarm. This is a great enhancement to the Apple's error recovery capabilities!

Instead of using the CTRL-G beep on your next program, here's an alarm system written to assist in performing error recovery on the Apple II. When the alarm system is used, your program will react to an error by immediately locking the keyboard, sounding a continuous two-tone alarm, and forcing the operator's attention to an error recovery subroutine. No way will recognizable errors escape your edits once they meet the Alarming Apple!

To use the alarm system, start with each of your subroutines clearly defined as either error detecting or error correcting. This means that you will classify most of your "normal" routines as error detecting routines. Arrange to have all of your routines invoked by a mainline. Then the mainline can invoke error correcting routines, as well, and still remain in control. This is illustrated by the program.

In the BASIC listing, the one error detecting routine is called TASK, while the error correcting routine is TRAP. The mainline is free to decide what to do after recovery: whether to continue the same error detecting routine or to take any other action. An intelligent mainline of this sort can avoid most error recovery hassles.

The key to the error recovery procedure is a machine language routine called ALARM. It is invoked from BASIC by executing a CALL 3529 and from machine language by executing a JSR $DC9. The alarm routine will then generate a two-tone alarm continuously. At the end of each cycle, it examines the keyboard for a CTRL-C. If none was found, it continues sounding the alarm. But when a CTRL-C is typed, the sound will stop and the routine will return. The effect is to produce a continuous sound, ignoring any input, until a CTRL-C is entered.

You may have your own ideas as to how the alarm should sound. The duration of the first tone is in $DA2 and its period is in $D9D. The second tone has its
I/O Enhancements

duration and pitch stored in $DBF and $DBA. The two that I employ are quite noisy, but you can experiment with other parameter pairs. Those periods that are relatively prime—having no common factor—will produce discord. They will be loudest when matching the Apple's speaker resonance.

When loading the routines, remember to set LOMEM greater than $D6D0, the highest location in the alarm routine, so the two won't overwrite each other. The BASIC routine shown here will run as it appears, and will invoke the machine language routine. If you are not bothering with the BASIC, simply JSR $DC9.

After you run the Alarming Apple and decide to use it for error recovery in your next program, consider these ideas:

Organize the program into error detecting routines, one or more error recovery routines, and an intelligent mainline.

Use an error flag in the recovery routines to inform the mainline.

Use a status flag in the error recovery routines to indicate success or failure of the recovery procedure to the mainline.

Let the mainline make all decisions regarding what to do next.

For instance, if you are heavily into structured programming, you might consider a mainline centered on a computed GOSUB with the returns of each routine setting a status number pointing to the next routine. Or you may want to use IFs and GOSUBs together in the mainline as each case is decided. The important thing is to route all control decisions—decisions that answer the question: "What next?"—through the mainline, including error recovery decisions; in fact, especially error recovery decisions.
REM ****************************
REM * SAMPLE BASIC CALL SEQUENCE *
REM * FOR ALARM PROMPT ROUTINE *
REM ****************************
REM TASK~3000:0FF~O:TASK~200:TRAP=300:ALARM=3529
REM ********************
REM MAIN LINE SEQUENCE
REM ********************
ERR=OFF:
GOSUB TASK: IF ERR THEN GOSUB TRAP
GOTO 32767: REM ***
REM INPUT ERR: REM *** USE FOR TEST ***
210 REM
211 REM
212 REM PUT ERROR DETECTING
213 REM HERE --- REPLACING
214 REM LINE 210........
215 REM
220 RETURN
299 REM
300 POKE 50,127: PRINT "ERROR": POKE 50,255: PRINT " TYPE A CTRL/C": CALL ALARM
320 REM
321 REM PUT ERROR RECOVERY
322 REM ROUTINE HERE......
323 REM
340 RETURN
RUNTIME UTILITIES

Introduction 42

Data Statement Generator 43
Virginia Lee Brady

An Edit Mask Routine in Applesoft BASIC 47
Lee Reynolds

Business Dollars and Sense in Applesoft 55
Barton M. Bauers, Jr.

Lower Case and Punctuation in Applesoft 62
James D. Childress
Introduction

Runtime utilities assist the actual execution of a program. This chapter contains four articles, each containing an Applesoft BASIC runtime utility program. "Data Statement Generator" by Virginia Lee Brady presents an interesting discussion of the Applesoft data statement, and provides a method for generating these statements from within your Applesoft program. The "Edit Mask" article by Lee Reynolds discusses the benefits of the formatting "masks" found in many high-level computer languages—and then shows how these can be implemented on the Apple. "Business Dollars and Sense" by Barton Bauers addresses the issue of rounding and formatting problems in business programs and posts an interesting solution to these problems. And "Lower Case and Punctuation in Applesoft" provides a simple and easy-to-use method of inputting lowercase letters and reserved punctuation marks into a BASIC character string. The routines presented in this chapter should find considerable use in practical programming applications.
Data Statement Generator
by Virginia Lee Brady

The BASIC "DATA" statement is extremely useful for storing data tables within a program. If the data is generated by a program, however, the usefulness of the Applesoft "DATA" statement declines significantly—unless you have a method to directly generate data statements under program control. This data statement generator provides such a method.

I had just finished adding several new data statements to a sewing program of mine that utilized a number of data statements, and now I was reading the information into their respective arrays. "BEEP," said the Apple, "**SYNTAX ERROR." I found the offending line; I'd left out one of the elements and Applesoft would not accept "RED" as a value for "YARDS." I entered the line again and this time I typed the wrong line number and erased my previous line. There ought to be a way, I decided, to let the Apple keep track of these things. I experimented with input statements, and while these allowed me to update the arrays, I couldn't save the information.

Using the information from Jim Butterfield's article on "PET BASIC" (MICRO:7) and the information in the Applesoft Manual, I developed a program that "writes" its own data statements. This routine automatically increments the line numbers and inputs the data elements in response to appropriate prompts. It's all POKEd into place and becomes a permanent part of the program.

How Applesoft is Stored

The BASIC program begins at $801 (2049 decimal) and there are only two bytes between the end of the program and the start of the simple variable table which begins at LOMEM:. Anytime a BASIC line is entered, altered, or deleted, the value of LOMEM: is changed and the program must be rerun to incorporate this new value. Therefore, LOMEM: must be set at some value past the end of the program to allow for expansion of the program without writing on top of the variable table.

To use this routine it is also necessary to recognize the following locations of a data statement in Applesoft:
2 bytes—pointer to next line of BASIC (to next pointer)
2 bytes—hex equivalent of the line number
1 byte—"83"—token for "DATA"
N bytes—ASCII equivalents of the program line
1 byte—"00"—indicates the end of the line

Then the sequence starts again until there are two bytes of "00" in the first two positions (total of three "00" bytes in a row.)

The Program

The program uses the fact that the locations $AF.0B (175-176 decimal) hold the value of the location where the next line number would go, or put another way, two less than this is where the "pointer to next line" would go. Call this PSN (for position). Thus the values to be POKEd into PSN and PSN + 1 are the low and high order bytes of the hex equivalent of LINE number. Then the DATA token (131 in decimal) is placed in PSN + 2. Since this program was designed to handle several elements in one data statement, a series of strings is next input as one string array. (It could just as easily have been done as several "INPUT A$"'s, but using an array allows you to change a string before it is POKEd into memory.) This is handled in lines 1035-1045. If there are no further changes, then the individual strings are concatenated into one long string with commas separating the individual substrings. Next this string is POKEd, one ASCII value at a time, into PSN + I + 2; then the "0" is POKEd into the end as the terminator.

Since PSN + I + 3 is the start of the next line (remember the value of I was incremented one extra time in the FOR-NEXT loop), call this NUMBER, convert it into hex, and POKE it into PSN-2 and PSN-1. If the program is to be continued, PSN is given the value of NUMBER + 2 and the sequence restarted. If this is to be the last entry, then place "0" into NUMBER and NUMBER + 1. All that remains is to reset the $AF.0B pointers to reflect the new value of the end of the program {NUMBER + 2). This is done in line 1085.

List the program—the new data statement is in place at the end of the program and can be read into the necessary string of numeric variables. If you want to use this program as a subroutine to an existing data program, where you already have some data statements being read in, you could use the fact that $7B.7C gives the line from which data is being read. Then insert a statement that sets LINE equal to PEEK(123) + PEEK(124)*256.

If your program uses trailers, then have a TRAILER$ that is the same as your trailer line (eg. "'0,0,0,0'"). To write over this, set PSN equal to PSN-6-LEN(TRAILER$) and your first data statement will start that much earlier and replace this trailer. At the end of the program, handle this as before and POKE the TRAILER$ into place... This way every time you update your program, the original trailer is "erased" and re-appended after the last data statement.
Notes and Cautions

It is important to remember that the line numbers you insert this way must be greater than those of an existing program line. If not, they will be placed at the end of the program, but will not be recognized as legitimate line numbers. (If you try to erase or list it, Applesoft, not finding it between the next lower and next greater line numbers will think it does not exist.) Also, do not try to Control-C out of the program once it has started the "POKEing" portion, since the pointers would be incorrect at this point and Applesoft would not know where to find the end of the program.

<table>
<thead>
<tr>
<th>Original Last Line</th>
<th>First Added Line</th>
<th>New Last Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINT LOW</td>
<td>POINT HIGH</td>
<td>LINE LOW</td>
</tr>
<tr>
<td>POINT HIGH</td>
<td>POINT LOW</td>
<td>LINE HIGH</td>
</tr>
<tr>
<td>&quot;DATA&quot;</td>
<td>&quot;DATA&quot;</td>
<td>&quot;END&quot;</td>
</tr>
<tr>
<td>data</td>
<td>&quot;END&quot;</td>
<td>Orig. End</td>
</tr>
<tr>
<td></td>
<td>Line 00/02</td>
<td>NEXT LOW</td>
</tr>
<tr>
<td></td>
<td>Line 00/02</td>
<td>NEXT HIGH</td>
</tr>
<tr>
<td></td>
<td>Next 00/02</td>
<td>Orig. End</td>
</tr>
<tr>
<td></td>
<td>NEXT HIGH</td>
<td>NEXT LOW</td>
</tr>
<tr>
<td>&quot;END&quot;</td>
<td>&quot;END&quot;</td>
<td>NEXT LOW</td>
</tr>
<tr>
<td></td>
<td>&quot;END&quot;</td>
<td>NEXT HIGH</td>
</tr>
<tr>
<td></td>
<td>&quot;END&quot;</td>
<td>New End</td>
</tr>
</tbody>
</table>

Note: Original Last Line
NEXT LOW/HIGH change from 0000 to 2002.

Figure 1: "MAP" of Two New DATA Statements being Added
50 HOME
60 LOMEM: 4000
70 LINE = 2000
80 GOTO 1000
90 REM CALCULATE HI/LOW BYTES
100 HI = INT (NUMBER / 256): LO = (NUMBER / 256 - HI) * 256: RETURN
1000 REM INPUT SUBSTRINGS
1010 PSN = PEEK (175) + PEEK (176) * 256
1015 INPUT "INPUT THE COLOR "; FS(1)
1016 INPUT "INPUT THE PATTERN "; FS(2)
1017 INPUT "INPUT THE YARDS IN DECIMAL "; FS(3)
1018 INPUT "INPUT THE FABRIC TYPE "; FS(4)
1020 REM ALLOW CHANGES
1035 FOR I = 1 TO 4: PRINT I; TAB(5)FS(I); NEXT I
1040 INPUT "ANY CHANGES "; Y$: IF LEFT$(Y$, 1) = "N" THEN 1050
1045 INPUT "WHICH ONE "; W: PRINT "CHANGE PART "; W; " TO " ; INPUT FS(W)
1050 FS = ": FOR I = 1 TO 3: FS = FS + FS(I) + ": NEXT ; FS = FS + FS(I)
1055 LINE = LINE + 5: NUMBER = LINE: GOSUB 100
1060 POKE PSN, LO: POKE PSN + 1, HI: POKE PSN + 2, 131
1065 FOR I = 1 TO LEN (FS): POKE PSN + I + 2, ASC (MID$(FS, I, I)); NEXT I
1070 POKE PSN + I + 2, 0: NUMBER = PSN + I + 3: GOSUB 100
1075 POKE PSN - 2, LO: POKE PSN - 1, HI
1080 INPUT "ADD MORE "; Y$: IF LEFT$(Y$, 1) = "Y" THEN PSN = NUMBER + 2: GOTO 1015
1085 POKE NUMBER, 0: POKE NUMBER + 1, 0: NUMBER = NUMBER + 2: GOSUB 100: POKE 175, LO: POKE 176, HI
1090 END
An EDIT Mask Routine in Applesoft BASIC

by Lee Reynolds

This article describes some techniques for producing formatted output using edit masks. The programs permit you to produce professional looking output, using a "mask" very similar to those used in many high level business oriented languages.

My work as a professional programmer in business applications has often called for the use of "edit masks", in such languages as COBOL, DIBOL, and the Commercial Subroutine Package of Data General FORTRAN. I have found the edit mask capability in these languages quite useful, so I decided to write a routine in Applesoft BASIC that I could use at home on my Apple II.

The Edit Mask

I should begin by first giving a brief explanation of what an edit mask is, for those readers who have never encountered the term before. An edit mask might be defined as a string of characters which specify operations on a number so as to produce an output string that contains the number's digits re-formatted for printing in certain specific ways. Some of the most common operations that can be carried out on any given number by means of edit masks are the following: (1) "suppressing" of zeroes, by replacing them with blanks in the output string, (2) inserting of a decimal point in a fixed position of the output string, (3) inserting of comma in the string to express thousands, millions, etc., (4) placing a dollar sign before the leftmost digit of the number string, and (5) appending a minus sign to the end of the string if the input number is negative.

The edit mask is used as a sort of "picture" of what the output string should be like after carrying out operations such as the above on the number to be edited. To achieve this, there are definite rules for the edit routine's interpretation of the characters that make up the mask. Perhaps the best way of explaining this is to give some examples of my routine's use.
The Edit Mask Program

The routine itself on the following listing is contained between line numbers 100 to 580. The statements preceding 100 are a "driver" routine you can use to input your edit mask and number to be edited in order to experiment with various types of editing.

The editing routine is called by a GOSUB 100. There are two arguments that must be passed to it: NUM is the number to be edited, and MASK$ is the edit mask string. NUM can contain any number of digits up to 9. I have made no provision for editing numbers that must be expressed in "scientific notation" with an Exponent field.

The result of the masking will be passed back to the calling program in the string OUT$, whose length is the same as MASK$.

There are six special characters which can appear in MASK$ that are treated in a distinctive way: these are the digit 9, the digit 0, the period, the comma, the minus sign, and the dollar sign. The mask can contain other characters also, but more about this later.

Explanation of Mask Characters

The digit 9 is the "numeric replacement" character. This means, wherever a 9 is present in the mask, it will be replaced in the result field (OUT$) by the corresponding digit of NUM, if any, in that position.

Thus, suppose we define MASK$ = "99999", and assume the number to be edited is NUM = 352. Then the result, after calling the edit routine, will be OUT$ = "352". (Note the two blanks preceding the ASCII digit 3. This is because the length of the mask exceeds the length of the number to edit by two.)

Next, the digit 0 is the "zero-suppress" character. This means wherever a 0 appears in the mask, it will be replaced in the result field by the corresponding digit of NUM only if that digit is not a zero; if the digit is a zero, then the corresponding position in the result field will be a blank.

To give an example, suppose MASK$ = "990990" and the number to be edited is NUM = 120563. Then the result will be OUT$ = "12 563". The zero in NUM was suppressed.

The most common usage of the zero-suppress character in a mask is to suppress leading zeroes of a number. Thus a mask of "00099" would suppress the first three digits of any five-digit number if they were zeroes, but would print them if they were not. Due to the way my routine operates, it turns out that leading zeroes are always suppressed, anyway. If you would rather change this feature of the routine, I will describe later how you could go about doing so.

The period in a mask is usually used as the decimal point position. It is what is called an "insertion character" in the mask because it is always inserted in the result field exactly in its corresponding position in the mask.
Let's consider some examples of masks containing a period, and what the result will be. Suppose our mask is "999.99" and our number to be edited is 312.44; then, as you would expect, the result will be OUT$ = "312.44". Next suppose we use the same mask but NUM = 33.6. The result is OUT$ = "33.60". There is a blank in position one and a zero in the last position. [If the last character of the mask had been a 0 instead of a 9, then the last character in the result would have been a blank.] Now, let's suppose that NUM = 124.556. In this case there is one more digit to the right of the decimal point in the number to edit than there is in the decimal part of the mask. When this, or something similar happens, my routine will truncate the extra digit(s), and replace it (them) by an asterisk to signal field overflow. The result then is OUT$ = "124.5".

My routine follows a similar rule in case the number of digits to the left of the decimal point in NUM exceeds the number allowed in MASK$. For example, if NUM = 1256.7, then the result will be OUT$ = "*56.70".

By the way, since it is conceivable that you might, either by mistake or by design, include two or more periods in your mask, the routine will treat only the rightmost period in the mask as the decimal point position. All other periods will be treated as insertion characters, and will appear in the corresponding positions of the result field as expected.

Next, let's consider the comma in an edit mask. An example of a mask containing two commas is the following: MASK$ = "99,999,999". If your number to edit contains either 7 or 8 digits, then the result field will contain both commas in the appropriate places, as you would expect. However, with 6 or fewer digits in NUM, either the first or both commas will be suppressed and replaced by blanks. Examples: if NUM = 1234567, the OUT$ = "1,234,567"; and if NUM = 1234, then OUT$ = "1,234" [note the five blank characters preceding the digit 1]; and lastly, if NUM = 123, then there will appear seven blanks preceding the digit 1: OUT$ = "123".

Thus we see that the comma is a special sort of insertion character which is suppressed if there are no preceding digits of the number to be edited.

Now consider the dollar sign used as an edit mask character. I have defined this character's usage in a special way. If the dollar sign is the very first character in the mask, then it is treated as a "floating dollar sign". That means that the dollar sign in the result field will "float" to the right, far enough to immediately precede the leftmost digit of NUM. Some examples: if MASK$ = "$99,999,999" and NUM = 11.45, then the result of editing is OUT$ = "$11.45" [note that there are four blanks preceding the dollar sign in the result field]. And if NUM = 2321, then we have this result: OUT$ = "$2,321.00" [one blank preceding the dollar sign].

Please note that I have defined this usage of the dollar sign as a "floating" dollar sign only when it is the first character in the mask. If it occurs elsewhere in the mask, then it becomes an insertion character.
The last special usage character in a mask is the trailing minus sign. If the
mask contains a minus sign as the very last character, then the rightmost position
of the result field will be a minus sign when the number to edit is negative, or will
be blank if the number is positive. Examples: if MASK$ = "99,999.99-" and
NUM = -1453.62, then the resultant OUT$ = "1,453,62-". While if NUM =
2246.7, then we have OUT$ = "2,246.70".

If a minus sign appears in a mask in any other position, it is treated as an in-
sertion character. Thus, for example, you could format a date, MMDDYY =
month, day, and year with the following mask: MASK$ = "09-99-99". If NUM =
101479, then OUT$ = "10-14-79".

You might be wondering what will happen if you edit a negative number
using a mask which does not contain a trailing minus sign. It depends upon
whether you have allotted enough digit positions in the mask to accommodate a
leading minus sign. If you have then the minus sign will take the place of the first
position containing a nine, zero, or comma that immediately precedes the left-
most digit of NUM. If you have not allotted enough digit positions in the mask,
then my routine will print the asterisk signaling field overflow.

Now, any character other than the six special cases discussed above may also
appear in a mask. In that case the character becomes an insertion character.
Suppose you define

MASK$ = "$BAL. DUE AS OF SEP/'78: 99,999.99"

If NUM = 1324.57, then the result of masking will be:

OUT$ = "BAL. DUE AS OF SEP/'78: $1,324.57"

From the above example, you can see that you are only restricted in using edit
masks by your imagination, perhaps after making modifications to my routine.
For example, you will note that the year in the above mask is '78 not '79. It could
not be '79 because the 9 is a numeric replacement character and in this case would
have been blanked out. However, if you change the numeric replacement char-
acter to some other more convenient character (perhaps an ampersand?) then this
difficulty could be avoided.

As already mentioned, another modification you might wish to make is to
allow outputting of leading zeroes in a numeric field if the corresponding edit
characters are 9's. To do this, you need to make three changes to the routine.

455 IF L-1 = II AND MID$ (MASK$,L-1,1) = "9" then 480
500 IF N$ = " " THEN N$ = "0"
525 IF N$ = " " THEN 460

When you incorporate this routine into your own programs, you may wish to
change the names of some of the local variables used by it in order not to conflict
with your own use of the same names. Here is a list of all variables used by my
routine.
Variables

- **MASK$**: the string containing the edit mask.
- **NUM**: the input number to edit
- **NUM$**: NUM converted to a string
- **LM**: length of MASK$
- **LN**: length of NUM$
- **PM**: position of rightmost decimal point in MASK$ (or zero if none)
- **PN**: position of decimal point in NUM$ (zero if none)
- **RM**: number of digit positions right of decimal point in MASK$
- **RN**: number of digit positions right of decimal point in NUM$
- **QM**: number of digit positions left of decimal point in MASK$
- **QN**: number of digits left of decimal point in NUM$
- **FD%**: flag telling whether mask has floating dollar sign (1 if yes, 0 if no)
- **MF%**: flag telling whether mask has trailing minus sign (1 if yes, 0 if no)
- **NF%**: flag telling whether NUM is negative (1) or positive (0)
- **M$**: current character of MASK$ being processed
- **N$**: current character of NUM$ being processed
- **I**: loop variable and temporary variable
- **J**: pointer to current digit in NUM$
- **II**: first position in MASK$ to process
- **I2**: last position in MASK$ to process

One final note: in using the driver routine to experiment with various edit masks, you should remember that if your mask will contain commas or colons, then you must enclose the entire mask by quotation marks, or else Applesoft will drop part of your mask when it executes the INPUT statement.

Notes on Converting to Other BASICS

I am not familiar with any other BASICs for microcomputers. I do, however, have some acquaintance with the BASIC languages for two minicomputers—the DEC PDP-11 and the Data General Nova 3. With this as background, I have compiled the following list of possible modifications you might have to make to my routine to get it to work on 6502 machines other than the Apple.

1) Applesoft allows variables to have names with more than two characters, although only the first two are used to distinguish between different names. If your BASIC does not allow this, you will have to change some of the names that my routine uses.

2) Some BASICs don't allow multiple statements per line, or if they do, the statement separator might not be the colon; two common alternatives are the back slash or the exclamation point.

3) If your BASIC does not have the "ON...GO TO" statement, then line number 85 will have to be replaced with something else, perhaps a couple of "IF...THEN GOTO..." statements.
4) Not all BASICs allow "NEXT" statements which do not specify the loop variable to end "FOR" loops. There are several lines in my program that may necessitate this type of change: 160, 190, 240, 280, 340, and 550. In all of these cases the implied FOR loop variable is "I".

5) You may have to DIMension your strings in your BASIC program, as is true in Apple's Integer BASIC, but not Applesoft.

6) String concatenation in Applesoft is accomplished with string expressions joined by means of the plus (+) sign; your BASIC may use the ampersand (&).

7) In comparing strings, Applesoft uses the combination of less than and greater than signs (< >); perhaps, as in Integer BASIC on the Apple, you are only allowed to test inequality with the number sign (#).

8) Please note that I have several statements in my program of the following general form: IF X THEN... This is "shorthand" for the equivalent IF X < > 0 THEN... I also have a number of statements like the following: IF...THEN 100 (where 100 can be any statement number). This is a "shorthand" for IF...THEN GOTO 100. I don't know whether all BASICs allow the abbreviated forms that I use.

9) I have made use of the following string functions: STR$, LEFT$, RIGHT$, MID$, and LEN. Your BASIC might call these by different names, or have different syntax rules about their arguments. Here are the Applesoft syntactic definitions for these functions, which you should keep in mind if you have to convert to different usages on your computer:

   STR$(X)
   converts the number X to a string

   LEFT$(A$,N)
   returns the leftmost N characters of string A$

   RIGHT$(A$,N)
   returns the rightmost N characters of string A$

   MID$(A$,M,N)
   returns the N consecutive characters of string A$, starting at position M

   LEN(A$)
   returns the number of characters in string A$
DECLARE NUM$ = STRING(1,CHR$(1))
DECLARE PN, QN, RN, RM, MF%, NF%, DF%, FD% = 0
DECLARE PROGRAM EDIT MASK
DECLARE PROGRAM PRINT "EDIT MASK ROUTINE": PRINT "THE EDIT MASK CAN
CONTAIN ANY INGREDIENT: PRINT "TION CHARACTERS, PLUS FOLLOWING SPECIAL
CHARACTERS: PRINT " IF $ IS FIRST CHAR., IT IS TREATED AS"
PRINT "A FLOATING DOLLAR SIGN"
PRINT " IF - IS LAST CHAR., IT WILL BE OUTPUT": PRINT "IF NUMBER TO
EDIT IS NEGATIVE, OR RE-": PRINT "PLACED BY BLANK IF POSITIVE"
PRINT " 9 CORRESPONDS TO A DIGIT TO PLACE IN": PRINT "THAT POSITION
OF THE MASK": PRINT " 0 CORRESPONDS TO A NONZERO DIGIT TO"
PRINT "PLACE IN THAT POSITION. IF YOU WANT A": PRINT "PRINT "PLACED •••"
IN THE MASK, ENCLOSING THE" PRINT " THE EDIT MASK IN QUOTES TO INPUT IT." PRINT
PRINT "EDIT MASK? " ; MASK$ \nINPUT "NUMBER TO EDIT?" ; NUM$ GOSUB 100: PRINT \nPRINT "PLACE NUMBER TO EDIT, NEW NUMBER, NEW MASK AND NUMBER?" ; N \nPRINT N GOTO 75, 70 \nGOTO 80 \n100 NUM$ = STR$(NUM$) ; LN = LEN (NUM$) ; LM = LEN (MASK$) \nQN = 0 : QM = 0 : RN = 0 : PN = 0 \n110 NF% = 0 : MF% = 0 : FD% = 0 : DF% = 0 \n110 OUT$ = "": IF NUM$ < 0 THEN NF% = 1: REM SET FLAG TELLING WHETHER INPUT
IS LAST CHAR.
120 IF RIGHTS (MASK$, I, 1) = "-" THEN MF% = 1: REM SET FLAG TELLING WHETHER
RIGHTMOST DIGIT IS NEGATIVE
130 IF LEFT$ (MASK$,1) = "S" THEN DF% = 1: REM SET FLAG TELLING WHETHER
LEFTMOST DIGIT IS FLOATING DOLLAR SIGN
140 FOR I = 1 TO LN: REM FIND POSITION OF DECIMAL POINT IN MASK
150 IF MID$ (MASK$,1,1) = "." THEN PN = I
160 NEXT I: IF DF% = 0 THEN DF% = 1: REM IF NO FLOATING DOLLAR SIGN IN M
ASK, SET FLAG SAYING "$" ALREADY OUTPUT TO EDITED FIELD
170 FOR I = 1 TO LN: REM FIND POSITION OF DECIMAL POINT IN NUMBER TO ED
IT \n180 IF MID$ (NUMS$,I,1) = "." THEN PN = I \n190 NEXT I \n190 NEXT I \n200 IF PN THEN RN = LN - PN: REM IF DECIMAL POINT IN NUMBER, COMPUTE \nNUMBER IN 0 OR MID$ (MASK$,I,1) = "9" THEN RM = RM + 1 \n240 NEXT \n250 IF RN = 0 AND PN = 0 THEN 300 \n260 IF RM = RN THEN 300 \n270 IF RM < RN THEN 290 \n290 FOR I = RN TO RM - 1: NUMS$ = NUMS$ + "0": NEXT : GOTO 300: REM ZERO-F
ILL RIGHTMOST DECIMAL POSITIONS OF NUMS
290 I = LN - RN + RM - 1: NUMS$ = LEFT$ (NUMS$,I) + "": REM TRUNCATE NUMS
TO MATCH MASK, PUT "" IN RIGHTMOST DIGIT \n300 QM = (LN) / RM: IF PN THEN QN = QN - 1: REM GET # DIGITS LEFT \n310 IF NF% AND MF% THEN QN = QN - 1: REM IGNORE MINUS SIGN IN NUMBER I
F TRAILING MINUS IN MASK \n320 IF PN THEN QN = QN - 1: REM FIND # DIGITS IN MASK LEFT \n330 IF MID$ (MASK$,I,1) = "0" OR MID$ (MASK$,I,1) = "9" THEN QM = QM + 1 \n340 NEXT \n350 IF QM > QN THEN 370: REM TRUNCATE NUMBER ON LEFT, MAKING LEFTMOST
DIGIT "\n360 IF LN (NUM$) = QN + QM - 1: IF NF% AND MF% THEN I = I - 1: REM DR
OP MINUS SIGN ALSO IF IGNORED BEFORE \n365 NUMS$ = "": RIGHTS (NUMS$,I): QM = QM
I1 = 1: IF FD% THEN I1 = 2: REM WILL IGNORE ANY FLOATING DOLLAR SIGN
IN MASK
I2 = LM: IF MF% THEN I2 = LM - 1: REM WILL IGNORE ANY TRAILING MINUS
IN MASK
370 I2 = LM: IF MF% THEN I2 = LM - 1: REM WILL IGNORE ANY TRAILING MINUS
IN MASK
380 I2 = LM: IF MF% THEN I2 = LM - 1: REM WILL IGNORE ANY TRAILING MINUS
IN MASK
370 I1 = 1: IF FD% THEN I1 = 2: REM WILL IGNORE ANY FLOATING DOLLAR SIGN
IN MASK
385 IF NF% AND MF% AND LEFT$ (NUM$,1) = "-" THEN QN = QN + 1: REM IF N
umber's MINUS SIGN WAS IGNORED BEFORE, PUT IT BACK IN
390 IF PN THEN NUM$ = DUM$ + RIGHT$ (NUM$,RM): REM DROP DECIMAL POINT
FROM NUMBER STRING
400 IF NF% AND MF% AND LEFT$ (NUM$,1) = "-" THEN NUM$ = RIGHTS (NUM$,LEN
(NUM$) - 1): REM DROP MINUS SIGN IF TRAILING MINUS IN MASK
405 J = LEN (NUM$): FOR I = I2 TO I1 STEP - 1: M$ = MID$ (MASK$,I,1): N$ = ":IF J > 0 THEN N$ = MID$ (NUM$,J,1)
410 IF M$ < > ": THEN 490
420 IF M$ < > ": THEN 490
430 IF N$ < > ": THEN 450
440 OUT$ = M$ + OUT$: J = J - 1: GOTO 550
450 IF N$ < > ": THEN 480
460 IF DF% THEN 440: REM IF FLOATING DOLLAR SIGN ALREADY OUTPUT, GO INS
ER BLANK
470 DF% = 1:OUT$ = ": + OUT$: GOTO 550
480 OUT$ = M$ + OUT$: GOTO 550
490 IF M$ < > ": THEN 520
500 IF N$ = ": THEN 460: REM IF ALL DIGITS OF NUMBER OUTPUT, GO OUTPUT
FLOATING DOLLAR SIGN OR BLANK
510 GOTO 440: REM GO OUTPUT THE DIGIT
520 IF M$ < > ": THEN 480: REM GO OUTPUT CURRENT CHARACTER IN MASK
530 IF NF% THEN 550: REM GO OUTPUT BLANK OR DIGIT
540 N$ = ": GOTO 440: REM OUTPUT BLANK
550 NEXT : IF DF% = 0 THEN OUT$ = ": + OUT$: REM IF FLOATING DOLLAR NO
T OUTPUT, APPEND IT ON LEFT
555 IF DF% AND FD% THEN OUT$ = ": + OUT$: REM IF DOLLAR SIGN IS ALREAD
Y OUTPUT, PUT BLANK IN PLACE OF MASK'S DOLLAR SIGN
560 IF MF% = 0 THEN RETURN: REM ALL DONE IF NO TRAILING MINUS IN MASK
570 N$ = ": IF NF% THEN N$ = ":: REM BLANK IF POSITIVE, MINUS SIGN IF NEGATIVE
580 OUT$ = OUT$ + N$: RETURN
Business Dollars and Sense in Applesoft

by Barton M. Bauers, Jr.

If you ever intend to do serious business programming in BASIC, then the information and programs presented here are invaluable. They show how to overcome the inherent rounding and formatting problems of BASIC in dealing with dollar and cents type of data.

If you purchased an Apple II Plus for business applications, that is applications which require the use of financial tables and calculations, then you may have encountered a rounding problem in executing your programs. Perhaps you have failed to recognize this problem, and are running programs which contain erroneous mathematical calculations! The purpose of this article is to acquaint you with the potential for rounding errors, and to suggest several possible solutions, depending on your needs. In addition, the process of creating text files, with some simple examples, will be addressed, since you will probably wish to use the subroutines discussed later in many programs which you write.

To start, let's demonstrate the problem. Try the following program:

PRINT 100.09 + 200.00 + .80 (rtn)

(Note that where (rtn) is indicated, it means to press the key marked RETURN.)

Your Apple should display:

300.89

Now type this program:

PRINT 300.89 — 100.09 — 200.00 — .80 (rtn)

The answer (which you'll agree should be zero) will appear as:

1.19907782E-08

This small error occurs because not all numbers between zero and one can be
exactly represented in binary arithmetic. Oddly enough, for most scientific work, such an error is insignificant, and will not affect the outcome of any programs. It is unlikely, however, that any usable system can be implemented in a business or financial situation unless absolute accuracy is obtained in recording and tabulating monetary amounts. When you program such an application—whether it be the family checkbook, or a complicated inventory control system—the ability to balance to the penny is a must!

There is, fortunately, a straightforward answer to the problem. While it is easy to discuss, it requires a bit of trickery to implement. If all values are carried within the computer as whole (integer) numbers, then there is no possibility of having rounding errors. The sacrifice you make, of course, is the necessity of performing all internal mathematical calculations in whole numbers, which requires that you, the programmer, remember where the decimal point belongs. Basically, therefore, by multiplying each monetary value by 100, and taking the INTeger value of the resultant figure, the problem is solved. This opens up additional problems, as we shall see.

Type in the following program:

```
10 DEF FN VL(X) = INT(X* 100)
20 INPUT "ENTER NUMBER:"; K
30 K = FN VL(K)
40 PRINT "NUMBER IS NOW:"; K
50 GOTO 20
RUN
```

Try some of the following examples:

```
1.00 (rtn)
```

The computer will respond with

```
100
```

Now try this one:

```
-2.99 (rtn)
```

The Apple answers with

```
-300
```

OOPS! Try this one now:

```
300.89
```

Your answer:

```
30088
```
Clearly, the use of integer values does not in itself solve the problem. The same rounding error which plagued the initial examples is contained in the integer value. The library function INT supplies the "...largest integer less than or equal to the given argument..." (quoted from the Applesoft II manual). In the negative direction, the rounding error will cause the integer value to one number smaller (further negative) than the argument whenever there is a rounding error: in the positive direction the integer is similarly smaller when the computer underrounds.

Referring back to the example used at the beginning of this article, it is easy to see that the value of the rounding error is extremely small—something like .00000001. Using the integer approach to eliminate the rounding problems, then, requires consideration for this small error. We are not concerned with values smaller than the second decimal place (pennies) in about 98% of business applications, therefore it is possible to add enough "cushion" to the integer conversion routine such that the small error which creeps in will never cause the Applesoft command INT to fall short during conversion.

To illustrate this process, type CTRL C (rtn) and rekey line 10 as follows;

```
10 DEF FN VL(X) = INT((X + .0001) * 100)
RUN (rtn)
```

Now try entering the previous examples.

```
<table>
<thead>
<tr>
<th>Number Entered</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>100</td>
</tr>
<tr>
<td>-2.99</td>
<td>-299</td>
</tr>
<tr>
<td>300.89</td>
<td>30089</td>
</tr>
</tbody>
</table>
```

This function works for both positive and negative numbers, because the 'ad­der' of .0001 is enough to offset any internal underrounding, both in a positive and a negative direction. Therefore, in any problem involving money calculations, you should add the following to your program:

```
15 DEF FN VL(X) = INT ((X + .0001) * 100)
```

```
aaa INPUT "ENTER AMOUNT":C
bbb C = FN VL(C)
```

Line 15 defines the function.

Line aaa requires keyboard entry of an amount which will be stored as variable C internally (you will naturally use whatever variable name you need here).
Line bbb converts C to an integer value, using the previously defined function, and 'pads' the value read in before conversion, to prevent underrounding.

Remember—all internal mathematics must now be performed with whole numbers.

A natural question at this point would be, "How do I print out the figures so that they once again look like dollars and cents?" This is part two of our story.

It would seem that by multiplying the integer number previously established by 01, we would again reduce the integer to a decimal number similar to the one originally typed in. Try it!

Type the following:

PRINT 30089* .01 (rtn)

Your answer:

300.89

Try some additional values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Value * .01</th>
</tr>
</thead>
<tbody>
<tr>
<td>-299</td>
<td>-2.99</td>
</tr>
<tr>
<td>-100</td>
<td>-1</td>
</tr>
<tr>
<td>180</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Again, the result is unacceptable for business applications. Again, it is clear that Applesoft BASIC, which handles scientific applications so well, is not equipped to yield usable formatting in dollars and cents. The author in fact, has seen commercial software which ignores this problem, and gives answers with the same errors demonstrated throughout the article. While some programmers might not consider the rounding problem serious, how can a businessman issue a check for $1.8?

The answer to the problem of restoring two decimal places to the internally generated integer values is a program which is named subroutine MASK. This program should be typed and SAVED, converted to a textfile, and EXEC'd into every business application where accurate dollars and cents calculations are required. Listing 1 shows the program steps for MASK. Type it and save it under the name DOLLAR MASK (it is assumed that you have at least one disk drive). After it is SAVED, you are ready to make a textfile out of DOLLAR MASK. To do this, if you have not already created a utility program for making textfiles, there is another short program which must be typed, SAVED, and made into a textfile. Prior to that exercise, however, let's look at the contents of the program MASK.

Line 50 is the value conversion function described earlier.
Line 15010 establishes the number of digits in the variable.

Line 15030 takes the right two characters (cents) and puts them in string variable XZ$. Note however that line 15060 puts a zero ahead of the value stored in XZ$ if XZ$ contains only one digit. Line 15090 removes a minus sign if it became embedded in XZ$, and replaces it with a zero, moving the minus sign to the left of the decimal point in XX$.

Line 15040 branches depending on whether the input string ZZ$ has 1, 2, or 3-9 digits.

Line 15100 puts all except the cents value (which is now stored in ZZ$) into the 'dollars' area, XX$.

To test this program, load it from the disk, and add the following additional lines:

60 INPUT "ENTER NUMBER:";CA  
70 CA = FN VL(CA)  
80 ZZ$ = STRS(CA)  
90 GOSUB 15000  
100 CA$ = -XW$  
110 PRINT " THE ANSWER IS: ";CA$  
120 END

Now type RUN and try some values which might be representative of a business application. Try some positive and negative values, so you can demonstrate that DOLLAR MASK really works.

After you have become familiar with the logic, it is easy to add other capabilities to the DOLLAR MASK. For example, if you want to remove the floating dollar sign from the program, delete the first part of line 15020, and drop XV$ from line 15110. Another example is shown in listing 3, a routine for adding check protecting characters [*] to the left of the masked number. The assumption in this subroutine is for a field of 30 digits, but you can easily increase or reduce it at your leisure.

To put the finishing touches on your program, it will be necessary to convert DOLLAR MASK into a textfile. Then, it can be added to any program you write by typing EXEC MASK. If you are not comfortable with the EXEC portion of the Apple DOS manual, then the program listed in listing 2 will do the job easily. To use this program, follow these steps:

1. Type the program in listing 2 TWICE, once with line number 10, and once with line number 63999. When typing it under line number 10, change the LIST reference to LIST 63999.

2. Type RUN.
3. The computer will ask NAME OF TEXTFILE — , to which you should respond CREATE EXEC FILE [rtn]. When the disk stops, you will have created a textfile named CREATE EXEC FILE. LOCK it, since it will permit you to set up standard subroutines as text files in the future.

Now you are ready to make DOLLAR MASK into a textfile. If you have already typed it and SAVEd it to disk under the name DOLLAR MASK, LOAD it into memory, and follow the steps below:

1. Type EXEC CREATE EXEC FILE
2. Type RUN 63999
3. Answer the inquiry with MASK [rtn]
4. You now have subroutine MASK stored on disk for future use.

Below is a summary on how to get MASK into your future business programs:

1. When writing a program do not use line numbers 15 or 15000 to 15120.
2. Insert the disk with MASK on it and type EXEC MASK.
3. You now have the subroutine and the function in your program.
4. Each time your program requires a value from the keyboard, such as CA, add the following line after you read the value in:
   \[ CA = FN VL(CA) \]
5. If you have occasion to output money data to the screen or to a printer, add the lines:
   \[ ZZ$ = STR$(CA) \]
   GOSUB 15000
   CA$ = XW$
   PRINT CA$
6. You now have a string variable CA$ to display the value previously stored in CA as a whole number.
7. Remember — the argument to use before you GOSUB 15000 is ZZ$, and the return argument is XW$.

63999 DS = CHR$ (4); INPUT "NAME OF TEXTFILE IS - ";AA$: PRINT DS"OPEN ";AA$: PRINT DS;"WRITE ";AA$: LIST 1,63998; PRINT DS;"CLOSE ";AA$: DEL 63999,63999
10 REM **********************
12 REM *
14 REM * BUSINESS DOLLARS *
16 REM * AND SENSE *
18 REM * BARTON BAUERS *
20 REM *
22 REM * DOLLAR MASK *
24 REM *
26 REM * COPYRIGHT (C) 1981 *
28 REM * MICRO INK, INC. *
30 REM * CHELMSFORD, MA 01824 *
32 REM * ALL RIGHTS RESERVED *
34 REM *
36 REM **********************
38 REM
50 DEF FN VL(X) = INT ((X + .0001) * 100)
14999 :  
15000 REM **SUBROUTINE**
15001 REM ARGUMENT IS ZZ$
15002 REM RESPONSE IS XW$
15005 :
15010 M% = LEN(ZZ$)
15020 XV$ = "$";XX$ = "";XY$ = "." 
15030 XZ$ = RIGHT$(ZZ$,2)
15040 ON M% GOTO 15060,15070,15100,15100,15100,15100,15100,15100
15050 PRINT "ERROR ON INPUT VALUE": XW$ = "": GOTO 15120 
15060 XZ$ = "0" + ZZ$: GOTO 15110 
15070 IF LEFT$(ZZ$,1) = "-" GOTO 15090 
15080 GOTO 15110 
15090 XZ$ = "0" + RIGHT$(ZZ$,1);XX$ = ":"; GOTO 15110 
15100 XZ$ = LEFT$(ZZ$, (M% - 2))
15110 XW$ = XV$ + XX$ + XY$ + XZ$
15120 RETURN

10 REM **********************
12 REM *
14 REM * BUSINESS DOLLARS *
16 REM * AND SENSE *
18 REM * BARTON BAUERS *
20 REM *
22 REM * CHECK PROTECT *
24 REM *
26 REM * COPYRIGHT (C) 1981 *
28 REM * MICRO INK, INC. *
30 REM * CHELMSFORD, MA 01824 *
32 REM * ALL RIGHTS RESERVED *
34 REM *
36 REM **********************
38 REM
50 DEF FN VL(X) = INT ((X + .0001) * 100)
15000 REM **SUBROUTINE**
15001 REM ARGUMENT IS ZZ$
15002 REM RESPONSE IS XW$
15005 :
15010 M% = LEN(ZZ$)
15020 XV$ = "$";XX$ = "";XY$ = "." 
15030 XZ$ = RIGHT$(ZZ$,2)
15040 ON M% GOTO 15060,15070,15100,15100,15100,15100,15100,15100
15050 PRINT "ERROR ON INPUT VALUE": XW$ = "": GOTO 15120 
15060 XZ$ = "0" + ZZ$: GOTO 15110 
15070 IF LEFT$(ZZ$,1) = "-" GOTO 15090 
15080 GOTO 15110 
15090 XZ$ = "0" + RIGHT$(ZZ$,1);XX$ = ":"; GOTO 15110 
15100 XZ$ = LEFT$(ZZ$, (M% - 2))
15110 XW$ = XV$ + XX$ + XY$ + XZ$
15120 RETURN
Lower Case and Punctuation in Applesoft

by James D. Childress

Getting lower case letters and punctuation into an Applesoft string can be a real problem... which is unfortunate, since many programs could benefit from that capability. The following article addresses that problem and the two accompanying programs provide a no-cost method to solve the problem!

While computer people may adapt to all caps, the general public still uses, and apparently likes, lower case. Printing with lower case is more familiar, more readable and more acceptable. Thus, we who work with computers should provide lower case in any printout that we expect or hope laymen to read. After all, computers should adapt to people; people should not have to adapt to computers.

Also, who among us hasn't wondered at how the Apple handles punctuations in strings? In INPUTs, we have found to our dismay that a "JONES, JOHN" results in an error message saying "?EXTRA IGNORED" and later finding the string variable as only "JONES" with nothing to tell us which Jones that may be. What wouldn't we give to get quotation marks and commas in the places we want!

So much for what should be or what we want. The Apple doesn't have lower case and seems rather whimsical about punctuation. Well, face it; there were a number of compromises made in the design of the Apple and Applesoft. Of course, some of these deficiencies can be conquered by money. We can buy one of the lower case boards and live more or less happily ever after. Unfortunately, we do not all or always have the option of buying a solution to a problem; most of us have more problems than money. And there are not always solutions for sale.

An alternative approach is an Applesoft program to produce the desired lower case and punctuation. I have looked for such a program and I found two possibilities (there likely are others but I am not acquainted with them):

1. Val J. Golding in "Lower Case Routine for Integral Data Printer," Call-Apple, v.2, p. 11 (April/May 1979) gave a program to POKE lower case characters into strings in the string array memory space.
2. Another program was published in Contact, v.1, p.5 [May 1978]; this program POKEs lower case into the beginning of program memory space.

Both of these are quite limited. Note: Both should work for punctuation problems within the same limitations.

Neither of these enables you to enter lower case or problem punctuations conveniently into string variables, nor to print statement strings in an Applesoft program as desired. The program in figure 1 does the job for string variables and the one in figure 2 for strings in print statements.

Use and Operation

The heart of these programs is the same as in the cited programs: use of the GET command to sneak things around the interpreter. The GET command handles input character-by-character so that each can be manipulated. [The identical GET routine is used for both programs—lines 63010 to 63120 in the first, and lines 63140 to 63150 in the second. Only one typing needs be done, a hint not to be ignored.]

The first program is intended for use as a subroutine. For example, a statement such as

30 INPUT "ACCOUNT NAME";NAME$(1)

can be replaced directly by

30 PRINT "ACCOUNT NAME"; GOSUB63000; NAME$(1) = BB$

In a run, the program would appear to behave normally except that there would be no ?EXTRA IGNORED's and NAME(1) would look quite strange on the CRT monitor ("",/7%2#!3%" for "lower case") and as lower case only on the printer.

In both programs, capitals are entered in a manner similar to the operation of MUSE's word processor program, Dr. Memory. A ctrl-A makes the next letter only capital; an ctrl-C makes all the following letters capital until either a ctrl-S or the end of the string. Unlike Dr. Memory, the control characters are not displayed. Instead, the capitalized letters are shown in inverse video. I like this way of doing things. If you would prefer the opposite video, just interchange the words NORMAL and INVERSE in lines 63020-63040 and 63080 and add an INVERSE to line 63000 in figure 1. You could do even more to tailor to your personal tastes; change the control characters, change the default operation from lower case to capitals, etc. These custom fittings are left as an exercise.

Another feature common to both programs is the motion of the cursor. The backspace works but that is all. And it will move the cursor back no further than the initial position. However, therein lurks a minor nuisance; if you try to backspace beyond that limit, the immediately preceding character will be wiped out or replaced by a white block. This is of no consequence; ignore it.
Since the string variables subroutine runs as a part of your program, you have to keep labels straight. This subroutine uses only AA$, AZ$, BB$, BB, BZ$, and ZZ and has no FOR loops. Also note that only the usual limitation applies for the length of strings.

In the use of the second program, you append it to the program in which you want to put lower case. A RUN 63000 initiates things; you simply give the line number in which lower case is wanted. The first string in that line is printed, terminated by # # to indicate the length limit. The cursor below this line indicates the place for the change. You can insert anything but we assume that a mixed capital and lower case rendition of the line above is what you will want. In any case, the length cannot be exceeded. If you go over the limit, the excess will be ignored. If you put in less, the remainder will be filled with spaces. If you don't want to change that particular string, simply hit RETURN.

After a RETURN, the next string in the same line will appear, ready to be changed. When all the strings of that one line have been dealt with, you are asked for the number of the next line.

As mentioned above, lower case is displayed by the Apple as keyboard symbols other than letters. These print properly as lower case on a printer that prints lower case. If you want to display, say, a table so that you can check data prior to printing, you need to program the display table and the printout table separately. For convenience in doing this, both programs provide an all-caps string BZ$ as well as the corresponding string BB$ with lower case.

Program Design

The GET routine, essentially the whole of figure 1, has already been mentioned. The GET command is followed by a series of IF's to implement the control character, backspace and RETURN functions. These are straight-forward and self-explanatory.

The second program, figure 2, consists of three parts. The first, lines 63020-63300, POKEs the new string into the program in the memory space.

Concluding Remarks

Although written for Applesoft, these programs can be adapted to other BASIC's. The first presents no problems. However, the program memory space search routine in the second will require modification for other computers. This modification should not be too difficult to implement for other Microsoft BASIC's.
62980 REM *********************************************
62981 REM *
62982 REM * LOWER CASE INSERT *
62983 REM * JAMES D. CHILDRESS *
62984 REM *
62985 REM * LOWER CASE INSERT *
62986 REM *
62989 REM * COPYRIGHT (C) 1981 *
62990 REM * MICRO INK, INC. *
62992 REM * CHELMSFORD, MA 01824 *
62993 REM * ALL RIGHTS RESERVED *
62994 REM *
62999 END

HOME: VTAB (3): PRINT "LOWER CASE INSERTION PROGRAM": PRINT:
LMAX = 62999: PRINT "NUMBER OF FIRST LINE TO BE RE-": INPUT "WRITE N •;LT:
PRINT:
LMAX = 62999: PRINT "NUMBER OF FIRST LINE TO BE RE-WRITTEN": INPUT "WRITE N •;LT:
NEXT: PRINT ":H:

IF TST > LMAX OR LN > LT THEN GOTO 63320

IF LN < LT THEN M = 256: PRINT:
M = 256: PEEK (M - 1) + PEEK (M - 2) + 2: GOTO 63030

K = 0: LL = 0: UL = 0

FOR J = M + 2 TO M + 255: TST = PEEK (J): IF TST = 0 THEN M = J + 3: GOTO 63030

IF TST = 58 THEN K = 1
IF TST = 186 OR TST = 132 THEN K = 1
IF K = 1 AND LL > 0 AND TST = 34 THEN UL = J - 1: GOTO 63120
IF K = 1 AND LL = 0 AND TST = 34 THEN LL = J + 1
NEXT

BB$ = "": BZ$ =": ZZ = 1: INVERSE:

IF ASC (AA$) = 13 THEN NORMAL:
GOTO 63160

IF ASC (AA$) = 1 THEN ZZ = 1: INVERSE:
BB = 0: GOTO 63140

IF ASC (AA$) = 3 THEN BB = 1: INVERSE:
GOTO 63140

IF ZZ = 1 OR BB = 1 THEN ZZ = 0: GOTO 63210

IF ASC (AA$) < 65 OR ASC (AA$) > 90 THEN 63210

AA$ = CHRS (ASC (AA$) + 32)

IF ASC (AA$) = 8 AND ASC (AA$) = 8 THEN BB$ = "": BZ$ = "": PRINT ""

IF LEN (BB$) < 2 AND ASC (AA$) = 8 THEN BB$ = "": BZ$ = "": GOTO 63130:

IF ASC (AA$) = 8 THEN BB$ = LEFT$ (BB$, LEN (BB$) - 2)

GOTO 63140

IF BB$ = "": GOTO 63110

PRINT : FOR I = LL TO UL:
PRINT CHR$ (ASC (DC$)):
MM = ASC (DD$)
POKE I, MM
NEXT

UL = 0: LL = 0: PRINT : GOTO 63110

PRINT : PRINT "NUMBER OF NEXT LINE TO BE REWRITTEN": INPUT ": END PROGRAM": LT

IF LT = 0 THEN END

GOTO 63020
Figure 2

62980 REM ****************************
62981 REM *
62982 REM *  LOWER CASE ENTRY *
62983 REM *  JAMES D. CHILDRESS *
62984 REM *
62985 REM *  LOWER CASE ENTRY *
62986 REM *
62989 REM *  COPYRIGHT (C) 1981 *
62991 REM *  MICRO INK, INC. *
62992 REM *  CHELMSFORD, MA 01824 *
62993 REM *  ALL RIGHTS RESERVED *
62994 REM *
62995 REM ****************************
62996 REM
63000 BB$ = "":BZ$ = "":BB = 0:ZZ = 0
63010 GET AA$:AZ$ = AA$: IF ASC (AA$) = 13 THEN NORMAL : GOTO 63130
63020 IF ASC (AA$) = 1 THEN ZZ = 1: INVERSE :BB = 0: GOTO 63010
63030 IF ASC (AA$) = 3 THEN BB = 1: INVERSE : GOTO 63010
63040 IF ASC (AA$) = 19 THEN BB = 0: NORMAL : GOTO 63010
63050 IF ZZ = 1 OR BB = 1 THEN ZZ = 0: GOTO 63080
63060 IF ASC (AA$) < 65 OR ASC (AA$) > 90 THEN 63080
63070 AA$ = CHR$ ( ASC (AA$) + 32)
63080 BZ$ = BZ$ + AZ$: PRINT AZ$: IF BB = 0 THEN NORMAL
63090 BB$ = BB$ + AA$: IF ASC (BB$) = 8 AND ASC (AA$) = 8 THEN PRINT "",
63100 IF LEN (BB$) < 2 AND ASC (AA$) = 8 THEN BB$ = LEFT$ (BB$, LEN (BB$) - 2)
63110 IF ASC (AA$) = 8 THEN BB$ = LEFT$ (BB$, LEN (BB$) - 2)
63120 GOTO 63010
63130 PRINT : RETURN
63140 END
4 GRAPHICS

Introduction

Graphing Rational Functions
Ron Carlson

A Hi-Res Graph Plotting Subroutine in Integer BASIC for the Apple II
Richard Pam

How to Do a Shape Table Easily and Correctly
John Figueras

Define Hi-Res Characters for the Apple II
Robert F. Zant

Apple II High Resolution Graphics Memory Organization
Andrew H. Eliason
**Introduction**

The graphics capability of the Apple needs no introduction. It is undoubtedly one of the most appreciated features of the computer. The articles and programs appearing in this section build around this tremendous capability by making it even easier to use and understand the Apple II hi-res graphics. The first two articles address the plotting of functions. "Graphing Rational Functions" by Ron Carlson presents a complete system in Applesoft for graphing any function at any scale on the Apple's screen. Richard Fam's "Hi-Res Graph Plot" provides another method for plotting functions in Integer BASIC.

"How to Do a Shape Table" by John Figueras provides a foolproof system for automatically generating shape tables. "Define Hi-Res Characters" by Robert Zant presents a method for generating a character table for use with Apple's hi-res character generator. And lastly, Andrew Eliason's "Hi-Res Memory Organization" article gives insight into the graphics screen representation in memory. Together, these articles and programs will help unlock the graphical capabilities of anyone's Apple!
Graphing Rational Functions

by Ron Carlson

One of the more interesting and educational applications for the Apple's high resolution graphics is plotting functions. This general purpose plotting routine—applied here to rational functions—can graph any function over any scale and is easy enough to be used by any student!

This is a general graphing program even though it is applied to graphing rational functions, such as:

\[
y = \frac{x(x - 4)(x + 3)}{(x - 1)(x + 5)}
\]

If you want to graph any type of function, either remove the denominator function, FN DEN(X), or merely DEF FN DEN (X) = 1. Therefore you could graph \( y = x \sin(x) \) by the following lines:

60 DEF FN NUM(X)=X*SIN(X)
70 DEF FN DEN(X)=1

This program has evolved from plotting x's on a printer to the versatile graphics output of the Apple II. Even the program on the Apple II went through changes, ranging from graphing with an origin in the center of the screen, graphing any quadrant and choice of scale, to this version of choosing the location of the origin on the screen and the scale. High school students appear to have no difficulty using either of these options.

The program is broken into several parts: first the directions and functions section explains to the user how to define the numerator and denominator functions and how to use the program. Any legal BASIC expression can be used for the definition of the numerator and denominator. Any non-rational function can be graphed by DEF FN DEN(X) = 1. I chose the definition method of inputting the function to make the program more easily transferable to other versions of BASIC.
Another section needed for the preparation is for arrangement of the scale and determination of the location of the origin. I use the low-resolution screen with a colored cursor in the center. The user can move the cursor up, down, left, or right by using the following keys: U, D, L, R, and F when finished. The relative final position of the cursor \((A, B)\) is changed to represent the location of the origin on the high-resolution grid of \(280 \times 192\).

The main body of the program is the graphing section. In order to graph functions, two problems had to be overcome. The first is that the upper left corner of the screen is the origin, making it effectively upside down. The second is that I wanted to have different origins for different applications.

---

**Sample conditions:** Origin \((200, 90)\)

Scale = 10

**Transformation formula:**

\[
\begin{align*}
    x' &= (Hx - Ox)/\text{scale} \\
    y' &= (Oy - Hy)/\text{scale}
\end{align*}
\]

\((Hx, Hy) = \text{Apple HI-Res coor.}\)

\((Ox, Oy) = \text{Origin location in HI-Res coor.}\)

**Sample points:**

\((240, 150) - (x', y')\)

\[
\begin{align*}
    x' &= (240 - 200)/10 = 4' \\
    y' &= (90 - 150)/10 = -6'
\end{align*}
\]

\((20, 40) - (x', y')\)

\[
\begin{align*}
    x' &= (20 - 200)/10 = -18' \\
    y' &= (90 - 40)/10 = 5'
\end{align*}
\]

\((200, 90) - (x', y')\)

\[
\begin{align*}
    x' &= (200 - 200)/10 = 0' \\
    y' &= (90 - 90)/10 = 0'
\end{align*}
\]

*Figure 1*
A mathematical transformation formula will change the HGR coordinates to \( x \) and \( y \) or \( x \) and \( y \) to HGR coordinates.

\[
(\text{real } x \text{ coor.}) = (\text{HGR } x \text{ coor.}) - (x \text{ coor. of origin})
\]

\[
X = H - A
\]

\[
(\text{real } y \text{ coor.}) = (y \text{ coor. of origin}) - (\text{HGR } y \text{ coor.}) \text{ and } Y = B - V.
\]

When the scale factor, \( S \), is considered, then the transformation formulas look like:

\[
X = (H - A)/S
\]

\[
V = B - Y*S
\]

Variables: Scale = 20
Origin: (50,60)

Sample points:

- \((30,180)\):
  \[
  x' = (30 - 50)/20 = -1' = (-1', -6')
  y' = (60 - 60)/20 = 0'
  \]

- \((50,60)\):
  \[
  x' = (50 - 50)/20 = 0' = (0', 0')
  y' = (60 - 60)/20 = 0'
  \]

- \((250,80)\):
  \[
  x' = (250 - 50)/20 = 10' = (10', -1')
  y' = (60 - 80)/20 = -1'
  \]

- \((180,30)\):
  \[
  x' = (180 - 50)/20 = 6.5' = (6.5', 1.5')
  y' = (60 - 30)/20 = 1.5'
  \]

Figure 2
To graph, start H, the HGR coordinate, at 0 and continue the loop until H is 279. Translate H to the real x coordinate and substitute X into the function. Check for an asymptote, and solve for the real y-coordinate. The transformation formula will give the HGR vertical coordinate, which can be checked to make sure it is on the screen, and plot the point. When the graphing loop is finished, POKE -16302,0 displays the bottom portion of the screen. The graph stays on the screen until the user depresses any key, thus giving plenty of time to make any important notes. The user is offered the choice of keeping the same function and changing the position of the origin and changing the detail by means of the scale, or starting over with a new function.

Sample run using: $$y = \frac{x(x + 3)(x - 4)}{(x + 1)(x - 1)(x + 5)}$$

Scale = 12

Figure 3
Graphing Rational Functions

Ron Carlson

---

10 REM ****************************
20 REM * GRAPHING RATIONAL *
30 REM * FUNCTIONS * 
40 REM * RON CARLSON * 
50 REM * FUNCTION GRAPH * 
60 REM * FUNCTION GRAPH * 
70 REM ****************************

50:
60 DEF FN NUM(X) = X
70 DEF FN DEN(X) = (X + 2)

---

500 PRINT "REMEMBER:"
REM POSITIONING THE ORIGIN OF THE SCREEN (40,40)
GR : COLOR= 3: PLOT 20,20:A = 20:B = 20
GET A$
A1 = A:B1 = B
IF A$ = "U" THEN B = B - 1: GOTO 710
IF A$ = "D" THEN B = B + 1: GOTO 710
IF A$ = "L" THEN A = A - 1: GOTO 710
IF A$ = "R" THEN A = A + 1: GOTO 710
IF A$ = "F" THEN 800
REM KEEP ON THE LOW RES SCREEN
IF B < 1 THEN B = 1
IF B > 39 THEN B = 39
IF A < 1 THEN A = 1
IF A > 39 THEN A = 39
REM BLANK OLD POSITION
COLOR= 0: PLOT A1,B1: COLOR= 3
REM PLOT NEW POSITION
PLOT A,B
GOTO 630
A = 7 * A:B = B * 192 / 40
REM CHANGE SCALE TO REFLECT HGR (280 BY 192)
TEXT : HOME : RETURN
END
A Hi-Res Graph-Plotting Subroutine in Integer BASIC for the Apple II

by Richard Fam

An Integer BASIC subroutine is presented which permits Hi-Res graph plotting. It includes X and Y axes generation with scale markers as well as the plotting of user specified points. This will make it easy to display the results of a variety of problems, functions, correlations, etc., from Integer BASIC.

The article entitled Apple II High Resolution Graphics Memory Organization, by Andrew H. Eliason is of tremendous value to those who wish to plot in Hi-Res graphics. The following graph plotting subroutine utilizes formulae given in this article.

The Graph Plot Subroutine

Referring to the listing: On being called by the GOSUB 9000 statement in the main program, the subroutine first clears page 1 of Hi-Res graphics memory at line 9023. This is quite a time-consuming process and the impatient experimenter may care to replace this line with a CALL statement to an equivalent machine language subroutine. I have actually tried this and found that it reduces the time execution for the complete plotting routine by approximately half.

Having set the graphics and Hi-Res modes in line 9060, the routine then proceeds to plot the X and Y axes. Scale markets are placed at 20-point intervals along the two axes.

The final stage in the subroutine involves the plotting of the points. The magnitude of these points is stored in matrix GPH which is dimensioned for 279 elements in the main program. Only values GPH(X) between 0 and 91 inclusive can be plotted.

As you may recall, the display area of Hi-Res graphics is a matrix comprised of 280 horizontal by 192 vertical points. The subroutine fetches elements of GPH, does the necessary calculations, and outputs the results on the screen. To prevent the disfigurement of the two axes, I have avoided the plotting of points less than one byte away from the Y-axis and on the X-axis itself.
For successful application of this graph plotting subroutine, observe the following rules:

a) Only an Apple II with a minimum of 16K bytes of memory can be used.

b) Ensure that the main program contains the statement DIM GPH(279).

c) Only values of GPH(X) such that 0 ≤ GPH(X) ≤ 191 where X ranges from 0 to 279, inclusive, will be plotted.

d) Set HIMEM:8191 to restrain intrusion into page 1 of Hi-Res graphics memory.

Here are two short programs demonstrating the performance of the high resolution graphics-plotting subroutine:

```
110 DIM GPH(279)
120 FOR I = 0 TO 279
130 GPH(I) = RND(191)
140 NEXT I
150 GOSUB 9000
160 END

110 DIM GPH(279)
120 FOR I = 0 TO 279
130 GPH(I) = I/2—30
140 NEXT I
150 GOSUB 9000
```
Fam  Hi-Res Graph Plot  77

10 REM ********************
11 REM *
12 REM * HI-RES GRAPH PLOTTING *
14 REM * RICHARD FAM *
15 REM * GRAPH- PLOT *
16 REM *
18 REM *
20 REM * COPYRIGHT (C) 1981 *
22 REM * MICRO INK, INC. *
24 REM * CHELMSFORD, MA 01824 *
26 REM * ALL RIGHTS RESERVED *
28 REM *
30 REM ********************
9000 REM
9001 REM
9007 REM * DATA IS STORED IN GPH(X)
9008 REM * CONSISTING OF 200 POINTS
9009 REM * 0 <= GPH(X) <=191
9010 REM *
9011 REM * SET HIMEM:8191
9012 REM *
9020 REM *
9021 REM * CLEAR SCREEN
9022 REM *
9023 FOR I=8192 TO 16383: POKE I,0: NEXT I
9030 REM *
9040 REM * SET HIRES MODE
9050 REM *
9060 POKE -16304,0: POKE -16297,0: POKE -16302,0
9140 REM *
9150 REM * PLOT Y-AXIS
9160 REM *
9170 FOR LV=0 TO 191: PT=1: IF (LV+9) MOD 20=0 THEN PT=7: POKE (LV MOD 8*1024+(LV/8) MOD 8*128+(LV/64)*40+8192),PT: NEXT LV
9200 REM *
9210 REM * PLOT X-AXIS
9220 REM *
9230 PT=0: FOR LH=0 TO 279: IF LH MOD 20<>0 THEN 9240: PT=PT+1: FOR MK=1 TO 2: POKE LH/7+16336-(1024*MK),64/(2 ^ ((PT+5) MOD 7))
9240 NEXT LH
9260 REM *
9270 REM * PLOT POINTS
9280 REM *
9290 FOR LH=8 TO 279: LV=191-GPH(LH): IF LV<0 OR LV>=191 THEN 9330
9310 BV=LV MOD 8*1024+(LV/8) MOD 8*128+(LV/64)*40+8192: POKE LH/7+BV,2 ^ (LV MOD 7)
9330 NEXT LH: RETURN
How to Do a Shape Table Easily and Correctly!

by John Figueras

The mechanism for generating shapes and characters in Apple High Resolution Graphics is cumbersome and prone to error. A very clear explanation of the mechanism and pitfalls is presented here. But, best of all, the program presented permits the user to create the shapes interactively, using the keyboard and display.

One of the most discouraging tasks facing the Apple owner is the creation of a shape table. The table is required for generation of shapes and characters for high resolution graphics, since Apple does not offer pre-formed plotting characters. Thus, if you want to label the axes of a graph, the shape table can be used to supply the characters required for the labels. It is also useful for producing special shapes for games.

If, like me, you have tried to prepare a shape table using Apple's procedure, I am sure you'll discover, as I did, that the procedure is time-consuming, tedious, and error-prone. In several attempts, I have yet to generate a shape table using the manual procedure given by Apple, that didn't end up with missing dots, spurious projections or an unpredicted shape. At first I thought the problem was of my own making, since Apple's directions are clear and apparently faultless. The use of the words "apparently faultless" in the last sentence implies that what I found was in fact the case: Apple's procedure for creating a shape table has some real glitches. I discovered these in the course of pursuing the work described below, and developed a procedure that circumvents the glitches and produces perfect results every time.

Apple's procedure for preparation of a shape table is carried out as follows: the shape is first laid out as a dot pattern on a grid (figure 1); a series of plotting vectors is superimposed on the pattern to trace out a continuous path that covers all points to be plotted. The plotting vectors are defined either as move-only or as plot-then-move vectors.
The shape in figure 1 is reproduced in figure 2 with the chain of plotting vectors superimposed. The plotting vector chain may start at any point, but in selecting this point you should know that the initial point in the shape is the point that gets plotted at coordinates \((X,Y)\) in the DRAW command. Therefore, your choice of initial point determines the justification of the shape or character with respect to the plotting location. If you want a center-justified character, then start the vector sequence at the center of the shape; a left-justified character must be started at the left side, and so on.

The Apple manuals give the impression that it is immaterial where you start the shape, but if you want to have your characters fall properly on a line, it is something you must attend to. Knowing justification of the shape is important in games where things bang together and in building up large patterns by plotting sub-units adjacent to each other—cases in which it is important to know where the boundaries of the shape fall relative to the point at which it is plotted.
The next step in preparing the shape table requires that the chain of plotting vectors in figure 2 be unfolded into a linear string, beginning with the initial point of the pattern. For the shape in figure 2, the following sequence of vectors is obtained after unfolding:

The plotting vector string is then broken up into groups of two or three, each group reading from right to left. To add a little more danger to the game, the rules require that no group of vectors may end with a move-up vector or with a plot-then-move vector, in which case the group will contain at most two plotting vectors.

The table in figure 3a shows how the above string is subdivided. In this case, because of the restrictions on termination, each group can contain only two vectors. The rules for formulating these vectors groups are actually quite soundly based, as will become clear in later considerations.

We are not done yet. In the next step, each plotting vector as it appears in the table in figure 3a is replaced by a 3-bit (octal) code. The code is shown in figure 4, along with the decimal equivalents. Note that the decimal code for a plot-then-move vector is obtained simply by adding decimal 4 to the corresponding move-only vector. There is a certain amount of method in this madness. The 3-bit code translation for the plotting vectors in figure 4, which represent our shape, is displayed in figure 3b.
The next opportunity for confusion (and error) appears now, when the bit-strings in Figure 3b are re-grouped and assembled into nybbles (figure 3c) and the nybbles are each translated into hexadecimal numbers (figure 3d). The pairs of hexadecimal numbers, of course, represent the content of one byte. This is the byte that is stored in the shape table. In essence, then, the shape table is a list of hexadecimal numbers, which, after translation into binary and re-grouping, represents the collection of 3-bit codes equivalent to the plotting vectors, which in turn represent the original shape. In the parlance of mathematics, the shape has been \textit{mapped} onto the set of hexadecimal numbers.

If by now the reader is feeling a tingle of impatience with this description, multiply that feeling by a factor of at least ten, and you will be on the verge of understanding what it feels like to carry out these steps. To add to the frustration, there are enough booby traps laid by Apple to ensure quite a decent probability that after you have gone through this travail, the shape that finally appears on your screen will be misshapen. With a computer at hand, it seems silly to be bogged down by a process like this—and that's what the rest of this article is about: a computer program in Applesoft BASIC that allows easy graphic input of a shape or character with automatic generation and storage of a correct shape table—graphics without tears, so to speak.

\begin{figure}[h]
\centering
\begin{tabular}{ccccccc}
\hline
\text{\textbf{a)}} & \text{\textbf{b)}} & \text{\textbf{c)}} & \text{\textbf{d)}} \\
\hline
\text{\textbf{\rightarrow \leftarrow}} & 00 & 111 & 011 & 0011 & 1011 & \text{3B} \\
\text{\textbf{\rightarrow \leftarrow}} & 00 & 111 & 111 & 0011 & 1111 & \text{3F} \\
\text{\textbf{\uparrow \uparrow}} & 00 & 100 & 100 & 0010 & 0100 & \text{24} \\
\text{\textbf{\uparrow \rightarrow}} & 00 & 100 & 001 & 0010 & 0001 & \text{21} \\
\text{\textbf{\uparrow \rightarrow}} & 00 & 100 & 001 & 0010 & 0001 & \text{21} \\
\text{\textbf{\uparrow \rightarrow}} & 00 & 100 & 001 & 0010 & 0001 & \text{21} \\
\text{\textbf{\uparrow \uparrow}} & 00 & 011 & 100 & 0001 & 1100 & \text{1C} \\
\text{\textbf{\leftarrow \rightarrow}} & 00 & 111 & 111 & 0011 & 1111 & \text{37} \\
\text{\textbf{\downarrow \downarrow}} & 00 & 110 & 010 & 0011 & 0010 & \text{32} \\
\hline
\end{tabular}
\caption{Translation of shape vectors to Hexadecimal Code.}
\end{figure}
Approach to a Solution

Computer programmers have their own mind-set. For some, it is structure: a beautiful program that reads like a novel. For others—start at the middle and develop a nice, tight, efficient algorithm. I am an input-output bug. To me, the proper questions that should be first answered are: how can I make it easy for users of the program to get their data into the program; and how can the output be made digestible?

In the present case, of course, the major problem is one of input. With the equipment at hand—an Apple keyboard, video screen and a couple of floppy disks—I settled on a display of a 15 x 15 grid and a cursor that can be moved by hitting appropriate keys (Up, Down, Left, and Right). The shape is created by plotting it as a dot pattern under control of the moveable cursor, using the P (for Plot) key to lay down the dot pattern. One necessary key is the Quit key, which informs the computer that the shape is done. A convenience key, E for Erase, is provided to accommodate some of my sloppy keyboard habits; it facilitates undoing the last plotted point. The selection of keys U,D,L and R for directing the cursor was modeled after the set of allowed plotting vectors (there are no diagonal moves in the set), and was a fortunate selection for easy formulation of the algorithm.

While the general format for input was quite clear, the approach to translating that input into a shape table was not immediately clear. Two procedures are possible: you can store all of the input data in some sort of two-dimensional array in memory and then analyze it, or you can take the input data as they are acquired and develop the shape table on the fly. I seriously considered the first path, and in fact, wrote a program that would translate the input pattern into a matrix of zeroes and ones. Further consideration showed that analysis of the pattern would be difficult, one of the major problems being that of ensuring proper plotting of the shape with respect to its starting point, i.e., justification. Moreover, the most efficient approach in terms of processing time and storage requirements for the shape table is to confine generation of the plotting vectors to the occupied cells of the grid as much as possible.

Such pattern tracing on an arbitrary two dimensional array presents a formidable search problem, particularly with disconnected patterns. The solution of the problem of efficiently tracing the input pattern was obvious as soon as I realized that the keystrokes used by a person entering the pattern on the grid constituted a continuous record of the pattern. By analyzing the keystroke pattern, I could produce a string of equivalents. The inspiration for this may be traceable in part to my knowledge of the way in which chemical structures are recorded at Chemical Abstracts Service of the American Chemical Society, where chemical typewriters, used for creating chemical structures, are connected to computers which record the keystrokes of the operator entering the structure. The record of keystrokes can then be "played back" to reproduce the structure exactly as it was keyed in. With this basic approach decided upon, the outline of the required algorithm became clear:

1) Select the position in memory at which the shape table is to be stored.
2) Generate and display the working \((15 \times 15)\) grid.

3) Input the starting coordinates for the shape (required for justification).

4) Generate the proper 3-bit codes that represent the plotting vectors, based on the keystrokes used to input the pattern.

5) Assemble the 3-bit codes (in groups of two or three, depending upon Apple’s strictures) into a byte.

6) Store the assembled byte in the shape table.

7) Provide for proper finishing-off of the current byte when the Quit key is hit.

8) Add an end-of-record mark (a zero byte) required by Apple as a shape terminator.

9) Store the table.

Most of these steps are straightforward, but two of them—generation of the 3-bit codes that represent plotting vectors, and their assembly into bytes (steps 4 and 5, above)—require further elaboration.

In Applesoft BASIC, the character returned by a keystroke is accessible with a "GET" command; the instruction \texttt{GET KEYS} will load the character accessed by the next keystroke into the variable \texttt{KEYS}. We may examine \texttt{KEYS} to determine whether it contains a D, L, U, or R and then do a table look-up (using the definitions in figure 4) to retrieve the decimal value associated with the direction implied by the keystroke. Each decimal value, of course, as stored in memory will generate the proper 3-bit binary code. Subsequently, the keystroke preceding the current one (which we thoughtfully saved in variable \texttt{KSVE$}) is examined. If \texttt{KSVE$} is a "P", then the current 3-bit code must represent a plot-then-move vector and decimal 4 as added to the decimal factor for the current key. If \texttt{KSVE$} is not a "P", then the current decimal key equivalent remains unaltered.

Assembly of the 3-bit codes into bytes involves only basic consideration of decimal to binary conversion. Byte assembly is done in the program as each 3-bit code becomes available, but for the purposes of discussion, let us assume that 3-bit codes, \(V_1, V_2, V_3\) are available in that order from the last three keystrokes. The first 3-bit code initializes the byte:

\[
V_1, \\
\text{BYTE} = V_1, \\
00000XXX
\]
The second 3-bit code must be added to the byte, but must first be left-shifted three bits if the \( V_1 \) bits already present are to remain unchanged. This is done by multiplying \( V_2 \) by 8:

\[
\text{BYTE} = \text{BYTE} + 8 \times V_2
\]

Now for \( V_3 \). To refresh your memory, you will observe in figure 4 that all plot-then-move 3-bit codes have their left-most bits "on." Since there are only two bits remaining unfilled in the byte, there is no way in which the plot status of the third 3-bit code can be entered into the byte. In this case, processing of the byte stops, and it is stored in the shape table, while \( V_3 \) is used to initialize the next byte. This is the reason that plotting vectors cannot be stored as end vectors in a byte, one of Apple's restrictions previously noted. In similar fashion, if \( V_3 \) corresponds to a move-up vector, with all bits zero, it is not loaded into the current byte, but is used to initialize the next byte. The reason for this is not so obvious, but is related to the aforementioned deduction that plotting vectors cannot appear as end vectors in the byte. Suppose that the zero move-up vector \( V_3 \) could be stored as an end vector; then everytime \( V_3 \) happened to be a plotting vector, the last two bits in the byte would be a zero, and undesired up-moves would be enabled whenever a plot-then-move vector happened to occur in \( V_3 \). Apple's restrictions make sense!

In the event that \( V_3 \) is neither a move-up nor a plot-then-move vector, it is added to the byte. Then it consists of an unambiguous two-bit code (figure 4) that can fit into the remaining two bits of the byte. Addition of \( V_3 \) requires a 6-bit left shift of \( V_3 \) to avoid changing the bits already present. This is done by multiplying \( V_3 \) by \( 64 = 2^6 \):

\[
\text{BYTE} = \text{BYTE} + 64 \times V_3
\]

<table>
<thead>
<tr>
<th>Plotting Vectors</th>
<th>3-Bit Codes</th>
<th>Decimal Equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \downarrow )</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>( \downarrow )</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 4: Representation of Plotting Vectors as 3-bit Codes and Decimal Equivalents.
Earlier, I mentioned glitches designed into Apple's shape procedure that would offer problems in obtaining correct shapes in graphics. There are actually two kinds of glitches—one predictable and the other not. The predictable one is a consequence of two facts: 1) Apple uses a zero byte as an end-of-record mark to terminate every shape; 2) the move-up vector is represented by a 3-bit code of 000. It follows that several move-up vectors in a row will generate an end-of-record mark and any part of the shape following thereafter will be forgotten. That's bad enough. Worse is the unexpected fact that move-up codes (000) that lie on the left part of the byte [most significant bits] are not recognized. For example, consider the two cases of a plot-then-move right command followed by a move-up command,

00000101 (decimal 5)

and a move-up command followed by a plot-then-move right command,

00101000 (decimal 40).

Presumably, these commands should give the same net result. That's what you think, and what I thought also! In fact, the move-up command implied in the left bits of decimal 5 is not recognized by the system, and the byte is interpreted as a plot-then-move right instruction only. Therefore, if you try to generate a 45° line with the sequence

plot-then-move-right: move-up: plot-then-move-right: move-up...

you will get a horizontal line, whereas the sequence

move-up: plot-then-move-right: move-up: plot-then-move-right...

will give the desired 45° line! There is nothing in Apple's literature that would lead the unwary to suspect that these two sequences will not plot alike. Now you know the source of those misshapen shapes.

The two problems described in the preceding paragraph—premature end-of-record mark and non-plotting up-vectors that appear in the left bits—arise from the definition of the up-vector as a zero 3-bit string. In fact, a concise statement of the problem is that any byte with a value less than decimal 8 can be expected to misbehave, unless it is the last byte in the shape table.

The solution to the problem lies in preventing the occurrence of these dubious bytes. This can be done easily—especially with a computer program—by introducing dummy right-and left-moves. The technique is simple: check the value of the assembled byte; if it is less than decimal 8, the second vector in the byte must correspond to the move-up (000) vector. In that case, replace the leftmost zero bits by a non-zero, move-right vector, transfer the move-up (000) vector to the next byte and follow it by a move-left vector. By placing the move-up (000) vector into the right-most three bits of the next byte, you ensure that it will be recognized as an up-vector. The succeeding move-left vector undoes the effect of
the move-right vector installed in the preceding byte so that the correct shape is maintained. Implementation of this routine in a computer program is actually quite easy, and resolves the problems introduced by the up-vector. Frankly, I don't see how anyone could be expected to obtain predictable shapes from Apple's procedure using hand-methods for creating shape tables, considering the inherent problems posed by the zero up-vector.

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>N, No. shapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 2</td>
<td>Empty</td>
</tr>
<tr>
<td></td>
<td>Directory</td>
</tr>
<tr>
<td></td>
<td>First Shape</td>
</tr>
</tbody>
</table>

2*N Bytes

Fig. 5: Memory Map for Shape Table
The Program(s)

Three programs were written to implement the computer-guided formulation of a shape table: A shape file initialization program (SHAPE 1), a shape creating program (SHAPE 2), a shape display program (figure 8). These will be discussed briefly. I hope that the following discussions coupled with the comments scattered through the programs will enable you to follow the programs without difficulty.

Shape File Initialization

The principle shape-creating program requires a previously allocated disk file for shape table storage. The initialization program (SHAPE 1) creates the disk file and also establishes the name and length of the file. The program allocates space for the shape table directory based on the number of shapes to be stored in the file, a number that is declared by you during initialization. The memory map for a shape table is stored in the first byte of the table; its maximum value is therefore 255, and this is the maximum number of shapes that can be stored in one shape table. The directory contains addressing information that allows random access to any shape in the table.

The directory falls between the first byte of the table and the beginning of the first shape. The amount of space allocated to the directory is determined by the number of shapes ultimately to be stored in the table; each shape requires two bytes in the directory for addressing. The shape tables themselves may be any length, up to a total length consistent with the $15 \times 15$ matrix in which the shapes are created. The shape tables are stored end-to-end as they are added to the file, each shape table ending in a zero byte as end-of-record mark. The layout of the shape file requires that any tables added to the file be accurately done, because once a table is buried in the file, it cannot be simply replaced unless the replacement has precisely the same length.

The file initialization program is also used for creating the cursor required for mapping shapes on the $15 \times 15$ working grid produced by the principal program. This relieves the user of the need to generate the cursor himself everytime he opens a new shape file. The cursor is stored as the first shape in the shape file, and the shape-creating program assumes that the cursor has already been stored for its use. As a consequence of this arrangement, you must remember that the user-generated shapes start with the second shape table in the file.

Although the file initialization program zeroes out all of the bytes in the directory, there is no substantial reason for doing this, except that the string of zero bytes makes it easy to determine where the directory ends and the shape tables begin in a memory dump. This advantage will last only until the directory is filled.

The Shape Creating Program

The BASIC program (SHAPE 2) that enables shape generation requires the use of dual floppy disks, but can be easily changed for single floppy use by replacing
"D2" in step 110 by "D1." [Similar adjustments will have to be made in the initialization and display programs, which store and access the shape file from disk D2.] Tape users will have to replace disk I/O by suitable tape I/O in steps 100, 110 and 1360.

The program loads a pre-existing shape file (created by the initialization program, if necessary) from disk, using the shape file name supplied by you on request from the program. The file is loaded into a memory location which you are also asked for by the program. A check is made (step 220) that there is room in the shape file directory for another entry. If not, you will be so advised and the program will abort. A pointer to the shape file required by the Apple system is set up in step 260. The 15 x 15 plotting grid is turned on (steps 300-330) and you will be asked to input the starting grid coordinates for the shape. Note, these are grid coordinates and not screen coordinates that are asked for. The cursor will be displayed on the center of the grid square that you have just selected as the starting point. Some user helps are displayed in the text area under the grid (steps 410-440), and you are off and running.

Manipulation of the R, L, D, and U keys will move the cursor in the appropriate directions. The REPEAT key will work with these commands. Pressing the P key will plot a small circle inside the square in which the cursor currently resides, and this plotted point will become part of the shape table being built in memory. An image of the cursor will persist in the initial square—as a “negative” image if you happened to plot at that square. The persistent cursor image serves as a reminder to you of the location of the start of the shape. The cursor is made to disappear and reappear in adjacent squares as you press the move keys by XDRAW commands at steps 500 and 530; the IF statement at step 1040 in the subroutine that draws the plotting circle is responsible for keeping the persistent image of the cursor at the starting square. The flag, FLAG, that appears in step 480 and elsewhere is used to allow the cursor to be turned off in a plotted square and to be turned on again when the cursor moves to the next square.

Keystrokes are recorded in step 570. A previous step (550) saves the previous two keystrokes in KI$ and KSVE$. The former record, KI$, is required to allow the erase feature, controlled by the E key and discussed below. KSVE is needed for proper generation of plot-then-move 3-bit codes, also discussed below. Interpretation of a keystroke takes place in steps 590-710, a sequence of IF's called a sieve. This particular form of key screen was chosen because it gives almost complete protection against inadvertent entry of incorrect keys. Once you are in the program, you will find that the keyboard is effectively locked out for all keys except those required by the program. If a non-applicable key is pressed, the sieve eventually routes the program through step 710 back to another key access at step 570. Inside the sieve, when a keystroke has been identified as a move command (L,R,U,D), the appropriate X- or Y- coordinate adjustment is made and the decimal value of the 3-bit code applicable to the move is stored where the variable KSVE$ is checked to see if the previous keystroke was a Plot command. If it was, SYMBOL is incremented by a 4 (remember figure 4?), and SYMBOL is then transmitted to the byte assembly area. More of this later.
If the current keystroke corresponds not to a Move command, but to a Plot command, the program sets the cursor disable flag, FLAG, calls the plot subroutine and then branches back to get the next keystroke (all of this is done in step 680). The Quit command forces a branch to a routine that closes out the current byte (starting at step 1080), adds a record mark (step 1170) and draws the completed shape (step 1170). At this juncture, you are asked a series of questions, the answers to which will allow you to:

1) forget the current shape and go back and try again without re-accessing the current shape file from disk;
2) keep the current shape, update the shape file directory and start a new shape;
3) forget the whole thing—add no new shapes to the file and quit;
4) load an updated shape file to disk and quit.

These alternatives will help you to avoid filling up the shape table with unwanted shapes, and allow you to experiment without being forced to save all of your experiments.

The closing out of the current byte preparatory to ending the current shape definition (step 1080) poses a problem if the last keystroke is a Plot command because a P command alone does not generate a vector. There is nothing to store after a final P command, unless it is followed by some sort of move. The problem is handled in steps 1100-1140 by adding an arbitrary up-move after a final Plot command to generate a plot-then-move-up vector. [Note that in figure 2, the concluding vector is a plot-then-move-down. This was done for the sake of clarity in drawing only. The point is mentioned in case some unusually perceptive reader notices that the foregoing description does not tally with the example in figure 2.) The final vector is either added to the current byte, in which it will appear as the only entry. If the last keystroke prior to closing the current shape table is anything other than a Plot command, the current byte can be closed out immediately without further ado.

The erase command has the very limited capability of erasing the last Plot command only. As discussed before, a Plot command alone does not result in formation of a vector until it is followed by a command. Therefore, if a Plot command is issued in error and no move command follows it, no vector will be generated and the shape table remains unchanged at this point. It is therefore possible to undo the Plot command simply, without the complication of analyzing the last byte for returning to the state that preceded the mistaken command (and it would be complicated!). At the point at which the Plot command is mistakenly issued, KSVE$ has a certain value. If we wish to go back to the condition prior to the mistaken Plot command, we must restore that value to KSVE$ so that when the correct command is issued it is properly interpreted when KSVE$ is examined subsequently. The character required for this purpose lies waiting in KI$. Thus, the erase command loads this previous value into KSVE$ and "unplots" the incorrect plotting circle by re-plotting with the color "black" [HCOLOR = 0 in step 720). Note that because of these limitations, no plot command can be undone after a move has been made.
Byte assembly using the 3-bit codes (stored currently in SYMBOL) occurs in 780-980. The variable CYCLE keeps track of the number of 3-bit codes entered into the current byte (called BYTE in the program). After the second 3-bit code is loaded into BYTE (step 820) a check is made (step 840) to see if the byte is less than 8; if it is, we know that the byte contains an unrecognizable move-up vector in the left five bits. In that case, a dummy move-right 3-bit code is inserted into the byte, the byte is stored (step 860) and a new byte is formed consisting of the required move-up (000) followed by a dummy move-left (110) to compensate for the dummy move-right. The resulting byte contains the bit string 0001 1000, decimal 24, generated in step 880. Statements 950-980 take care of the cases in which the third 3-bit code is a plot-then-move code or a move-up only code, which require that the current byte be stored, and the current 3-bit code be loaded into the next byte.

The Display Program

It is likely that your disk or tape will be replete with shape files tailored to various uses, now that creating shape tables is so easy. A convenient display program will become essential in order to find out which shapes are stored where. The display program that accomplishes this (figure 8) is an example of how shape files may be used in a program. The program constructs a 6 × 6 grid on the high resolution screen and displays one shape per grid cell. To identify the location of the shapes in the shape table, each occupied cell carries the shape index in the upper left-hand corner. The numerals required for plotting these indices are extracted from a shape table called NUMERALS that you will have to create at storage location 20000 (decimal) by means of the shape creating program. The numerals are restricted to a 5 × 7 grid, and are formatted as illustrated by the example in figure 1. Sufficient space is reserved in the display squares to accommodate three-digit numerals from 1 through 255. "Aha," you ask, "how can 255 shapes be displayed in a 6 × 6 grid?" The program provides for paging through the shape table, 36 shapes at a time. The paging is activated by hitting any alphanumeric key on the Apple keyboard.

The display program opens by getting the shape files that it needs—one for numerals (step 50) and the table to be displayed (step 90). Pointers to the tables are set up (steps 70 and 120). Starting at step 180, each shape I is accessed in a FOR...NEXT loop. A grid-specific index is calculated (step 190) by taking the current shape index I modulo 36 (step 190). For the first shape in each group of 36 (I modulo 36 = 1), the screen is cleared (step 240) and the 6 × 6 grid is displayed (steps 250-330). The row and column positions for the I—the shape in the grid are found (steps 360, 370). The shape index is "unpacked" into its separate digits (steps 380-410) and these digits are plotted in the correct grid cell in the upper left-hand corner (steps 430-480). The NUMERALS shape table is accessed in step 420 by placing the pointer to the NUMERALS shape table in (decimal) addresses 232 and 233, so that subsequent DRAW commands will refer to this table. In similar fashion, when the shapes to be plotted are required, the address of the shape table must be entered into addresses 232, 233. This program illustrates how any number of shape tables may be used inside a program simply by supplying the correct pointers at the time that shapes are to be DRAWn XDRAWn.
Parting Words

The 15 x 15 grid used for shape creation is the largest practical size for the Apple screen with space provided for text. A larger grid can be accommodated by eliminating the text area, but this will compromise the required starting coordinate input. However, the number of cells could be increased by decreasing cell size and using a smaller plotting figure. If you try this, it is convenient to select a plotting grid with odd numbers of X and Y segments so that the central plotting area falls on a grid square and not at the intersection of two grid lines. This is of help in centering shapes.

You should also be aware, if it is not obvious by now, that the location of a shape on the grid has no bearing on where it plots in high resolution graphics, except with regard to the initial point of the shape, which alone determines justification. You may use any convenient subsection of the full grid for plotting, and it does not have to be the same subsection for each shape.
REM ***************************
REM * *
REM * HOW TO DO A SHAPE TABLE *
REM * JOHN FIGUERAS *
REM * SHAPE2 *
REM * COPYRIGHT (C) 1981 *
REM * MICRO INK, INC. *
REM * CHELMSFORD, MA 01824 *
REM * ALL RIGHTS RESERVED *
REM *
REM ***************************

REM PRINT TAB(6);"****CREATE A SHAPE TABLE****"
PRINT
PRINT TAB(5);"J. FIGUERAS, ROCHESTER, N.Y.;"
PRINT
PRINT TAB(17);"9112179;"
PRINT
REM INPUT "SHAPE TABLE NAME ";NAME$
REM DISK ACCESSES USE DISK D2
D$ = CHR$(4): PRINT D$;"NOMON C,I,O"
PRINT D$;"BLOAD +NAME$+,A+,STR$(ASVE)+",,V0,D2"

REM GET CAPACITY MAX OF FILE
MAX = PEEK(ASVE+2) + 256 * PEEK(ASVE+3)
MAX = (MAX - 2) / 2
REM GET NO. OF SHAPES IN TABLE
N = PEEK(ASVE)
REM GET FILE LENGTH
INDEX = PEEK(ASVE+2*N+2) + 256 * PEEK(ASVE+2*N+3)
REM COMPUTE ADDRESS OF NEXT FREE BYTE
ADDR = ASVE + INDEX
REM CLEAR TEXT AND GET INITIAL PLOT COORDS
PRINT: PRINT: PRINT: PRINT "ENTER STARTING COORDS"
INPUT "X ";X:X = 10 * X - 5
INPUT "Y ";Y:Y = 10 * Y - 5
DRAW 1 AT X,Y:XS = X:YS = Y
Figueras  How to Do a Shape Table  93

540 REM SAVE LAST TWO KEYSTROKES. KIS IS NEEDED FOR ERASE R OUTLINE
550 KIS = KSVES: KSVES = KEYS
560 REM GET NEW KEYSTONE
570 GET KEYS
580 REM GO TO SIEVE TO GET 3-BIT PLOT VECTOR FROM KEYS AND KSVES
590 IF KEYS < > "U" THEN 610
600 SYMBOL = 0: Y = Y - 10: GOTO 760
610 IF KEYS < > "R" THEN 630
620 SYMBOL = 1: X = X + 10: GOTO 760
630 IF KEYS < > "D" THEN 650
640 SYMBOL = 2: Y = Y + 10: GOTO 760
650 IF KEYS < > "L" THEN 670
660 SYMBOL = 3: X = X - 10: GOTO 760
670 IF KEYS < > "P" THEN 690
680 FLAG = 1: GOSUB 1000: GOTO 530
690 IF KEYS = "Q" THEN 1080
700 REM NEXT STATEMENT PROTECTS FROM KEYING ERROR
710 IF KEYS < > "E" THEN 570
720 HCOLOR = 0: FLAG = 0: GOSUB 1000
730 REM SET UP PRE-PLOT STATUS
740 KSVES = KIS: HCOLOR = 3: GOTO 500
750 REM ADJUST 3-BIT VECTOR FOR PLOT
760 IF KSVES = "P" THEN SYMBOL = SYMBOL + 4
770 REM LOAD 3-BIT VECTOR INTO BYTE
780 CYCLE = CYCLE + 1
790 IF CYCLE < > 1 THEN 810
800 BYTE = SYMBOL: GOTO 480
810 IF CYCLE < > 2 THEN 900
820 BYTE = BYTE + 8 * SYMBOL
830 REM PROTECT AGAINST PREMATURE END-OF-RECORD
840 IF BYTE > 7 THEN 480
850 REM ENTER DUMMY RIGHT MOVE AND STORE BYTE
860 BYTE = BYTE + 8: POKE ADDR, BYTE: ADDR = ADDR + 1
870 REM ENTER UP MOVE AND DUMMY LEFT MOVE IN NEW BYTE
880 BYTE = 24: CYCLE = 2: GOTO 480
890 REM ID THIRD 3-BIT VECTOR IS A MOVE ONLY, FINISH BYTE; ELSE LOAD BY
900 IF SYMBOL > 3 THEN 930
910 BYTE = BYTE + 64 * SYMBOL
920 REM STORE BYTE
930 POKE ADDR, BYTE: ADDR = ADDR + 1
940 REM STORE 3-BIT VECTOR IN NEXT BYTE IF NEEDED
950 IF SYMBOL = 0 OR SYMBOL > 3 THEN 980
960 REM PREPARE FOR NEXT BYTE. GET NEXT 3-BIT VECTOR
970 CYCLE = 0: GOTO 480
980 CYCLE = 1: BYTE = SYMBOL: GOTO 480
990 REM PLOT ROUTINE
1000 FOR Y2 = Y - 3 TO Y + 3 STEP 6: HPLOT X - 1, Y2 TO X + 1, Y2: NEXT
1010 FOR Y2 = Y - 2 TO Y + 2 STEP 4: HPLOT X - 2, Y2 TO X + 2, Y2: NEXT
1020 FOR Y2 = Y - 1 TO Y + 1: HPLOT X - 3, Y2 TO X + 3, Y2: NEXT
1030 REM TURN OFF CURSOR IN PLOTTED SQ.
1040 IF X = XS AND Y = YS THEN RETURN
1050 XDRAW 1 AT X, Y: RETURN
1060 REM PREPARE BYTE FOR QUIT
1070 REM CLOSE OUT BYTE FOR MOVE-ONLY
1080 IF KSVES < > "F" THEN 1150
1090 REM USE PLOT-THEN-UP VECTOR TO END
1100 IF CYCLE < > 2 THEN 1120
1110 POKE ADDR, BYTE: ADDR = ADDR + 1
1120 IF CYCLE < > 1 THEN 1140
1130 BYTE = BYTE + 32: GOTO 1150
1140 BYTE = 4
1150 POKE ADDR, BYTE: ADDR = ADDR + 1
1160 REM ADD RECORD MARK. DISPLAY NEW SHAPE
1170 POKE ADDR, 0: ADDR = ADDR + 1: XDRAW N AT 200, 75
1180 INPUT "SAVE SHAPE? Y/N "; KIS
1190 IF KIS = "Y" THEN 1220
1200 N = N - 1: GOTO 180
1210 REM GET INDEX FOR NEXT FREE BYTE
1220 N = N + 1: ADDR = ADDR + 16
1230 IF N < MAX THEN 1270
1240 PRINT "WARNING: TABLE FULL WITH THIS SHAPE"
1250 IF N > MAX THEN 1310
1260 REM STORE INDEX IN DIRECTORY

How to Do a Shape Table  93

Figueras
1270 POKE ASVE + 2 * N, ADDR - 256 * INT (ADDR / 256)
1280 POKE ASVE + 2 * N + 1, INT (ADDR / 256)
1290 INPUT "DONE? Y/N "; KI$
1300 IF KI$ = "N" THEN 160
1310 INPUT "SAVE TABLE? Y/N "; KI$
1320 REM RESPONSE PROTECTED AGAINST RANDOM KEY HIT
1330 IF KI$ = "Y" THEN 1360
1340 IF KI$ = "N" THEN 1370
1350 GOTO 1310
1360 PRINT DS; "SAVE" + NAMES + ",A" + STRS (ASVE) + ",L" + STRS (ADDR)
1370 END

10 REM ***************
12 REM *
14 REM * HOW TO DO A SHAPE TABLE *
16 REM * JOHN FIGUERAS *
17 REM *
18 REM * SHAPE3 *
20 REM *
22 REM * COPYRIGHT (C) 1981 *
23 REM * MICRO INK, INC. *
24 REM * CHELMSFORD, MA 01824 *
25 REM * ALL RIGHTS RESERVED *
26 REM *
27 REM ***************
28 REM
30 REM **** DISPLAY SHAPE TABLE ****
32 REM LOAD NUMERALS SHAPE FILE
35 PRINT : PRINT : PRINT "Hit any key for each page of table"
40 DS = CHRS (4) : PRINT DS; "NOMON C,I,o"
50 PRINT DS; "LOAD NUMERALS,A20000,D2"
60 REM SET UP POINTER TO NUMERALS
70 NHI = 78:NL = 32
80 REM GET TABLE FOR DISPLAY
90 INPUT "SHAPE TABLE NAME "; NAME$
100 INPUT "STARTING ADDRESS "; ADDR
110 REM SET UP POINTER TO SHAPE TABLE
120 AHI = INT (ADDR) : ALO = ADDR - 256 * AHI
130 REM GET NO. OF SHAPES FOR DISPLAY
140 NN = PEEK (ADDR)
150 REM INITIALIZE SCREEN
160 HGR : POKE - 16302,0
170 HCOLOR = 2: SCALE= 1: ROT= 0
180 FOR I = 1 TO NN
190 IMOD = I - 36 * INT (I / 36)
200 IF IMOD < > 1 THEN 350
210 GET KEYS
220 REM SCLEAR SCREEN AND CREATE GRID
230 REM GRID WILL HOLD 36 SHAPES
240 CALL 62450
250 HPL0T 0,0 TO 269,0 TO 269,180 TO 0,180 TO 0,0
260 FOR L = 45 TO 269 STEP 45
270 FOR J = 0 TO 180 STEP 10
280 HPL0T L,J
290 NEXT J: NEXT L
300 FOR L = 30 TO 180 STEP 30
310 FOR J = 0 TO 269 STEP 45
320 HPL0T J,L
330 NEXT J: NEXT L
340 REM CALCULATE GRID SQUARE COORDS.
350 IF IMOD = 0 THEN IMOD = 36
360 ROW = INT ((IMOD - 1) / 6)
370 COL = IMOD - 6 * ROW - 1
380 C1 = INT (I / 100)
Ed. note: In order for SHAPE 3 to work correctly, you will have to create a shape table called "Numerals." Remember that this will be an 11 entry table, with the numerals filling slots 2-11 (since slot 1 is filled with the 'cursor' — see article). Use SHAPE 1 to locate "Numerals" at 20000, and use SHAPE 2 to fill it. Put the numeral "1" in first, and work up to numeral "0."
Define Hi-Res Characters for the Apple II

by Robert F. Zant

The Apple contributed software bank, Volume 3, contains a very interesting and useful high resolution character generator. The hand method described to generate the character table, unfortunately, is somewhat less than exciting. The following routine relieves you of the burden of the hand method, and allows you to exploit the generator to the maximum.

The characters are represented in the table in a coded, reverse-image format. The code is based on a 7 by 8 dot matrix representation for each character. The format for an "L" is depicted below. Note that a border is left at the top and side so that characters will be separated on the screen.

```
...*.*
.*...*
.*...*
.*...*
.*...*
.*...*
.*...*
```

The coded table entry is derived from the format by substituting a zero for each dot, and a one for each asterisk. Each line of the matrix is thereby coded into one byte. The high order bit is set to zero in each byte. Eight bytes are required to encode each character. The code for the "L" depicted above would be

```
02,02,02,02,42,7E,00
```

The following program assists in defining characters and substituting them into the character table. Each character is defined in a regular dot matrix format, rather than in reverse-image. The program automatically calculates the binary code for the equivalent rotated version. The letter "L" would be entered as:
Note that the dot matrix must remain intact, and must contain only dots and asterisks. The command to store the character, the CTRL S, must be entered after the matrix, on the ninth line. A carriage return is required after each command.

At the beginning of the run, the operator specifies the table position (0 to 127) for the first character to be defined. Thereafter, characters are automatically stored at succeeding locations in the table. Separate runs of the program can be used to define characters in non-contiguous table locations.
10 REM ******************************
12 REM *
14 REM * DEFINE HI-RES CHARACTERS *
16 REM * ROBERT F. ZANT *
18 REM *
20 REM * CHARACTERS *
22 REM *
24 REM * COPYRIGHT (C) 1981 *
25 REM * MICRO INK, INC. *
26 REM * CHELMSFORD, MA 01824 *
27 REM * ALL RIGHTS RESERVED *
28 REM *
29 REM ******************************
30 REM
100 TEXT : CALL - 936
200 VTAB 5: PRINT "ENTER DECIMAL EQUIVALENT"
300 PRINT "OF FIRST 'ASCII' CHARACTER"
350 PRINT "(MAXIMUM VALUE OF 127)"
400 INPUT B
425 IF B > 0 AND B < 128 THEN 450: PRINT "RE-ENTER": GOTO 400
450 B = 26624 + B ^ 8
500 CALL - 936
600 PRINT "CHANGE THE DOTS IN THE FOLLOWING MATRIX"
700 PRINT "TO ASTERISKS TO DESCRIBE A FIGURE."
750 PRINT "USE 'ESC C', 'ESC D', '-', '->' AND '<-' TO EDIT."
775 PRINT "(LEAVE DOTS THAT ARE NOT REPLACED)"
800 PRINT "ENTER A 'CTRL S' TO STORE THE FIGURE"
900 PRINT "ENTER A 'CTRL Q' TO QUIT"
1000 REM PRINT MATRIX
1100 VTAB 9
1200 FOR I = 0 TO 7
1300 PRINT "......"
1400 NEXT I
1500 VTAB 9
2000 REM GET INPUT CHARACTER
2100 CALL - 657
2200 IF PEEK (512) = 147 THEN 3000
2300 IF PEEK (512) = 145 THEN 9000
2500 GOTO 2000
3000 REM ENCODE CHARACTER
3050 A = B: REM SAVE BEGINNING OF CHARACTER
3100 REM LOOK THROUGH MATRIX
3200 FOR I = 1064 TO 1960 STEP 128
3250 C = 0
3300 FOR J = 0 TO 6
3400 IF PEEK (I + J) = 174 THEN 3700
3500 IF PEEK (I + J) < > 170 THEN 4000
3600 C = C + 2 ^ J
3700 NEXT J
3800 POKE B,C:B = B + 1
3900 NEXT I
3950 GOTO 1000
4000 REM ERROR IN MATRIX
4100 VTAB 20
4200 PRINT "MATRIX CONTAINS INVALID CHARACTER"
4250 PRINT "RE-ENTER";B = A
4300 FOR I = 1 TO 1000: NEXT I
4400 VTAB 20: CALL - 958
4500 GOTO 1500
9000 END
Apple II High Resolution
Graphics Memory Organization

by Andrew H. Eliason

This section on graphics would not be complete without some explanation of how the Apple displays its high resolution screens. The following informative article explains how the Apple's Hi-Res memory is displayed on the screen. A demonstration program is included which clarifies the concepts presented.

One of the most interesting, though neglected, features of the Apple II computer is its ability to plot on the television screen in a high resolution mode. In this mode, the computer can plot lines, points and shapes on the TV display area in greater detail than is possible in the color graphics mode (GR) which has a resolution of 40 x 48 maximum.

In the high resolution (Hi-Res) mode, the computer can plot to any point within a display area 280 points wide and 192 points high. While this resolution may not seem impressive to those who have used plotters and displays capable of plotting hundreds of units per inch, it is nonetheless capable of producing a very complex graphic presentation. This may be easily visualized by considering that a full screen display of 24 lines of 40 characters is "plotted" at the same resolution. An excellent example of the Hi-Res capability is included in current Apple II advertisements.

Why, then, has relatively little software appeared that uses the Hi-Res features? One of the reasons may be that little information has been available regarding the structure and placement of words in memory which are interpreted by Hi-Res hardware. This information is essential to users who wish to augment the Apple Hi-Res routines with their own, or to explore the plotting possibilities directly from BASIC. In a fit of curiosity and Apple-insomnia, I have PEEKed and POKEd around in the Hi-Res memory area. The following is a summary of my findings. Happy plotting!

Each page of Hi-Res Graphics Memory contains 8192 bytes. Seven bits of each byte are used to indicate a single screen position per bit in a matrix of 280H x 192V. The eighth bit of each byte is not used in Hi-Res and the last eight bytes of every 128 are not used.
[Note: Subsequent to the original publication of this article, Apple Computer began to produce machines which had the added capability of plotting two more colors, blue and orange, to the Hi-Res screen. If the previously unused most significant bit of any byte is set to 1, the hardware will generate the new colors (blue instead of violet, orange instead of green) for each of the other bits within the corresponding byte. If your Apple II is a "six color" machine, you will see the results of this when you enter a number between 128 and 255 into the program given in the article. It has also been discovered that the six color hardware will cause an orange dot to appear to the left (!) of column zero in the associated row, under some circumstances, when the sixth bit of the last byte of the eight "unused" bytes is set.]

The bits in each byte and the bytes in each group are plotted in ascending order in the following manner. First consider the first two bytes of page 1. (Page 2 is available only in machines with at least 24K).

<table>
<thead>
<tr>
<th>BYTE</th>
<th>8192</th>
<th>8193</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCREEN POSITION</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13</td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>0 1 2 3 4 5 6 0 1 2 3 4 5 6</td>
<td>V G V G V G V G V G V G</td>
</tr>
<tr>
<td>(Bit 7 not used)</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

V = VIOLET
G = GREEN

Figure 1 represents the screen position and respective bit and word positions for the first 14 plot positions of the first horizontal line. If the bit is set to 1 then the color within the block will be plotted at the position indicated. If the bit is zero, then black will be plotted at the indicated position. You can see that even bits in even bytes plot violet, even bits in odd bytes plot green and vice versa. Thus all even horizontal positions plot violet and all odd horizontal positions plot green. To plot a single white point, you must plot the next higher or lower horizontal position along with the point, so that the additive color produced is white. This is also true when plotting single vertical lines.

The memory organization for Hi-Res is, for design and programming considerations, as follows: Starting at the first word, the first 40 bytes (0-39) represent the top line of the screen (40 bytes × 7 bits = 280). The next 40 bytes, however, represent the 65th line (i.e., vertical position 64). The next 40 bytes
represent three lines at positions 8, 72 and 136, the next group at positions 16, 80 and 142, and so on until 1024 bytes have been used. The next 1024 bytes represent the line starting at vertical position 1 (second line down) in the same manner. Eight groups of 1024 represent the entire screen. The following simple program provides a good graphic presentation as an aid to understanding the above description. Note that there is no need to load the Hi-Res machine language routines with this program. Set HIMEM:8191 before you type in the program.

```
100 REM SET HIMEM:8191
110 REM Hi-Res GRAPHICS LEARNING AID
120 POKE -16304,0: REM SET GRAPHICS MODE
130 POKE -16297,0: REM SET Hi-Res MODE
140 REM CLEAR PAGE - TAKES 20 SECONDS
150 FOR I = 8192 TO 16383: POKE I,0: NEXT I
160 INPUT "ENTER BYTE (1 to 127)", BYTE
170 POKE -16302,0: REM CLEAR MIXED GRAPHICS
180 FOR J=8192 to 16383: REM ADDRESS'
190 POKE J,BYTE: REM DEPOSIT BYTE IN ADDRESS
200 NEXT J
210 POKE -16301,0: REM SET MIXED GRAPHICS
220 GOTO 160
999 END
```

An understanding of the above, along with the following equations will allow you to supplement the Hi-Res graphics routines for memory efficient programming of such things as: target games, 3D plot with hidden line suppression and 3D rotation, simulation of the low resolution C=SCRN (X,Y) function, etc. Also, you may want to do some clever programming to put Flags, etc., in the unused 8128 bits and 512 bytes of memory!

**Hi-Res Graphics Equations and Algorithms**

Where:

- \( FB \) = ADDRESS OF FIRST BYTE OF PAGE.
- \( FB \) = ADDRESS OF FIRST BYTE OF PAGE.
- \( PAGE \, 1 = 8192 \quad PAGE \, 2 = 16384 \)
- \( LH \) = HORIZONTAL PLOT COORDINATE. 0 TO 279
- \( LV \) = VERTICAL PLOT COORDINATE. 0 TO 191
- \( BV \) = ADDRESS OF FIRST BYTE IN THE LINE OF 40
- \( BY \) = ADDRESS OF THE BYTE WITHIN THE LINE AT BV
- \( BI \) = VALUE OF THE BIT WITHIN THE BYTE WHICH CORRESPONDS TO THE EXACT POINT TO BE PLOTTED.

Given: \( FB,LH,LV \)

\[
BV = LH \mod 8 \times 1024 + (LV/8) \mod 8 \times 128 + (LN/64) \times 40 + FB
\]

\[
BY = LH/7 + BV
\]

\[
BI = 2 \times (LH \mod 7)
\]
To Plot a Point (Without Hi-Res Plot Routine):

\[
\begin{align*}
\text{LH} &= X \mod 280 : \text{LV} = Y \mod 192 \text{ (OR)} \\
\text{LV} &= 192 - Y \mod 192 \\
\text{FB} &= 8192 \\
\text{BV} &= \text{LV} \mod 8 \times 128 + (\text{LV}/64) \times 40 + \text{FB} \\
\text{BY} &= \text{LH}/7 + \text{BV} \\
\text{BI} &= 2 \times (\text{LH} \mod 7) \\
\text{WO} &= \text{PEEK} (\text{BY}) \\
\text{IF (WO/BI) MOD 2 THEN RETURN} \\
\text{POKE BY, BI + WO} \\
\text{RETURN}
\end{align*}
\]

To Remove a Point, Substitute:

\[
\begin{align*}
\text{IF (WO/BI) MOD 2 = 0 THEN RETURN} \\
\text{POKE BY, WO-BI}
\end{align*}
\]

To Test a Point for Validity, the Statement:

"IF (WO/BI) MOD 2" IS TRUE FOR A PLOTTED POINT AND FALSE (= 0) FOR A NON PLOTTED POINT.
5 EDUCATION

Introduction 104

Apple Pi 105
   Robert J. Bishop

Sorting Revealed 109
   Richard C. Vile, Jr.

Solar System Simulation with or without an Apple II 134
   David A. Partyka

Programming with Pascal 143
   John P. Mulligan
Introduction

Over the past several years, the computer has clearly made its mark on the concept of "education." Most secondary school curricula now offer some sort of computer programming. And even more importantly, CAI (Computer Assisted Instruction) is being introduced in a wide variety of less mathematical subjects. Two virtues of the computer, patience and accuracy, make it well adapted to the task of education. This section should help you to appreciate these noble virtues of your Apple!

"Apple Pi" by Bob Bishop uses an Integer BASIC program to find the value of Pi to the 1000th decimal place. "Sorting Revealed" by Richard Vile discusses computer sorting, and offers five Integer BASIC programs which graphically demonstrate various types of sorts. "Solar System Simulation" by Dave Partyka discusses the motions of the planets and uses an Applesoft hi-res graphics program to show those motions. The last article in the section, "Programming with Pascal" by John Mulligan, provides an overview of Apple Pascal and several sample Pascal procedures. So explore the learning possibilities available with your Apple.
Apple Pi

*by Robert J. Bishop*

Did you ever want to know the value of Pi to 1000 decimal places? The following article briefly describes a method to calculate this value, and then implements the method using Apple Integer BASIC.

Everyone knows that the value of Pi is about 3.1416. In fact, its value has been known this accurately as far back as 150 A.D. But it wasn’t until the sixteenth century that Francisco Vieta succeeded in calculating Pi to ten decimal places.

Around the end of the sixteenth century the German mathematician, Ludolph von Ceulen, worked on calculating the value of Pi until he died at the age of 70. His efforts produced Pi to 35 decimal places.

During the next several centuries a great deal of effort was spent in computing the value of Pi to even greater precision. In 1699 Abraham Sharp calculated Pi to 71 decimal places. By the mid 1800's its value was known to several hundred decimal places. Finally, in 1873, an English mathematician, Shanks, determined Pi to 707 decimal places, an accuracy which remained unchallenged for many years.

I was recently rereading my old copy of Kasner & Newman's *Mathematics and Imagination* (Simon & Schuster, 1940), where I found the series expansion:

\[
\pi = \sum_{k=1}^{\infty} \frac{4(-1)^{k+1}}{(2k-1)239^{2k-1}} - \sum_{k=1}^{\infty} \frac{16(-1)^{k+1}}{(2k-1)5^{2k-1}}
\]

The book indicated that this series converged rather quickly but "... it would require ten years of calculation to determine Pi to 1000 decimal places." Clearly this statement was made before modern digital computers were available. Since then, Pi has been computed to many thousands of decimal places. But Kasner & Newman's conjecture of a ten-year calculation for Pi aroused my curiosity to see just how long it would take my little Apple II computer to perform the task.
Program Description

My program to compute the value of Pi is shown in listing 1. It was written using the Apple II computer's integer BASIC and requires a 16K system (2K for the program itself; 12K for data storage). The program is fairly straightforward but a brief discussion may be helpful.

The main calculation loop consists of lines 100 through 300; the results are printed in lines 400 through 600. The second half of the listing contains the multiple precision arithmetic subroutines. The division, addition, and subtraction routines start at lines 100, 2000, and 3000, respectively.

In order to use memory more efficiently, PEEK and POKE statements were used for arrays instead of DIM statements. Three such arrays are used by the program: POWER, TERM, and RESULT. Each is up to 4K bytes long and starts at the memory locations specified in line 50 of the program.

The three arrays mentioned above each store partial and intermediate results of the calculations. Each byte of an array contains either one or two digits, depending on the value of the variable, TEN. If the number of requested digits for Pi is less than about 200, it is possible to store two digits per byte; otherwise, each byte must contain no more than one digit. (The reason for this distinction occurs in line 1070 where an arithmetic overflow can occur when trying to evaluate higher order terms of the series if too many digits are packed into each byte.)

The program evaluates the series expansion for Pi until the next term of the series results in a value less than the requested precision. Line 1055 computes the variable, ZERO, which can be tested to see if an underflow in precision has occurred. This value is then passed back to the main program where, in line 270, it determines whether or not the next term of the series is needed.

Results

Figure 1 shows the calculated value of Pi to 1000 decimal places. Running the program to get these results took longer than it did to write the program! [The program ran for almost 40 hours before it spit out the answer.] However, it took less than two minutes to produce Pi to 35 decimal places, the same accuracy for which Ludolph von Ceulen spent his whole life striving!

Since the program is written entirely in BASIC it is understandably slow. By rewriting all or part of it in machine language its performance could be vastly improved. However, I will leave this implementation as an exercise for anyone who is interested in pursuing it.

Note: You must set HIMEM:5120.
0 REM ****************************
1 REM *
2 REM *
3 REM *
4 REM *
5 REM *
6 REM *
7 REM * COPYRIGHT (C) 1981 *
8 REM * MICRO INK, INC. *
9 REM * CHELMSFORD, MA 01824 *
10 REM * ALL RIGHTS RESERVED *
11 REM *
12 REM ****************************
13 REM
14 CALL -936: VTAB 10: TAB 5: PRINT "HOW MANY DIGITS DO YOU WANT ";
15 INPUT SIZE
16 CALL -936
20 TEN=10: IF SIZE>200 THEN 50
30 TEN=100: SIZE=(SIZE+1)/2
50 POWER=5120: TERM=6144: RESULT=7168
60 DIV=1000: ADD=2000: SUB=3000: INIT=4000: COPY=5000
70 DIM CONSTANT(2): CONSTANT(1)=25: CONSTANT(2)=239
100 REM MAIN LOOP
125 FOR PASS=1 TO 2
150 GOSUB INIT
200 GOSUB COPY
210 POINT=TERM: DIVIDE=EXP: GOSUB DIV
220 IF SIGN>0 THEN GOSUB ADD
230 IF SIGN<0 THEN GOSUB SUB
240 EXP=EXP+2:SIGN=-SIGN
250 POINT=POWER: DIVIDE=CONSTANT(PASS): GOSUB DIV
260 IF PASS=2 THEN GOSUB DIV
270 IF ZERO<>0 THEN 200
300 NEXT PASS
400 REM PRINT THE RESULT
500 PRINT : PRINT
510 PRINT "THE VALUE OF PI TO ";(TEN/100+1)*SIZE;" DECIMAL PLACES": PRINT
520 PRINT PEEK (RESULT);":
530 FOR PLACE=RESULT+1 TO RESULT+SIZE
540 IF TEN=10 THEN 570
560 IF PEAK (PLACE)<10 THEN PRINT "0";
570 PRINT PEEK (PLACE);
580 NEXT PLACE
590 PRINT
600 END
990 REM
1000 REM DIVISION SUBROUTINE
1010 DIGIT=0: ZERO=0
1020 FOR PLACE=POINT TO POINT+SIZE
1030 DIGIT=PEEK (PLACE) OR (QUOTIENT+RESIDUE)
1040 QUOTIENT=DIGIT/DIVIDE
1050 RESIDUE=DIGIT MOD DIVIDE
1055 ZERO=ZERO OR (QUOTIENT+RESIDUE)
1060 POKE PLACE,QUOTIENT
1070 DIGIT=TEN*RESIDUE
1080 NEXT PLACE
1090 RETURN
1200 REM
2000 REM ADDITION SUBROUTINE
2010 CARRY=0
2020 FOR PLACE=SIZE TO 0 STEP -1
2030 SUM= PEEK (RESULT+PLACE)+ PEEK (TERM+PLACE)+CARRY
2040 CARRY=0
2050 IF SUM<TEN THEN 2080
2060 SUM=SUM-TEN
2070 CARRY=1
2080 POKE RESULT+PLACE,SUM
2090 NEXT PLACE
2100 RETURN
2990 REM
3000 REM SUBTRACTION SUBROUTINE
3010 LOAN=0
3020 FOR PLACE=SIZE TO 0 STEP -1
3030 DIFFERENCE= PEEK (RESULT+PLACE)- PEEK (TERM+PLACE)-LOAN
3040 LOAN=0
3050 IF DIFFERENCE>=0 THEN 3080
3060 DIFFERENCE=Difference+TEN
3070 LOAN=1
3080 POKE RESULT+PLACE,Difference
3090 NEXT PLACE
3100 RETURN
3110 REM
4000 REM INITIALIZE REGISTERS
4010 FOR PLACE=0 TO SIZE
4020 POKE POWER+PLACE,0
4030 POKE TERM+PLACE,0
4040 IF PASS=1 THEN POKE RESULT+PLACE,0
4050 NEXT PLACE
4060 POKE POWER,16/PASS * 2
4070 IF PASS=1 THEN DIVIDE=5
4080 IF PASS=2 THEN DIVIDE=239
4090 POINT=POWER: GOSUB DIV
4100 EXP=1:SIGN=3-2*PASS
4110 RETURN
4990 REM
5000 REM COPY "POWER" INTO "TERM"
5010 FOR PLACE=0 TO SIZE
5020 POKE TERM+PLACE,PEEK(Power+PLACE)
5030 NEXT PLACE
5040 RETURN

THE VALUE OF PI TO 1000 DECIMAL PLACES:

3.14159265358979323846264338327950288419
7169399375105820974944592307816406286208
99966280348253421170679821480865132823066
4709384464695505800321172535940812848111745
02841027019385211055596446622948954930381
9644288109756659334461284756482337867831
6527120190914564856692346034861045432664
82133936077260249141273724587006606315588
1748815209209628292540917153643678925903
60011330530548820466521384146951941516160
9433057270365759591953092186117381932611
7931051185480744623799627495673518857527
2489122793818301194912983367336244065664
3086021394946395224737190702179860943702
7705392171762931767523846748184676694051
3200056812714526356082778577134275778960
9173637178721468404901224953430146549585
3710507922796892589235420199561121290219
6086403441815981362977477130996051870721
1349999998372978049951059731732816096318
5950244594553469083026425223082533446850
3526193118817101000313783875288658753320
8381420617177669147303598253490428755468
7311595628638823537875937519577818577805
321712268066130019278766119590921642019
89

Figure 1
Sorting Revealed

by Richard C. Vile, Jr.

The following article presents a truly fresh approach to understanding the basics of sorting. In addition to a discussion of various sorting methods, programs are presented that demonstrate the sorting algorithms in action.

It has often been said that a picture is worth a thousand words. Sadly, this maxim is frequently ignored by professional educators, especially when dealing with such bone-dry subjects as mathematics and computer science. This article will present a detailed example of the use of a simple, yet effective, visual technique for giving insight into the basis for certain algorithms. Our approach will be to show the algorithm in action. Our medium will be the Apple II personal computer, but any computer which provides a memory-mapped display will do. The vehicle for the demonstration will be one of the staples of the computer science curriculum — the joy of pedants and the bane of poor benighted students — viz. sorting algorithms.

Sorting Theory

Unfortunately, we must stoop to pedantry to begin with. The reader who is already well-versed in sorting lore may skip directly to Sorting Implemented.

Sorting is such a varied and vast topic that large portions of entire books have been devoted to it. Perhaps the best known compendium of sorting facts and theory is to be found in Knuth's robust volume Sorting and Searching (The Art of Computer Programming Vol. 111, Addison Wesley, 1973). Our demonstration will be limited to just a few of the better known sorting algorithms, although the techniques could be applied to others as well. We shall provide programs that allow the visualization of five different sorting algorithms: bubble sort, Shell sort, insertion sort, selection sort, and quicksort. Of these, we shall discuss the bubble sort and quicksort in some detail prior to the presentation of the programs. Details of the others may be found in almost any good introductory computer science text, as well as in most texts on data structures.
Apart from the specific details of the algorithms used, the theory connected with sorting deals with efficiency. When people who are "in the know" discuss sorting, they will frequently bandy about certain terminology which they don't bother to explain. In hopes of increasing the number of cognoscenti involved in such discussions, we shall now attempt to lay out some of the more common terms for you.

To simplify matters somewhat, let us assume that all of our sorting will take place entirely in memory. Sorting methods that involve storing intermediate stages on disk files or magnetic tape, so-called external sorts, will be beyond our scope, although presumably not beyond our ken. The objects to be sorted will be assumed to be numbers, either integer or floating point, stored in memory in an array of one dimension and of a given size. The size of the array being sorted will be a hit personality throughout the discussion, so we give it a name: N.

Number of elements to sort = N

To fully comprehend one of the definitions given later, it is necessary to indulge in a bit of mathematics. We shall need to understand two functions. In particular:

\[ \log_2 x = \text{base 2 logarithm of } x \]
\[ \lfloor x \rfloor = \text{floor of } x \]

Actually, we are interested in the combination of these functions as applied to the friendly value N:

\[ \lfloor \log_2 N \rfloor \]

i.e. the floor of the base 2 logarithm of N. Before you run screaming to the nearest math anxiety clinic, at least read the next few sentences of explanation.

Suppose you have a pile of N coconuts. [Why coconuts, you ask? Why not, we reply!] Think about the following process:

1. Subdivide the pile into two piles which are as nearly equal in size as possible.
2. Take the smaller of the two piles from step 1. If it consists of one coconut, then stop. Otherwise, repeat from step 1.

Now how many times did you do step 1? The answer is the value of \( \lfloor \log_2 N \rfloor \)!

So, without worrying about picky details, the floor of the base 2 logarithm of N is the number of times you can divide N by 2 and still retain a non-zero quotient. Figure 1 pictures a simple case.

An alternate way of thinking about the situation involves collecting coconuts. The procedure is as follows:

1. Begin with a single coconut.
2. If doubling the number, $k$, of coconuts which you already have would cause your total to exceed $N$ coconuts ($2k$ is greater than or equal to $N$), then stop.

3. Collect $k$ more coconuts, giving you $2k$, and repeat step 2 now thinking of the new total as the value of $k$.

Now how many times did you execute step 3? The answer will again be $\lfloor \log_2 N \rfloor$. Before you go on, try to convince yourself (without flying to Tahiti to collect real coconuts) the two procedures yield the same result.

We shall return to this value, the "coconut number", later.

To talk about the efficiency of any algorithm, we need some quantities that we can measure. For sorting algorithms, we concentrate on two: the number of comparisons and the number of interchanges.

A comparison occurs whenever a member of the collection of numbers is compared to something else: a value fished out of a hat, or another member of the collection. Thus, a statement such as IF $A[I] > A[I+1]$ THEN ... counts as a comparison, as well as IF $A[I] > \text{MAX}$ THEN ...

An interchange occurs whenever a member of the collection of numbers is moved from one place to another in the computer's memory, and possibly some other number takes its place. The classic interchange may be described by the sequence of three statements:

\[
\text{TEMP} = A[I] \\
A[J] = \text{TEMP}
\]

(assuming, of course, that $I \neq J$). Not all sorting algorithms use this classic form, but there is usually an easily identified interchange step whose repetition we can count.
Trying to count the number of comparisons and/or interchanges which take place during the course of execution of a sorting algorithm will give an approach to measuring the efficiency of that algorithm. In addition to comparisons and interchanges, there will also be overhead involved in a sorting algorithm: i.e. the computing time used in loop control, recursion, etc. This is more difficult to measure theoretically and is therefore usually deduced from empirical observations.

Being armed with a few terminological weapons, we may now attack some of the more familiar sorting buzz phrases. Assume we are speaking of the number of comparisons made during the execution of some sorting algorithm. Then we may speak of an \( N^2 \) sorting algorithm (pronounced N-squared). This means that "on the order of" \( N \times N \) comparisons will be made in the course of sorting an array of size \( N \). Well, that was relatively painless — at least as a definition! The interesting (painful) part comes when we try to prove that a given algorithm is an \( N^2 \) algorithm. We shall get to that in the next section.

Another phrase which is frequently encountered when casually "talking sorts" is: that's an \( N \log N \) sort (pronounced \( N \log N! \)). What that actually means is that the expected number of comparisons in carrying out the sorting algorithm for an array of size \( N \) is:

\[
N \times \lfloor \log_2 N \rfloor
\]

That is, \( N \) multiplied by the coconut number. Again, this is easy enough to say, but perhaps a bit harder to appreciate than the \( N^2 \) description. After all, why should we be concerned with these numbers, and what is the significance of the difference between them?

Consider briefly, table 1. It shows values for \( N \), \( N^2 \), \( \log_2 N \), and \( N \log_2 N \). Assuming that overhead is relatively constant, or at least negligible from one algorithm to the next, we see that there is an ever increasing difference between \( N^2 \) and \( N \log N \) (from now on, we assume that \( \log N \) means \( \log_2 N \)). To make the comparison more concrete, let us assume that a comparison costs \(.001\$\), and that we need to sort an array containing 1,048,576 numbers. Using an \( N^2 \) sort will cost \$10,995,116.27, whereas using an \( N \log N \) sort will only put us out \$209.72. Of course, a single comparison of two numbers on today's monster computers—or "big iron" as they are sometimes referred to in the trade—costs considerably less than \(.001\$. But even at \(.0000001\$\) per comparison—a rate of 10,000,000 comparisons per penny—the cost differential will be \( 2^4 \) for the \( N \log N \) sort—\$1,099.51 for the \( N^2 \) sort! With that kind of comparison, you can see why no commercially viable sorting package is going to use the \( N^2 \) sorting approach.

Some Sorting Algorithms

We now present two of the more well known sorting algorithms in some detail. We will attempt informally to prove that the first is an \( N^2 \) algorithm. The second algorithm discussed is an example of an \( N \log N \) algorithm, but we shall spare the reader any attempts at proof.
Table 1

<table>
<thead>
<tr>
<th>N</th>
<th>N^2</th>
<th>log N</th>
<th>N log N</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4096</td>
<td>6</td>
<td>384</td>
</tr>
<tr>
<td>128</td>
<td>16,384</td>
<td>7</td>
<td>896</td>
</tr>
<tr>
<td>256</td>
<td>65,536</td>
<td>8</td>
<td>2,048</td>
</tr>
<tr>
<td>512</td>
<td>262,144</td>
<td>9</td>
<td>4,608</td>
</tr>
<tr>
<td>1,024</td>
<td>1,048,576</td>
<td>10</td>
<td>10,240</td>
</tr>
<tr>
<td>2,048</td>
<td>4,194,304</td>
<td>11</td>
<td>22,528</td>
</tr>
<tr>
<td>4,096</td>
<td>16,777,216</td>
<td>12</td>
<td>49,152</td>
</tr>
<tr>
<td>8,192</td>
<td>67,108,864</td>
<td>13</td>
<td>106,496</td>
</tr>
<tr>
<td>16,384</td>
<td>268,435,456</td>
<td>14</td>
<td>229,376</td>
</tr>
<tr>
<td>32,768</td>
<td>1,073,741,824</td>
<td>15</td>
<td>491,520</td>
</tr>
<tr>
<td>65,536</td>
<td>4,299,967,296</td>
<td>16</td>
<td>1,048,576</td>
</tr>
<tr>
<td>131,072</td>
<td>17,179,869,184</td>
<td>17</td>
<td>2,097,152</td>
</tr>
<tr>
<td>262,144</td>
<td>68,719,476,736</td>
<td>18</td>
<td>4,718,592</td>
</tr>
<tr>
<td>524,288</td>
<td>274,877,906,944</td>
<td>19</td>
<td>9,961,472</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1,099,511,626,776</td>
<td>20</td>
<td>20,971,520</td>
</tr>
</tbody>
</table>

Bubble Sort

This algorithm is probably the most widely known and loathed by students of introductory computer science. Many an instructor has droned on about its properties to unwilling students of FORTRAN! For many of these students, it is their only taste of the vast menu of sorting techniques.

We assume that N elements, which we shall denote by $A(1), A(2), \ldots, A[N]$, are to be arranged in ascending order; in short, sorted. The bubble sort operates by making repeated "sweeps" through the array, causing various elements to "bubble — up" in the process. We shall see that for each sweep, at least one element is guaranteed to be positioned in its correct final slot in the array.

The heart of each sweep is the idea of comparing two adjacent entries in the array:

$$A[I] \quad A[I + 1]$$

If $A[I]$ has a greater value than $A[I + 1]$, then the two elements are known to be out of correct order and need to be swapped. This is accomplished by the use of the classic interchange, which we illustrate here in BASIC and Pascal in figure 2.

Now consider the iterations of this fundamental step which are necessary to bring the entire array into sorted order. First, suppose we are just beginning. Then we can make no assumptions about the sizes of the array elements, relative to their positions in the array. Thus, suppose we iterate the fundamental compare-maybe-swap step over values of $I$ ranging from 1 to N-1 (why not 1 to N?). That is, we will successively compare $A[1]$ and $A[2], A[2]$ and $A[3]$, and so on, until we reach $A[N-1]$ and $A[N]$. Positions of various elements will change through swapping. In particular, the largest numerical value in the original array is guaranteed to wind up in $A[N]$ after the sweep is completed. To convince yourself that this is true, ask: "If the largest value is originally in $A[I]$, then what other array entries will it be swapped with?"
The last paragraph has indicated that we can reach a picture such as that shown in figure 3, after one sweep of the array. What has been accomplished? We have partially sorted the original array. How much of the resulting array is now in correct order? One element — the last. Note that this is the same as the number of sweeps we have made. Now suppose we make a second sweep through the array, comparing $A[1]$ and $A[2]$, $A[2]$ and $A[3]$, etc. until we reach $A[N-2]$ and $A[N-1]$. It is not necessary to compare $A[N-1]$ and $A[N]$, since we know that $A[N]$ is already in its correct final position. Moreover, $A[N-1]$ is now also guaranteed to be the second largest element in the array, and therefore in its correct final position. Thus the original array has been divided into two pieces: the elements $A[1], A[2], \ldots A[N-2]$, still possibly unsorted, and the elements $A[N-1]$ and $A[N]$, both where they 'should be'. We have made two passes and put two elements in their correct positions.
Continuing this process by making passes through less and less of the array will cause more and more of the ‘tail end’ of the array to be in correct final order and leave less and less of the beginning of the array to still be sorted. Altogether it will take N-1 passes through the array to guarantee that it is totally sorted. The reason that it does not require N passes is that the last pass causes two elements to wind up in their correct places, instead of just one. Figure 4 gives both a BASIC and a Pascal version of the complete bubble sort algorithm.

BASIC

10 FOR I = 1 TO N-1
20 FOR J = 1 TO N-I
30 IF A(J) <= A(J+1) THEN 70
40 TEMP = A(J)
50 A(J) = A(J+1)
60 A(J+1) = TEMP
70 NEXT J
80 NEXT I

Pascal

for I := 1 to N-1 do
for J := 1 to N-I do
begin

Temp := A[J];
end;

Figure 4: Bubble sort algorithm in both BASIC and Pascal

Now let us see if we can count the number of comparisons that will be made. Each sweep through the array corresponds to one pass through the inner loop of the algorithm. The number of comparisons made will be the same as the value of the upper limit of this loop, which according to figure 4 is N-1. The value of I is varied by the outer loop and runs from 1 to N-1. Thus, there will be:

N-1 comparisons the first time through the loop.
N-2 comparisons the second time through the loop.
N-3 comparisons the third time through the loop.
...
N-(N-2) = 2 comparisons the (N-2)nd time through the loop
N-(N-1) = 1 comparisons the (N-1)st time through the loop.

The total number is therefore:

\[ (N-1) + (N-2) + ... + 3 + 2 + 1 \]

This number is known in mathematics as a ‘triangular’ number, and by a formula from algebra may be expressed solely in terms of N as \( \frac{1}{2} N^2 - N \). Consequently, there are about \( N^2 \) comparisons made.
The inefficiency of the bubble sort is compensated for by its simplicity, especially from a pedagogical point of view. It is totally trivial to program, as we have seen. Consequently, it is quite acceptable for sorting tasks that only involve 'small' values of N.

Quicksort

Quicksort, invented by C.A.R. Hoare, is probably the most 'elegant' of the sorting techniques yet devised. It is an NlogN sort, which is based on a very simple idea and in its most compact form may be programmed in very few lines of code. In fact, probably the greatest difficulty in grasping how it works involves understanding the administrative details of how to apply the basic step which motivates its operation. One has the tendency to say, 'You mean, that's all there is to it?', or 'But what do you mean by simply apply the same procedure to both halves?'. Nonetheless, once appreciated, it is an algorithm you will never forget. That should be reward enough for the effort expended in understanding it in the first place.

The basic idea underlying Quicksort is to perform inter-changes of non-adjacent array elements in hopes of bringing order to the array more quickly [bubble sort has already demonstrated the inefficiency of interchanging adjacent entries]. The idea is applied using the concept of a partition of the array elements.

To partition the elements $A[P], A[P+1], \ldots, A[Q]$ of the array $A$, where $P \geq 1$, $P \leq Q$, $Q \leq N$, requires that some value $X$ which actually occurs as one of the entries $A[P], A[P+1], \ldots, A[Q]$ be placed into its correct final position, say $K$, and that the remaining elements are arranged so that $A[I] \leq A[K]$ for $I < K$ and $A[J] \geq A[K]$ for $J > K$. The results are pictured in figure 5.

For convenience in implementation [although this may not be the optimal choice in theory], we shall always choose $A[P]$ as the value $X$, which is to be inserted into its correct final resting place. To accomplish our end result, we adopt the following 'double-barreled' scan.

Start with $I = P + 1$ and $J = Q$. Scan forward from $I$ [i.e. in increasing $I$-value order] until we find $A[I]$ for which $A[I] \geq X$. Scan backward from $J$ [i.e. in decreasing $J$-value order] until we find $A[J]$ for which $A[J] \leq X$. Then interchange $A[I]$ and $A[J]$, since they are both in the 'wrong half' of the partition according to the above narrative.

![Figure 5: Partitioning A(P)-A(Q)]
definition. Continue this procedure until \( J \leq I \). As a final act, interchange \( A[P] \) and \( A[I] \), where \( I \) now has its 'final' value. This puts \( X = A[P] \) into its correct final position in the array. You should convince yourself that it also achieves the picture shown in figure 5. Actually, there is one case which fails. See if you can discern what it is — we'll come back to it later on.

An example may make things a bit clearer. Figure 6 shows an unsorted array of 16 elements, which is to be partitioned for \( P = 1, Q = 16 \). Shown are the first values of \( I \) and \( J \) for which an interchange of the partitioning process will take place. See if you can draw the final picture showing the array with the partition complete and the value of \( K \). The answer is shown in figure 7.

---

**BEFORE FIRST INTERCHANGE**

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\( P = 1 \quad A[P] = 10 \quad N = 16 \)

\( \geq A[P] \)

\( I = 4 \)

\( \leq A[P] \)

\( J = 12 \)

**AFTER FIRST INTERCHANGE**

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\( \leq A[1] \)

\( I \)

\( \geq A[1] \)

\( J \)

---

**Figure 6: Partition of A: P = 1**

---

**Figure 7: Partition step complete A(7) in correct position.**
When one gets down to programming the partitioning process, several details that may not have been previously obvious suddenly force themselves into the spotlight. To highlight these, we present in figure 8 a Pascal procedure for the partition step. The first item which may catch your eye is that array A is indicated in the parameter list to be of size $N + 1$, instead of $N$. The reason may be seen by studying the second repeat statement of figure 8:

```
repeat
  I := I + 1
  until A[I] ≥ Value;
```

```
procedure
  Partition(
    var A: array[1..N+1] of integer;
    Lower, Upper: integer;
    var J: integer);
var
  Value, Temp: integer;
begin
  if Lower < Upper then begin
    I := Lower;  {Lower bound in A for partition step}
    J := Upper;  {Upper bound in A for partition step}
    Value := A(Lower);  {Comparison value for partitioning}
    while I < J do begin  {Partitioning loop}
      repeat  {Find element in right half to switch}
        J := J - 1
        until A(J) <= Value;
      repeat  {Find element in left half to switch}
        I := I + 1
        until A(I) >= Value;
      if I <= J then begin  {Perform the switch}
        Temp := A(J);
        A(J) := A(I);
        A(I) := Temp
      end  {of if I <= J}
    end  {of while I < J}
    A(Lower) := A(J);  {Insert A(Lower) into its }
    A(J) := Value;    {correct final position in A}
  end  {of if Lower < Upper}
end  {of Procedure Partition};
```

Figure 8
As with all loops, the programmer should be sure that there is a way out! In this case, if the elements $A[1], A[2], \ldots, A[N]$ of the array are assumed to be randomly distributed among all possible values, then there is no guarantee that any of them satisfies the condition $A[I] \geq \text{Value}$. Thus, we have extended the array and stored a value in $A[N+1]$ which is guaranteed to be greater than or equal to any other value that could occur in the original array. In Pascal, the predefined identifier Maxint serves the purpose, and we may assume that the assignment $A[N+1] := \text{Maxint;}$ has occurred in the calling routine. Now, even if all elements of $A$ are strictly less than $A[1]$, the repeat loop will terminate when it bumps into the Maxint value stored in $A[N+1]$. Such a value, which is not part of the data being manipulated, but instead serves to protect against some dire circumstances, is known as a sentinel.

This approach raises two further questions: first, do we face a similar problem with $J$; and second, do we face the possibility of erroneously swapping $A[N+1]$ with some element of $A$? The first question is easily answered by realizing that $\text{Value} := A[\text{Lower}]$. Thus, if $J$ is decreased so far that $J := \text{Lower}$, then $A[J] \leq \text{Value}$ is automatically true. Thus, the first repeat loop is guaranteed to stop because of this choice. To answer the second question, let's look closely at what happens when $N = \text{Upper}$ and $A[I] < \text{Value}$ for all $I, I = 2, 3, \ldots, N$. The repeat statement:

```
repeat
  J := J - 1
until A[J] \leq \text{Value}
```

immediately succeeds. $J$ starts at $N+1$, $J-1 = N$ and $A[N] < \text{Value}$ by our assumption. Thus, $J$ stops at the value $N$ after the first time through the loop. On the other hand, the repeat statement for $I$ will continue to fail, again by our assumption, until $I = N+1$. Now $I + N + 1$ and $J = N$. This means that the test $I < J$ will fail. Therefore, the interchange shown inside the while loop will be skipped. Aha!, you say — caught you — nothing happens and Quicksort is a shame! Fortunately, that is not true. The last two statements in the procedure:

```
A[\text{Lower}] := A[J];
A[J] := \text{Value};
```

will be carried out, causing $A[\text{Lower}]$ and $A[N]$ to be swapped.

To assimilate the code of the procedure, simulate its action on the array of figure 6. As a final note, the procedure protects itself from funny initial values for $\text{Lower}$ and $\text{Upper}$, by first checking to make sure that $\text{Lower} < \text{Upper}$. This will turn out to be necessary in one version (the recursive one) of the complete Quicksort algorithm, but must be moved back to the caller for the other version (the 'straight' or iterative one).

Now that we have studied the innards of the Quicksort algorithm, it is time to investigate how the partition step fits into the larger scheme of things. Once the original array $A$ has been partitioned, we are left with one element in its correct final resting place and two subarrays that remain to be sorted. The beauty of
Quicksort is its simplicity. Once the two subarrays are both sorted, the entire array is automatically sorted. This is true because of the condition — guaranteed by the partition step — that all elements in the first half of the array are less than or equal to all the elements in the second half of the array. Not convinced? Think about it! Or, consider the following analogy: a school teacher wishes to arrange test papers in alphabetical order. The papers are divided into two piles [partitioning step] with all papers in the left-hand pile belonging to students whose names begin with letters A to M, and all papers in the right-hand pile belonging to students with names beginning with letters N to Z. Now, if the left-hand pile is arranged [by whatever method] into alphabetical order and likewise the right-hand pile, then all that remains to put the whole collection into alphabetical order is to place the left-hand pile on top of the right-hand pile.

To continue the Quicksort algorithm, one applies the basic step to both subarrays obtained from the first partitioning step. That will produce in each case two new subarrays [or better, sub-subarrays], to which the partitioning process is applied in turn. Since we started with a finite number of elements in array A, sooner or later this will produce sub-sub...subarrays with 0 elements. Such subarrays are sorted by default. Thus, they need not be partitioned any further. Moreover, when both subarrays of a given subarray reach this state, they form together with their partition element a sorted subarray, which may then be ignored while the remaining unsorted subarrays are processed. Eventually, the original two subarrays will have been sorted and voila!, A will have been sorted. Figure 9 shows the implementation of this scheme as a Pascal procedure that must be invoked from outside itself with initial values for Lower and Upper, which are presumably 1 and N in most cases. Once it gets going, it calls itself on behalf of the subarrays, and the sub-subarrays, etc., until it completely sorts A. Figure 10 shows the progress of the sort as applied to a small array, with N = 8. Study it carefully. Figure 11 presents the calling structure to Sort for the array in figure 10. The root of the tree represents the original call to Sort from outside. The interior nodes of the tree represent calls to Sort from within itself. Each node is labeled with the values of Lower and Upper which were passed on the corresponding call.

procedure
    Sort(
        var A: array[1..N+1] of integer;
        Lower,Upper: integer);
    var
        J: integer;
    begin
        Partition(A,Lower,Upper,J);
        Sort(A,Lower,J-1);
        Sort(A,J+1,Upper);
    end {of Procedure Sort};

Figure 9
<table>
<thead>
<tr>
<th>A</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 9 1 13 5 21 7 20</td>
<td>Partition(A,1,8);</td>
</tr>
<tr>
<td>10 9 1 7 5 21 13 20</td>
<td>Partition(A,1,4);</td>
</tr>
<tr>
<td>5 9 1 7 10 21 13 20</td>
<td>Partition(A,1,1);</td>
</tr>
<tr>
<td>5 9 1 7 10 21 13 20</td>
<td>Partition(A,3,4);</td>
</tr>
<tr>
<td>5 1 9 7 10 21 13 20</td>
<td>Partition(A,3,3);</td>
</tr>
<tr>
<td>1 5 9 7 10 21 13 20</td>
<td>Partition(A,5,4);</td>
</tr>
<tr>
<td>1 5 7 9 10 21 13 20</td>
<td>Partition(A,6,8);</td>
</tr>
<tr>
<td>1 5 7 9 10 20 13 21</td>
<td>Partition(A,6,7);</td>
</tr>
<tr>
<td>1 5 7 9 10 20 13 21</td>
<td>Partition(A,6,6);</td>
</tr>
<tr>
<td>1 5 7 9 10 13 20 21</td>
<td>Partition(A,8,7);</td>
</tr>
<tr>
<td>1 5 7 9 10 13 20 21</td>
<td>Partition(A,9,8);</td>
</tr>
</tbody>
</table>

**Figure 10:** Complete trace of Quicksort for N = 8. Boxed entries are known to be in the correct slot.

The leaves of the tree represent calls to Sort in which the passed values of Lower and Upper correspond to subarrays with 0 elements. Such subarrays are already sorted and "nothing" will happen on these calls.

**EXERCISE:** Determine whether or not the Partition procedure may be modified to return whenever the passed array has either 0 or 1 elements. If so, make the necessary changes to the code.

The recursive implementation of Quicksort is without a doubt one of the most "beautiful" algorithms yet devised in any branch of computer science. Unfortunately, the performance of Quicksort in such an implementation, even though superior to most $N^2$ algorithms, is still not quite as good as it could be.
We shall not attempt to explain the technical reasons for this, other than to say that recursion involves more than a modicum of overhead. However, we shall attempt to formulate the algorithm in a non-recursive or iterative fashion for comparison.

Now look back at the recursive implementation of Quicksort shown in figure 9. Since Sort calls itself, this means that the variable J, which is used locally within Sort, must be given a different "incarnation" on each call. Otherwise, the recursive calls would cause its former value to be lost, which in turn would mean that the procedure would get mixed up about where the subarrays began and ended. In languages, such as Pascal, which support recursive procedures, the uniqueness of J on each call is guaranteed. In a language like BASIC, there aren't even procedures, let alone recursive ones! Thus, in such a language, we must "fake it" in some way or another.

What is it about the variable J that's so important? It remembers the dividing point between the two subarrays determined by any partition step. This enables the two halves to be sorted separately by successive calls to Sort. Another way to approach matters would be to save information about subarrays that still need sorting and retrieve it as necessary. An appropriate data structure for preserving
such information is a stack. The Lower and Upper values for one "half" of a partition may be saved by pushing them onto the stack, while the other "half" is being sorted. When the other half has been completely sorted, the Lower and Upper values for the saved half may be popped off the stack and the sorting of that half commenced. Of course, while sorting a given half, new pairs of bounds for smaller subarrays will be determined and bounds for one subarray of each such pair will in turn be pushed onto the stack. If a point is reached at which we try to pop the bounds of a subarray from the stack, and find that the stack is empty, then we will know that the original array is completely sorted. As a performance enhancement, we shall always sort the smaller of any given pair of subarrays first. This is in distinction to the algorithm of figure 9, which always sorts the left subarray first. Sorting the smaller subarray first will cause a minimum number of entries to be saved on the stack.

The actual code of an iterative implementation of the Quicksort algorithm is presented in Listing 5, using Apple Integer BASIC.

Sorting Implemented

The Apple II Integer BASIC programs of Listings 1-5 provide implementations of visual sorts for the following five methods: Bubble sort, straight insertion sort, selection sort, Shell sort, and Quicksort. The visual display arranges the array to be sorted as a table of up to 100 positive two digit integers — the user may request fewer if so desired to speed up the completion of the algorithm. The basic table uses the random number generator for INTEGER BASIC. For skeptical viewers, the values 0 to N may be generated in a permuted order and filled into the first N + 1 slots of the tableau. The modification needed in order to accomplish this is shown in figure 12. Figure 13 shows a typical tableau, this one prior to the beginning of Shellsort. Notice that extra information is displayed in the small area surrounding the display. By studying the listing and carefully monitoring this information, extra insight into the nature of the algorithms may be gained.

```
80 FOR I = 0 to N: A(1) =:NEXT I
90 FOR I = 0 to N
100 L = RND (N + L): IF A(L) = 0 THEN 100
105 A(L) = 1: X = L: GOSUB DISPLAY
110 NEXT I
```

*Figure 12: Modification to Display generation: will seed the Initial array with exactly the numbers 0 to N in some permuted order.*

All values generated are positive and less than 100. This is done because of horizontal space constraints in the display and does not reflect any inherent limitations in the algorithms themselves.

The programs each carry out one of the sorting algorithms. As the array is sorted, the values displayed on the screen are modified to reflect the changes taking place internally. Various devices are used to highlight this: some visual and some aural. The audio effects are programmed using the Programmer's Aid ROM. Thus, you may have to remove or modify certain statements to run the programs if you don't own PA.
Each time a number is moved from one place to another in the array, that value is highlighted in the display. This is accomplished by momentarily displaying the value in reverse video, then switching back to normal mode. If your Apple has been modified for lower case, this probably won’t work. You can get a good idea of how each algorithm does its job just by watching the pattern of flashes on the screen.* In addition to this, as mentioned above, each sort prints on the border of the display some additional information about what is happening. Each program begins with a prologue giving the name of the sort and prompting the user for the number of elements to be sorted. The value of PDL(1) is used by the programs to control the speed at which the display is generated. Thus to slow down the progress of the program, simply turn up the PDL(1) control.

While each algorithm is in progress, two tones will be sounded periodically. One tone is generated each time an array element is copied from one place to another, that is, for each interchange. A different tone is sounded whenever an array element is compared to another or to a fixed value, that is, for each comparison. Listening to the pattern of sounds thus produced gives a very definite auditory tattoo to each algorithm. The calls to Programmer’s Aid which produce these tones are localized in subroutines to facilitate their removal or replacement should you not have the PA ROM. For example, in the bubble sort demo, you may defeat the sounds by inserting the two statements:

901 RETURN
951 RETURN
Even if you do have PA, you may want to use these statements in order to (a) speed up the program a little or (b) hear only comparisons or only interchanges.

*NOTE: If you stop the program with a Control-C at just the right (or wrong—depending on your point of view) moment, you may find that everything is being displayed in reverse video. To return to normal display mode, simply type:

POKE 50,255

and all should be well.
10 REM ************************
20 REM * 
30 REM * BUBBLE * 
40 TEXT : CALL -936
50 GO SUB INTRO
60 COL=X MOD 10
70 ROW=X/10
80 VTAB 2+ROW+3: TAB 7+3*COL
90 IF A(X)<10 THEN PRINT " ";
100 PRINT A(X);
110 RETURN
120 IF KEY<>ASC("Q") THEN 810
130 TEXT : CALL -936: END
140 IF KEY>PEEK(KBD): IF KEY<128 THEN 810
150 PRINT " > ";
160 POKE CLR,O: RETURN
170 POKE CLEAR,O: CALL -936: GOTO 48
180 VTAB 23: TAB 1: PRINT "TEMP="; TAB 20
190 PRINT "BUBBLE SORT"
200 RETURN
210 IF KEY<>PEEK(KBD): IF KEY<128 THEN 200
220 POKE CLR,0: GOSUB WAIT
230 NEXT J
240 IF FLAG=0 THEN 208
250 NEXT I
260 VTAB 24: TAB 21: PRINT "FINISHED";
270 IF PEEP(KBD)<128 THEN 210
280 POKE CLR,0: CALL -936: GOTO 48
290 REM ****************************************
300 REM * SORTING REVEALED * 
310 REM * RICHARD C. VILE, JR. * 
320 REM * BUBBLE * 
330 REM * COPYRIGHT (C) 1981 * 
340 REM * MICRO INK, INC. * 
350 REM * CHELMSFORD, MA 01824 * 
360 REM * ALL RIGHTS RESERVED * 
370 REM ************************
380 REM *** TO REMOVE SOUND, 901 IS ADDED ***
390 REM
400 REM : CALL -936: POKE TIME,5: CALL MUSIC
905 FOR DE=1 TO PDL (1): NEXT DE
910 RETURN
950 REM *** TO REMOVE INTERCHANGE SOUNDS, ADD 951 ***
951 RETURN
952 POKE PITCH,49: POKE TIME,3: CALL MUSIC
955 FOR DE=1 TO PDL (1): NEXT DE
960 RETURN

1000 VTAB 10: TAB 5: PRINT "I WILL SORT UP TO 100 POSITIVE"
1001 TAB 5: PRINT "INTEGERS INTO ASCENDING"
1002 TAB 5: PRINT "ORDER USING THE BUBBLE SORT."
1008 VTAB 15: TAB 10: INPUT "VALUE OF N, PLEASE",NUM:N=NUM-1
1009 IF NUM<=0 THEN 805
1010 IF NUM<100 THEN RETURN
1015 TAB 10
1020 PRINT "TOO BIG!!!!": GOTO 1000
206 NEXT I
208 VTAB 24: TAB 15: PRINT "FINISHED";
210 IF PEEK (KBD)<128 THEN 210
220 POKE CLR,0: CALL -936: GOTO 48
500 TEXT : CALL -936
510 VTAB 1: FOR I=0 TO 9: TAB 7+3*I: PRINT I;: NEXT I
515 VTAB 2: TAB 7: FOR I=0 TO 9: PRINT "-";: NEXT I
520 FOR J=0 TO 9: VTAB 3+2*J: TAB 4: PRINT J;":": NEXT J
525 VTAB 23: TAB 13: PRINT "INSERTION SORT"
530 RETURN
600 COL=x MOD 10
610 ROW=x/10
620 VTAB 2*ROW+3: TAB 7+3*COL
630 IF A(X)<10 THEN PRINT ";
635 PRINT A(X);
640 RETURN
650 COL=x MOD 10:ROW=x/10
655 VTAB 2*ROW+3: TAB 7+3*COL
660 PRINT "I";
670 RETURN
800 IF KEY<> ASC("O") THEN 810
805 TEXT : CALL -936: END
810 KEY= PEEK (KBD): IF KEY<128 THEN 810
820 POKE CLR,0: RETURN
900 REM ***TO REMOVE SOUNDS, 901 INSERTED***
901 RETURN
902 POKE PITCH,10: POKE TIME,5: CALL MUSIC
905 GOSUB DELAY
910 RETURN
950 REM ***TO REMOVE SOUNDS, 951 INSERTED***
951 RETURN
952 POKE PITCH,49: POKE TIME,3: CALL MUSIC
955 GOSUB DELAY
960 RETURN
975 FOR DE=1 TO PDL (1): NEXT DE
980 RETURN
1000 VTAB 10: TAB 5: PRINT "$ WILL SORT UP TO 100"
1001 TAB 5: PRINT "INTEGERS INTO ASCENDING"
1002 TAB 5: PRINT "ORDER USING THE INSERTION SORT."
1003 VTAB 15: TAB 10: INPUT "VALUE OF N PLEASE",NUM:N=NUM-1
1010 IF N>10 THEN 1013
1012 TEXT : CALL -936: END
1013 IF NUM<=100 THEN RETURN
1015 TAB 10
1020 PRINT "THAT'S TOO BIG!!!": GOTO 1000

0 I=J=Y=N
10 REM ***************
12 REM *
14 REM * SORTING REVEALED *
16 REM * RICHARD C. VILE, JR. *
18 REM *
20 REM *
22 REM *
24 REM * COPYRIGHT (C) 1981 *
25 REM * MICRO INK, INC. *
26 REM * CHELMSFORD, MA 01824 *
27 REM * ALL RIGHTS RESERVED *
28 REM *
29 REM ***************
30 REM
40 TEXT : CALL -936
41 DIM A(99)
42 KBD=-16384:CLR=-16368:TITLE=500:INTRO=1000
44 DISPLAY=600:WAIT=800:CMP=900:INT=950
46 MUSIC=-10473:TIME=766:TIMBRE=765:PITCH=767
47    DELAY = 975: ERASE = 650
48    GOSUB INTRO
50    GOSUB TITLE
100   FOR I = 0 TO N
105   A(I) = RND(100): X = I: GOSUB DISPLAY
110   NEXT I
150   FOR I = 0 TO N - 1
151     MAX = 0
152     VTAB 23: TAB 32: PRINT "I = "; IF I < 10 THEN PRINT "; "; PRINT I;
155     FOR J = 1 TO N - I
156        KEY = PEEK(KBD): IF KEY < 128 THEN 158
157        POKE CLR, 0: GOSUB WAIT
158     GOSUB DELAY
159     GOSUB CMP
160    IF A(J) <= A(MAX) THEN 200
163    MAX = J
165    VTAB 24: TAB 32: PRINT "M = "; IF MAX < 10 THEN PRINT " "; PRINT MAX;
168    POKE 50, 63
175    X = J: GOSUB DISPLAY
178    POKE 50, 255
185    X = J: GOSUB DISPLAY
200    NEXT J
202    TEMP = A(MAX): GOSUB INT
204    A(N - I) = TEMP: X = N - I: POKE 50, 63: GOSUB DISPLAY: GOSUB INT: POKE 50, 255
212    NEXT I
215    VTAB 24: TAB 15: PRINT "FINISHED";
218    IF PEEK(KBD) < 128 THEN 218
220    POKE CLR, 0: CALL -936: GOTO 48
500    TEXT : CALL -936
510    VTAB 1: FOR I = 0 TO 9: TAB 7 + 3 * I: PRINT I: NEXT I
515    VTAB 2: TAB 7: FOR I = 0 TO 9: PRINT " "; NEXT I
520    FOR J = 0 TO 9: VTAB 3 + 2 * J: TAB 4: PRINT J; " "; NEXT J
525    VTAB 23: TAB 13: PRINT "SELECTION SORT"
530    RETURN
600    COL = X MOD 10
610    ROW = X / 10
620    VTAB 2 * ROW + 3: TAB 7 + 3 * COL
630    IF A(X) < 10 THEN PRINT " ";
635    PRINT A(X);
640    RETURN
800    IF KEY# ASC("O") THEN 810
805    TEXT : CALL -936: END
810    IF PEEK(KBD) < 128 THEN 810
820    POKE CLR, 0: RETURN
900    REM *** TO REMOVE SOUNDS, 901 INSERTED ***
901    RETURN
902    POKE PITCH, 10: POKE TIME, 5: CALL MUSIC
905    GOSUB DELAY
910    RETURN
950    REM *** TO REMOVE SOUNDS, 951 INSERTED ***
951    RETURN
952    POKE PITCH, 49: POKE TIME, 3: CALL MUSIC
955    GOSUB DELAY
960    RETURN
975    FOR DE = 1 TO PDL (1): NEXT DE
980    RETURN
1000   VTAB 10: TAB 5: PRINT "I WILL SORT UP TO 100"
1001   TAB 9: PRINT "INTS INTO ASCENDING"
1002   TAB 5: PRINT "ORDER USING THE SELECTION SORT."
1008   VTAB 15: TAB 10: INPUT "VALUE OF N PLEASE", N
1009   N = N - 1
1010   IF N > 0 THEN 1013
1012   TEXT : CALL -936: END
1013   IF N <= 100 THEN RETURN
1015   TAB 10
1020   PRINT "TO BIG!!!!": GOTO 1000
**SORTING REVEALED**

**RICHARD C. VILE, JR.**

**MICRO INK, INC.**

**CHELMSFORD, MA 01824**

**COPYRIGHT (C) 1981**

**ALL RIGHTS RESERVED**

```plaintext
100 DIM A(99), INCS(5)
105 MUSIC = -10473; PITCH = 767; TIME = 766; TIMBRE = 765; POKE TIMBRE, 32
110 KBD = -16384; CLR = -16368; TITLE = 400; INTR0 = 1000
120 DISPLAY = 500; WAIT = 800; CMP = 900; INT = 950
125 DELAY = 975; ERASE = 550
130 TEXT : CALL -936
140 GOSUB INTRO
150 GOSUB TITLE
160 FOR I = 0 TO N
170 A(I) = RND(100); X = I: GOSUB DISPLAY
180 NEXT I
190 INCS(1) = 10; INCS(2) = 6; INCS(3) = 4; INCS(4) = 2; INCS(5) = 1
200 FOR I = 1 TO 5
210 SPAN = INCS(I)
211 IF SPAN > N THEN 370
215 VTAB 24; TAB 12; PRINT "SPAN = ";
216 IF SPAN < 10 THEN PRINT " "; PRINT SPAN;
220 FOR J = SPAN TO N
230 Y = A(J): GOSUB INT
233 VTAB 23; TAB 28; PRINT "J = ";: IF J < 10 THEN PRINT " "; PRINT J
235 TAB 26; PRINT "A(J) = ";: IF A(J) < 10 THEN PRINT " ";
236 POKE 50, 63: PRINT A(J): POKE 50, 255
240 FOR K = J - SPAN TO 0 STEP -SPAN
245 GOSUB CMP
250 IF Y > A(K) THEN 320
255 TAB 28; PRINT "A(K) = ";: IF A(K) < 10 THEN PRINT " ";
256 POKE 50, 63
257 GOSUB CMP
258 X = K + SPAN: A(K) = Y
260 POKE 50, 63
265 GOSUB INT
270 A(X + SPAN) = A(X)
280 X = X + SPAN: GOSUB DISPLAY
285 KEY = PEEK (KBD): IF KEY < 128 THEN 290
287 POKE CLR, 0: GOSUB WAIT
290 GOSUB DELAY
295 POKE 50, 255: GOSUB DISPLAY
300 X = K: GOSUB ERASE
310 NEXT K
320 POKE 50, 63
325 GOSUB INT
330 A(X + SPAN) = Y: X = K + SPAN: GOSUB DISPLAY
340 GOSUB DELAY
350 POKE 50, 255: GOSUB DISPLAY
360 NEXT J
370 NEXT I
380 VTAB 24; TAB 12; PRINT "FINISHED ";
390 IF PEAK (KBD) < 128 THEN 390
395 POKE CLR, 0: CALL -936; GOTO 140
400 TEXT : CALL -936
420 VTAB 1: FOR I = 0 TO 9: TAB 7 + 3*I: PRINT I: NEXT I
430 VTAB 2: TAB 6: FOR I = 0 TO 9: PRINT " "; NEXT I
440 FOR J = 0 TO 9: VTAB 3 + 2*J: TAB 4: PRINT J: " "; NEXT J
450 VTAB 23: TAB 10: PRINT " SHELL SORT" 
460 RETURN
500 COL = X MOD 10
510 row = X / 10
520 VTAB 2*row + 3: TAB 7 + 3*COL
530 IF A(X) < 10 THEN PRINT " ";
540 PRINT A(X)
549 RETURN
```
Vile  Sorting Revealed  131

550 COL=X MOD 10:ROW=X/10
555 VTAB 2*ROW+3: TAB 7+3*COL
560 PRINT * *;
599 RETURN
800 IF KEY<> ASC("Q") THEN 810
805 TEXT : CALL -936: END
810 IF PEEK (KBD)<128 THEN 810
820 POKE CLR,0: RETURN
900 REM ***TO REMOVE SOUNDS, 901 INSERTED***
901 RETURN
902 POKE PITCH,10: POKE TIME,5: CALL MUSIC
905 GOSUB DELAY
910 RETURN
950 REM ***TO REMOVE SOUNDS, 951 INSERTED***
951 RETURN
952 POKE PITCH,49: POKE TIME,3: CALL MUSIC
955 GOSUB DELAY
960 RETURN
975 FOR DE=1 TO PDL (1): NEXT DE
980 RETURN
1000 VTAB 10: TAB 5: PRINT "I WILL SORT UP TO 100"
1001 TAB 5: PRINT "INTEGERS INTO ASCENDING"
1002 TAB 5: PRINT "ORDER USING THE SHELL SORT."
1008 VTAB 15: TAB 10: INPUT "VALUE OF N PLEASE",N
1009 N=N-1
1010 IF N>0 THEN 1013
1012 TEXT : CALL -936: END
1013 IF N<100 THEN RETURN
1015 TAB 10
1020 PRINT "THAT'S TOO BIG!!!": GOTO 1000

10 REM ********************
12 REM *                    *
14 REM * SORTING REVEALED  *
16 REM * RICHARD C. VILE, JR. *
18 REM *                    *
20 REM * QUICK              *
22 REM *                    *
24 REM * COPYRIGHT (C) 1981 *
25 REM * MICRO INK, INC.    *
26 REM * CHELMSFORD, MA 01824 *
27 REM * ALL RIGHTS RESERVED *
28 REM *                    *
29 REM ********************
30 REM                    *
32 REM
35 DIM A(200),STACK(24)
36 KBD=-16384:CLR=-16368:TITLE=5000:INTRO=10000
37 DISPLAY=6000:CMPI=6500:DELAY=6600
38 MUSIC=10473:TIME=766:TIMBRE=765:PITCH=767
40 TEXT : CALL -936
42 PRINT "MICRO APPLE, VOLUME I"
44 VTAB 3: PRINT "SEE 'SORTING REVEALED'"
46 VTAB 5: PRINT "BY RICHARD C. VILE, JR."
47 IF PEEK (KBD)<160 THEN 47
48 GOSUB INTRO
50 GOSUB TITLE
100 FOR I=0 TO N-1
105 A(I)= RAND (100):X=I: GOSUB DISPLAY
110 NEXT I
115 A(N+1)=32767
120 P=0:Q=N
125 TOP=0:MAXTP=0
130 IF P=Q THEN 170
135 K=Q+1
137 VTAB 23: TAB 34: PRINT "P= "; IF P<100 THEN PRINT " "; IF P<10 THEN PRINT " "; PRINT P
138 TAB 34: PRINT "O= "; IF K<100 THEN PRINT " "; IF K<10 THEN PRINT " "; PRINT K;
139 GOSUB 1145
140 IF J-P<0-J THEN 150
143 GOSUB 400
144 GOTO 160
150 GOSUB 500
160 TOP=TOP+2
161 IF TOP=MAXTP THEN MAXTP=TOP
162 VTAB 24: TAB 23: PRINT (TOP/2);
163 IF PEEK (KBD)>=128 THEN GOSUB 8000
165 GOTO 130
170 IF TOP=0 THEN 208
175 Q=STACK(TOP):P=STACK(TOP-1):TOP=TOP-2
176 GOSUB 7500
177 VTAB 24: TAB 23: PRINT (TOP/2);
179 IF PEEK (KBD)>=128 THEN GOSUB 8000
180 GOTO 130
208 VTAB 24: TAB 4: PRINT "FINISHED";
209 TAB 15: PRINT "MAXTOP= "; (MAXTP/2);
210 IF PEEK (KBD)<128 THEN 210
220 POKE CLR,0: CALL -936: GOTO 48
400 STACK(TOP+1)=P
405 STACK(TOP+2)=J-1
410 P=J+1
415 GOSUB 7000
499 RETURN
500 STACK(TOP+1)=J+1
505 STACK(TOP+2)=0
510 Q=J-1
515 GOSUB 7000
599 RETURN
1145 V=A(P):I=P:J=K
1160 J=J-1: IF A(J)<=V THEN 1170
1162 GOSUB DELAY
1165 GOSUB CMP: GOTO 1160
1170 I=I+1: IF A(I)>=V THEN 1180
1172 GOSUB DELAY
1175 GOSUB CMP: GOTO 1170
1180 IF J<=I THEN 1200
1185 TEMP=A(I)
1186 A(I)=A(J):X=I: GOSUB DISPLAY
1188 A(J)=TEMP:X=J: GOSUB DISPLAY
1195 IF PEEK (KBD)<128 THEN 1160
1196 GOSUB 8000
1199 GOTO 1160
1200 A(P)=A(J):X=P: GOSUB DISPLAY
1202 A(J)=V: X=J: GOSUB DISPLAY
1999 RETURN
5000 TEXT : CALL -936
5010 VTAB 1: FOR I=0 TO 9: TAB 7+3*I: PRINT I: ; NEXT I
5020 VTAB 2: TAB 7: FOR I=0 TO 9: PRINT "---": ; NEXT I
5030 FOR J=0 TO 19: VTAB 3+J: TAB 3
5040 VTAB 23: TAB 3: PRINT "QUICKSORT PARTITION=========>";
5045 VTAB 24: TAB 15: PRINT "PENDING:0";
5060 FOR R=10 TO 22: TAB 39: PRINT ":": NEXT R
5099 RETURN
6000 COL=X MOD 10
6010 ROW=X/10
6020 POKE 50,63
6030 VTAB ROW+3: TAB 7+3*COL
6040 IF A(X)<10 THEN PRINT " ";
6050 PRINT A(X);
6060 POKE 50,255
6070 VTAB ROW+3: TAB 7+3*COL
6080 IF A(X)<10 THEN PRINT " ";
6090 PRINT A(X);
6100 REM ***TO REMOVE INT SOUND, 6101 INSERTED***
6101 RETURN
6110 POKE PITCH,49: POKE TIME,3: CALL MUSIC
6199 RETURN
6500 REM ***TO REMOVE COMP. SOUNDS, 6501 INSERTED***
6501 RETURN
Vile  Sorting Revealed  133

6510 POKE PITCH,10: POKE TIME,5: CALL MUSIC
6599 RETURN
6600 FOR DE=0 TO PDL (1): NEXT DE
6699 RETURN
7000 VTAB 21-TOP: TAB 37
7005 TOS=STACK(TOP+1): NOS=STACK(TOP+2)
7010 IF NOS<100 THEN PRINT " "; IF NOS<10 THEN PRINT " "; PRINT NOS
7015 TAB 37: IF TOS<100 THEN PRINT " "; IF TOS<10 THEN PRINT " "; PRINT TOS;
7499 RETURN
7500 VTAB 21-TOP: TAB 37: PRINT " ": TAB 37: PRINT " ";
7999 RETURN
8000 POKE CLR,0
8005 IF PEEK(KBD)<128 THEN 8005
8010 POKE CLR,0
8099 RETURN
10000 VTAB 10: TAB 5: PRINT "I WILL SORT UP TO 100 POSITIVE"
10010 TAB 5: PRINT "INTEGERS INTO ASCENDING ORDER USING HOARE'S QUICKSORT."
10030 VTAB 15: TAB 10: INPUT "VALUE OF N PLEASE",N
10040 IF N>0 THEN 10060
10050 TEXT : CALL -936: END
10060 IF N<=199 THEN RETURN
10070 TAB 10
10090 PRINT "TOO BIG!!!": GOTO 10000
Solar System Simulation with or without an Apple II

by David A. Partyka

Astronomy is a science of observation. Through years of observation, mathematical laws have been derived to explain certain phenomena, like the motion of the planets around the sun. Now, using your Apple and the program and information provided here, you can explore the inner solar system using Hi-Res graphics; don't let Kepler's laws go to waste—be an indoor astronomer!

There are unlimited applications for a micro with high resolution graphics. Some of the more fascinating aspects are the simulation of objects around us. This article and program deals with the simulated motion of the first six planets of our solar system.

Each planet moves in an elliptical orbit of varying distance from the sun. The closer the orbit to the sun, the less time it takes that planet to complete its orbit. Mercury, the closest planet takes 88 days, while Saturn the farthest of the first six takes 29 years. Because the planets move in elliptical orbits, their distance from the sun and orbital speed is constantly changing. Using Johann Kepler's (1571-1630) second law of planetary motion, "The line joining the planet to the sun sweeps out equal areas in equal time," we can calculate the time it takes the planet to travel from point W to point R (figure 1). As you can see, the line RV joining the sun S to the planet R will vary in length as the planet travels around its orbit. Being at its minimum distance at W, the planet must travel faster for the line RV to sweep an equal area as when the planet is at its maximum distance Z.

To calculate the area SWR (figure 1) we use the formula

\[ \text{Area} = \frac{ab}{2} (H-e \sin H). \]

Variable \( a \) being the length of the major axis, \( b \) the length of the minor axis, \( e \) the eccentricity of the ellipse \((c/a)\) and \( H \) (figure 2), the angle in RADIANS from the center of the ellipse to point \( q \); point \( q \) being on a circle of radius \( a \), intercepted by a perpendicular line from the major axis going through point \( R \) to the circle.
By using Equation (1), we can calculate the number of days it takes the planet to travel any degree of angle from the area. By dividing the total area of the ellipse, \((\text{total area} = \pi ab)\), by the number of days to complete the orbit we have the area swept out per day. Rearranging equation (1), we get

\[ H - e \sin H = \frac{\text{area} \cdot 2}{ab} \]

and a problem. The term \(H - e \sin H\) can't be simplified for the angle \(H\) because of the term \(\sin H\). Given the daily area we could still calculate the angle \(H\) by using a loop routine until we got the correct answer, but this would considerably slow the simulation down.

Instead I use the angle \(A\) (figure 1) at the other focus of the ellipse. By dividing 360 degrees by the number of days to complete the orbit we get the number of degrees per day for angle \(A\). Using the equation

\[ \text{RV} = 2a - \left(\frac{P}{1 + e\cos(180 - A)}\right) \]

we get the distance between the sun and the planet for each value of \(A\). Using another equation

\[ \cos V1 = \frac{R - \text{RV}}{\text{RV} \cdot e} \]

we get the angle \(V1\) that the planet lies in relation to the sun (figure 2). The value \(P\) in equation (4) being a perpendicular line from the focus to the ellipse and equal to \(a(1 - e^2)\). By increasing angle \(A\) at the daily rate we get the X,Y coordinates for each day and plot it on the screen.
Using angle A also causes a problem. Increasing angle A at a daily rate doesn't increase the area SWR (figure 1) at a daily rate. Even though there is an error, it isn't accumulative. The difference returns to zero at four points in the orbit, two points being at the minimum point W and the maximum point Z. The other two points vary with eccentricity but zero out before the $\frac{3}{4}$ position and after the $\frac{3}{4}$ position of its orbit. For Mercury, the fastest planet, the error amounts to about .65 degrees and even less for the other planets. One more equation,

$$\cos H = \frac{a-RV}{ae}$$

is a link between equation (1) and equation (3) and can be used to calculate the error of using angle A.

Now that the calculations are out of the way, let me describe this program. To keep the program small I chose only the first six planets. If you want to add the other three planets it can be done with little trouble (see listing 1). The planets are plotted in order from the sun, Mercury, Venus, Earth, Mars, Jupiter, then Saturn. You can choose any combination of planets to display, from one to all six. The planets are assigned scaling factors so their orbits will use the full plotting area when selected planets are used.
You can plot the position of the planets or planet for any day, ie July 8, 1980, or for any length of time from when you choose, ie. 100 days starting at Oct 3, 1980. You can plot any length of time with any amount of time between plots, ie. plot 900 days with 30 days between plots. Then you can choose whether to plot single points, only one dot per planet, or continuous plots. Each dot remains on the screen. Using single point plots it appears as if you are above the solar system looking down on the planets as they orbit the sun. With continuous plots you can see the orbit for the length of time you choose to plot with the amount of time between plots. When doing a plot, the first plot is always the date you choose, then it continues with what you requested. Figure 3 is an example of plotting all the planets for Aug 11, 1980; 0 was the response for the number of days to plot with any number for days between plots. The constellation names, planet names, and degrees don't show on the actual display but are shown here for reference.

Figure 3: This is an example of the display for all six planets for Aug. 11, 1980 (224 days from Jan. 0).
Figure 4 is an example of plotting the planets Mercury, Venus, and Earth on May 29, 1980 for 44 days with 4 days between plots. In this example May 29 was the first plot followed by the 11 plots for 44 days at 4 day intervals. Around the plotting area is a circle that has plots at 10 degree intervals with a double plot at the zero point. Use this to get the longitude of degrees that the planet lies in relation to the sun.

This program is set up for Jan. 0, 1980, or if you prefer, Dec. 31, 1979. To change the reference date, just add the number of days difference from Jan. 0, 1980 to the values W, ie. W1, W2, W3, etc.

Some of the things you can do with this program are to determine the dates of superior conjunction, inferior conjunction, opposition, and greatest elongation. You can demonstrate the retrograde motion of the outer planets, whether a planet is a morning or evening object, or when two or more planets appear close to each other in the sky. What else you do depends on your knowledge of astronomy. The program is simple so any additions or changes you make should be easy.

Figure 4: This is a display of the planets Mercury, Venus, and the Earth. This example is for continuous plots starting May 29th (day 150), for 44 days with 4 days between plots.
If you’re wondering how accurate this program is, I used an almanac for 1980 that gave the dates of special events for the planets, and all 20 dates that I tried worked. The display that I got for each date corresponded to what the almanac said was happening. I also have a book that gives the location of the planets 22 years ago, and the display I got was accurate enough not to make changes to the program.

### Listing 1

<table>
<thead>
<tr>
<th>planet</th>
<th>Sidereal revolution in days</th>
<th>Distance from Sun in million miles</th>
<th>Longitude of perihelion in degrees</th>
<th>Eccentricity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mercury</td>
<td>87.969</td>
<td>43.403</td>
<td>28.597</td>
<td>77.1</td>
</tr>
<tr>
<td>Venus</td>
<td>224.701</td>
<td>67.726</td>
<td>66.813</td>
<td>131.3</td>
</tr>
<tr>
<td>Earth</td>
<td>365.256</td>
<td>94.555</td>
<td>91.445</td>
<td>102.6</td>
</tr>
<tr>
<td>Mars</td>
<td>686.980</td>
<td>154.936</td>
<td>128.471</td>
<td>335.7</td>
</tr>
<tr>
<td>Jupiter</td>
<td>4332.125</td>
<td>507.046</td>
<td>460.595</td>
<td>13.6</td>
</tr>
<tr>
<td>Saturn</td>
<td>10825.863</td>
<td>937.541</td>
<td>838.425</td>
<td>95.5</td>
</tr>
<tr>
<td>Uranus</td>
<td>30676.15</td>
<td>1859.748</td>
<td>1699.331</td>
<td>172.9</td>
</tr>
<tr>
<td>Neptune</td>
<td>59911.13</td>
<td>2821.686</td>
<td>2760.386</td>
<td>58.5</td>
</tr>
<tr>
<td>Pluto</td>
<td>90824.2</td>
<td>4551.386</td>
<td>2756.427</td>
<td>223.0</td>
</tr>
</tbody>
</table>

### Listing 2

<table>
<thead>
<tr>
<th>Address</th>
<th>Old</th>
<th>New</th>
<th>From monitor load the Hi-Res subroutines in the normal location, C00 to FFF. Make these changes then move the subroutines to 3C00 by keying 3C00 C00.FFFM then RETURN.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C01</td>
<td>20</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>C65</td>
<td>0B</td>
<td>3B</td>
<td></td>
</tr>
<tr>
<td>C7E</td>
<td>0C</td>
<td>3C</td>
<td></td>
</tr>
<tr>
<td>CE3</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>D0A</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>D62</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>D6B</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>D93</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>D9F</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>DCD</td>
<td>0E</td>
<td>3E</td>
<td></td>
</tr>
<tr>
<td>DD5</td>
<td>0E</td>
<td>3E</td>
<td></td>
</tr>
<tr>
<td>DF6</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>E02</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>E3D</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>EBF</td>
<td>0C</td>
<td>3C</td>
<td></td>
</tr>
<tr>
<td>EC6</td>
<td>0E</td>
<td>3E</td>
<td></td>
</tr>
<tr>
<td>EC9</td>
<td>0C</td>
<td>3C</td>
<td></td>
</tr>
<tr>
<td>ED8</td>
<td>0C</td>
<td>3C</td>
<td></td>
</tr>
<tr>
<td>EF1</td>
<td>0D</td>
<td>3D</td>
<td></td>
</tr>
</tbody>
</table>

Old Values | High-Res Commands | New Values
---|---|---
3072 C00 | INIT | 15360 3C00
3086 C0E | CLEAR | 15374 3C0E
3780 EC4 | PLOT | 16068 3EC4
3761 EB1 | POSN | 16049 3EB1
3786 ECA | LINE | 16074 3ECA
3805 EDD | SHAPE | 16093 3EDD
**SOLAR SYSTEM SIMULATION**

David A. Partyka

---

```
10 REM ***************************
20 REM * SOLAR SYSTEM SIMULATION *
30 REM * DAVID A. PARTYKA *
35 REM * SOLAR *
40 REM * *
45 REM *
50 REM * COPYRIGHT (C) 1981 *
55 REM * MICRO INK, INC. *
60 REM * CHELMSFORD, MA 01824 *
65 REM * ALL RIGHTS RESERVED *
70 REM *
75 REM ***************************
80 REM
85 GOTO 1000
90 REM (10-110) PLOT X AND Y VALUES
100 H PLOT X,Y
110 RETURN
150 REM (200-300) CALCULATE THE X AND Y PLANET POSITIONS
200 D = Z - INT (Z / SRD) * SRD
205 REM D IS FOR DAYS
210 B = Q - (D / SRD * Q2)
220 RV = A - (P / (1 + E * COS (B)))
225 REM RV IS THE RADIUS VECTOR OR DISTANCE FROM THE SUN TO THE PLANET
230 V = PE / RV - E2
240 IF V > 1 THEN V = VL
245 IF V < - 1 THEN V = - VL
250 V1 = ATN (V / SQR (- V * V + 1)) + T
255 REM V1 IS THE ANGLE THAT THE PLANET LIES FROM THE SUN. THE 0 POINT BEING AT THE RIGHT, INCREASING COUNTERCLOCKWISE.
260 IF D < SRD / 2 THEN V1 = Q2 - V1
270 V1 = V1 + J
280 X = COS (V1) * RV: Y = SIN (V1) * RV * FA
290 X = X * TT + X1: Y = Y * TT + Y1
300 RETURN
900 REM (1000) DISPLAY PRIMARY PAGE, SET TEXT MODE
1000 POKE -16300,0: POKE -16303,0
1010 T = 1.5708
1020 Q = 3.14159265
1030 Q2 = 6.2831853
1040 VL = .99999999
1050 FA = 29 / 32
1055 REM FA IS THE RATIO OF X TO Y TO PLOT A CIRCLE AN THE APPLE INSTEAD
1060 X1 = 140: Y1 = 96
1700 HOME: PRINT : PRINT : PRINT : PRINT
1800 PRINT "DO YOU WANT TO DISPLAY "
1810 PRINT : PRINT "THE SAME PLANETS AS YOUR LAST RUN"
1815 PRINT : INPUT "Y OR N ";AS
1820 PRINT : PRINT
1830 IF AS = "N" THEN 2000
1840 IF AS < > "Y" THEN 1800
1850 IF SI < > 0 THEN 4000
1855 PRINT : PRINT
1860 PRINT : PRINT "YOU HAVEN'T PICKED THE PLANETS YET"
1870 PRINT : PRINT : PRINT
2000 PRINT "CHOOSE THE PLANETS YOU WANT TO DISPLAY"
2005 PRINT
2010 PRINT "ENTER A 1 FOR YES AND A 0 FOR NO"
2011 PRINT
2012 REM (2020-2079) GET SPECIFIC VALUES FOR EACH PLANET
2013 REM SI=ORBITAL PERIOD: PI=A1*(1-E1*E1)/2
2014 REM E1=PERCENTRICITY: U1=P1/E1:X1=I/E1
2015 REM REM AI= MINIMUM AND MAXIMUM DISTANCE FROM THE SUN
2016 REM J1=LONGITUDE OF PERIHELION IN RADIANS
2017 REM W1=DAYS FROM 0 DEGREES TO PERIHELION FOR 1980
2018 REM TT=SCALING FACTOR TO USE FULL PLOTTING AREA IF SELECTED PLANET S ARE DISPLAYED
2020 INPUT "DISPLAY MERCURY ";ME
2021 SI = 87.969
2022 E1 = .2056
2023 A1 = 43.403 + 28.597
2024 P1 = AI * (1 - E1 * E1) / 2
2025 X1 = 1 / E1
2026 U1 = P1 / E1
```
2027  J1 = 77.1 * Q / 180
2028  W1 = 37.58
2029  IF ME = 1 THEN TT = 2.3
2030  INPUT "DISPLAY VENUS ";VE
2031  S2 = 224.701
2032  E2 = .0068
2033  A2 = 67.726 + 66.813
2034  P2 = A2 * (1 - E2 * E2) / 2
2035  K2 = 1 / E2
2036  U2 = P2 / E2
2037  J2 = 131.3 * Q / 180
2038  W2 = 140.5
2039  IF VE = 1 THEN TT = 1.5
2040  INPUT "DISPLAY EARTH ";EA
2041  S3 = 365.256
2042  E3 = .0167
2043  A3 = 94.555 + 91.445
2044  P3 = A3 * (1 - E3 * E3) / 2
2045  K3 = 1 / E3
2046  U3 = P3 / E3
2047  J3 = 102.6 * Q / 180
2048  W3 = 289
2049  IF EA = 1 THEN TT = 1.05
2050  INPUT "DISPLAY MARS ";MA
2051  S4 = 686.980
2052  E4 = .0934
2053  A4 = 154.936 + 128.471
2054  P4 = A4 * (1 - E4 * E4) / 2
2055  K4 = 1 / E4
2056  U4 = P4 / E4
2057  J4 = 13.6 * Q / 180
2058  W4 = 289
2059  IF MA = 1 THEN TT = .6
2060  INPUT "DISPLAY JUPITER ";JU
2061  S5 = 4332.125
2062  E5 = .0478
2063  A5 = 507.046 + 460.595
2064  P5 = A5 * (1 - E5 * E5) / 2
2065  K5 = 1 / E5
2066  U5 = P5 / E5
2067  J5 = 13.6 * Q / 180
2068  W5 = 1604
2069  IF JU = 1 THEN TT = .19
2070  INPUT "DISPLAY SATURN ";SA
2071  S6 = 10822.863
2072  E6 = .0555
2073  A6 = 937.541 + 838.425
2074  P6 = A6 * (1 - E6 * E6) / 2
2075  K6 = 1 / E6
2076  U6 = P6 / E6
2077  J6 = 95.5 * Q / 180
2078  W6 = 2115
2079  IF SA = 1 THEN TT = .1
3900  HOME : PRINT : PRINT
4000  PRINT : PRINT "DO YOU WANT ": PRINT
4010  INPUT "POINT (0) OR CONTINUOUS (1) PLOTS ";TY
4015  IF TY < > 0 AND TY < > 1 THEN 4000
4020  PRINT : PRINT : PRINT
4030  PRINT : PRINT "DO YOU WANT TO START AT": PRINT
4040  PRINT "A SPECIFIC DATE (0) ": PRINT
4050  INPUT "OR THE BEGINNING OF THE YEAR (1) ";DT
4051  IF DT < > 0 AND DT < > 1 THEN 4020
4052  IF DT = 1 THEN 4060
4053  PRINT : PRINT : PRINT
4054  INPUT "ENTER # OF DAYS SINCE JAN 0, 1980 ";DE
4057  Z1 = DE
4060  PRINT : PRINT : PRINT "ENTER # OF DAYS TO PLOT ";DN
4070  PRINT : PRINT : PRINT
4080  INPUT "ENTER # OF DAYS BETWEEN PLOTS ";DA
4082  IF DA < > 0 THEN 4800
4084  PRINT : PRINT
4086  PRINT "0 NOT ALLOWED": GOTO 4070
4090  REM 4800 INIT HIGH RES, FULL SCREEN, PAGE 2
4800  HCR2
4802  REM (4805-4860) PLOT REFERENCE POINTS AND OUTER 10 DEGREE CIRCLE
500 FOR L1 = 0 TO Q2 STEP 1 / 36 * Q2
530 X = X1 + COS (L1) * 105.9
540 Y = Y1 - SIN (L1) * 105.9 * FA
570 GOTO 1000
580 NEXT L1
590 FOR X = 248: Y = 96: W = 3
600 NEXT L1
650 SET UP VALUES FOR MERCURY AND PLOT
620 GOSUB 200: F1 = X: G1 = Y
630 IF TY = 1 THEN 5140
640 X = M1: Y = N1: HCOLOR= 0: GOSUB 100
650 X = F1: Y = G1: M1 = X: N1 = Y: HCOLOR= 3: GOSUB 100
660 REM (5200-5240) SET UP VALUES FOR VENUS AND PLOT
670 IF VE = 0 THEN 5300
690 GOSUB 200: F2 = X: G2 = Y
700 IF TY = 1 THEN 5240
710 X = M2: Y = N2: HCOLOR= 0: GOSUB 100
720 X = F2: Y = G2: M2 = X: N2 = Y: HCOLOR= 3: GOSUB 100
730 REM (5300-5240) SET UP VALUES FOR EARTH AND PLOT
740 IF EA = 0 THEN 5400
760 GOSUB 200: F3 = X: G3 = Y
770 IF TY = 1 THEN 5340
780 X = M3: Y = N3: HCOLOR= 0: GOSUB 100
790 X = F3: Y = G3: M3 = X: N3 = Y: HCOLOR= 3: GOSUB 100
800 REM (5400-5440) SET UP VALUES FOR MARS AND PLOT
810 IF MA = 0 THEN 5500
830 GOSUB 200: F4 = X: G4 = Y
840 IF TY = 1 THEN 5440
850 X = M4: Y = N4: HCOLOR= 0: GOSUB 100
860 X = F4: Y = G4: M4 = X: N4 = Y: HCOLOR= 3: GOSUB 100
870 REM (5500-5540) SET UP VALUES FOR JUPITER AND PLOT
880 IF JU = 0 THEN 5600
900 GOSUB 200: F5 = X: G5 = Y
910 IF TY = 1 THEN 5540
920 X = M5: Y = N5: HCOLOR= 0: GOSUB 100
930 X = F5: Y = G5: M5 = X: N5 = Y: HCOLOR= 3: GOSUB 100
940 REM (5600-5640) SET UP VALUES FOR SATURN
950 IF SA = 0 THEN 6000
970 GOSUB 200: F6 = X: G6 = Y
980 IF TY = 1 THEN 5640
990 X = M6: Y = N6: HCOLOR= 0: GOSUB 100
1000 X = F6: Y = G6: M6 = X: N6 = Y: HCOLOR= 3: GOSUB 100
1010 REM (7000) PLOT POINT 297 190 TO INDICATE END OF SIMULATION THEN W
1020 WAIT FOR INPUT OF ANY CHARACTER TO START AGAIN
1030 Z1 = 0: DE = 0
1040 GOTO 1000
Programming with Pascal

by John P. Mulligan

This overview of Pascal discusses the features of the language and provides a sample program illustrating its structure and ease of use.

One of the first things I realized after purchasing my Apple II computer system, was that programming in BASIC was really a pain. Although BASIC is very suitable for programming games and relatively simple programming systems, I feel that its usefulness declines in direct proportion to the complexity of the application. There are a number of important reasons for this.

First of all, it is very difficult to program in BASIC using Structured programming techniques. Structured programming is a concept that has become widely accepted over the last few years as a method for simplifying program design and coding, and any subsequent maintenance. Basically, the program is designed by continuously breaking the main problem down into smaller problems, and then by writing one program module to solve each of the smaller problems. The modularization additionally serves to enhance readability and logic design.

Another aspect of programming that helps in understanding logic flow is the concept of “prettyprinting”, if I may borrow a term. This is simply writing the program in such a way as to promote ease of reading, and to indicate logic flow by indentation. Apple’s BASICS are notoriously difficult to read, although this is understandable because the BASIC Interpreter needs to parse the text directly at execution time and needs the text formatted in a specific manner.

The last, and for me, most important fault I see with BASIC is that it is tediously slow. This again is due to the interpretation of the BASIC textual statements. In some applications, this is quite acceptable, but for high volume processing, this becomes increasingly important. Until recently, I overcame this factor by doing most of my programming on the Apple II in Assembly language.

Now that I have aired my grievances about BASIC, let’s turn to Pascal. Pascal was first developed by Niklaus Wirth, who tried to develop the perfect programming language. This language is actually based on the ALGOL 60 programming language which is, like Pascal, a procedure-oriented language. The language developed by Wirth was named after the French mathematician Blaise Pascal, and was designed as a language to teach programming concepts. Although originally used on minicomputer systems, it is excellent for microprocessor-based systems as well.
The nice thing about Pascal is that it has all of those traits that BASIC lacks, and more. The Apple II implementation of U.C.S.D. Pascal is a very excellent programming system that is convenient, sophisticated and quite powerful. However, rather than concentrating on the operating system and the program development aspects of the system, I would like to talk about Pascal itself.

First of all, it is a compiler language. The program text is input to the compiler, and a Pascal P-code object module is generated that is executed by the Pascal P-machine emulation program. This speeds up program execution at least ten times over an equivalent BASIC program. Speed advantages are not the only benefit to program compilation. The program text can be written free form, which promotes the use of prettyprinting, and this in turn increases readability.

For example, I have written a program that sorts an array of integer numbers using the QUICKSORT algorithm. This is one of the most efficient sorting techniques that has been yet discovered, but it is somewhat confusing at first glance. Essentially, the array is sorted by the following means: First, the array is split into two halves and a routine is called for each half which first estimates a value that is in the middle of the range. When this is done, the array section being operated on is scanned and all values less than or greater than the estimated value are placed on their respective half of the array section. When this is accomplished, the array section is split and the procedure is again called.

Look at the program example, TESTSORT. The first thing to remember about Pascal programs is that 'first is last'. In other words, any variable, constant, or procedure must be defined before it is referred to. That is why the executable statements for any program or procedure are the last statements in that program or procedure.

A procedure is basically a program subroutine that is, or should be, accomplishing a discrete function within the program. Any procedure may also be composed of one or more procedures. In the example, procedure PRINT is a stand-alone procedure, while procedure SPLIT is constructed using four sub-modules, SWITCH, BUBBLE, MOVEUP and MOVEDN.

Notice also that variables and constants are always declared prior to their use at the beginning of the program or procedure. Additionally, these data areas are global to the lexical level of the program at which they are defined. In other words, the constant MAXMEMS is available to any statement in the program because it is defined at the highest level, but the variable HOLD can only be accessed from within procedure SWITCH.

This feature of defining variables for a sub-module allows the technique of recursion to be used. Simply put, this means that a procedure is able to call itself as a subroutine. This is in fact what the procedure SPLIT is doing. By using recursion, the programmer can keep the coding simple, and yet write extremely efficient programs. In this example, SPLIT is initially called from the main program logic, and the value 0 and the variable ACTMEMS are passed as parameters. At the end of the SPLIT processing, the size of the array segment being manipulated is
evaluated. At this point, the array is broken into two halves and the procedure is called again for each half. This process continues until the array segment to be passed to the SPLIT procedure is twelve items or less. At this point, a simple bubble sort is called for efficiency reasons and the return is made from the subroutine call.

This use of recursion is possible because new and unique variables are generated for each recursional level. This process allows the variables to be at the proper value when the return to the next higher level is completed. Because of this, however, a lot of memory is gobbled up in the process and there is an effective limit to the number of recursion levels possible. In the case of the Apple, a minimum of six words are used at each level in addition to any variables used, and each word is considered by the P-machine to be 16 bits. For this reason, the example is limited to 285 members in the array.

The use of this recursion technique is what makes the QUICKSORT algorithm so efficient, however. The first sort that I wrote in Pascal was a simple bubble sort that took about 70 seconds to sort 100 items in the array. Using QUICKSORT, this same array will be sorted in about five seconds. The maximum of 285 elements is sorted consistently in 16 seconds. Even though a machine language sort would run circles around these figures, try doing some sorts in BASIC. I'm not even sure that QUICKSORT could be written in BASIC.

There is one last feature of the U.C.S.D. Pascal system that I feel merits a lot of attention. With this system, machine language subroutines can be linked into and called from Pascal host programs. These routines are essentially members of Partitioned Data Sets (PDS) that are called UNITs. These UNITs each have a unique name, and up to 16 of these UNITs may reside on any one of a number of subroutine libraries that the programmer can generate. In the TESTSORT program, I wanted to use the routines NOTE and RANDOMIZE, which are machine language procedures that are used to manipulate the Apple's speaker and in generating random numbers, respectively. These routines reside in an Apple supplied UNIT called Applestuff. This unit is included in the program, and at the end of compilation is automatically linked in from the system library. Any of the functions and procedures listed at the beginning of the program above the statement IMPLEMENTATION are now available to the Pascal host program.

I've tried to highlight some of the main features of this very professional software system as simply as possible, and in doing so, have tried to indicate the usefulness of this product without being tedious. Pascal is an exciting development on the microcomputer horizon which will allow the serious software analyst to develop professional applications for microcomputer systems. Oh yes, there is one last critical point that I have neglected to mention. Programs written in U.C.S.D. Pascal can be run on any computer system using the U.C.S.D. Operating System, and there are a lot of micros out there in addition to Apple now using this operating system. Think about it for a moment. The implications are truly amazing.
1: D 1 (*$L PRINT:*$)  
2: D 1 PROGRAM TESTSORT;  
3: D 1 (*----------------------------------------*)  
4: D 1 (* )  
5: D 3 (* QUICKSORT ARRAY OF INTEGER *)  
6: D 3 (*)  
7: D 3 (*----------------------------------------*)  
8: D 3 ($)  
9: D 3 ($)  
10: D 3 )  
11: D 3 FUNCTION PADDLE(SELECT: INTEGER): INTEGER;  
12: D 3 FUNCTION BUTTON(SELECT: INTEGER): BOOLEAN;  
13: D 4: D 1 PROCEDURE TTLOUT(SELECT: INTEGER; DATA: BOOLEAN);  
14: D 5: D 3 FUNCTION KEYPRESS: BOOLEAN;  
15: D 6: D 3 FUNCTION RANDOM: INTEGER;  
16: D 7: D 1 PROCEDURE RANDOMIZE;  
17: D 8: D 1 PROCEDURE NOTE(PITCH; DURATION: INTEGER);  
18: D 3 ($)  
19: D 3 ($)  
20: D 3 IMPLEMENTATION  
21: D 3 USES APPLESTUFF;  
22: D 3 (####)  
23: D 3 (####)  
24: D 3 (####)  
25: D 3 (####)  
26: D 3 (####)  
27: D 3 (####)  
28: D 3 (####)  
29: D 3 (####)  
30: D 3 (####)  
31: D 3 (####)  
32: D 3 (####)  
33: D 3 (####)  
34: D 3 (####)  
35: D 3 (####)  
36: D 3 (####)  
37: D 3 (####)  
38: D 3 (####)  
39: D 3 (####)  
40: D 3 (####)  
41: D 3 (####)  
42: D 3 (####)  
43: D 3 (####)  
44: D 3 (####)  
45: D 3 (####)  
46: D 3 (####)  
47: D 3 (####)  
48: D 3 (####)  
49: D 3 (####)  
50: D 3 (####)  
51: D 3 (####)  
52: D 3 (####)  
53: D 3 (####)  
54: D 3 (####)  
55: D 3 (####)  
56: D 3 (####)  
57: D 3 (####)  
58: D 3 (####)  

FUNCTION PADDLE(SELECT: INTEGER): INTEGER;  
FUNCTION BUTTON(SELECT: INTEGER): BOOLEAN;  
PROCEDURE TTLOUT(SELECT: INTEGER; DATA: BOOLEAN);  
FUNCTION KEYPRESS: BOOLEAN;  
FUNCTION RANDOM: INTEGER;  
PROCEDURE RANDOMIZE;  
PROCEDURE NOTE(PITCH; DURATION: INTEGER);  

IMPLEMENTATION  
USES APPLESTUFF;  

VAR OUT : INTERACTIVE;  
CON : INTERACTIVE;  
NUM : ARRAY[0..MAXMEMS] OF INTEGER;  
ACTMEMS, IX : INTEGER;  
P, D, IX : INTEGER;  
CHRCTR : CHAR;  

PROCEDURE PRINT(TEXT: STRING);  
PROCEDURE PRINT THE INTEGER ARRAY;  
VAR IX,CTR : INTEGER;  

BEGIN  
PAGE(OUT);  
14 WRITE(OUT,TEXT);  
31 WRITE(OUT);  
38 WRITE(OUT);  
45 IX := 0;  
48 CTR := 0;  
51 REPEAT  
51 WRITE(OUT,'$44.NUM[IX]:6');  
81 IX := IX + 1;  
87 CTR := CTR + 1;  
93 IF CTR = 12 THEN  
99 BEGIN  
99 CTR := 0;  
102 WRITE(OUT);  
109 END;  
109 UNTIL IX > ACTMEMS;  
117 WRITE(OUT);  
124 END;
PROCEDURE SPLIT(X,Y:INTEGER);
**PROCEDURE** SPLIT IS A PROCEDURE WHICH
ACTUALLY DOES THE SORTING.  THE
SORT ALGORITHM USED IS THE QUICKSORT METHOD.

VAR F,L,MID : INTEGER;
ODDPASS : BOOLEAN;

PROCEDURE SWITCH(SW1,SW2:INTEGER);

PROCEDURE BUBBLE(BB1,BB2:INTEGER);

PROCEDURE MOVEUP:

PROCEDURE MOVEDN:

BEGIN
  ODDPASS := FALSE;
  REPEAT
    IF NUM[F] >= NUM[L] THEN
      BEGIN
        SWITCH(F,L);
        F := F + 1;
        MID := L;
        EXIT(MOVEUP)
      END
      ELSE
      BEGIN
        L := L - 1;
        UNTIL NOT (L > F);
        END;

BEGIN
  ODDPASS := TRUE;
  REPEAT
    IF NUM[L] < NUM[F] THEN
      BEGIN
        SWITCH(F,L);
        L := L - 1;
        MID := F;
      END
      ELSE
      BEGIN
        F := F + 1;
        MID := L;
        EXIT(MOVEUP)
      END
      ELSE
      BEGIN
        L := L - 1;
        UNTIL NOT (L > F);
        END;
Jill
Education
119 1 7:4 57 EXIT(MODEDIN)
120 1 7:3 61 END
121 1 7:2 61 ELSE
122 1 7:3 63 F := F + 1;
123 1 7:1 71 UNTIL NOT (L > F);
124 1 7:0 81 END;
125 1 7:0 96 (**Ps)
126 1 7:0 96 (************************************************************************)
127 1 7:0 96 (*) MAIN LOGIC FOR SPLIT (*)
128 1 7:0 96 (*)
129 1 7:0 96 (*)
130 1 7:0 96 (************************************************************************)
131 1 3:0 0 BEGIN=
132 1 3:1 0 F := X;
133 1 3:1 3 L := Y;
134 1 3:1 6 MID := ((F + L) DIV 2);
135 1 3:1 13 IF NUM[F] < NUM[MID] THEN
136 1 3:2 40 SWITCH(F,MID);
137 1 3:1 44 IF NUM[F] > NUM[L] THEN
138 1 3:2 71 SWITCH(F,L);
139 1 3:1 75 ODDPASS := TRUE;
140 1 3:1 78 WHILE L > F DO
141 1 3:2 03 IF ODDPASS THEN
142 1 3:3 06 MOVEUP
143 1 3:2 06 ELSE
144 1 3:3 90 MOVEIN;
145 1 3:1 94 IF (MID - X) > 12 THEN
146 1 3:2 101 SPLIT(X,MID)
147 1 3:1 103 ELSE
148 1 3:2 107 BUBBLE(X,MID);
149 1 3:1 111 IF (Y - MID) > 12 THEN
150 1 3:2 118 SPLIT(MID,Y)
151 1 3:1 120 ELSE
152 1 3:2 124 BUBBLE(MID,Y);
153 1 3:0 128 END;
154 1 3:0 142 (**Ps)
155 1 3:0 142 (************************************************************************)
156 1 3:0 142 (*) PROGRAM TESTSORT LOGIC (*)
157 1 3:0 142 (*)
158 1 3:0 142 (*)
159 1 3:0 142 (************************************************************************)
160 1 1:0 0 BEGIN
161 1 1:1 0 RESET(OUT,'PRINTER:');
162 1 1:1 41 RESET(CON,'CONSOLE:');
163 1 1:1 62 PAGE(CON);
164 1 1:1 72 GOTOXY(05,06); WRITE('************************************************************************');
165 1 1:1 120 GOTOXY(05,07); WRITE('*');
166 1 1:1 168 GOTOXY(05,08); WRITE('* INPUT NUMBER OF ELEMENTS *');
167 1 1:1 216 GOTOXY(05,09); WRITE('* LESS THAN 285: *');
168 1 1:1 264 GOTOXY(05,10); WRITE('*');
169 1 1:1 312 GOTOXY(05,11); WRITE('************************************************************************');
170 1 1:1 360 GOTOXY(26,09);
171 1 1:1 365 UNIFORM(1);
172 1 1:1 368 READLN(ACTHEMS);
173 1 1:1 387 RANDOMIZE;
174 1 1:1 390 FOR IY := 0 TO ACTHEMS DO NUM[IY] := (IY + RANDOM MOD 3452);
175 1 1:1 446 GOTOXY(06,13); WRITE('PRINT UNSORTED ARRAY (Y/N)? ')>
176 1 1:1 491 UNIFORM(1);
177 1 1:1 494 READ(CHRCTR);
178 1 1:1 505 IF CHRCTR = 'Y' THEN
BEGIN
GOTOXY(06,14); WRITE('START PRINTER AND HIT ANY KEY ');
UNITCLEAR(1);
READ(CHRCTR);
GOTOXY(00,00);
PRINT('BEFORE THE SORT - ')
END;
GOTOXY(12,16); WRITE('SORT INITIATED ');
P := 18;
D := 100;
NOTE(P,D);
SPLIT(0,ACTMEMS);
P := 18;
NOTE(P,D);
PAGE(CON);
GOTOXY(05,14); WRITE('START PRINTER AND HIT ANY KEY ');
UNITCLEAR(1);
READ(CHRCTR);
PRINT('AFTER THE SORT - ')
END.
<table>
<thead>
<tr>
<th>Before the Sort</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>3303</td>
</tr>
<tr>
<td>1786</td>
<td>1291</td>
</tr>
<tr>
<td>1813</td>
<td>2632</td>
</tr>
<tr>
<td>1160</td>
<td>3653</td>
</tr>
<tr>
<td>422</td>
<td>2612</td>
</tr>
<tr>
<td>929</td>
<td>2442</td>
</tr>
<tr>
<td>1781</td>
<td>3351</td>
</tr>
<tr>
<td>112</td>
<td>2869</td>
</tr>
<tr>
<td>3314</td>
<td>648</td>
</tr>
<tr>
<td>1025</td>
<td>2014</td>
</tr>
<tr>
<td>2574</td>
<td>2309</td>
</tr>
<tr>
<td>3527</td>
<td>1064</td>
</tr>
<tr>
<td>2746</td>
<td>272</td>
</tr>
<tr>
<td>3154</td>
<td>368</td>
</tr>
<tr>
<td>362</td>
<td>553</td>
</tr>
<tr>
<td>1209</td>
<td>1877</td>
</tr>
<tr>
<td>1563</td>
<td>3476</td>
</tr>
<tr>
<td>951</td>
<td>1454</td>
</tr>
<tr>
<td>3462</td>
<td>1592</td>
</tr>
<tr>
<td>644</td>
<td>3243</td>
</tr>
<tr>
<td>1750</td>
<td>1634</td>
</tr>
<tr>
<td>3486</td>
<td>1601</td>
</tr>
<tr>
<td>2205</td>
<td>3427</td>
</tr>
<tr>
<td>1199</td>
<td>1158</td>
</tr>
</tbody>
</table>
AFTER THE SORT -

<table>
<thead>
<tr>
<th>23</th>
<th>112</th>
<th>179</th>
<th>196</th>
<th>213</th>
<th>248</th>
<th>272</th>
<th>286</th>
<th>302</th>
<th>308</th>
<th>317</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>347</td>
<td>348</td>
<td>383</td>
<td>398</td>
<td>404</td>
<td>419</td>
<td>422</td>
<td>423</td>
<td>492</td>
<td>519</td>
</tr>
<tr>
<td>522</td>
<td>531</td>
<td>553</td>
<td>569</td>
<td>575</td>
<td>592</td>
<td>595</td>
<td>392</td>
<td>600</td>
<td>605</td>
<td>606</td>
</tr>
<tr>
<td>448</td>
<td>653</td>
<td>657</td>
<td>663</td>
<td>670</td>
<td>676</td>
<td>683</td>
<td>711</td>
<td>712</td>
<td>713</td>
<td>736</td>
</tr>
<tr>
<td>753</td>
<td>785</td>
<td>830</td>
<td>849</td>
<td>854</td>
<td>855</td>
<td>874</td>
<td>880</td>
<td>886</td>
<td>900</td>
<td>919</td>
</tr>
<tr>
<td>929</td>
<td>951</td>
<td>986</td>
<td>989</td>
<td>991</td>
<td>1010</td>
<td>1019</td>
<td>1019</td>
<td>1025</td>
<td>1058</td>
<td>922</td>
</tr>
<tr>
<td>1073</td>
<td>1085</td>
<td>1087</td>
<td>1103</td>
<td>1122</td>
<td>1133</td>
<td>1139</td>
<td>1151</td>
<td>1156</td>
<td>1158</td>
<td>1160</td>
</tr>
<tr>
<td>1167</td>
<td>1168</td>
<td>1177</td>
<td>1199</td>
<td>1201</td>
<td>1201</td>
<td>1206</td>
<td>1209</td>
<td>1214</td>
<td>1246</td>
<td>1246</td>
</tr>
<tr>
<td>1258</td>
<td>1261</td>
<td>1270</td>
<td>1291</td>
<td>1314</td>
<td>1320</td>
<td>1323</td>
<td>1333</td>
<td>1348</td>
<td>1352</td>
<td>1381</td>
</tr>
<tr>
<td>1394</td>
<td>1400</td>
<td>1415</td>
<td>1421</td>
<td>1426</td>
<td>1433</td>
<td>1445</td>
<td>1445</td>
<td>1454</td>
<td>1471</td>
<td>1481</td>
</tr>
<tr>
<td>1505</td>
<td>1506</td>
<td>1540</td>
<td>1563</td>
<td>1568</td>
<td>1570</td>
<td>1574</td>
<td>1592</td>
<td>1601</td>
<td>1610</td>
<td>1639</td>
</tr>
<tr>
<td>1634</td>
<td>1640</td>
<td>1675</td>
<td>1691</td>
<td>1708</td>
<td>1725</td>
<td>1750</td>
<td>1756</td>
<td>1764</td>
<td>1770</td>
<td>1781</td>
</tr>
<tr>
<td>1791</td>
<td>1805</td>
<td>1813</td>
<td>1817</td>
<td>1820</td>
<td>1836</td>
<td>1841</td>
<td>1845</td>
<td>1857</td>
<td>1877</td>
<td>1892</td>
</tr>
<tr>
<td>2076</td>
<td>2128</td>
<td>2154</td>
<td>2164</td>
<td>2171</td>
<td>2205</td>
<td>2211</td>
<td>2234</td>
<td>2238</td>
<td>2239</td>
<td>2243</td>
</tr>
<tr>
<td>2257</td>
<td>2298</td>
<td>2301</td>
<td>2308</td>
<td>2310</td>
<td>2313</td>
<td>2338</td>
<td>2360</td>
<td>2371</td>
<td>2393</td>
<td>2399</td>
</tr>
<tr>
<td>2401</td>
<td>2406</td>
<td>2433</td>
<td>2442</td>
<td>2456</td>
<td>2458</td>
<td>2471</td>
<td>2541</td>
<td>2545</td>
<td>2566</td>
<td>2569</td>
</tr>
<tr>
<td>2561</td>
<td>2593</td>
<td>2602</td>
<td>2603</td>
<td>2612</td>
<td>2612</td>
<td>2632</td>
<td>2653</td>
<td>2665</td>
<td>2691</td>
<td>2703</td>
</tr>
<tr>
<td>2746</td>
<td>2760</td>
<td>2774</td>
<td>2786</td>
<td>2824</td>
<td>2834</td>
<td>2854</td>
<td>2869</td>
<td>2876</td>
<td>2903</td>
<td>2907</td>
</tr>
<tr>
<td>2909</td>
<td>2911</td>
<td>2954</td>
<td>2967</td>
<td>2971</td>
<td>2982</td>
<td>3003</td>
<td>3004</td>
<td>3005</td>
<td>3020</td>
<td>3034</td>
</tr>
<tr>
<td>3053</td>
<td>3056</td>
<td>3116</td>
<td>3121</td>
<td>3124</td>
<td>3154</td>
<td>3157</td>
<td>3164</td>
<td>3173</td>
<td>3208</td>
<td>3210</td>
</tr>
<tr>
<td>3220</td>
<td>3221</td>
<td>3243</td>
<td>3244</td>
<td>3258</td>
<td>3258</td>
<td>3262</td>
<td>3279</td>
<td>3303</td>
<td>3314</td>
<td>3326</td>
</tr>
<tr>
<td>3345</td>
<td>3351</td>
<td>3354</td>
<td>3390</td>
<td>3391</td>
<td>3420</td>
<td>3427</td>
<td>3433</td>
<td>3451</td>
<td>3462</td>
<td>3476</td>
</tr>
<tr>
<td>3482</td>
<td>3486</td>
<td>3502</td>
<td>3526</td>
<td>3527</td>
<td>3573</td>
<td>3609</td>
<td>3661</td>
<td>3690</td>
<td>3477</td>
<td>3476</td>
</tr>
</tbody>
</table>
6
GAMES

Introduction 154

Spelunker 155
  Thomas R. Mimlitch

Life for Your Apple 168
  Richard F. Suitor

Apple II Speed Typing Test with Input Time Clock 173
  John Broderick, CPA

Ludwig Von Apple II 175
  Marc Schwartz and Chuck Carpenter
Introduction

Now who can resist a good "fun and games" program for the Apple? Certainly not us—so here are some of the more interesting games we’ve come across. "Spelunker" by Thomas Mimlitch brings the excitement of an adventure game to your Apple. The program, written in integer BASIC, is guaranteed to provide hours of enjoyment and suspense. "Life for Your Apple" by Dick Suitor brings the famous Life simulation to your Apple. It is written in integer BASIC, and machine language for speed. The "Speed Typing Test" by John Broderick (explained by its title) although not really a game, is a fun program written in integer BASIC.

The last program capitalizes on the musical capabilities of the Apple. "Ludwig Von Apple" by Marc Schwartz and Chuck Carpenter, plays a catchy tune. So, although you didn’t buy your Apple solely for playing games, they can be instructive — and fun!
Adventure fans, look out: Spelunker is here for the Apple. If you dare to enter the world of Spelunker, be prepared to spend a while—Spelunker can be quite mesmerizing! As a break, you might want to inspect techniques and style used in this model game program. But remember—the world of Spelunker is not for the faint of heart!

This game is an adventure fantasy series in which you become directly involved in exploration of a mysterious cavern in southwest Kentucky called Devils' Delve. If you have never played before, you should take a guide along. The guide will read the chamber descriptions as you enter each room for the first time and supply some hints and clues to help you when you are stuck. Only the guide should use the room descriptions, word lists, and the map of the caverns. However, younger players may need some of these aids to help them.

Spelunker is an interactive game. You must converse with the program to explore the caverns and locate their treasures. You can talk in sentences if you wish, but the program will use only one verb and one noun to establish meaning. For this reason, it is best to converse in verb/noun phrases. In the case of moving from chamber to chamber, for example, enter "GO W" or simply "W" and the verb "GO" will be implied. The Spelunker program will move you into the next room to the west upon receiving this command. Other examples might include "TAKE LIGHT" or "JUMP DOWN".

With this brief introduction you should be ready to explore the caverns of Spelunker. While you are about it, try drawing a map of the cave. You may also wish to discover exactly what vocabulary is understood by the program. The material that follows is for the guide only—so don’t ruin your first adventure by peeking at it.

For the Guide Only

In the 16K Apple II version of Spelunker, the chamber descriptions are not part of the program because of limited memory size. These room descriptions have been prepared for the adventurer’s guide. The guide may read each room description as the adventurer enters the chamber for the first time.
1. **Mouth:** You are at the mouth of a large cavern. The sides of the entrance slope steeply upward, and a mysterious passage leads west into the cave.

2. **Tree room:** A towering, withered tree stands in what appears to be a dried up river bed. From it you seem to hear echoing sounds saying, "Water... water... water..."

3. **Writing room:** Do not read this description if the room is dark. The writing room is a large, oval chamber with tall ceilings and massive stalagmites. The smooth eastern wall has some writing on it—cryptic characters that spell out, "THE SPIRITS OF THE FRUIT."

4. **Pit room:** A small chamber with an immense stalagtite hanging from the center of the ceiling, directly over the mouth of a bottomless pit.

5. **South lake shore:** You stand at the edge of a misty lake that stretches endlessly out before you to the north.

6. **West lake shore:** You are standing on a damp, sandy shoreline with a very low passage leading off to the west. A clammy draft issues from the low-ceilinged passage.

7. **North lake shore:** A small, sandy beach on the northern edge of Misty Lake.

8. **Maze room:** Also known as the swiss cheese room. You lose your sense of direction because twisting passages are coming and going at all points of the compass.

9. **Frozen river room:** What appears to be a petrified river bed slopes gently upward leading toward the west. It has a low, four-foot ceiling.

10. **Swift river room:** You hear swiftly running water, as you enter this room, and you see a narrow, churning, underground river flowing to the south.

11. **Hub room:** A magnificently decorated chamber with crystalline designs and intricate rock formations. A narrow, fast moving river flows through the hub room.

12. **Ice room:** Mysteriously, ice forms very quickly in this chamber, encapsulating anything left there for too long. There is so much ice that you can't even get into the room; however, you see an exit on the other side of the chamber.

13. **Chimney room:** A small, smoke filled chamber with a fire burning in a natural fireplace in the north wall. Apparently, a chimney leads far up through the rock and out of the cavern.

14. **Gold room:** As you enter this room, the first thing that you notice is a pile of golden treasures nestled into a nook on the far side. Before you take another
step, a foul-smelling ogre jumps out from a hole in the side wall and rushes forward to protect his gold.

15. **Bones room**: Lining the walls of this chamber are the skeletons of pirates long since dead. An ominous curse is uttered by all of the skeletons in unison as you enter the room, and the curse shadows your travels throughout the cavern.

16. **Bat room**: The ceiling is all but invisible for the tens of thousands of bats sleeping there. In one corner of this room lies an old, rusted chest. As you open the chest, the bats begin to stir. Inside the chest is a king’s ransom in jewels: diamonds, rubies and emeralds.

17. **Ghost room**: An eerie feeling of demonic power lurks in this chamber.

18. **Misty Lake**: You are in the middle of Misty Lake. A strange glow emanates from the bottom of the lake. You turn off your light and notice an enormous, bright pearl nestling inside a gigantic clam. The clam is at the bottom of the lake, in only ten feet of water.

19. **Swift River**: This narrow, fast flowing river is outside the cavern. It runs south for a few yards and then disappears underground.

Having been exposed to a fantasy program called *Adventure* which seems to reside on many large timesharing networks, I was challenged to see if this type of game could be handled on a micro. Thus the dream stage began. I thought up monsters, treasures, a cave structure, tools, tricks and battles. The major goals emerged:

- Pseudo-English input commands (verb-noun phrases)
- Interconnected rooms one could travel through
- Objects one could take, put, carry and use
- Monsters/treasures; do battle, take rewards
- Secrets to be discovered

The obvious method was to tabularize as much data as possible so that similar functions could be implemented as subroutines. This left only special handling routines to be added.

The program was organized into five major sections. Lines numbered 30xxx initialize the tables and variables. Lines numbered 4xxx to 10xxx print out the current location and status for the player. Lines numbered 1xxx read and decode the input string. Lines in the 2xxx range perform the command action, if possible. In lines with 3xxx numbers the monsters have an opportunity to react to their environment. Each of these sections was developed, tested and integrated separately from the others.
### Verb Table

#### Sensitive Noun Types

<table>
<thead>
<tr>
<th>Verb</th>
<th>Type</th>
<th>Direction</th>
<th>Location</th>
<th>Weapon</th>
<th>Monster</th>
<th>Treasure</th>
<th>Tools</th>
<th>Foods</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GO</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 JUMP</td>
<td>11</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 RUN</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 WALK</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 DRIVE</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 CLIMB</td>
<td>3</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 DIG</td>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 CARRY</td>
<td>116</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 DROP</td>
<td>116</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 PUT</td>
<td>116</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 TAKE</td>
<td>116</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 USE</td>
<td>36</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 WISH</td>
<td>36</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14 THROW</td>
<td>4</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 HELP</td>
<td>8</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 KILL</td>
<td>8</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 STOP</td>
<td>40</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 HIT</td>
<td>8</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 FIGHT</td>
<td>8</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 RUB</td>
<td>16</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21 START</td>
<td>32</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22 DRINK</td>
<td>64</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23 EAT</td>
<td>64</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24 BITE</td>
<td>64</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Input Commands

A list of verbs and nouns was developed and categorized as to nature or function. After entering these tables into the program, I worked on the routine to read and decode input commands. Each word was picked out of the input string, then searched for in the noun and verb lists. The first recognized verb and noun numbers were the output of this routine, and this output controlled the action routines. I later added an edit to compare the noun type and verb type to see if they were compatible.

### VERBS

- BITE
- CARRY
- CLIMB
- DIG
- DRINK
- DRIVE
- DROP
- EAT
- FIGHT
- GO
- HELP
- HIT
- JUMP
- KILL
- PUT
- RUB
- RUN
- START
- STOP
- TAKE
- THROW
- USE
- WALK
- WISH

### NOUNS

- APPLE
- AX
- BATs
- BOMB
- BONES
- CAVE
- CHEST
- CLAM
- CURSE
- DOWN
- E
- FIRE
- GHOST
- GOLD
- ICE
- KNIFE
- LAKE
- LAMP
- LIGHT
- N
- NE
- NW
- OGRE
- PEARL
- RAFT
- RIVER
- ROPE
- S
- SE
- SW
- TENT
- TREE
- TRUCK
- UP
- W
- WATER
### Noun Table

<table>
<thead>
<tr>
<th>Noun</th>
<th>Type</th>
<th>Status (Location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 N</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>2 NE</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>3 E</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>4 SE</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>5 S</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>6 SW</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>7 W</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>8 NW</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>9 UP</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>10 DOWN</td>
<td>Direction</td>
<td>0</td>
</tr>
<tr>
<td>11 CAVE</td>
<td>Location</td>
<td>0</td>
</tr>
<tr>
<td>12 LAKE</td>
<td>Location</td>
<td>0</td>
</tr>
<tr>
<td>13 RIVER</td>
<td>Location</td>
<td>0</td>
</tr>
<tr>
<td>14 TREE</td>
<td>Location</td>
<td>0</td>
</tr>
<tr>
<td>15 AX</td>
<td>Weapon</td>
<td>4 = Pit</td>
</tr>
<tr>
<td>16 BOMB</td>
<td>Weapon</td>
<td>3 = Writing</td>
</tr>
<tr>
<td>17 CURSE</td>
<td>Weapon</td>
<td>15 = Bones</td>
</tr>
<tr>
<td>18 FIRE</td>
<td>Weapon</td>
<td>13 = Chimney</td>
</tr>
<tr>
<td>19 KNIFE</td>
<td>Weapon</td>
<td>1 = Mouth</td>
</tr>
<tr>
<td>20 CLAM</td>
<td>Monster</td>
<td>18 = Misty Lake</td>
</tr>
<tr>
<td>21 BATS</td>
<td>Monster</td>
<td>16 = Bat</td>
</tr>
<tr>
<td>22 BONES</td>
<td>Monster</td>
<td>15 = Bone</td>
</tr>
<tr>
<td>23 GHOST</td>
<td>Monster</td>
<td>17 = Ghost</td>
</tr>
<tr>
<td>24 OGRE</td>
<td>Monster</td>
<td>14 = Gold</td>
</tr>
<tr>
<td>25 CHEST</td>
<td>Treasure</td>
<td>16 = Bat</td>
</tr>
<tr>
<td>26 GOLD</td>
<td>Treasure</td>
<td>14 = Gold</td>
</tr>
<tr>
<td>27 PEARL</td>
<td>Treasure</td>
<td>18 = Misty Lake</td>
</tr>
<tr>
<td>28 LAMP</td>
<td>Treasure</td>
<td>12 = Ice</td>
</tr>
<tr>
<td>29 RAFT</td>
<td>Tool</td>
<td>5 = South Shore</td>
</tr>
<tr>
<td>30 ROPE</td>
<td>Tool</td>
<td>9 = Frozen River</td>
</tr>
<tr>
<td>31 TENT</td>
<td>Tool</td>
<td>1 = Mouth</td>
</tr>
<tr>
<td>32 TRUCK</td>
<td>Tool</td>
<td>1 = Mouth</td>
</tr>
<tr>
<td>33 LIGHT</td>
<td>Tool</td>
<td>1 = Mouth</td>
</tr>
<tr>
<td>34 WATER</td>
<td>Food</td>
<td>0</td>
</tr>
<tr>
<td>35 APPLE</td>
<td>Food</td>
<td>0</td>
</tr>
<tr>
<td>36 ICE</td>
<td>Water</td>
<td>12 = Ice</td>
</tr>
</tbody>
</table>

### Objects to Take and Put

Parallel to the noun list is the status list which gives the room number where an object currently resides. A \(-1\) indicates that the object is in the possession of the player. In the output section, objects in the current room (LOC) were printed and the objects in the players possession were also reported. The second action routine was added next—the TAKE and PUT routine. TAKE changed the status of a noun to \(-1\), while PUT set its status equal to LOC. Again I tested the program and played with it, moving things all over the caves.
Cave Room Structure

The map was finalized, giving each room a number. The interconnections were entered into the N, E, S and W arrays, with a positive number indicating an exit in that direction to the room number specified. A series of statements were inserted to print out the current room descriptions, but at the time only the room name was printed. Later I discovered that there was not enough memory to put in the complete descriptions in any event.

MOVE—the first of the action routines—was coded next. If there was a possible move in the requested direction, the LOC variable was set to the new room and its description was printed. This portion was a lot of fun to test and debug.

<table>
<thead>
<tr>
<th>Room Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Room</td>
</tr>
<tr>
<td>1 Mouth</td>
</tr>
<tr>
<td>2 Tree</td>
</tr>
<tr>
<td>3 Writing</td>
</tr>
<tr>
<td>4 Pit</td>
</tr>
<tr>
<td>5 South Lake Shore</td>
</tr>
<tr>
<td>6 West Lake Shore</td>
</tr>
<tr>
<td>7 North Lake Shore</td>
</tr>
<tr>
<td>8 Maze</td>
</tr>
<tr>
<td>9 Frozen River</td>
</tr>
<tr>
<td>10 Swift River Room</td>
</tr>
<tr>
<td>11 Hub</td>
</tr>
<tr>
<td>12 Ice</td>
</tr>
<tr>
<td>13 Chimney</td>
</tr>
<tr>
<td>14 Gold</td>
</tr>
<tr>
<td>15 Bones</td>
</tr>
<tr>
<td>16 Bats</td>
</tr>
<tr>
<td>17 Ghost</td>
</tr>
<tr>
<td>18 Misty Lake</td>
</tr>
<tr>
<td>19 Swift River</td>
</tr>
<tr>
<td>20 Intersect 1</td>
</tr>
<tr>
<td>21 Intersect 2</td>
</tr>
<tr>
<td>22 Intersect 3</td>
</tr>
<tr>
<td>49 Falls (over)</td>
</tr>
<tr>
<td>50 Home</td>
</tr>
</tbody>
</table>

Monsters, Treasures and Battles

The monsters and treasures were merely noun objects in the caves, like all of the other things. A relationship was defined between the monster, his treasure, the player, and the player's use of weapons. Thus grew up the monster table and the weapons table. The monster table identifies the monster, determines his
strength, defines his treasure, identifies his home chamber, and determines how quickly he moves about the caves. The monsters move through the caverns to find their treasures if they are stolen. In the table are certain base probability factors for the monster to kill the player, steal all the player's treasures, or steal only the treasure that originally belonged to the monster.

The weapons table details the power of each of the player's weapons and determines which monsters they are effective against. The next action routine was ready to implement the ATTACK routine. This is invoked whenever a weapon is used, put, thrown, and so on. Any monsters in the room are attacked, and their life forces are decreased by a random amount limited by the force of the weapon used. When a monster's life force is reduced to zero, it is eliminated.

<table>
<thead>
<tr>
<th>Monster Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monster name</td>
</tr>
<tr>
<td>Monster number</td>
</tr>
<tr>
<td>Reward</td>
</tr>
<tr>
<td>Reward number</td>
</tr>
<tr>
<td>Move delay</td>
</tr>
<tr>
<td>Move increment</td>
</tr>
<tr>
<td>Attack count</td>
</tr>
<tr>
<td>Kill probability</td>
</tr>
<tr>
<td>Steal all probability</td>
</tr>
<tr>
<td>Steal own probability</td>
</tr>
<tr>
<td>Home room number</td>
</tr>
<tr>
<td>Life force quotient</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Weapon Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weapon name</td>
</tr>
<tr>
<td>Weapon number</td>
</tr>
<tr>
<td>Power</td>
</tr>
<tr>
<td>Attacks Monster No. 1</td>
</tr>
<tr>
<td>Attacks Monster No. 2</td>
</tr>
<tr>
<td>Attacks Monster No. 3</td>
</tr>
</tbody>
</table>

Of course, it is not fair to let the player cut the demons to shreds without allowing them to fight back. Thus came the REACTION routines. Happy monsters are those that have their own treasures in their room and have not been attacked. Any monsters that are not happy will seek someone to vent their anger upon, and that person is the player. A very intricate set of probabilities decides the outcome of this anger. The more the monster has been hurt by the player's attacks, the weaker his counterattack will become. But also, the more times he has countered in vain, the madder he gets! Nothing is more deadly than a mad monster.
Lots of testing and refinements later, SPELUNKER took its maiden voyage. Surely a program like this is never finished. The framework has been laid for all sorts of adventures—whatever one can imagine. And, now that I have more memory, I can expand the scope and capabilities of the program.
REM
*************************
REM *
SPELUNKER *
THOMAS R. MIMLITCH *
REM *
COPYRIGHT (C) 1981 *
REM *
MICRO INK, INC. *
REM *
CHELMSFORD, MA 01824 *
REM *
ALL RIGHTS RESERVED *
REM *
*************************
GOTO 30000:
TO INITIALIZE
PRINT ": INPUT IN$: IN$( LEN(IN$)+1)=" GO IN ": I=1
NOUN=0: VERB=0
GOSUB 1500: GOSUB 1600: GOSUB 1700
IF W3$1" THEN 1010
NTYP=NTYP(NOUN): VTyp=VTYP(VERB)
IF (VTYP MOD (NTYP*2))>=NTYP THEN 2000
PRINT "ICH VERSTEHE NICHT"
GOTO 3000
GOTO 2000
W3$="": S=O: FOR I=1 TO LEN(IN$): IF S=O THEN 1520:
IF INS(I,I)=" ": INS(I,I)=SPC$(S+1)
RETURN
IF NOUN=O THEN RETURN: FOR J=1 TO NUMN: IF W3$1NOUNS$(J*5-4,J*5) THEN NEXT J: IF J>=NUMN THEN RETURN: NOUN=J: W2$=W3$
RETURN
IF VERB=O THEN RETURN: FOR J=1 TO NUMV: IF W3$1VERBS$(J*S-4,J*5) THEN NEXT J: IF J>NUMV THEN RETURN: VERB=J: W1$=W3$
RETURN
NLOC=O
IF NOUN>8 THEN 2200
IF (NOUN MOD 2)=1 THEN 2100
IF LOC#11 AND LOC#6 THEN 1070
GOTO 2100+NOUN*10
NLOC=N(LOC): GOTO 2190
NLOC=0: GOTO 2190
NLOC=E(LOC): GOTO 2190
NLOC=15: IF LOC#8 THEN NLOC=8: GOTO 2190
NLOC=S(LOC): GOTO 2190
NLOC=22: IF LOC#8 THEN NLOC=8: GOTO 2190
NLOC=W(LOC): GOTO 2190
NLOC=12: IF LOC#8 THEN NLOC=8: GOTO 2190
IF RAFT=1 THEN NLOC= ABS (NLOC)
RAFT=0: PLOC=LOC
IF NLOC>O THEN LOC=NLOC
IF NLOC#12 THEN 2900
IF M(SD)<5 THEN 2900
IF PLOC=6 THEN S(12)=O
IF PLOC=l1 THEN W(12)=O
GOTO 2900
IF (NOUN=9 OR NOUN=10) AND ROPE=O THEN GOTO 1070
IF NOUN=9 THEN 2250
IF LOC#5 AND LOC#13 THEN 1070
IF LOC#5 THEN LOC=4
IF LOC=13 THEN LOC=50
GOTO 3000
IF NOUN=10 THEN 2300
IF LOC#4 THEN 1070
LOC=S: GOTO 3000
IF VERB=8 OR VERB=11 THEN 2320: GOTO 2350
IF NUMP=8 THEN 1070
IF NOUN=34 AND (LOC=19 OR LOC=10 OR LOC=5 OR LOC=7 OR LOC=6 OR LOC=11) THEN 2345
IF STA(NOUN)#LOC THEN 1070
2335 IF NOUN=28 AND M(50)>0 THEN 1070
2345 STA(NOUN)=1: GOTO 3000
2350 IF VERB=9 OR VERB=10 OR VERB=14 THEN 2370: GOTO 2400
2370 IF STA(NOUN)=1 THEN 1070
2380 STA(NOUN)=LOC
2385 IF NOUN=33 THEN 2420
2390 GOTO 2420
2395 IF VERB=12 THEN 2900
2410 IF STA(NOUN)=1 THEN 1070
2420 FOR WT=1 TO NUMW*5-4 STEP 5
2425 IF NOUN#WT(WT) THEN 2480
2430 FOR D=2 TO 4
2435 IF STA(WT(WT+D)) MOD 100 = LOC THEN 2470
2440 FOR M=1 TO NUMM*10-9 STEP 10
2445 IF WT(WT+D) = M(M) THEN 2460
2446 HT= RND(WT(WT+1))/(CURSE+1)
2448 M(M+9)=M(M+9)-HT
2450 IF M(M+4)=0 THEN M(M+4)=1
2455 IF M(M+9)>0 THEN 2460
2456 PRINT "ASSAULT ON \";NOUNS$(M(M)*5-4,M(M)*5);\";HT;\" UNITS"
2457 STA(M(M))=0
2460 NEXT M
2470 NEXT D
2480 NEXT WT
2490 IF NOUN=16 OR VERB=10 THEN 2500
2492 STA(16)=0: GOTO 2493+ RND (4)
2493 N(LOC)=0: GOTO 2500
2494 E(LOC)=0: GOTO 2500
2495 S(LOC)=0: GOTO 2500
2496 W(LOC)=0
2500 IF NTYP=32 THEN 2900
2510 IF NOUN=33 THEN 2520: IF VERB=12 THEN LIGHT=1: GOTO 2900
2520 IF NOUN=29 THEN 2530: LIGHT=1
2530 IF NOUN=30 THEN 2540: LIGHT=1
2540 REM
2550 IF NOUN=11 THEN ROPE=0
2560 IF STA(30)=LOC THEN ROPE=1
2570 IF LOC=12 THEN 3000
2580 W(12)=6:S(12)=11
3000 REM: RE-ACTION
3010 FOR M=1 TO NUMM*10-9 STEP 10
3020 IF STA(M(M))=0 THEN GOSUB 3800
3030 NEXT M
3040 IF STA(35)=0 AND STA(34)=2 THEN 3800
3050 GOTO 4000
3060 REM: MGNS SUB
3070 MRR=STA(M(M)) MOD 100
3080 IF (STA(M(M+1)) MOD 100)=MRR AND M(M+4)=0 THEN 3900
3090 IF MRR=LOC THEN 3860
3100 M(M+2)=M(M+2)+M(M+3) MOD 6
3110 IF M(M+2)=0 THEN RETURN
3120 GOTO 3850+ RND (4)
3130 NLOC=N(M(MRR)): GOTO 3855
3140 NLOC=E(MRR): GOTO 3855
3150 NLOC=S(MRR): GOTO 3855
3160 NLOC=W(MRR): GOTO 3855
3170 IF NLOC<1 THEN RETURN
3180 IF MRR=0 THEN RETURN
3190 IF MRR=3 THEN RETURN
3200 STA(M(M))=NLOC+STA(M(M))-MRR: RETURN
3210 GOTO 3850+ RND (4)
3220 IF STA(M(M+1))=MRR AND M(M+1)=0 THEN 3900
3230 IF MRR=LOC THEN 3860
3240 M(M+2)=M(M+2)+M(M+3) MOD 6
3250 IF M(M+2)=0 THEN RETURN
3260 M(M+4)=M(M+4)+1
3270 KP=(M(M+5)-(STA(M(M+1))=-1)*40+9*(M(M+4)-2)*M(M+9)/100+CURSE
3280 IF KP>60 THEN KP=60
3290 SAP=(M(M+6)+9*(M(M+4)-2)*M(M+9)/100+CURSE
3300 IF SAP>70 THEN SAP=70
3310 IF SRR=80 THEN SRR=80
3320 PRINT "ATTACK BY \";NOUNS$(M(M)-1)*5+1,M(M)*5)
3330 IF R1= RND (100): R2= RND (100): R3= RND (100)
3340 IF KP>R1 THEN 3920
3350 IF SRR>R2 THEN 3940
3360 IF STA(M(M+1))=-1 THEN RETURN
3370 GOTO 3960
3895 RETURN
3900 STA(M(M))=M(M+8)
3905 STA(M(M+1))=M(M+8)
3910 RETURN
3920 VTAB 23: TAB 1: PRINT "THE";NOUNS((M(M)-1)*5+1,M(M)*5);"KILLED YOU!"
3924 PRINT KP,R1
3925 END
3940 FOR I=1 TO NUMN
3945 IF NTYP(I)=16 AND STA(I)=-1 THEN STA(I)=M(M+8)
3950 NEXT I
3957 PRINT "ALL YOUR REWARDS STOLEN"
3959 GOTO 3900
3960 PRINT "HE TOOK BACK HIS VALUABLE"
3965 GOTO 3900
4000 REM OUTPUT
4020 FOR I=3 TO 9: VTAB I: TAB 2: PRINT " ": NEXT I
4040 GOTO 4000+100*LOC
4060 POKE 50,63: VTAB 3: TAB 2: PRINT LOS$: POKE 50,255: PRINT " ": RETURN
4080 VTAB 23: TAB 1:
4095 IF LIGHT=1 OR LOC<3 OR LOC=19 THEN 9100
4097 PRINT "IT IS VERY DARK"
4099 GOTO 9100
4100 LOC$="MOUTH ": GOSUB 4070
4199 GOTO 4090
4200 LOC$="TREE ROOM ": GOSUB 4070
4299 GOTO 4090
4300 LOC$="WRITING ROOM": GOSUB 4070
4399 GOTO 4090
4400 LOC$="PIT ": GOSUB 4070
4499 GOTO 4090
4500 LOC$="SOUTH LAKE ": GOSUB 4070
4599 GOTO 4090
4600 LOC$="WEST LAKE ": GOSUB 4070
4699 GOTO 4090
4700 LOC$="NORTH LAKE ": GOSUB 4070
4799 GOTO 4090
4800 LOC$="MAZE ROOM ": GOSUB 4070
4899 GOTO 4090
4900 LOC$="FROZEN RIVER": GOSUB 4070
4999 GOTO 4090
5000 LOC$="RIVER ROOM ": GOSUB 4070
5099 GOTO 4090
5100 LOC$="HUB ROOM ": GOSUB 4070
5199 GOTO 4090
5200 LOC$="ICE ROOM ": GOSUB 4070
5299 GOTO 4090
5300 LOC$="CHIMNEY ": GOSUB 4070
5399 GOTO 4090
5400 LOC$="GOLD ROOM ": GOSUB 4070
5499 GOTO 4090
5500 LOC$="BONES ": GOSUB 4070
5510 IF STA(35)1-1 THEN CURSE=CURSE+15
5599 GOTO 4090
5600 LOC$="BATS ": GOSUB 4070
5699 GOTO 4090
5700 LOC$="GHOST ROOM ": GOSUB 4070
5799 GOTO 4090
5800 LOC$="MISTY LAKE ": GOSUB 4070
5899 GOTO 4090
5900 LOC$="SWIFT RIVER ": GOSUB 4070
5999 GOTO 4090
6000 LOC$="INTERSECTION": GOSUB 4070
6099 GOTO 4090
6100 GOTO 6000
6200 GOTO 6000
6999 GOTO 4090
8899 LOC$="OVER FALLS ": GOSUB 4070
8910 VTAB 23: TAB 1: GOTO 9090
9000 LOC$="YOUR HOME ": GOSUB 4070
9005 AMT=0
9010 IF STA(25)1-1 THEN AMT=AMT+13
9020 IF STA(26)1-1 THEN AMT=AMT+22
9030 IF STA(27)1-1 THEN AMT=AMT+8
166 Games

9040 IF STA(28)=-1 THEN AMT=AMT+5
9050 VTAB 23: TAB 1
9060 IF AMT=0 THEN 9090
9070 PRINT "YOU HAVE FOUND $";AMT;""; RND (900)+100;" IN TREASURES"
9080 IF AMT>13 THEN PRINT "NICE SPELUNKING!"
9090 PRINT "GOOD-BYE"
9099 END

9100 FOR I=2 TO 10: VTAB I: TAB 33: PRINT "": NEXT I
9105 FOR I=1 TO NUMN-1
9110 IF STA(I) MOD 100=LOC THEN 9360
9120 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9130 J=(J+1) MOD 4: IF J=0 THEN 9350
9140 PRINT "";
9150 NEXT I
9160 NEXT I

9170 FOR I=1 TO NUMN-1
9180 IF STA(I)>=0 THEN 9480
9190 PRINT NOUNS$((I-1)*5+1,I*5);• ";
9200 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9210 NEXT I
9220 END

9300 FOR I=1 TO NUMN-1
9310 FOR I=1 TO NUMN-1
9320 IF STA(I) MOD 100=LOC THEN 9360
9330 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9340 FOR J=1 TO LOC THEN 9350
9350 PRINT "": TAB 2
9360 NEXT J
9370 NEXT I
9380 IF STA(I)=0 THEN 9480
9390 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9400 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9410 NEXT I
9420 FOR I=1 TO NUMN-1
9430 IF STA(I)=0 THEN 9480
9440 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9450 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9460 NEXT I
9470 IF STA(I)=0 THEN 9480
9480 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9490 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9500 NEXT I
9510 IF STA(I)=0 THEN 9480
9520 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9530 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9540 NEXT I
9550 IF STA(I)=0 THEN 9480
9560 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9570 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9580 NEXT I
9590 IF STA(I)=0 THEN 9480
9600 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9610 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9620 NEXT I
9630 IF STA(I)=0 THEN 9480
9640 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9650 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9660 NEXT I
9670 IF STA(I)=0 THEN 9480
9680 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9690 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9700 NEXT I
9710 IF STA(I)=0 THEN 9480
9720 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9730 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9740 NEXT I
9750 IF STA(I)=0 THEN 9480
9760 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9770 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9780 NEXT I
9790 IF STA(I)=0 THEN 9480
9800 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9810 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9820 NEXT I
9830 IF STA(I)=0 THEN 9480
9840 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9850 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9860 NEXT I
9870 IF STA(I)=0 THEN 9480
9880 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9890 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9900 NEXT I
9910 IF STA(I)=0 THEN 9480
9920 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9930 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9940 NEXT I
9950 IF STA(I)=0 THEN 9480
9960 PRINT NOUNS$((I-1)*5+1,I*5);" ";
9970 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9980 NEXT I
9990 IF STA(I)=0 THEN 9480
1000 PRINT NOUNS$((I-1)*5+1,I*5);" ";
1001 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
1002 NEXT I

3000 REM INITIALIZE ROUTINE
3010 DIM INS(40),NOUNS$(255),VERBSS(255),W1$(5),W2$(5),W3$(5),NTYP(50),VTYP(50)
3020 DIM N(50),E(50),S(50),W(50)
3030 TEXT : CALL -936
3040 DIM LOC$(26),SPC$(5),M(6*10)
3050 SPCs=*
3060 NUMW=6
3070 DIM WT(5*NUMW)
3080 LOC=1
3090 REM INITIALIZE VARIABLES
3100 REM SHOULD BE READ AND DATA STATTS
3110 NUMW$(LEN(NOUNS$)+1)="N NE E SE S SW W NW UP DOWN"
3120 NOUNS$(LEN(NOUNS$)+1)="CAVE LAKE RIVER TREE"
3130 NOUNS$(LEN(NOUNS$)+1)="AX BOMB CURSE FIRE KNIFE"
3140 NOUNS$(LEN(NOUNS$)+1)="CLAM BATS GHOST OGRE"
3150 NOUNS$(LEN(NOUNS$)+1)="CHEST GOLD PEARL LAMP"
3160 NOUNS$(LEN(NOUNS$)+1)="RAFT ROPE TENT TRUCK LIGHT"
3170 NOUNS$(LEN(NOUNS$)+1)="WATER APPLE ICE"
3180 NOUNS$(LEN(NOUNS$)+1)="*****"
3190 NUMN=37
3200 VERBSS$(LEN(VERBSS$)+1)="GO JUMP RUN WALK DRIVE CLIMB"
3210 VERBSS$(LEN(VERBSS$)+1)="DIG"
3220 VERBSS$(LEN(VERBSS$)+1)="CARRY DROPPUT TAKE USE HIT FIGHT"
3230 VERBSS$(LEN(VERBSS$)+1)="HELP KILL STOP HIT FIGHT"
3240 VERBSS$(LEN(VERBSS$)+1)="RUB"
3250 VERBSS$(LEN(VERBSS$)+1)="START DRIVE"
3260 VERBSS$(LEN(VERBSS$)+1)="DRINK EAT BITE"
3270 VERBSS$(LEN(VERBSS$)+1)="*****"
3280 NUMW=26
30310 FOR I=1 TO 10:NTYP(I)=1: NEXT I
30320 FOR I=11 TO 14:NTYP(I)=2: NEXT I
30330 FOR I=15 TO 19:NTYP(I)=4: NEXT I
30340 FOR I=20 TO 24:NTYP(I)=8: NEXT I
30350 FOR I=25 TO 28:NTYP(I)=16: NEXT I
30360 FOR I=29 TO 33:NTYP(I)=32: NEXT I
30370 FOR I=34 TO 35:NTYP(I)=64: NEXT I
30380 NTYP(36)=32
30410 FOR I=1 TO 6:VTYP(I)=1: NEXT I
30412 VTYP(2)=11:VTYP(6)=3
30420 VTYP(7)=2
30430 FOR I=8 TO 11:VTYP(I)=16: NEXT I
30432 VTYP(12)=36:VTYP(13)=36:VTYP(14)=4
30440 FOR I=15 TO 19:VTYP(I)=8: NEXT I
30442 VTYP(17)=40
30450 VTYP(20)=16
30460 FOR I=21 TO 22:VTYP(I)=32: NEXT I
30470 FOR I=23 TO 25:VTYP(I)=64: NEXT I
30500 FOR I=1 TO 14:STA(I)=0: NEXT I
30510 STA(15)=4:STA(16)=3:STA(17)=15
30520 STA(18)=13:STA(19)=1:STA(20)=18
30530 STA(21)=16:STA(22)=15:STA(23)=17
30540 STA(24)=14:STA(25)=16:STA(26)=14
30550 STA(27)=18:STA(28)=12:STA(29)=5
30560 STA(30)=9:STA(31)=1:STA(32)=1
30570 STA(33)=1:STA(34)=0:STA(35)=0
30580 STA(36)=12
30600 FOR I=1 TO 50:N(I)=0:E(I)=0:5(I)=0:W(I)=0: NEXT I
30610 N(1)=50:N(3)=2:N(5)=18:N(7)=9:N(8)=8:N(9)=7
30630 N(19)=1:N(20)=8:N(22)=11
30640 FOR I=2 TO 20:E(I)=E(I+1)-E(I): NEXT I
30650 E(21)=11
30670 S(13)=11:S(18)=5
30670 S(21)=22:S(22)=16
30690 W(12)=6=W(14)=11=W(15)=11=W(20)=14=W(21)=17=W(22)=21
30700 POKE 50,63
30710 VTAB 24: GOSUB 31999: VTAB 1: GOSUB 31999: VTAB 16: GOSUB 31999
30720 VTAB 2: TAB 1
30730 FOR I=2 TO 23: PRINT " ";: TAB 29: IF I<11 THEN PRINT " ";: TAB 39: PRINT " ";: NEXT I
30800 FOR I=1 TO 60:M(I)=0: NEXT I
30810 M(1)=24:M(2)=26:M(4)=2:M(6)=60:M(7)=30:M(8)=55:M(9)=14:M(10)=100
30830 M(21)=23:M(24)=6:M(29)=17:M(30)=50
30840 M(31)=20:M(32)=27:M(33)=1:M(36)=90:M(37)=60:M(38)=65:M(39)=18:M(40)=60
30850 M(41)=36:M(42)=28:M(43)=1:M(47)=60:M(49)=12:M(50)=25
30860 M(51)=22:M(53)=1:M(59)=15:M(60)=75
30890 NUM=6
30900 WT(1)=15:WT(2)=100:WT(3)=24:WT(4)=0:WT(5)=0
30910 WT(6)=16:WT(7)=150:WT(8)=24:WT(9)=22:WT(10)=36
30930 WT(16)=19:WT(17)=50:WT(18)=24:WT(19)=20:WT(20)=0
30940 WT(21)=33:WT(22)=30:WT(23)=23:WT(24)=2:WT(25)=0
30950 WT(26)=36:WT(27)=40:WT(28)=21:WT(29)=0:WT(30)=0
30990 GOTO 4000
31999 TAB 1: PRINT " ";: RETURN
32000 PRINT (PEEK (202)+ PEEK (203)*256)-(PEEK (204)+ PEEK (205)*256): END
LIFE for your Apple

by Richard F. Suitor

Perhaps the best known computer game/simulation of all time is Life. First appearing over a decade ago, Life is now found on many mainframes, minis, and micros. It is used as a biology model, as a math aid, and of course, as a game! Now, Dick Suitor's ultra-fast Life program will bring life to your Apple, too!

A listing of LIFE for the Apple II is described briefly here. The generation calculations are in assembly language. The display is initiated in BASIC and the routines are called from BASIC, which will slow down the generation time if desired.

The entire \( 40 \times 48 \) low resolution graphics display is used. An unoccupied cell is 0 (black). An occupied one is 11 (pink). During the first half of a generation, cells that will die are set to color 8 (brown). Those to be born are set to color 3 (violet). During this stage, bit 3 set indicates a cell is alive this generation; bits 0 and 1 set indicate a cell will be alive the next generation. During the second half (mop-up) those with bits 0 set are set alive (color 11), the rest are set to zero.

The BASIC program allows you to set individual cells alive, and to set randomly 1 in N alive in a rectangular region. The boundaries \( X = 0 \) and 39, \( Y = 0 \) and 47 do not change, but may be initialized. At the start of the program, NO PADDLE INTERVAL? is requested. If during the program the paddle reads close to 255 (as it will if none is connected) the number input here will be used instead. Zero is fastest—several generations per second. Entering 200 gives a few seconds per generation.

When \( X \) and \( Y \) coordinates are requested, put in the coordinates for any cells to be set alive. A negative \( X \) terminates this phase. Setting \( X = N \) and a negative \( Y \) will initialize a rectangular region to 1 in N randomly occupied and terminate the initialization. The boundaries of the rectangular region must be input and may be anywhere in the full display. A glider gun can be fit vertically in the display. However, don't initialize for \( Y > = 40 \) (other than random) for the scrolling during initialization input will wipe it out.

Before RUNning the BASIC program, set LOMEM: 2500 to avoid overwriting the subroutines.
1 REM ****************************
2 REM *
3 REM * LIFE FOR YOUR APPLE *
4 REM *
5 REM * COPYRIGHT (C) 1981 *
6 REM * MICRO INK, INC. *
7 REM * CHELMSFORD, MA 01824 *
8 REM * ALL RIGHTS RESERVED *
9 REM *
10 REM ****************************
11 REM
12 CALL -936: PRINT "MICRO/APPLE VOLUME 1."
13 VTAB 3: PRINT "SEE 'LIFE FOR YOUR APPLE'"
14 VTAB 5: PRINT "BY RICHARD F. SUITOR"
15 VTAB 7: PRINT "BASED ON JOHN CONWAY'S GAME OF LIFE"
18 IF PEEK (-16384)<160 THEN 18
50 TEXT :GEN=2088:MOP=2265
60 DIM A$(7)
70 K1=1:K2=1:KBD=-16384
99 GOTO 1000
100 REM
102 POKE -16302,0
103 GOTO 130
104 FOR I=1 TO K3
105 CALL GEN
107 FOR K=1 TO K1: NEXT K
110 CALL MOP
112 FOR K=1 TO K2: NEXT K
120 NEXT I
130 REM
131 KX= PDL (0)-10
132 IF KX>240 THEN KX=KX1
135 IF KX<0 THEN KX=0
140 K1=KX*6
150 K2=KX*2
155 K3=500/(K1+50)+1
156 IF PEEK (KBD)=174 THEN 156
157 IF PEEK (KBD)=160 THEN 1000
158 IF PEEK (KBD)<141 THEN 104
165 TEXT
170 END
1000 GR
1010 CALL -936
1020 INPUT "NO PADDLE TIME INTERVAL ",KX1
1100 COLOR=11: INPUT "INPUT X,Y ",X,Y
1105 IF Y<0 THEN 1800
1110 IF X<0 OR Y<0 THEN 2500
1120 IF X>39 OR Y>39 THEN 1100
1130 PLOT X,Y: GOTO 1100
1800 INPUT "X DIRECTION LIMITS ",I1,I2
1810 IF I1<0 OR I2>39 OR I1>I2 THEN 1800
1820 INPUT "Y DIRECTION LIMITS ",J1,J2
1830 IF J1<0 OR J2>47 OR J1>J2 THEN 1820
2000 CALL -936: GR
2001 POKE -16302,0
2002 CALL -1998
2005 FOR I=11 TO 12
2010 FOR J=J1 TO J2: COLOR=11: IF RND (X) THEN COLOR=0
2020 PLOT I,J
2030 NEXT J
2040 NEXT I
2100 GOTO 100
2500 POKE -16302,0
2510 COLOR=0
2520 FOR K=40 TO 47
2530 HLIN 0,39 AT K
2540 NEXT K
2550 GOTO 100
9000 END
Games

0800 1 ;*****************************************************
0800 2 ;* GAME OF LIFE FOR APPLE II *
0800 3 ;* BY *
0800 4 ;* RICHARD SUITOR *
0800 5 ;* LIFE *
0800 6 ;* COPYRIGHT (C) 1981 *
0800 7 ;* MICRO INK, INC. *
0800 8 ;* ALL RIGHTS RESERVED *
0800 9 ;*****************************************************
0800 10 ; L
0800 11 ; 800 12 ;
0800 13 ;
0800 14 ;
0800 15 ;
0800 16 ; ORG $800
0800 17 ; OBJ $800
0800 18 ; L
0800 19 ; ENTER AT GENO AND MOPO ALTERNATELY
0800 20 ; 2088 AND 2265 DEC. RESP.
0800 21 OLLN EPZ $02 ; OLD HORIZ LINE
0800 22 NWLN EPZ $04 ; NEW LINE
0800 23 SUM1 EPZ $06 ; # OF OCC. CELLS IN 3X3
0800 24 SUM2 EPZ $07 ; 1,2 FOR OLD, NEW
0800 25 BUF1 EQU $0940 ; A C VERT. OCC. #
0800 26 BUF2 EQU $0942 ;
0800 27 BUF3 EQU $093F ;
0800 28 BUF4 EQU $0970 ;
0800 29 BUF5 EQU $0972 ;
0800 30 BUF6 EQU $096F ;
0800 31 NWLN LDA NWLN+01
0800 32 OLLN+01 STA NWLN
0800 33 LWLN LDA NWLN
0800 34 OLLN+01 STA OLLN
0800 35 NWLN CLC
0800 36 NWLN+01 ADC 80
0800 37 OLLN+01 STA NWLN
0800 38 NWLN LDA NWLN+01
0800 39 NWLN+01 ADC 80
0800 40 NWLN+01 CMP 08
0800 41 NWLN+01 BNE SAME
0800 42 NWLN+01 STA NWLN
0800 43 NWLN+01 ADC 27
0800 44 NWLN+01 CMP 52
0800 45 NWLN+01 BPL LAST
0800 46 NWLN+01 STA NWLN
0800 47 NWLN+01 LDA 04
0800 48 NWLN+01 STA NWLN+01
0800 49 NWLN+01 CLC
0800 50 NWLN+01 RTS
0800 51 NWLN+01 RTS
0800 52 NWLN+01 RTS
0800 53 NWLN+01 RTS
0800 54 NWLN+01 RTS
0800 55 NWLN+01 RTS
0800 56 NWLN+01 RTS
0800 57 NWLN+01 RTS
0800 58 NWLN+01 RTS
0800 59 NWLN+01 RTS
0800 60 NWLN+01 RTS
0800 61 NWLN+01 RTS
0800 62 NWLN+01 RTS
0800 63 NWLN+01 RTS
0800 64 NWLN+01 RTS
0800 65 NWLN+01 RTS
0800 66 NWLN+01 RTS
0800 67 NWLN+01 RTS
0800 68 NWLN+01 RTS
0800 69 NWLN+01 RTS
0800 70 NWLN+01 RTS
0800 71 NWLN+01 RTS
0800 72 NWLN+01 RTS
0800 73 NWLN+01 RTS
0800 74 NWLN+01 RTS
0800 75 NWLN+01 RTS

0800 80 GEN1 JSR INIT
0800 81 GEN2 JSR NWLN
0800 82 GEN3 JSR GEN2
0800 83 GEN4 JSR NWLN
0800 84 GEN5 JSR GEN3
0800 85 GEN6 JSR NWLN
0800 86 GEN7 JSR GEN2
0800 87 GEN8 JSR NWLN
0800 88 GEN9 JSR GEN3
0800 89 GEN10 JSR NWLN
0800 90 GEN11 JSR GEN3
0800 91 GEN12 JSR NWLN
0800 92 GEN13 JSR GEN3
0800 93 GEN14 JSR NWLN
0800 94 GEN15 JSR GEN3
0800 95 GEN16 JSR NWLN
0800 96 GEN17 JSR GEN3
0800 97 GEN18 JSR NWLN
0800 98 GEN19 JSR GEN3
0800 99 GEN20 JSR NWLN
0800 A0 GEN21 JSR GEN3
0800 A1 GEN22 JSR NWLN
0800 A2 GEN23 JSR GEN3
0800 A3 GEN24 JSR NWLN
0800 A4 GEN25 JSR GEN3
0800 A5 GEN26 JSR NWLN
0800 A6 GEN27 JSR GEN3
0800 A7 GEN28 JSR NWLN
0800 A8 GEN29 JSR GEN3
0800 A9 GEN30 JSR NWLN
0800 AA GEN31 JSR GEN3
0800 AB GEN32 JSR NWLN
0800 AC GEN33 JSR GEN3
0800 AD GEN34 JSR NWLN
0800 AE GEN35 JSR GEN3
0800 AF GEN36 JSR NWLN
0800 B0 GEN37 JSR GEN3
0800 B1 GEN38 JSR NWLN
0800 B2 GEN39 JSR GEN3
0800 B3 GEN40 JSR NWLN
0800 B4 GEN41 JSR GEN3
0800 B5 GEN42 JSR NWLN
0800 B6 GEN43 JSR GEN3
0800 B7 GEN44 JSR NWLN
0800 B8 GEN45 JSR GEN3
0800 B9 GEN46 JSR NWLN
0800 BA GEN47 JSR GEN3
0800 BB GEN48 JSR NWLN
0800 BC GEN49 JSR GEN3
0800 BD GEN50 JSR NWLN
0800 BE GEN51 JSR GEN3
0800 BF GEN52 JSR NWLN
0800 C0 GEN53 JSR GEN3
0800 C1 GEN54 JSR NWLN
0800 C2 GEN55 JSR GEN3
0800 C3 GEN56 JSR NWLN
0800 C4 GEN57 JSR GEN3
0800 C5 GEN58 JSR NWLN
0800 C6 GEN59 JSR GEN3
0800 C7 GEN60 JSR NWLN
0800 C8 GEN61 JSR GEN3
0800 C9 GEN62 JSR NWLN
0800 CA GEN63 JSR GEN3
0800 CB GEN64 JSR NWLN
0800 CC GEN65 JSR GEN3
0800 CD GEN66 JSR NWLN
0800 CE GEN67 JSR GEN3
0800 CF GEN68 JSR NWLN
0800 D0 GEN69 JSR GEN3
0800 D1 GEN70 JSR NWLN
0800 D2 GEN71 JSR GEN3
0800 D3 GEN72 JSR NWLN
0800 D4 GEN73 JSR GEN3
0800 D5 GEN74 JSR NWLN
0800 D6 GEN75 JSR GEN3
0800 D7 GEN76 JSR NWLN
0800 D8 GEN77 JSR GEN3
0800 D9 GEN78 JSR NWLN
0800 DA GEN79 JSR GEN3
0800 DB GEN80 JSR NWLN
0800 DC GEN81 JSR GEN3
0800 DD GEN82 JSR NWLN
0800 DE GEN83 JSR GEN3
0800 DF GEN84 JSR NWLN
0800 E0 GEN85 JSR GEN3
0800 E1 GEN86 JSR NWLN
0800 E2 GEN87 JSR GEN3
0800 E3 GEN88 JSR NWLN
0800 E4 GEN89 JSR GEN3
0800 E5 GEN90 JSR NWLN
0800 E6 GEN91 JSR GEN3
0800 E7 GEN92 JSR NWLN
0800 E8 GEN93 JSR GEN3
0800 E9 GEN94 JSR NWLN
0800 EA GEN95 JSR GEN3
0800 EB GEN96 JSR NWLN
0800 EC GEN97 JSR GEN3
0800 ED GEN98 JSR NWLN
0800 EE GEN99 JSR GEN3
0800 EF GEN100 JSR NWLN
0800 F0 GEN101 JSR GEN3
0800 F1 GEN102 JSR NWLN
0800 F2 GEN103 JSR GEN3
0800 F3 GEN104 JSR NWLN
0800 F4 GEN105 JSR GEN3
0800 F5 GEN106 JSR NWLN
0800 F6 GEN107 JSR GEN3
0800 F7 GEN108 JSR NWLN
0800 F8 GEN109 JSR GEN3
0800 F9 GEN110 JSR NWLN
0800 FA GEN111 JSR GEN3
0800 FB GEN112 JSR NWLN
0800 FC GEN113 JSR GEN3
0800 FD GEN114 JSR NWLN
0800 FE GEN115 JSR GEN3
0800 FF GEN116 JSR NWLN
0854 1003   76  BPL GEN4
0856 FE7009   77  INC BUF2,X
0859 2908   78  GEN4 AND 08
085B F006   79  BEQ GEN5
085D FE7009   80  INC BUF2,X
0860 FE4009   81  INC BUF1,X
0863 88   82  GEN5 DEY
0864 CA   83  DEX
0865 10CE   84  BPL GEN6
0867 A026   85  LDY 26
0869 18   86  CLC
086A A5D09   87  LDA BUF1+27
086D 6D5A09   88  ADC BUF1+26
0870 8506   89  STA SUM1
0872 A6809   90  LDA BUF2+27
0875 6D8A09   91  ADC BUF2+26
0878 8507   92  STA SUM2
087A 18   93  ;COMP OCC #S IN 3X3 & CHANGE COLOR
087A A06   94  GNLP CLC
087B A506   95  LDA SUM1
087D 793F09   96  ADC BF1M,Y
0880 38   97  SEC
0881 F9409   98  SBC BF1P,Y
0884 8506   99  STA SUM1
0886 C903  100  CMP 03
0888 F00E  101  BEQ GEN9
088A 9004  102  BCC GEN8
088C C904  103  CMP 04
088E F00E  104  BEQ GEN10
0890 B102  105  GEN8 LDA (OLLN),Y
0892 F00A  106  BEQ GEN10
0894 298F  107  AND 0F
0896 5004  108  BVC GEN16
0898 B102  109  GEN9 LDA (OLLN),Y
089A 0930  110  ORA 30
089C 9102  111  GN16 STA (OLLN),Y
089E 18  112  GN10 CLC
089F A507  113  LDA SUM2
08A1 796F09  114  ADC BF2M,Y
08A4 38  115  SEC
08A5 F97209  116  SBC BF2P,Y
08AB 8507  117  STA SUM2
08AA C903  118  CMP 03
08AC F00E  119  BEQ GN12
08AE 9004  120  BCC GN11
08B0 C904  121  CMP 04
08B2 F00E  122  BEQ GN13
08B4 B104  123  GN11 LDA (NWLN),Y
08B6 F00A  124  BEQ GN13
08B8 298F  125  AND 0F
08BA 5004  126  BVC GN15
08BC B104  127  GN12 LDA (NWLN),Y
08BE 0930  128  ORA 03
08C0 9104  129  GN15 STA (NWLN),Y
08C2 88  130  GN13 DEY
08C3 F002  131  BEQ GN14
08C5 10B3  132  BPL GNLP
08C7 4C2B08  133  GN14 JMP GEN1
08CA A904  134  INIT LDA 04
08CC 8505  135  STA NWLN+01
08CE A900  136  LDA 00
08D0 8504  137  STA NWLN
08D2 BD6B09  138  STA BF1P+$26
08D5 8D9B09  139  STA BF2P+$26
08DA 60  140  RTS
08D9 141  ;MOP UP, IF COLOR AND 3=0, REMOVE (COL=0)
08DD 20CA08  142  ;OTHERWISE, ALIVE (COL=11)
08DB 200008  143  MOP0 JSR INIT
08DC 9001  144  MOP1 JSR NWLN
08DF 9001  145  BCC MOP2
08E1 60  146  RTS
08E2 A027  147  MOP2 LDY 27
08E4 B102  148  MOP3 LDA (OLLN),Y
08E6 F00A  149  BEQ MOP5
08E8 297F  150  AND 7F
172

**Ed. note:** To use the LIFE program—

1. LOAD APPLE LIFE
2. BLOAD LIFE
3. LOMEM:2500
4. RUN

The LOMEM instruction is very important!
Apple II Speed Typing Test
With Input Time Clock

by John Broderick, CPA

So, you think you are a pretty fast typist?! Well, then you'll definitely want to take the Apple speed typing test! Find out how many wpm's you're really pushing! And of special interest to the inquisitive—the timed input subroutine used in this program can be used in your own programs as well!

The speed typing test is a must for all Appleliers, like myself, who consider themselves expert typists. However, I did not set out to write a typing test, but to make an input subroutine (GOSUB 8400) which puts the user under the pressure of a time clock.

Try the program below:

2000 call-936:
2010 VW = 10: rem set VTAB
2020 TT = 1: rem set TAB
2030 GOSUB 8400
2040 GOTO 2000

You should hear and see the time at the bottom of the screen with the seconds and tenths of seconds flying by as you type in an alpha-numeric string.

Subroutine 8400 reads the keyboard in line 8434 with K equal to the ASCII number. Line 8447 subtracts 159 from ASCII so that now K is equal to the position of the equivalent character in string A$ (line 8406). So you can see that we are slowly building up two words in W$ at line 8447 by adding (to the end of string W$) the next letter coming in on the keyboard until the ASCII equivalent of carriage return (141) is detected at line 8444.

Now when the princess falls into the snake pit, if she doesn't make the right decision fast enough, the snakes will probably get her.
**SPEED TYPING TEST**

**COPYRIGHT (C) 1981 MICRO INK, INC. CHELMSFORD, MA 01824**

**ALL RIGHTS RESERVED**

REM DEFINE VV=VTAB & TT=TAB
REM THEN GOSUB 8400- THIS DOES THE SAME AS AN ORDINARY INPUT W$

PRINT "MICRO/APPLE VOLUME 1"
PRINT •SEE 'SPEED TYPING TEST"
PRINT •WITH INPUT TIME CLOCK’
PRINT •By JOHN BRODERICK•

IF PEEK (-16384)<>160 THEN

DIM TYPE$(250): POKE 33,36
CALL -936

INPUT "DO YOU WISH TO MAKE UP YOUR OWN TEST SENTENCE Y/N?",TYPE$

IF TYPE$f=•y• THEN

PRINT : PRINT "ENTER TEST SENTENCE Now•:

INPUT TYPE$:

TYPE$=•NoW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR COUNTRY•

CALL -936: PRINT :ERR=O:
PRINT •you ARE TAKING A SPEED TYPING TEST!•

PRINT •TYPE THE NEXT SENTENCE APPEARING ON THE SCREEN AS FAST AS YOU CAN •

FOR I=1 TO 4000: NEXT I: REM ---BODY OF PROGRAM---

CALL -936:ERR=O

VV=13:
REM SET SUBPOINT VTAB

TT=1:
REM SET SUBROUTINE TAB

VTAB (9): TAB 1: PRINT TYPE$:
GOSUB 8400

VTAB (16): TAB 1

IF W$=TYPE$ THEN 510:

REM COMPUTE ERRORS 210- 410

FOR I=LEN(W$) TO LEN(TYPE$):W$(I+l)=B$(1,1): NEXT I

FOR I=1 TO LEN(TYPE$): IF I> LEN(W$) THEN ERR=ERR+l: IF I> LEN(W$) THEN NEXT I

IF W$(I,I)=TYPE$(I,I) THEN ERR=ERR+l: NEXT I

PRINT : PRINT : CALL -198: PRINT • •;ERR;• ERRORS•;• HIT RETURN•: GOTO 520

T=(X*23)+J:L=LEN(TYPE$): IF L<l THEN 520

L=L-(ERR*6): IF L<2 THEN 506

WPM=(L*l2*20)/T

VTAB (24): TAB 30: PRINT WPM;"WPM": VTAB (16): TAB 1: RETURN

PRINT "CORRECT- HIT RETURN•: PRINT : PRINT • •;ERR;• ERRORS• HIT RETURN": GOTO 520

CALL -198: PRINT • •;ERR;" ERRORS•; • HIT RETURN"
Ludwig Von Apple II

by Marc Schwartz

We all know how great the Apple is at generating tunes. Well, Ludwig Von Apple proves this point again. A simple program with a simple verse, Ludwig shows just how easy it can be to make beautiful music with your Apple!

Owners of the Apple II know from demonstration tapes that the Apple can make sounds. Not all know that it can make music. Having prepared a horse racing program, I decided that it would be fitting to start out the game with the bugle call heard at the track. The following program does just that!

A few words of explanation are in order. The series of "POKEs" in line 30 to 240 set up a musical tone subroutine that is called in line 460. Each note is represented by a four digit code in A$. The first three digits of the code determine the note, and the last digit determines the length of the note. Line 410 decodes the first three digits by converting each digit to ASCII (Apple ASCII), subtracting 176 from each to give three numbers, from zero to nine, and then multiplying the first number by the second and adding the third. This is one of many possible ways of generating all the numbers from zero to a large number (ninety in this case) using single digits.

Line 420 takes the number just generated and subtracts it from forty. This is done because the subroutine as written is a bit confusing if you want to make music, since the tones go up as the numbers go down. This step corrects for that. Line 440 determines how long each tone will be. As "ASC(A$(Z + 3) – 176)" increases, the note lengthens: a "1" produces a very short note, and a "6" makes a very long note. For some reason, higher tones come out more brief than lower tones. Line 450 determines the tempo. A larger number speeds up the tune; a smaller one slows it down. Tempo numbers can go from 1 to 255.

When the program reaches line 470, it returns to line 400 to begin decoding the next four digits and playing the next note.

I don't think that Chopin would need to worry about competition from anyone using this program, but it is fun to have a musical computer.
1 REM ************************
2 REM *   
3 REM * LUDWIG VON APPLE II  
4 REM *  MARC SCHWARTZ 
5 REM *  
6 REM *  
7 REM *  
8 REM * COPYRIGHT (C) 1981 
9 REM *  MICRO INK, INC.  
10 REM * CHELMSFORD, MA 01824  
11 REM *  ALL RIGHTS RESERVED 
12 REM *  
13 REM ************************ 
16 CALL -936
18 VTAB 5: PRINT "LUDWIG VON APPLE II"
19 IF PEEK (-16384)<160 THEN 19
20 DIM AS(255)
30 POKE 2,173
40 POKE 3,48
50 POKE 4,192
60 POKE 5,165
70 POKE 6,0
80 POKE 7,32
90 POKE 8,168
100 POKE 9,252
110 POKE 10,165
120 POKE 11,1
130 POKE 12,208
140 POKE 13,4
150 POKE 14,198
160 POKE 15,24
170 POKE 16,240
180 POKE 17,5
190 POKE 18,198
200 POKE 19,1
210 POKE 20,76
220 POKE 21,2
230 POKE 22,0
240 POKE 23,96
300 AS$="001100715211720172017201"
310 AS$(25)="5211521152110071521100710012"
400 FOR Z=1 TO LEN(AS$)-3 STEP 4
410 Z1=( ASC(AS$(Z))-176)*( ASC(AS$(Z+1))-176)+ ASC(AS$(Z+2))-176
420 Z2=40-Z1
430 POKE 0,Z2
440 POKE 24, ASC(AS$(Z+3))-176
450 POKE 1,75
460 CALL 2
470 NEXT Z
480 IF PEEK(-16384)=160 THEN 400
490 IF PEEK (-16384)< 141 THEN 480
500 END
Another Version, by C.R. (Chuck) Carpenter

The machine language routine used by Marc is put into the BASIC program by use of the POKE statement. I was curious to see the type of program used to activate the Apple II on-board speaker. To do this, I converted the decimal values used for the POKE statements into HEX with my TI Programmer. Then I loaded the values into the computer using the system monitor commands that are part of the Apple II functions.

Once I had the program loaded, I used the monitor commands to list an assembled version of the routine, as shown in figure 1. The assembler provides a listing of the program and the mnemonics used with the machine language opcodes. This made it easier to determine what was happening in Marc's program. At this point I wanted to see what would happen if I ran the program by itself—as a machine language routine only.

Because it is somewhat easier to call the routine from a BASIC routine, I entered the BASIC routine shown in figure 2. This way I could also change the values stored in memory location $0000 by using the POKE statement. To initialize the beginning of the routine, I entered a value of $05 into location $0000. According to Marc, this would produce a high frequency output tone and this turned out to be the case.

Now that I had everything set up, I was curious to see why the duration of playing time is not the same for the different tones. To start with, I entered the program with 3 different values at location $0000. As I ran the program I timed the length of playing with a stop watch. The value of 5 played for .18 min., 10 played for .45 min. and 15 played for .85 min. This was in agreement with Marc's findings. As it turns out, the length of time a particular frequency plays is a function of the duration of a cycle. The output continues for a number of cycles and the shorter cycles (higher frequencies) get done sooner. To get the correct musical timing you would need to include variable delay time for each note played. (The time between zero crossings adds up to the same total time per note.)
<table>
<thead>
<tr>
<th>Address</th>
<th>OPCODE</th>
<th>SENSE</th>
<th>Assembler Instruction</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-</td>
<td>0F</td>
<td>???</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001-</td>
<td>00</td>
<td>BRK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002-</td>
<td>AD 30 C0</td>
<td>LDA</td>
<td>$C030</td>
<td></td>
</tr>
<tr>
<td>0005-</td>
<td>A5 00</td>
<td>LDA</td>
<td>$00</td>
<td></td>
</tr>
<tr>
<td>0007-</td>
<td>20 A8 FC</td>
<td>JSR</td>
<td>$FCA8</td>
<td></td>
</tr>
<tr>
<td>000A-</td>
<td>A5 01</td>
<td>LDA</td>
<td>$01</td>
<td></td>
</tr>
<tr>
<td>000C-</td>
<td>D0 04</td>
<td>BNE</td>
<td>$0012</td>
<td></td>
</tr>
<tr>
<td>000E-</td>
<td>C6 18</td>
<td>DEC</td>
<td>$18</td>
<td></td>
</tr>
<tr>
<td>0010-</td>
<td>F0 05</td>
<td>BEQ</td>
<td>$0017</td>
<td></td>
</tr>
<tr>
<td>0012-</td>
<td>C6 01</td>
<td>DEC</td>
<td>$01</td>
<td></td>
</tr>
<tr>
<td>0014-</td>
<td>4C 02 00</td>
<td>JMP</td>
<td>$00002</td>
<td></td>
</tr>
<tr>
<td>0017-</td>
<td>60</td>
<td>RTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0018-</td>
<td>00</td>
<td>BRK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

>LIST
  10 POKE 0,5
  99 END

>CALL 2

>10 POKE 0,10
>RUN

>CALL 2

>10 POKE 0,15
>RUN

>CALL 2
REFERENCE

Introduction 180

An Apple II Programmer's Guide
Rich Auricchio 181

Exploring the Apple II DOS
Andy Hertzfeld 186

Applesoft II Shorthand
Allen J. Lacy 191

The Integer BASIC Token System in the Apple II
Frank D. Kirschner 198

Creating an Applesoft BASIC Subroutine Library
N.R. McBurny 204
Introduction

Everyone should want to know more about his Apple—the way it works and what it can do. This chapter presents five articles which explore in depth one facet of the Apple. Not only are these articles informative, they're also great to have on hand as a reference [hence the chapter title!].

"An Apple II Programmer's Guide" by Rick Auricchio is an overview of the basics of machine language programming. "Exploring DOS" by Andy Hertzfeld provides a quick look into the Apple's disk operating system. "Applesoft II Shorthand" by Allen Lacy provides a look into Applesoft commands and presents a program which can replace the commands with a 'shorthand.' "Integer BASIC Token System" by Frank Kirschner discusses the token system by which integer BASIC programs are stored in memory. "Creating an Applesoft BASIC Subroutine Library" by N.R. McBurny demonstrates the increased flexibility the EXEC command can provide you. From these articles, you're guaranteed to gain a fairly broad base of Apple knowledge.
The new Apple II reference manual provides a good amount of documentation on many of the useful monitor subroutines. Well before the days of that manual, MICRO published a fairly complete guide to those routines. And here it is—a clear and concise programmer's guide which uncovers many monitor features. It should interest novice and expert alike.

Most of the power of the Apple II comes in a "secret" form—almost undocumented software. After several months of coding, experimenting, digging, and writing to Apple, most of the Apple's pertinent software details have come to light.

Although most of the ROM software has been printed in the Apple Reference Manual, its Integer BASIC has not been listed; as a result, this article will be limited to Monitor software. Perhaps when a source listing of Integer BASIC becomes available, we'll be able to interface with some of its many routines.

First Things First

When I took delivery of my Apple (July 1977), all I had was a "preliminary" manual—no goodies like listings or programming examples. My first letter to Apple brought a listing of the Monitor. Seeing what appeared to be a big jumble of instructions, I set out dividing the listing into logical routines while deciphering their input and output parameters. Once this was done, I could look at portions of the code without becoming dizzy.

The Monitor's code suffers from a few ills:

1. Subroutines lack a descriptive "preamble" stating function, calling sequences, and interface details.

2. Many subroutines have several entry points, each of which does something slightly different.

3. Useful routines are not documented in a concise form for user access.
I will concede that, while using a "shoehorn" to squeeze as much function as possible into those tiny ROMs, some shortcuts are to be expected. However, those valuable Comment Cards don't use up any memory space in the finished product—'nuff said.

The Good Stuff

The best way to present the Apple's software interface details is to describe them in tabular form, with further explanation about the more complex ones.

Table 1 outlines the important data areas used by the Monitor. These fields are used both internally by the Monitor, and in user communication with many Monitor routines. Not all of the data fields are listed in table 1.

Table 2 gives a quick description of most of the useful Monitor routines: it contains Name, Location, Function, Input/Output parameters, and Volatile (clobbered) Registers.

Don't hesitate to experiment with these routines—since all the important software is in ROM, you can't clobber anything by trying them out (except what you might have in RAM, so beware).

Using the "User exits"

The Monitor provides a few nice User Exits for us to get our hands into the Monitor. With these, it is a simple matter to "hook in" special I/O and command-processing routines to extend the Apple's capabilities.

Two of the most useful exits are the KEYIN and COUT exits. These routines, central to the function of the Monitor, are called to read the keyboard and output characters to the screen. By placing the address of a user routine in CSWH/L or KSWH/L, we will get control from the Monitor whenever it attempts to read the keys or output to the screen.

As an example of this exit's action, try this: with no I/O board in I/O Slot 5, key-in "Kc5" (5, followed by control K, then Return). You'll have to hit Reset to stop the system.

Here's what happens: Setting the key-board to device 5 causes the Monitor to install $C500 as the "user-exit" address in KSWH/L. This, of course, is the address assigned to I/O Slot 5. Since no board is present, a BRK opcode eventually occurs; the Monitor prints the break and the registers, then reads for another command. Since we still exit to $C500, the process repeats itself endlessly. Reset removes both user exits; you must "re-hook" them after every Reset.

These two exits can enable user editing of keyboard input, printer driver programs, and many other ideas. Their use is limited to your ingenuity.
Table 1: MONITOR Data Areas in Page Zero

<table>
<thead>
<tr>
<th>Name</th>
<th>Loc.</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>WNDLEFT</td>
<td>20</td>
<td>Scrolling window: left side (0-$27)</td>
</tr>
<tr>
<td>WNDWDTH</td>
<td>21</td>
<td>Scrolling window: width (1-$28)</td>
</tr>
<tr>
<td>WNDTOP</td>
<td>22</td>
<td>Scrolling window: top line (0-$16)</td>
</tr>
<tr>
<td>WNDBTM</td>
<td>23</td>
<td>Scrolling window: bottom line (1-$17)</td>
</tr>
<tr>
<td>CH</td>
<td>24</td>
<td>Cursor: horizontal position (0-$27)</td>
</tr>
<tr>
<td>CV</td>
<td>25</td>
<td>Cursor: vertical position (0-$17)</td>
</tr>
<tr>
<td>COLOR</td>
<td>30</td>
<td>Current COLOR for PLOT/HLIN/VLIN functions</td>
</tr>
<tr>
<td>INVFLG</td>
<td>32</td>
<td>Video Format Control Mask: $FF=Normal, $7F=Blinking, $3F=Inverse</td>
</tr>
<tr>
<td>PROMPT</td>
<td>33</td>
<td>Prompt character: printed on GETLN CALL</td>
</tr>
<tr>
<td>CSWL</td>
<td>36</td>
<td>Low PC for user exit on COUT routine</td>
</tr>
<tr>
<td>CSWH</td>
<td>37</td>
<td>High PC for user exit on COUT routine</td>
</tr>
<tr>
<td>KSWL</td>
<td>38</td>
<td>Low PC for user exit on KEYIN routine</td>
</tr>
<tr>
<td>KSWH</td>
<td>39</td>
<td>High PC for user exit on KEYIN routine</td>
</tr>
<tr>
<td>PCL</td>
<td>3A</td>
<td>Low User PC saved here on BRK to Monitor</td>
</tr>
<tr>
<td>PCH</td>
<td>3B</td>
<td>High User PC saved here on BRK to Monitor</td>
</tr>
<tr>
<td>A1L</td>
<td>3C</td>
<td>A1 to A5 are pairs of Monitor work bytes</td>
</tr>
<tr>
<td>A1H</td>
<td>3D</td>
<td></td>
</tr>
<tr>
<td>A2L</td>
<td>3E</td>
<td></td>
</tr>
<tr>
<td>A2H</td>
<td>3F</td>
<td></td>
</tr>
<tr>
<td>A3L</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>A3H</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>A4L</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>A4H</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>A5L</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>A5H</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>ACC</td>
<td>45</td>
<td>User AC saved here on BRK to Monitor</td>
</tr>
<tr>
<td>XREG</td>
<td>46</td>
<td>User X saved here on BRK to Monitor</td>
</tr>
<tr>
<td>TREG</td>
<td>47</td>
<td>User Y saved here on BRK to Monitor</td>
</tr>
<tr>
<td>STATUS</td>
<td>48</td>
<td>User P status saved here on BRK to Monitor</td>
</tr>
<tr>
<td>SPNT</td>
<td>49</td>
<td>User Stack Pointer saved here on BRK</td>
</tr>
</tbody>
</table>

Page 2 ($0200-$02FF) is used as the KEYIN Buffer.

Pages 4-7 ($0400-$07FF) are used as the Screen Buffer.
Page 8 ($0800-$08FF) is the "secondary" Screen Buffer.

Another useful exit is the Control Y command exit. Upon recognition of Control Y, the Monitor issues a JSR to location $03F8. Here the user can process commands by scanning the original typed line or reading another. This exit is often very useful as a shorthand method of running a program. For example, when you're going back and forth between the Monitor and the Mini-Assembler, typing "F666G" is a bit tiresome. By placing a JMP $F666 in location $02F8, you can enter the Mini-Assembler via a simple Control Y.

Upon being entered from the Monitor at $03F8, the registers are garbage. Locations A1 and A2 contain converted values from the command (if any), and an RTS gets you neatly back into the Monitor. Figure 1 shows this in more detail.
Table 2: MONITOR ROUTINES

<table>
<thead>
<tr>
<th>Name</th>
<th>Loc.</th>
<th>Steps</th>
<th>On</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLOT</td>
<td>F800</td>
<td>AC</td>
<td></td>
<td>Plot a point. CCLR contains color in both halves of byte ($00-$FF). AC: y-coord, Y: x-coord.</td>
</tr>
<tr>
<td>CLRSCR</td>
<td>F832</td>
<td>AC,Y</td>
<td></td>
<td>Clear screen - graphics mode.</td>
</tr>
<tr>
<td>INSTDSP</td>
<td>F8D0</td>
<td>ALL</td>
<td></td>
<td>Disassemble instruction at PCH/PCL.</td>
</tr>
<tr>
<td>PRNTYX</td>
<td>F940</td>
<td>AC</td>
<td></td>
<td>Print contents of Y and X as 4 hex digits.</td>
</tr>
<tr>
<td>PRBL2</td>
<td>F94C</td>
<td>AC,X</td>
<td></td>
<td>Print blanks: X is number to print.</td>
</tr>
<tr>
<td>PREAD</td>
<td>FB1E</td>
<td>AC,Y</td>
<td></td>
<td>Read paddle. X: paddle number 0-3.</td>
</tr>
<tr>
<td>SETTXT</td>
<td>FB39</td>
<td>AC</td>
<td></td>
<td>Set TEXT mode.</td>
</tr>
<tr>
<td>SETGR</td>
<td>FB40</td>
<td>AC</td>
<td></td>
<td>Set GRAPHIC mode (GR).</td>
</tr>
<tr>
<td>VTAB</td>
<td>FC22</td>
<td>AC</td>
<td></td>
<td>VTAB to row in AC (0-$17).</td>
</tr>
<tr>
<td>CLREOP</td>
<td>FC42</td>
<td>AC,Y</td>
<td></td>
<td>Clear to end-of-page.</td>
</tr>
<tr>
<td>HOME</td>
<td>FC58</td>
<td>AC,Y</td>
<td></td>
<td>Home cursor and clear screen.</td>
</tr>
<tr>
<td>SCROLL</td>
<td>FC70</td>
<td>AC,Y</td>
<td></td>
<td>Scroll up one line.</td>
</tr>
<tr>
<td>CLREOL</td>
<td>FC9C</td>
<td>AC,Y</td>
<td></td>
<td>Clear to end-of-line.</td>
</tr>
<tr>
<td>NXTA4</td>
<td>FCB4</td>
<td>AC</td>
<td></td>
<td>Increment A4 (16 bits), then do NXTA1.</td>
</tr>
<tr>
<td>NXTA1</td>
<td>FCBA</td>
<td>AC</td>
<td></td>
<td>Increment A1 (16 bits). Set carry if result &gt;= A2.</td>
</tr>
<tr>
<td>RDKEY</td>
<td>FD0C</td>
<td>AC,Y</td>
<td></td>
<td>Get a key from the keyboard.</td>
</tr>
<tr>
<td>RDCHAR</td>
<td>FD35</td>
<td>AC,Y</td>
<td></td>
<td>Get a key, also handles ESCAPE functions.</td>
</tr>
<tr>
<td>GETLN</td>
<td>FD6A</td>
<td>ALL</td>
<td></td>
<td>Get a line of text from the keyboard, up to the carriage return. Normal mode for Monitor. X returned with number of characters typed in.</td>
</tr>
<tr>
<td>CROUT</td>
<td>FD8E</td>
<td>AC,Y</td>
<td></td>
<td>Print a carriage return.</td>
</tr>
<tr>
<td>PRBYTE</td>
<td>FDDA</td>
<td>AC</td>
<td></td>
<td>Print contents of AC as 2 hex digits.</td>
</tr>
<tr>
<td>COUT</td>
<td>FDED</td>
<td>AC,Y</td>
<td></td>
<td>Print character in AC; also works for CR, BS, etc.</td>
</tr>
<tr>
<td>PRERR</td>
<td>FF2D</td>
<td>AC,Y</td>
<td></td>
<td>Print &quot;ERR&quot; and bell.</td>
</tr>
<tr>
<td>BELL</td>
<td>FF3A</td>
<td>AC,Y</td>
<td></td>
<td>Print bell.</td>
</tr>
<tr>
<td>RESET</td>
<td>FF59</td>
<td>--</td>
<td></td>
<td>RESET entry to Monitor - initialize.</td>
</tr>
<tr>
<td>MON</td>
<td>FF65</td>
<td>--</td>
<td></td>
<td>Normal entry to 'top' of Monitor when running.</td>
</tr>
</tbody>
</table>
| SWEET16| F689 | None  |     | SWEET16 is a 16-bit machine language interpreter. [See: SWEET16: The 6502 Dream Machine, Steve Wozniak,] [BYTE, Vol. 2, No. 11, November 1977, pages 150-159.]

Command typed:

*1234.F5A7Yc

Upon entry at $03F8, the following exists:

A1L ($3C) contains $34
A1H ($3D) contains $12
A2L ($3E) contains $A7
A2H ($3F) contains $F5

Figure 1
Hardware Features

One of the best hardware facilities of the Apple II, the screen display, is also the "darkest"—somewhat unknown. Here's what I've found out about it.

The screen buffer resides in memory pages 4 through 7, locations $0400 through about $07F8. The Secondary screen page, although not accessed by the Monitor, occupies locations $0800 through $0BF8. Screen lines are not in sequential memory order; rather, they are addressed by a somewhat complex calculation carried out in the routine BASCALC. BASCALC computes the base address for a particular line and saves it; whenever the cursor's vertical position changes, BASCALC recomputes the base address. Characters are stored into the screen buffer by adding the base address to the cursor's horizontal position.

I haven't made too much use of directly storing characters into the screen buffer; usually just storing new cursor coordinates will do the trick via the Monitor routines. Be careful, though—only change vertical position via the VTAB routine since the base address must get recomputed!

Characters themselves are internally stored in 6-bit format in the screen buffer. Bit 7 ($80), when set, forces normal (white-on-black) video display for the character. If Bit 7 is reset, the character appears inverse (black-on-white) video. Bit 6 ($40), when set, enables blinking for the character; this occurs only if Bit 7 is off. Thus an ASCII "A" in normal mode is $81; in inverse mode, $01; in blinking mode, $41.

Reading the keyboard via location $C000 is easy; if Bit 7 ($80) is set, a key has been pressed. Bits 0-6 are the ASCII keycode. To enable the keyboard again, its strobe must be cleared by accessing location $C010. Since the keyboard is directly accessible, there is no reason you can't do "special" things in a user program based on some keyboard input. If you get keys directly from the keyboard, you can bypass ALL of the Control and Escape functions.
Exploring the Apple II DOS

by Andy Hertzfeld

The Apple DOS (Disk Operating System) is one of the most useful enhancements available for your Apple. The power of this disk system is great—yet for the longest time no reference information was available on it. This guide—originally prepared for the old DOS 3.1—filled this information gap until the new DOS 3.2 manual was released. Its concise and concrete explanations are still invaluable to DOS enthusiasts!

The DOS resides in the highest portion of your system's memory and is about 10K bytes long. Its exact size depends on how many file buffers you choose to allocate (one file buffer is needed for each simultaneously open file). Each file buffer is 595 bytes long and the system provides you with three to start with (you must have at least one).

The DOS communicates with the rest of the system via the input and output hooks CSW and KSW located at $36 - $39 (This article uses "$" to indicate a hexadecimal number). Through these hooks it is given control every time a character is input or output. This is a nice scheme because it allows the DOS to be called from any environment [BASIC, Monitor, Mini-Assembler, etc.] but it has the drawback of activating the DOS when a command is typed as input to a user program which is usually not what you want. Also, since the reset button resets the hooks, the DOS is disabled whenever the system is reset, which isn't so great.

The process of loading the DOS into memory for the first time is called "bootstrapping." Bootstrapping is initiated when control is transferred to the PROM on the disk controller card. Memory pages 3 and 8 are blown by a bootstrap. There are two different types of disks you can boot from: masters and slaves. The distinction is that a master disk can be used to bootstrap on a system of arbitrary memory size while a slave will only work properly on a system with the same memory size that created it. This is because since the DOS sits at the top of memory, its addresses [for JSRs, JMPs, etc.] will be different on systems with different memory sizes. A master disk cleverly solves this problem by loading into low memory first and then relocating itself up to where it belongs. Note that this means that a master bootstrap will blow a lot of additional memory.
All addresses in this article are for a 48K system. If your system has memory size X, subtract 48K – X from the addresses that are given here.

A call to the routine at $9DBF will initialize or re-initialize the DOS. This routine should be called after every reset to restore the hooks. It is exactly like typing "3D0" "G" as Apple’s documentation recommends but is a little bit safer since the $3D0 location is often destroyed by various programs.

Every diskette has a volume number from 1 to 254. It is assigned when the diskette is initialized and there is currently no easy way to change it. The volume number of the current disk is stored at $B7F6. Before most DOS commands the system checks to see if the current volume number matches the last volume number used. If it doesn’t, a “volume mismatch” error is generated. While this “feature” may be nice for large business applications that don’t want the wrong disks inserted, it is very annoying to most average users, especially when you want to transfer a number of programs between two disks with different volume numbers. (This constraint has been eliminated in DOS 3.2. Ed.) After much searching, I located the place where the volume check is performed and devised a patch to disable it. It’s only two bytes long; just enter the monitor and type: “BDFE: A9 00”. This will disable all volume checking until the next bootstrap. It works by replacing the comparison instruction which performs the volume check with a “LDA #0” instruction which sets the “equality” or Z flag, effectively forcing the match to succeed.

Binary files of arbitrary length can be saved on disk with the “BSAVE” command. Each BSAVEd file has an implicit starting address and length associated with it; when the file is BLOADed it is loaded at the starting address. Unfortunately, there is no way provided for a user to find out the starting address and length of a BSAVEd file; this makes copying files that you are not intimately familiar with very difficult.

Fortunately, when a file is BLOADed, the directory record of the file is always placed in a buffer in a fixed location. The buffer contains the starting address and length of the file as well as other useful information. The length is kept at memory locations $A960-$A961 while the starting address is stored at $A972-$A973 (with the least significant byte first, as usual). Thus to retrieve the starting address and length of a BSAVEd program you can simply BLOAD it and then PEEK at the above locations.

Some people might wish to alter the names of some of the DOS commands to suit their own tastes (it is, after all, a personal computer). For example, I know many folks would like to abbreviate the “CATALOG” command to a simple “C”. This is surprisingly easy to do; since the DOS lives in RAM the contents of its command table are easily changed. The command table is located from $A884-$A907. Each command name is represented as an ASCII string with the high bits off, except for the last character of the string, which has its high-order bit set. The strings are associated with the commands by their position in the command table (the first string corresponds to the INIT command, the second to the LOAD command, etc.). The position of every command is given in table 1.
TABLE 1: POSITION OF COMMANDS IN THE COMMAND TABLE

The position refers to which string in the command table is associated with the command. 1 means it’s the first string, etc.

<table>
<thead>
<tr>
<th>Position</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INIT</td>
</tr>
<tr>
<td>2</td>
<td>LOAD</td>
</tr>
<tr>
<td>3</td>
<td>SAVE</td>
</tr>
<tr>
<td>4</td>
<td>RUN</td>
</tr>
<tr>
<td>5</td>
<td>CHAIN</td>
</tr>
<tr>
<td>6</td>
<td>DELETE</td>
</tr>
<tr>
<td>7</td>
<td>LOCK</td>
</tr>
<tr>
<td>8</td>
<td>UNLOCK</td>
</tr>
<tr>
<td>9</td>
<td>CLOSE</td>
</tr>
<tr>
<td>10</td>
<td>READ</td>
</tr>
<tr>
<td>11</td>
<td>EXEC</td>
</tr>
<tr>
<td>12</td>
<td>WRITE</td>
</tr>
<tr>
<td>13</td>
<td>POSITION</td>
</tr>
<tr>
<td>14</td>
<td>OPEN</td>
</tr>
<tr>
<td>15</td>
<td>APPEND</td>
</tr>
<tr>
<td>16</td>
<td>RENAME</td>
</tr>
<tr>
<td>17</td>
<td>CATALOG</td>
</tr>
<tr>
<td>18</td>
<td>MON</td>
</tr>
<tr>
<td>19</td>
<td>NOMON</td>
</tr>
<tr>
<td>20</td>
<td>PR#</td>
</tr>
<tr>
<td>21</td>
<td>IN#</td>
</tr>
<tr>
<td>22</td>
<td>MAXFILES</td>
</tr>
<tr>
<td>23</td>
<td>FP</td>
</tr>
<tr>
<td>24</td>
<td>INT</td>
</tr>
<tr>
<td>25</td>
<td>BSAVE</td>
</tr>
<tr>
<td>26</td>
<td>BLOAD</td>
</tr>
<tr>
<td>27</td>
<td>BRUN</td>
</tr>
<tr>
<td>28</td>
<td>VERIFY</td>
</tr>
</tbody>
</table>

Thus you can dream up your own names for the commands by storing new strings in the command table. For example, to change the name of the INIT command to "DNEW" you would enter the monitor and type "A884: 44 4E 45 D7". However, some caution is required when you change the length of a command name; in general you will probably have to rewrite the entire command table to achieve the desired affect.

The error message table is stored starting at address $A971. By using the same techniques described for the command table, you can rewrite error messages to be whatever you like.
It is hard to use the input and output hooks in conjunction with the DOS. You cannot simply change the hooks, as they are the DOS's only contact with the rest of the system. Also, if you change only one of them, the DOS has the nasty habit of changing it back. Fortunately, the DOS has its own internal hooks it uses for keyboard input and video output. Its output hook is at $AAS3 - $AAS4 and the input hook immediately follows at $AASS - $AA56. If you change the contents of these addresses instead of the usual hooks at $36 - $39, everything should work just fine. For example, let's say you wanted to divert output to a line printer without disabling the DOS. If the line printer output routine is located at $300, all you would have to do is enter the monitor and type "$AA53: 00 03".

To execute a DOS command from a BASIC program, simply print it, prefixing it with a "control-D". The prefix character is stored at memory location $AAB2, with its high-order bit set. Thus, if you don't like control-D and wish to use some other prefix character, all you have to do is store a different character value into $AAB2.

I am very curious to find out the primitive instructions the DOS uses to communicate with the disk controller, but without proper documentation it is very difficult to determine what does what. I have managed to find out the primitives that turn the drive on and off, though. If your controller card is in slot S, referencing memory location $C089 + $S0 will power up the disk and start it spinning while referencing $C088 + $S0 will turn it back off.

This article is merely the tip of the proverbial iceberg; most of the DOS's internals still remain a mystery to me. I hope Apple eventually distributes complete documentation, but until then other curious users can use this article as a starting point for their own explorations. Table 2 contains a summary of important addresses in the DOS for easy reference, including some not mentioned in the above commentary.

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B7F6</td>
<td>holds the volume number of the current diskette</td>
</tr>
<tr>
<td>$9DBF</td>
<td>routine to re-initialize the DOS</td>
</tr>
<tr>
<td>$AAB2</td>
<td>location of printing command character, initially set to control-D</td>
</tr>
<tr>
<td>$A972 - $A973</td>
<td>starting address of most recently loaded program, lsb first</td>
</tr>
<tr>
<td>$A960 - $A961</td>
<td>length of most recently loaded program</td>
</tr>
<tr>
<td>$A884 - $A907</td>
<td>the DOS command table</td>
</tr>
<tr>
<td>$A971 - $AA24</td>
<td>the DOS error message table</td>
</tr>
<tr>
<td>$AA53 - $AA54</td>
<td>the internal hook address to output a character</td>
</tr>
<tr>
<td>$AA55 - $AA56</td>
<td>the internal hook address to input a character</td>
</tr>
<tr>
<td>$C089 + $S0, S = slot</td>
<td></td>
</tr>
</tbody>
</table>

(*continued*)
no.*  
$C088 + $S0, S = slot

no.*  
$BD00

address to power up the disk

address to power down the disk

routine which reads in the directory off the
disk. It is called by virtually every DOS
command. (RWTS)

All addresses given (except those marked with an asterisk) refer to a system
with 48K bytes of memory. If your system has memory size X, subtract (48K-X)
from each address.
Applesoft II Shorthand

by Allen J. Lacy

If you want to make Applesoft a little easier to use, try this program which permits entire commands to be input with a single control key. Since the command lookup is table driven, you can select the keys to conform to your own preferences. The techniques used provide a valuable understanding of how to add your own modifications.

The routine Shorthand ties into the input hooks at $38 and $39 (56 and 57 decimal) and uses a table inside the RAM version of Applesoft II. In Applesoft's table, each command is represented as an ASCII string with the high bit off except for the last character of the string which has the high bit set. The routine also uses a monitor routine to read a key. If it is a control character, shorthand gets an address from its internal table. If the high byte of the address is 0, the routine passes the control character back. If the address is not 0 shorthand passes the command stored at that location back.

To Use with ROM Version

Shorthand could be adapted to run with the ROM version of Applesoft II. The addresses in Shorthand would have to be changed. I do not have access to a ROM card and so do not know the addresses. But if the ROM version is just a relocated RAM version, the addresses in Shorthand and table 2 just need $C800 added to them.

Shorthand does not use all of the control keys because some have special functions. These functions are shown in table 1. If you do not mind losing these functions, these keys can be used also. The choices for which command is tied to which key is shown in the program listing. If you do not like my choices, you can change the command addresses stored in table 2. The addresses are for the RAM version and will not work for the ROM version.

Use of Shorthand

Shorthand is relocatable and can be placed anywhere in memory. I normally load it at $300—$3AE, which is where I assembled it. But it can be placed anywhere. Applesoft's HIMEM: can be used to protect some upper memory.
Example: A 32K system without DOS can have Shorthand loaded at $7F51-7FFF and then HIMEM: can be set to 32593. To bring up Shorthand use the following steps:

1. LOAD and RUN the Applesoft TAPE

2. Enter the monitor by pressing RESET or do a CALL—151

3. Type
   300.3AER
   or type
   7F51.7FFF

4. Start tape with Shorthand on it and press RETURN, stop the tape when it has loaded

5. Type
   OG
   Press Return

6. Type
   POKE 1144,0
   Press RETURN

7. If Shorthand is at $300—$3AE type
   POKE 56,0; POKE 57,3
   If Shorthand is at $7F51—$7FFF type
   POKE 56,81: POKE 57,127

8. Press RETURN

9. If Shorthand is at 7F51 type
   HIMEM: 32593
   Press RETURN

Another good place to store Shorthand is between Applesoft II and your program. The problem is that Applesoft's LOMEM: does not set the lowest memory used by Applesoft, but sets the point at which Applesoft will start storing variables. But the monitor can be used to set pointers. To do this use the following steps:

1. LOAD and RUN the Applesoft II tape

2. Enter the monitor by pressing RESET or do a CALL—151

3. Type
   3000.30AER

4. Start the tape with Shorthand on it and press RETURN
   When it has loaded stop the tape.
5. Type
   67:BO 30
   Press RETURN

6. Type 30AF:0
   Press RETURN

7. Type
   0G
   Press RETURN

8. Type
   NEW
   Press RETURN

9. Type
   POKE 1144,0
   Press RETURN

10. Type
    POKE 56,0:POKE = 57,48
    Press Return

Shorthand will now be tied in.

Step 5 sets the pointer which tells Applesoft II where to start storing a program to $30B0. Step 6 sets the byte just below the start point to 0. I do not know why Applesoft wants this, but it will bomb if it is not done. Step 8 causes Applesoft to reset the rest of its pointers to reflect the new start point.

Now every time you want to type one of the commands stored in the table just press the control key and another key at the same time.

Example: To enter INPUT press the control key at the same time as the I.

I have made labels for my keyboard showing which command is under which key. To return full control to the keyboard, use the command IN#O. To turn Shorthand back on just POKE the correct values back into 56 and 57. Shorthand does not have to be turned off when you are finished programming and want to run a program, unless the program wants one of the control keys which Shorthand uses, for input. I normally set the hooks when I bring up Applesoft and leave them set.

The routine should work with DOS. I do not have DOS so these techniques are not tested. Since DOS communicates with the rest of the system via the input and output hooks at $36—39, you cannot set the hooks to tie in shorthand without turning off DOS. But DOS has its own internal hooks. Unfortunately the hooks are at different places for different memory sizes. In a 48K system the input hook is at $AA55, $AA56 (43605, 43606 decimal). For smaller systems subtract 48K—X from the numbers, where x is the memory size.
Reference

If you have DOS, use the following procedure to activate Shorthand:

1. Load Shorthand
   BLOAD Shorthand-A [see note below].
2. Use POKEs to set 43605, 43606. If Shorthand is at 0300
   POKE 43605,0: POKE 43606,3

Two versions of Shorthand are included. Shorthand-A, which is listed below, is for Applesoft in RAM and is relocatable. Shorthand-B has been modified by the editor to work with Applesoft in ROM. This ROM version is on the disk as Shorthand-B but is not listed here and is not relocatable.

Table 1

<table>
<thead>
<tr>
<th>Control Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control U</td>
<td>—</td>
</tr>
<tr>
<td>Control H</td>
<td>—</td>
</tr>
<tr>
<td>Control M</td>
<td>RETURN</td>
</tr>
<tr>
<td>Control J</td>
<td>Line feed</td>
</tr>
<tr>
<td>Control G</td>
<td>BELL</td>
</tr>
<tr>
<td>Control X</td>
<td>Kill input line</td>
</tr>
<tr>
<td>Control C</td>
<td>Stops a running program</td>
</tr>
<tr>
<td>Control D</td>
<td>Is used by DOS</td>
</tr>
</tbody>
</table>

Table 2

ROM VERSION: Add $C800

| 8D0 END      | 8D3 FOR      | 8D6 NEXT   | 8DA DATA   |
| 8DE INPUT   | 8E3 DEL      | 8E6 D/M    | 8E9 READ   |
| 8ED GR      | 8EF TEXT     | 901 HLIN   | 905 VLIN   |
| 909 HGR2    | 90D HGR      | 910 HCOLOR = | 917 HPLOT |
| 91C DRAW     | 920 XDRAW    | 925 HTAB   | 929 HOME   |
| 92D ROT =    | 931 SCALE =  | 937 SHLOAD | 93D TRACE  |
| 942 NOTRACE  | 949 NORMAL   | 94F INVERSE | 956 FLASH |
| 95B COLOR =  | 961 POP      | 964 VTAB   | 968 HIMEM: |
| 96E LOMEM:  | 974 ONERR    | 979 RESUME | 97F RECALL |
| 985 STORE    | 98A SPEED =  | 990 LET    | 993 GOTO   |
| 997 RUN      | 99A IF       | 99C RESTORE| 9A3 &      |
| 9A4 GOSUB    | 9A9 RETURN   | 9AF REM    | 9B2 STOP   |
| 9B6 IN       | 9B8 WAIT     | 9BC LOAD   | 9D0 CONT   |
| 9D4 LIST     | 9D8 CLEAR    | 9DD GET    | 9E0 NEW    |
| 9E3 TAB(     | 9E7 TO       | 9E9 FN     | 9EB SPC(   |

(continued)
Lacy  Applesoft Shorthand  195

9EF THEN 9F3 AT 9F5 NOT 9F8 STEP
9FC + 9FD – 9FE * 9FF /
A00 A01 AND A04 OR A06 >
A07 = A08 > A09 SGN A0C INT
AOF ABS A12 USR A15 FRE A18 SCRn
A1D PDL A20 POS A23 SR A26 RND
A29 LOG A2C EXP A2F COS A32 SIN
A35 TAN A38 ATN A3B PEEK A3F LEN
A42 STR$ A46 VAL A4B ASC A4C CHR$
A50 LEFT$ A55 RIGHT$ A5B MID$

0800 1 ;***************
0800 2 ;* APPLESOFT SHORTHAND *
0800 3 ;* BY *
0800 4 ;* ALAN LACY *
0800 5 ;* SHORTHAND *
0800 6 ;* *
0800 7 ;* *
0800 8 ;* COPYRIGHT (C) 1981 *
0800 9 ;* MICRO INK, INC. *
0800 10 ;* ALL RIGHTS RESERVED *
0800 11 ;*
0800 12 ;*
0800 13 ;***************
0800 14 ;
0800 15 ZP EPZ $1E ;R15 OF SWEET 16
0800 16 ;
0800 17 ;**********************************
0800 18 ;* LOCATIONS 47B-47F NOT *
0800 19 ;* USED BY SCREEN DISPLAY *
0800 20 ;**********************************
0800 21 ;
0800 22 SW EQU $0476 ;SWITCH
0800 23 CT EQU $0479 ;CHAR COUNT
0800 24 XSAV EQU $047A
0800 25 YSAV EQU $047B
0800 26 POIN EQU $047C
0800 27 ZPS EQU $047E
0800 28 ;*
0800 29 RKEY EQU $FD1B ;KEY READ CODE
0800 30 SW16 EQU $F689 ;SWEET 16
0800 31 ORG $300
0800 32 OBJ $800
0800 33 ;
0800 34 ;**********************************
0800 35 ;*
0800 36 ;* START LOCATION OF SHORTHAND *
0800 37 ;*
0800 38 ;**********************************
0800 39 ;
0800 40 SH STX XSAV ;SAVE X REG
0800 41 STY YSAV ;SAVE Y REG
0800 42 PHA ;SAVE ACC
0800 43 LDA ZP ;SAVE ZERO PAGE
0800 44 STZ ZPS ;LOCATIONS
0800 45 LDA ZP+1
0800 46 STZ ZPS+1
0800 47 ;
SWEET16 is used to store so KP program counter in $1E and this is used to find the location of the table in shorthand.

```
0311 2089F6  JSR SW16 ;ENTER SWEET16
0314 00  HEX 00 ; LEAVE SWEET16
0315 58 KP EQU * ; KNOWN POINT
0315 AD7804 LDA SW
0318 D03B BNE NBYT ; CHECK SW
031A 48 PLA ; RESTORE ACC
031C 60 JSR RKEY ; READ A KEY
031E 48 PHA ; STORE KEY VAL
031F C99B CMP $9B ; CONTROL KEY?
0321 9014 BCC CTR
0323 68 61 PLA ; RESTORE ACC
0326 201BFD JSR RKEY ; READ A KEY
0329 851E CMP f$9B ; CONTROL KEY?
032A 9014 BCC CTR
032C 8564 ADC ITAB-KP ; OFFSET FROM KP
032E C8 INY
0330 BE7D04 STA POIN+l ; STORE IN POIN
0332 8D7904 STA CT
0334 118 NBYT PLA
0336 88 DEY
0338 8D7C04 STA POIN
033A A9FF LDA I$FF ; SET sw
033C 8D7804 STA sw
033E 8D7904 LDA 80 ; SET CT TO 0
0340 100 BGE RET ; IFO RETURN
0342 8D7D04 STA POIN+1 ; STORE IN POIN
0344 88 DEY
0346 B11E 103 LDA (ZP), Y
0348 8D7C04 STA POIN
034A 9FF LDA $FF ; SET SW
034C 8D7804 STA SW
034E 8D900 LDA 80 ; SET CT TO 0
0350 8D7904 STA CT
0352 109 ;
0354 110 ;
0356 111 ;
0358 112 ; NBYT IS USED TO PASS THE CHAR-
035A 113 ; ACTERS FROM THE TABLE IN APPL-
035C 114 ; SOFT AS IF THEY WERE TYPED IN
035E 115 ;
0360 116 ;
0362 117 ;
0364 68 NBYT PLA
0366 AC7904 LDY CT ; LOAD CHAR CT
0368 AD7C04 LDA POIN ; STORE POIN IN
036A 851E 121 STA 2P ; ZERO PAGE
```
Lacy

**Zero Page Variables:**

- **ZP** 001E

**Absolute Variables/Labels:**

- **SW** 0478 CT 0479 XSAV 047A YSAV 047B POIN 047C
- **ZPS** 047E RKEY FD1B SM16 F689 SH 0300 KP 0315 RET 0323
- **RT** 0324 CTR 0337 NBYT 0355 END 0370 TAB 0379 LS 03AF
The Integer BASIC Token System in the Apple II

by Frank D. Kirschner

Most BASIC interpreters 'tokenize' the code as they scan it—storing space-saving 'tokens' in memory. The Apple Integer BASIC interpreter is no exception. Here is a discussion of how that Integer BASIC uses tokens, what the tokens mean, and where these tokens are stored in memory. The information presented here should lend considerable insight into the Apple's fast and efficient Integer BASIC.

There are two primary methods of storing BASIC programs in microcomputers. One involves storing the entire program, letter by letter and symbol by symbol somewhere in memory, and interpreting the ASCII codes on execution. This is typical of BASIC compilers and some interpreters, like the TRS-80 Level I. A more memory-efficient system uses tokens, eight bit bytes each of which represent a BASIC word or symbol. The TRS-80 Level II uses this method, as does the Apple II, to which the examples that follow apply.

When in Integer BASIC, the Apple stores characters as they are entered in a character buffer [hex locations 0200 to 02FF]. When "return" is entered, BASIC "parses" the entry (that is, interprets the ASCII characters and breaks the instruction into executable parts). It determines what is a command, what are variables, data and so forth. If it is legal and is preceded by a number between 0 and 32767 [a line number], it stores it in memory in a fashion discussed below. If there is no line number, it simply executes the command and awaits further instructions.

The way the programs are stored is quite clever. When BASIC is initiated (control B or E000 G from the monitor) several things happen. First, the highest available user memory [RAM] is stored in memory locations 004C [low byte] and 004D [high byte], called the HIMEM pointer. Also, locations 00CA and 00CB, the start-of-program pointer, get the same numbers, since there is no program as yet. As program steps are entered, they are stored starting at the top of memory, highest line numbers first, and the start-of-program pointer is decreased accordingly. See figure 1. When a line with a higher number than some already in memory is entered, they are shuffled to preserve the order. One application: if you
enter a program and then hit control B, the program is not scratched (or erased); only the start-of-program pointer is affected. Since powering up the Apple fills the memory with a pattern of ones and zeros (it looks like FF FF 00 00 ...) from the monitor, it is easy to find the start of the program and then manually reset CA and CB to that location.

See figure 2 for an example of the way program instructions are stored in memory (all numbers are in hex). As an example, power up the Apple, bring up BASIC, and enter

100 PRINT 0,50

Enter the monitor (by pushing "reset"), and then examine the program by entering

3FF4.3FFF RETURN

(Locations are for a 16K Apple. Subtract 2000 hex for a 4K or add 4000 hex for a 32K Apple.) You will see this:

3FF4 - 0C 64 00 62
3FF8 - B0 00 00 49 B5 32 00 01
which means:

- **0C**: There are 12 bytes in this line
- **64 00**: It is line 100 (decimal)
- **62**: PRINT (see table 1 for complete list of tokens)
- **B0**: The next two bytes are a number (rather than tokens)
- **00 00**: The number 0
- **49**: The comma in a PRINT statement
- **B5**: Another number follows
- **32 00**: The number 50
- **01**: End of BASIC line

To demonstrate the use of this information, return to BASIC and try to enter the following BASIC line:

```
100 DEL 0,50
```

You will get a syntax error, because the Apple Interpreter does not allow the command DEL in deferred execution mode. Now do this: reenter the monitor and change the **62** (PRINT) to **09** (DEL) and the **49** (, for PRINT) to **0A** (, for DEL) by entering:

```
3FF7: 09 RETURN
3FFB: 0A RETURN
```

Reenter BASIC (control C) and list. Try this instruction by adding lines between 0 and 50, running the program, and then listing it. This allows you to write a program which will carry out some functions only the first time it is run and then automatically delete those lines.

![Figure 2](image-url)

Tokens for BASIC statements:

- **Line number (Lo byte, Hi byte)**: This is line 100 (Decimal)
- **Number of bytes in BASIC line**: (also, one less than the number of bytes from the beginning of the next line.)
In addition to inserting instructions which cannot be entered as deferred commands, you can modify the program under program control. As an example, here is a program which will stop and start listing a long program by hitting a key on the keyboard.

Bring up BASIC
Enter: 257 LIST 0: RETURN
HIT RESET, 3FF6.3FFF RETURN
You will see
3FF6 - 0A 01
3FF8 - 01 74 B0 00 00 03 5B 01
What this means:
3FF6: 0A Ten bytes in line
3FF7,8: 01 01 LINE 257
3FF9: 74 TOKEN FOR LIST
3FFA: B0 Means "Number follows"
3FFB,C: 00 00 LINE TO BE "LISTED" (LO, HI)
3FFD: 03 TOKEN FOR COLON
3FFF: 01 End of BASIC LINE
Now enter 3FF7: FF FF RETURN
Cont. C, List
You have 65535 LIST 0: RETURN
Now enter
100 X = PEEK (-16384): POKE -16368, 0: 1F
X 127 THEN 0: GOTO 100
Reset, 3FCF.3FFF RETURN
Change line number from 100 to 65534 by entering 3FDO; FE FF RETURN
Change GOTO 100 to GOTO 65534 by entering 3FF3: FE FF
Change the 0 in "THEN 0" to 65533 by entering 3FEE: FD FF

In a like manner, enter these remaining steps (under each number which has to be entered through the monitor, the Hex equivalent, in reverse order as it must be entered, appears):

65533 I = I PEEK(I): IF I > PEEK (76) +
(FD FF)
   256*PEEK (77) THEN END: GOTO
   65531
   (FB FF)
65532 X = PEEK (-16384):POKE -16386,0:
(FC FF)
   IF X 127 THEN 65534
   (FE FF)
65531 POKE 16374, PEEK (I + 1): POKE 16380
(FB BB)
   PEEK (1 + 2) GOSUB 65535
   (FF FF)
32767 I = PEEK (202) 256* PEEK (203)
The steps must be entered in reverse order (i.e. descending line numbers) because
the interpreter orders them by their number when entered, and will not re-order
lines when the numbers have been changed through the monitor.

The reason for making all these line numbers very high is so the applications
program will fit "under" the list program. Now, in the monitor, move the start of
program and HIMEM pointers below the program:

3A: 49 3F RETURN
4C: 49 3F RETURN

Hit control C and list. Nothing is listed. The program has been stored in a portion
of memory temporarily inaccessible to BASIC. Load your applications program,
make sure all the line numbers are less than 32767, and change HIMEM through
the monitor (4C: 00 40) and execute RUN 32767. The program will list until you
hit a key and then resume when you hit a key again. It uses the fact that each line
begins with the number of bytes in the line followed by the line number. Numbers
of successive lines are found and POKEd into the appropriate location in line
75535, which then lists each line.

Using these methods you can exercise considerably more control over the
BASIC interpreter in your microcomputer.

---

**Table 1**

Apple II Integer BASIC Tokens

<table>
<thead>
<tr>
<th>BASIC Command or Function</th>
<th>Hex Token</th>
<th>BASIC Command (Continued)</th>
<th>Hex Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>31</td>
<td>LOAD</td>
<td>04</td>
</tr>
<tr>
<td>(</td>
<td>3F</td>
<td>MAN</td>
<td>0F</td>
</tr>
<tr>
<td>)</td>
<td>72</td>
<td>NEW</td>
<td>0B</td>
</tr>
<tr>
<td>ASC (</td>
<td>3C</td>
<td>NEXT</td>
<td>59</td>
</tr>
<tr>
<td>)</td>
<td>72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>28</td>
<td>NO DSP</td>
<td>79</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>29</td>
<td>NO TRACE</td>
<td>7A</td>
</tr>
<tr>
<td>AUTO</td>
<td>0D</td>
<td>PDL</td>
<td>32</td>
</tr>
<tr>
<td>,</td>
<td>0A</td>
<td></td>
<td>3F</td>
</tr>
<tr>
<td>CALL</td>
<td>4D</td>
<td></td>
<td>72</td>
</tr>
<tr>
<td>CLR</td>
<td>0C</td>
<td>PEEK</td>
<td>2E</td>
</tr>
<tr>
<td>COLOR =</td>
<td>66</td>
<td></td>
<td>3F</td>
</tr>
<tr>
<td>CON</td>
<td>60</td>
<td></td>
<td>72</td>
</tr>
<tr>
<td>DEL</td>
<td>09</td>
<td>PLOT</td>
<td>67</td>
</tr>
<tr>
<td>,</td>
<td>0A</td>
<td></td>
<td>68</td>
</tr>
<tr>
<td>DIM</td>
<td>4F</td>
<td>POKE</td>
<td>64</td>
</tr>
</tbody>
</table>

(continued)
<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
<th>Example Usage</th>
<th>Additional Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIM 24</td>
<td>String array</td>
<td>POP 77</td>
<td>If used alone</td>
</tr>
<tr>
<td>PRINT 63</td>
<td>Numeric variable</td>
<td>PRINT 62</td>
<td>First</td>
</tr>
<tr>
<td>END 51</td>
<td>String variable</td>
<td>PR# 29</td>
<td>Includes #</td>
</tr>
<tr>
<td>FOR 55</td>
<td>Numeric variable</td>
<td>REM 5D</td>
<td></td>
</tr>
<tr>
<td>TO 57</td>
<td>RETURN 5B</td>
<td>RND 2F</td>
<td></td>
</tr>
<tr>
<td>STEP 58</td>
<td>RND 2F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOSUB 5C</td>
<td>String variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOTO 5F</td>
<td>Numeric variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GR 4C</td>
<td>Numeric variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HIMEM: 10</td>
<td>String variable</td>
<td>SAVE 05</td>
<td>Includes #</td>
</tr>
<tr>
<td>HLIN 69</td>
<td>Numeric variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>, 6A</td>
<td>String variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AT 6B</td>
<td>Numeric variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF 60</td>
<td>Numeric variable</td>
<td>SGN 30</td>
<td></td>
</tr>
<tr>
<td>THEN 24</td>
<td>When followed by a line no.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>THEN 25</td>
<td>When followed by GOSUB or a BASIC operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT 54</td>
<td>Numeric variable</td>
<td>VLIN 6C</td>
<td></td>
</tr>
<tr>
<td>INPUT 52</td>
<td>String variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT 53</td>
<td>Input if followed by ...</td>
<td>AT 6E</td>
<td></td>
</tr>
<tr>
<td>, 27</td>
<td>VTAB 6F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot; 28</td>
<td>First</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot; 29</td>
<td>Second</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN # 7F</td>
<td>Includes #</td>
<td>AND 1D</td>
<td></td>
</tr>
<tr>
<td>LEN (3B</td>
<td>Includes (</td>
<td>OR 1E</td>
<td></td>
</tr>
<tr>
<td>LET 5E</td>
<td>MOD 1F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIST 74</td>
<td>NOR 1D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Creating an Applesoft BASIC Subroutine Library

by N.R. McBurney

There's more than one way to run a BASIC program on your Apple with DOS. Using EXEC files offers increased flexibility over the RUN command. In this article the author uses the power of the EXEC command to link Applesoft programs from a common library of disk-resident subroutines.

DISK FULL! Well, of course it was full. I had over a dozen lengthy programs stored on it. In each of those programs over fifty percent of the code was identical BASIC routines. Besides the problem of disk space, maintaining identical copies of software is almost impossible. After any given period of time identical software will differ. This is a corollary to somebody's DP axiom that "identical data bases aren't."

The first problem is to find a way to append the subroutines to the main programs. To do this, we need to know how BASIC programs are stored in RAM. With ROM BASIC, the user program usually starts in location 2049 ($801). RAM (cassette) BASIC normally starts at 12289 ($3001). All of the examples in this article assume and were executed with the ROM version of Applesoft BASIC. This start address is stored in locations 103-104 ($67-$68). Similarly, the end of the program is pointed to by locations 175-176 ($AF-$B0). This is shown graphically in figure 1, step 1. If we change the start of the program pointer to the end of our program (figure 1, step 2), then load our subroutines (figure 1, step 3) we need only change our start of program pointer back to its original value (figure 1, step 4), to have successfully merged our two programs. To do this manually, first:

LOAD MAIN PROGRAM

where MAIN PROGRAM is the name of the file containing your BASIC program minus your subroutines. Next type:

I = PEEK(176) * 256 + PEEK(175) - 2
As we stated before, decimal locations 176-175 ($AF-$B0) contain the address of the end of the program currently in RAM. Now type:

```
POKE 104,INT(I/256)
POKE 103,1-INT(I/256)★256
```

Decimal locations 103-104 contain the address of the start of the BASIC program. This is normally 2049 ($801). The above two statements changed the starting address to now point to the end of our main program. Now we type:

```
NEW
LOAD SUBROUTINES
```

where SUBROUTINES is the name of the file containing the routines required by MAIN PROGRAM. SUBROUTINES has now been loaded behind our first program, leaving our original program still intact in RAM. Finally we type in:

```
POKE 103,1
POKE 104,8
```

These two statements changed the pointer to the start of our program back to its original value (2049 decimal, $801 hex). Assuming you haven’t made any typing errors, if you now type LIST, you will see that you have successfully appended SUBROUTINES to MAIN PROGRAM.

When we have many programs using the same set of subroutines, this process will save core, but it doesn’t result in user-oriented software. There’s an easier way! The process can be handled with an EXEC file. Apple DOS EXEC files allow you to store on a text file what you would normally type in at the keyboard. When you EXEC a file, Apple DOS processes each line exactly as if it had been typed in at the keyboard. This is an extremely powerful tool. This article explores just one use of that power.
Program listing 1 contains code for a generalized EXEC file writer. It requests an EXEC file name, creates or replaces the file and then writes the quoted lines contained in the program's DATA statements onto the named text file. Any apostrophes ('') in the DATA statements are converted to quotes ("") before writing to the EXEC file. This feature allows us to write PRINT statements to the EXEC file.

If we run the program shown in listing 1, we produce the EXEC file shown in listing 2. Let's look at a simple example of how to use this EXEC file.

For this example, the file called MAIN PROGRAM (our main program) contains the instructions shown in listing 5. Our subroutine file, SUBROUTINES, contains the instructions shown in listing 6. If we type EXEC MERGE (the name of the EXEC file in listing 2), we would have the following (user input is underlined):

```plaintext
] LOAD MAIN PROGRAM
] EXEC MERGE

] SUBROUTINES LOADED....
] LIST

(listing appears)
```

What we've just done is create a library routine loader! While this approach has proven adequate for development work, expecting an end user to remember which main program must be used with which EXEC file is expecting a human being to adapt to the requirements of the computer. Unfortunately, this kind of design mentality has been prevalent in the industry and is responsible for much of the public's distaste for computers. A more professional approach is possible.

There are several ways that the linking operation can be made invisible to the user and more production oriented for the developer. Our previous example could have included both the LOAD MAIN PROGRAM and RUN statements. Listing 3 contains an EXEC file with these changes. Using this EXEC file results in the following:

```plaintext
] EXEC TITLE DEMO

(Apple HOME's)
FIRST LINE OF TITLE
SECOND LINE
```

The problem with this approach is that it requires a separate EXEC file to execute each program. Every disk file requires a minimum of one sector of overhead plus one sector minimum for the program. This approach is not
completely compatible with our original goal of minimizing storage requirements. A better approach, in my opinion, is to write a menu program that (invisible to the user) determines the names of the programs to be linked together by our EXEC file. These file names are stored by the menu program in RAM, and then the linking EXEC file is EXECed under program control. The EXEC file retrieves the names from RAM and runs the combined program. Listing 7 contains a sample menu program that illustrates this concept.

In our menu, program lines 1000-1190 display the menu shown in figure 2. Lines 1200-1340 request the user to enter the number of his request (the line ENTER YOUR REQUEST NUMBER... is "crawled" along the bottom of the screen). Line 1290 checks to see if a key has been depressed, and if it has, line 1340 converts it from ASCII code to a digit. Lines 1350-1450 map the request number into a main program name. Since all of the programs require the same subroutine file, the name of that file is set in line 1530. The loop in lines 1550-1580 POKEs the two file names into locations 768-829 (\$330-$333D). Locations 768-829 are generally available to the user. Finally, line 1610 EXECs the file MASTER MERGE shown in listing 4 and runs the desired combined programs.

I have been using various permutations of the techniques described in this article for several months and have found them to be extremely workable. The only obvious restriction is that subroutine line numbers must be larger than the last line of the main program. In practice, I've limited my main programs to lines 1-29999 and my subroutines to lines 30000-65000. The small amount of discipline that this restriction imposes is more than offset by the twin benefits of more effective disk space utilization and easier software maintenance.

<table>
<thead>
<tr>
<th>SUBROUTINE LIBRARY USAGE DEMONSTRATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) MAIN PROGRAM DEMO</td>
</tr>
<tr>
<td>2) DISPLAY 10 TITLES</td>
</tr>
<tr>
<td>3) DIAMOND FORMAT</td>
</tr>
<tr>
<td>4) BLOCK TITLES</td>
</tr>
<tr>
<td>5) VERTICAL TITLE</td>
</tr>
</tbody>
</table>

ENTER YOUR REQUEST NUMBER...
Listing 1: Generalized EXEC File Writer Program

1000 REM ******************************
1001 REM *
1002 REM * CREATING AN APPLESOF T* *
1003 REM * SUBROUTINE LIBRARY * *
1004 REM * W.R. MCBURNEY * *
1005 REM *
1006 REM * EXEC FILE WRITER * *
1007 REM *
1008 REM * COPYRIGHT (C) 1981 * *
1009 REM * MICRO INK, INC. * *
1010 REM * CHELMSFORD, MA 01824 * *
1011 REM * ALL RIGHTS RESERVED * *
1012 REM *
1013 REM *
1014 REM ******************************
1015 REM GENERALIZED EXEC FILE WRITER:
1016 REM QUOTES IN THE DATA STATEMENTS ARE USED ONLY AS DELIMITERS AND WILL NOT
1017 REM APPEAR ON THE EXEC FILE. APOSTROPHES IN THE DATA STATEMENTS WILL NOT
1018 REM APPEAR AS QUOTATION MARKS IN THE EXEC FILE.
1019 REM APOSTROPHES WITH QUOTES
1020 REM READ IN LINE AND REPLACE
1021 REM APOSTROPHES WITH QUOTES
1022 REM CHECK FOR CORRECT ERROR CODE
1023 REM (42=OUT OF DATA)
1024 REM (#42=OUT OF DATA)
1025 REM *
1026 IF PEEK (222) = 42 THEN GOTO 1255
1027 PRINT "ERROR "; PEEK (222)
1028 PRINT "IN LINE "+ PEEK (218) + PEEK (219) * 2556
1029 STOP
1030 POK E 216,0
1031 PRINT DS;"NOMON O"
1032 PRINT DS;"CLOSE";FILES
1033 REM
1034 REM BEGIN DATA STATEMENTS DEFINING
1035 REM TEXT TO BE PLACED IN EXEC FILE
1036 DATA "I=PEEK(176)*256+PEEK(175)-2:POKE 104,INT(I/256):POKE 103,I-IN T(I/256)*256"
1037 DATA "LOAD SUBROUTINES"
1038 DATA "POKE 103,1:POKE 104,8;PRINT 'SUBROUTINES LOADED...';CHR$ (7)"
Listing 2: EXEC File MERGE

LOAD MAIN PROGRAM
HOME: I = PEEK(176) * 256 + PEEK(175) - 2: POKE 10
4, INT(I/256): POKE 103, I - INT(I/256) * 256
LOAD SUBRoutines
HOME: POKE 103, I: POKE 104, 8
PRINT "SUBROU TINES LOADED..."; CHR$(7)
RUN

Listing 3: EXEC File TITLE DEMO

LOAD MAIN PROGRAM
HOME: I = PEEK(176) * 256 + PEEK(175) - 2: POKE 10
4, INT(I/256): POKE 103, I - INT(I/256) * 256
LOAD SUBRoutines
HOME: POKE 103, I: POKE 104, 8
RUN

Listing 4: EXEC File MASTER MERGE

MAIN$ = "": FOR I = 1 TO 30: MAIN$ = MAINS + CHR$(PEEK(K(767 + I))): NEXT: PRINT CHR$(4); "LOAD " + MAIN$"!
HOME: I = PEEK(176) * 256 + PEEK(175) - 2: POKE 104, INT(I/256): POKE 103, I - INT(I/256) * 256
SUBRS$ = "": FOR I = 1 TO 30: SUBRS$ = SUBRS$ + CHR$(PEEK(K(798 + I))): NEXT: PRINT CHR$(4); "LOAD " + SUBRS$!
HOME: POKE 103, I: POKE 104, 8
RUN

Listing 5: MAIN Program

100 REM
110 REM DEMONSTRATION MAIN PROGRAM
120 REM
130 HOME
140 TITLES = "FIRST LINE OF TITLE"
150 GOSUB 10000
160 TITLES = "SECOND LINE"
170 GOSUB 10000
180 END

Listing 6: SUBROUTINE File

10000 REM
10010 REM DEMONSTRATION SUBROUTINE TO
10020 REM PRINT A CENTERED TITLE LINE
10030 REM
10040 L = LEN(TITLES)
10050 PRINT TAB(20 - L / 2); TITLES
10060 RETURN
Listing 7: MENU Program

1000 REM
1010 REM MENU DEMONSTRATION PROGRAM
1020 REM
1030 HOME
1040 REM
1050 REM DISPLAY THE MENU
1060 REM
1070 PRINT TAB(10);"SUBROUTINE LIBRARY"
1080 PRINT TAB(9);"USAGE DEMONSTRATION"
1090 INVERSE
1100 FOR I = 4 TO 14
1110 HTAB 5
1120 VTAB I
1130 PRINT TAB(35)
1140 NEXT
1150 VTAB 5: HTAB 7: PRINT "1) MAIN PROGRAM DEMO"
1160 VTAB 7: HTAB 7: PRINT "2) DISPLAY 10 TITLES"
1170 VTAB 9: HTAB 7: PRINT "3) DIAMOND FORMAT"
1180 VTAB 11: HTAB 7: PRINT "4) BLOCK TITLES"
1190 VTAB 13: HTAB 7: PRINT "5) VERTICAL TITLE"
1200 REM
1210 REM REQUEST AND WAIT FOR INPUT
1220 REM
1230 AS = "ENTER YOUR REQUEST NUMBER..."
1240 VTAB 22
1250 HTAB 5
1260 AS = MID$(AS,2) + LEFT$(AS,1)
1270 PRINT AS
1280 FOR I = 1 TO 8
1290 X = PEEK(-16384)
1300 IF X > 128 THEN 1330
1310 NEXT
1320 GOTO 1240
1330 POKE -16368,0
1340 X = X - 176
1350 REM
1360 REM DETERMINE WHICH PROGRAM TO APPEND
1370 REM SUBROUTINES TO AND THEN RUN THAT
1380 REM PROGRAM VIA THE EXEC FILE
1390 REM
1400 IF X = 1 THEN MAIN$ = "MAIN PROGRAM"
1410 IF X = 2 THEN MAIN$ = "TEN TITLES"
1420 IF X = 3 THEN MAIN$ = "DIAMOND"
1430 IF X = 4 THEN MAIN$ = "BLOCK TITLES"
1440 IF X = 5 THEN MAIN$ = "VERTICAL TITLE"
1450 IF MAIN$ = "" THEN PRINT CHRS(7): GOTO 1240
1460 REM NORMAL
1470 REM
1480 REM POKE NAME OF MAIN PROGRAM INTO LOCATIONS $300-$31E
1490 REM AND NAME OF SUBROUTINE FILE INTO
1500 REM LOCATIONS $31F-$33D.
1510 REM
1520 K1 = 767: K2 = 798
1530 SUBRS$ = "SUBROUTINES"
1540 MAIN$ = LEFT$(MAIN$ + "", 30)
1550 FOR I = 1 TO 30
1560 POKE K1 + I, ASC (MID$(MAIN$,I,1))
1570 POKE K2 + I, ASC (MID$(SUBRS$,I,1))
1580 NEXT
1590 HOME
1600 PRINT CHRS(7)
1610 PRINT CHRS(4);"EXEC MASTER MERGE"
## Language Index

### APPLESOF BASIC

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RENUMBER</td>
<td>Applesoft Renumbering, Childress</td>
<td>8</td>
</tr>
<tr>
<td>SEARCH/CHANGE</td>
<td>Search/Change, Childress</td>
<td>15</td>
</tr>
<tr>
<td>DATA-GEN</td>
<td>Data Statement, Brady</td>
<td>46</td>
</tr>
<tr>
<td>EDIT MASK</td>
<td>Edit Mask, Reynolds</td>
<td>53</td>
</tr>
<tr>
<td>DOLLAR MASK</td>
<td>Business Dollars, Bauers</td>
<td>61</td>
</tr>
<tr>
<td>CHECK PROTECT</td>
<td>Business Dollars, Bauers</td>
<td>61</td>
</tr>
<tr>
<td>LC INSERT</td>
<td>Lower Case, Childress</td>
<td>65</td>
</tr>
<tr>
<td>LC ENTRY</td>
<td>Lower Case, Childress</td>
<td>66</td>
</tr>
<tr>
<td>FUNCTION GRAPH</td>
<td>Graphing, Rational Functions, Carlson</td>
<td>73</td>
</tr>
<tr>
<td>SHAPE 1</td>
<td>How to Do a Shape Table, Figueras</td>
<td>91</td>
</tr>
<tr>
<td>SHAPE 2</td>
<td>How to Do a Shape Table, Figueras</td>
<td>92</td>
</tr>
<tr>
<td>SHAPE 3</td>
<td>How to Do a Shape Table, Figueras</td>
<td>94</td>
</tr>
<tr>
<td>CHARACTERS</td>
<td>Define Hi-Res Characters, Zant</td>
<td>98</td>
</tr>
<tr>
<td>SOLAR</td>
<td>Solar System, Partyka</td>
<td>140</td>
</tr>
<tr>
<td>EXEC FILE WRITER</td>
<td>Creating a Subroutine Library, McBurney</td>
<td>208</td>
</tr>
</tbody>
</table>

### INTEGER BASIC

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDIT</td>
<td>Program Edit Aid, Hill</td>
<td>19</td>
</tr>
<tr>
<td>ALARM PROMPT</td>
<td>Alarming Apple, Irwin</td>
<td>39</td>
</tr>
<tr>
<td>GRAPH PLOT</td>
<td>Hi-Res Graph Plot, Fam</td>
<td>77</td>
</tr>
<tr>
<td>GRAPHICS-ORG</td>
<td>Hi-Res Graphics Memory, Eliason</td>
<td>102</td>
</tr>
<tr>
<td>APPLE PI</td>
<td>Apple Pi, Bishop</td>
<td>107</td>
</tr>
<tr>
<td>BUBBLE</td>
<td>Sorting Revealed, Vile</td>
<td>126</td>
</tr>
<tr>
<td>INSERT</td>
<td>Sorting Revealed, Vile</td>
<td>127</td>
</tr>
<tr>
<td>SELECT</td>
<td>Sorting Revealed, Vile</td>
<td>128</td>
</tr>
<tr>
<td>SHELL</td>
<td>Sorting Revealed, Vile</td>
<td>130</td>
</tr>
<tr>
<td>QUICK</td>
<td>Sorting Revealed, Vile</td>
<td>131</td>
</tr>
<tr>
<td>SPELUNKER</td>
<td>Spelunker, Mimlitch</td>
<td>153</td>
</tr>
<tr>
<td>APPLE LIFE</td>
<td>Life for Your Apple, Suitor</td>
<td>169</td>
</tr>
<tr>
<td>TYPE TEST</td>
<td>Speed Typing Test, Broderick</td>
<td>174</td>
</tr>
<tr>
<td>LUDWIG</td>
<td>Ludwig Von Apple, Schwartz</td>
<td>176</td>
</tr>
</tbody>
</table>

### MACHINE LANGUAGE

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND</td>
<td>Program Edit Aid, Hill</td>
<td>20</td>
</tr>
<tr>
<td>EDIT PLUS</td>
<td>A Little Plus, Peterson</td>
<td>27</td>
</tr>
<tr>
<td>ZOOM</td>
<td>Zoom and Squeeze, Little</td>
<td>32</td>
</tr>
<tr>
<td>SLOW LIST</td>
<td>Slow List, Sander-Cederlof</td>
<td>35</td>
</tr>
<tr>
<td>ALARM</td>
<td>Alarming Apple, Irwin</td>
<td>38</td>
</tr>
<tr>
<td>LIFE</td>
<td>Life for Your Apple, Suitor</td>
<td>170</td>
</tr>
<tr>
<td>SHORTHAND</td>
<td>Applesoft Shorthand, Lacy</td>
<td>195</td>
</tr>
</tbody>
</table>
Author Index

(Biographies included)

Auricchio, Richard ......................................... 181
Software engineer for Apple Computer, Inc. His previous experience includes work on operating system development using the Xerox Sigma-9 mainframe.

Bauers, Barton M., Jr ......................................... 55
Executive vice president of LFE Corporation, Fluids Control Division. His programming background includes experience with Fortran, PL-1, and BASIC. Bauers holds a Masters degree in science in industrial engineering with a concentration in operations research.

Bishop, Bob ...............................................
Senior member of the technical staff at Apple Computer, Inc., working on research and development. Bishop is author of Applevision.

Brady, Virginia Lee .......................................... 43
Registered nurse. Brady bought an Apple computer, took computer courses, and now works as a programmer at the Maryland Institute for Emergency Medical Services (shock trauma) in Baltimore, Maryland.

Broderick, John, CPA ....................................... 173

Carlson, Ron ............................................... 69
Computer instructor at Plymouth-Canton High School, Canton, Michigan. He was previously a math teacher for ten years and a senior instructor at Sycor, a minicomputer manufacturer. His articles have appeared in MICRO, Creative Computing, Personal Computing and Recreational Computing. Carlson has conducted computer education seminars for schools and organizations. He is currently completing a textbook on high school computer instruction.

Carpenter, Chuck .......................................... 175
Senior systems engineer at Xerox Corporation in Dallas, Texas. Carpenter has published articles in several computer magazines and is currently writing the Creative Computing Apple Cart column.

Childress, J.D ............................................... 5, 12, 62
President of CareWare, Inc., a firm specializing in microcomputer software for the health-care industry. He holds a Ph.D. in Physics.
Chipchase, Frank ............................................ 9
Chief engineer for International Multifoods Corporation. Before purchasing an Apple, Chipchase had had no experience with computers. He has written a utility program for the Apple which is being marketed.

Eliason, Andrew H ........................................... 99

Fam, Richard ............................................... 75

Figueras, John .............................................. 78
Scientific programmer at Eastman Kodak Research Labs. He holds a Ph.D. in organic chemistry.

Hertzfeld, Andrew .......................................... 186
Employed at Apple Computer, Inc., since August 1979.

Hill, Alan .................................................. 17
Apple owner and enthusiast since early 1978. He enjoys writing utility programs and is the author of Master Catalog, Amper-Reader, Amper-Search, as well as Amper-Sort II.

Irwin, Paul ................................................ 37
Programmer/analyst operating a software and consulting business centered on microcomputer applications. Irwin holds a B.S. in Physics and Math. He is currently serving as president of the Ottawa 6502 user group.

Kirschner, Frank D .......................................... 198

Lacy, Allen ................................................ 191
Professional programmer who writes programs for large IBM, and desk-top computers.

Little, Gary ................................................ 29
Articled law student and Apple hobbyist. Past president and current treasurer of Apples British Columbia Computer Society in Vancouver.

McBurney, N.R ............................................ 204
Southern region manager of custom applications for General Electric Information Services Co. He has held a variety of scientific, data processing and management positions at General Electric. McBurney holds a BA in Mathematics and owns an Apple.

Mimlitch, Thomas, R ....................................... 155
Director of technical services for the MicroAge Computer Store in Columbus, Ohio. Mimlitch integrates microcomputer hardware and software systems. A computer science graduate (BA), Mimlitch has pursued interests in artificial intelligence, analog simulations, and operating systems.
Mulligan, John ............................................. 143
  Systems designer for Bio-science Laboratories, a division of the Dow Chemical Company.

Partyka, David ............................................. 134
  Works as a programmer on an IBM 3031 OS system for the May Department Stores, Co. He's been programming for the past three years and was an operator for four years before that.

Peterson, Craig ............................................. 25
  Numerical control engineer for his company which uses an Apple II.

Reynolds, Leon M ........................................... 47
  Computer programmer for 15 years. He reads everything he can about the Apple and has a library of several hundred programs.

Sander-Cederlof, R.B ......................................... 33
  Owner of a company called S-C Software where he specializes in producing Apple software, such as the S-C Assembler II. He has been programming computers since 1957, including the IBM 704, Bendix G-15 and CDC 6600. During the infancy of time-sharing he wrote two interactive Fortran systems and a data base management system for the Control Data 3300. During seven years at Texas Instruments, he created the software for TI's manufacturing robots, and several online test systems. He purchased an Apple in 1977. Sander-Cederlof publishes, writes and edits two newsletters: AppleGram and Apple Assembly Line.

Schwartz, Marc ............................................ 175

Suitor, Richard F ........................................... 168
  Suitor grew up expecting to be a physicist, but his mind was warped by early exposure to the awesome collections of vacuum tubes and blinking lights that evolved into the micros of today. In 1978 he obtained an Apple. Final degeneration was immediate; having decided his case was chronic, he has joined Software Resources of Cambridge, Massachusetts.

Vile, Richard C. Jr ......................................... 109
  Project leader in the compiler and languages areas for Bell Northern Research, Inc.

Zant, Robert F ............................................... 96
  Professor of information systems at North Texas State University. Zant has 17 years experience in computing as a programmer, analyst, educator, and consultant.
DISK VOLUME 001

* A 005 MICRO/APPLE
* A 009 RENUMBER
* A 013 SEARCH/CHANGE
* I 005 EDIT
* B 002 FIND
* B 002 EDIT PLUS
* B 002 ZOOM
* B 002 SLOW LIST
* I 005 ALARM PROMPT
* B 002 ALARM
* A 006 DATA-GEN
* A 017 EDIT MASK
* A 005 DOLLAR MASK
* A 005 CHECK PROTECT
* A 007 LC INSERT
* A 005 LC ENTRY
* A 012 FUNCTION GRAPH
* I 006 GRAPH- PLOT
* A 006 SHAPE1
* A 019 SHAPE2
* A 008 SHAPE3
* A 007 CHARACTERS
* I 005 GRAPHICS- ORG
* I 010 APPLE PI
* I 009 BUBBLE
* I 010 INSERT
* I 009 SELECT
* I 010 SHELL
* I 012 QUICK
* A 022 SOLAR
* I 040 SPELUNKER
* I 006 APPLE LIFE
* B 003 LIFE
* I 011 TYPE TEST
* I 005 LUDWIG
* B 002 SHORTHAND-A
* B 002 SHORTHAND-B
* A 007 EXEC FILE WRITER
Notice to Purchaser

When this book is purchased, this pocket should contain

A. One floppy disk entitled MICRO/Apple, Volume 1.
B. A warranty card pertaining to the disk.

If either is missing, make sure you ask the seller for a copy.

The publisher hereby grants the retail purchaser the right to make one copy of the disk for back-up purposes only. Any other copying of the disk violates the copyright laws and is expressly forbidden.
MICRO/Apple 1
Edited by Ford Cavallari

30 Articles by 29 Authors
More than 30 Programs on Diskette!
No Need to Type in Hundreds of Lines of Code!


MICRO and the Apple have grown up together. MICRO began in 1977, the year the Apple II was first available commercially, and MICRO’s first issue bore on its cover a large picture of the then little known Apple II.

This volume, MICRO/Apple 1, while especially for beginning-to-intermediate-level Apple users, will interest even sophisticated users. It contains some of the most valuable general-interest articles and programs published in the magazine since 1977—brought up to date by the authors and MICRO staff. The programs have been tested and entered on the diskette which comes with the book (13-sector DOS 3.2 format).

Subsequent volumes of MICRO/Apple will contain comprehensive reference materials, more advanced machine language routines, and educational primers. These volumes will not only contain articles published in MICRO but other original material, some of it too lengthy to fit into the magazine format.

About the Editor

Ford Cavallari received a degree in mathematics from Dartmouth. While there, he made extensive use of the college’s time-sharing and microcomputer facilities and helped convert several important BASIC academic programs to run on Apple II systems. His work with the Apple has ranged from large-scale computer architecture projects to tiny, recreational graphics programs. He is a founding member of the Computer Literacy Institute. As Apple Specialist on the staff of MICRO, The 6502 Journal, he serves as Editor of the MICRO/Apple book series.

$24.95 in U.S./Canada
(Including floppy disk)

MICRO INK, Inc.
P.O. Box 6502
Chelmsford, Massachusetts 01824