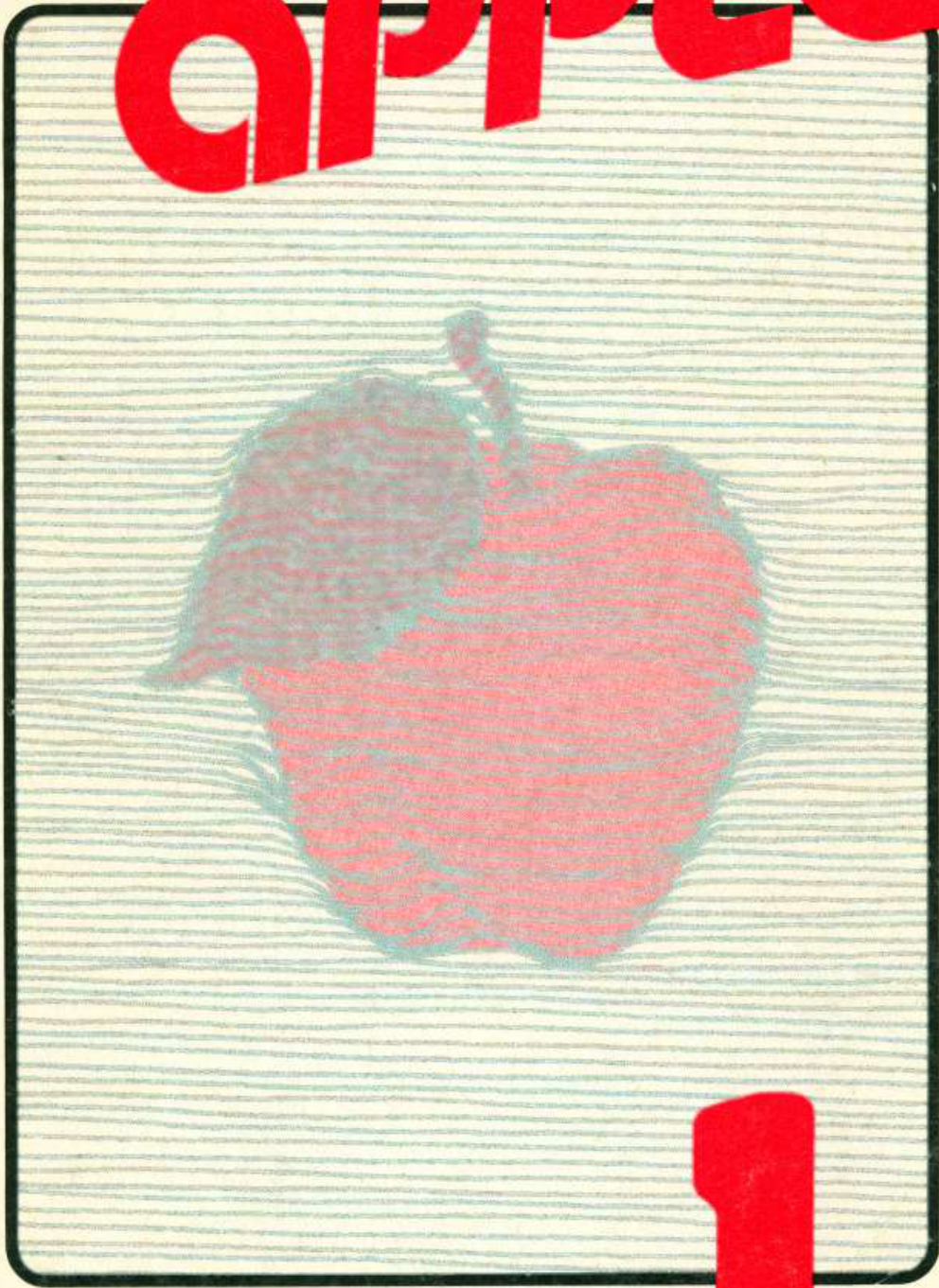


MICRO APPLE



1

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.

Contents

	INTRODUCTION	1
1	BASIC AIDS	3
	Applesoft Renumbering	5
	<i>James D. Childress</i>	
	Better Utilization of Apple Computer Renumber and Merge Program	9
	<i>Frank D. Chipchase</i>	
	SEARCH/CHANGE in Applesoft	12
	<i>James D. Childress</i>	
	An Apple II Program Edit Aid	17
	<i>Alan G. Hill</i>	
2	I/O ENHANCEMENTS	23
	A Little Plus for Your Apple II	25
	<i>Craig Peterson</i>	
	Zoom and Squeeze	29
	<i>Gary B. Little</i>	
	A Slow List for Apple BASIC	33
	<i>R.B. Sander-Cederlof</i>	
	Alarming Apple	37
	<i>Paul Irwin</i>	
3	RUNTIME UTILITIES	41
	Data Statement Generator	43
	<i>Virginia Lee Brady</i>	
	An EDIT Mask Routine in Applesoft BASIC	47
	<i>Lee Reynolds</i>	
	Business Dollars and Sense in Applesoft	55
	<i>Barton M. Bauers, Jr.</i>	
	Lower Case and Punctuation in Applesoft	62
	<i>James D. Childress</i>	
4	GRAPHICS	67
	Graphing Rational Functions	69
	<i>Ron Carlson</i>	
	A Hi-Res Graph Plotting Subroutine in Integer BASIC for the Apple II	75
	<i>Richard Fam</i>	

	How to Do a Shape Table Easily and Correctly	78
	<i>John Figueras</i>	
	Define Hi-Res Characters for the Apple II	96
	<i>Robert F. Zant</i>	
	Apple II High Resolution Graphics Memory Organization	99
	<i>Andrew H. Eliason</i>	
5	EDUCATION	103
	Apple Pi	105
	<i>Robert J. Bishop</i>	
	Sorting Revealed	109
	<i>Richard C. Vile, Jr.</i>	
	Solar System Simulation with or without an Apple II	134
	<i>David A. Partyka</i>	
	Programming with Pascal	143
	<i>John P. Mulligan</i>	
6	GAMES	153
	Spelunker	155
	<i>Thomas R. Mimlitch</i>	
	Life for Your Apple	168
	<i>Richard F. Sutor</i>	
	Apple II Speed Typing Test with Input Time Clock	173
	<i>John Broderick, CPA</i>	
	Ludwig Von Apple II	175
	<i>Marc Schwartz and Chuck Carpenter</i>	
7	REFERENCE	179
	An Apple II Programmer's Guide	181
	<i>Rick Auricchio</i>	
	Exploring the Apple II DOS	186
	<i>Andy Hertzfeld</i>	
	Applesoft II Shorthand	191
	<i>Allen J. Lacy</i>	
	The Integer BASIC Token System in the Apple II	198
	<i>Frank D. Kirschner</i>	
	Creating an Applesoft BASIC Subroutine Library	204
	<i>N.R. McBurney</i>	
	LANGUAGE INDEX	212
	AUTHOR INDEX — Biographies included	213
	DISK INFORMATION	216

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

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

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 - 0C 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, 0C, 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.


```

62987 REM *****
62988 REM *
62989 REM * APPLESOFT RENUMBERING *
62990 REM * JAMES D. CHILDRESS *
62991 REM *
62992 REM * RENUMBER *
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--";BGN: INPUT
" ENDING LINE--";TRM: INPUT " TOTAL NUMBER OF LINES (ROUGHLY)--"
;D: PRINT
63020 INPUT "RENUMBERED BEGINNING LINE--";SK: INPUT "INCREMENT--";ADD
63030 START = 256 * PEEK (104) + PEEK (103):M = START + 2:DD = INT (D /
4): DIM LS(D),LN(DD),LM(DD),LOC(DD),NA$(DD),ND(DD),INS(DD),IMS(DD)
63040 L = L + 1:LS(L) = M:LC = 256 * PEEK (M + 1) + PEEK (M): IF LC > 6
2900 THEN 63100
63050 FOR J = M + 2 TO M + 255:TST = PEEK (J): IF TST = 0 THEN M = J +
3: GOTO 63040
63060 IF TST = 171 THEN NA$(K + 1) = "GOTO": GOSUB 63250
63070 IF TST = 176 THEN NA$(K + 1) = "GOSUB": GOSUB 63250
63080 IF TST = 196 AND PEEK (J + 1) > 47 AND PEEK (J + 1) < 58 THEN NA
$(K + 1) = "THEN": GOSUB 63250
63090 NEXT
63100 FOR J = 1 TO L:LNU = 256 * PEEK (LS(J) + 1) + PEEK (LS(J)): IF L
NU > TRM OR LNU > 62900 THEN PRINT : PRINT "RENUMBERING COMPLETED T
ROUGH LINE ";LUN;".": GOTO 63350
63110 IF LNU < BGN THEN 63190
63120 SK$ = "0000" + STR$(SK):SK$ = RIGHT$(SK$,5)
63130 FOR I = 1 TO K: IF LNU < > INS(I) THEN NEXT : GOTO 63180
63140 FOR KA = 1 TO ND(I): POKE LOC(I) + 1 + ND(I) - KA, VAL ( MID$( SK$
,6 - KA,1)) + 48: NEXT
63150 IF LNU = INS(I) THEN IMS(I) = SK
63160 IF LEN ( STR$( SK)) > ND(I) THEN PCR = 1
63170 NEXT
63180 SO = INT (SK / 256): POKE (LS(J) + 1),SO: POKE (LS(J)),SK - 256 *
SO
63190 FOR I = 1 TO K: IF LNU = LN(I) THEN LM(I) = SK: IF LNU < BGN THEN
LM(I) = LNU
63200 NEXT
63210 SK = SK + ADD:LUN = LNU
63220 NEXT
63250 K = K + 1:LN(K) = LC:SU = PEEK (J + 1) - 48
63260 FOR KA = J + 2 TO J + 6:CPR = PEEK (KA): IF CPR = 0 OR CPR = 58 OR
CPR = 44 THEN GOTO 63290
63270 SU = 10 * SU + CPR - 48
63280 NEXT
63290 LOC(K) = J:ND(K) = KA - 1 - J:INS(K) = SU:J = KA - 1: IF CPR = 44 THEN
NA$(K + 1) = "COMMA":J = KA: GOTO 63250
63300 RETURN
63310 END
63350 IF PCP < > 1 THEN END
63360 PRINT : PRINT "NOTE: YOU MUST MAKE THE FOLLOWING CHAN-": PRINT "GE
S MANUALLY.": PRINT
63370 FOR I = 1 TO K: IF LEN ( STR$( IMS(I))) < = ND(I) THEN NEXT : END
63380 PRINT "LINE ";LM(I)," : INSERT ";IMS(I);" AFTER ";NA$(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:

HIMEM: 36352 (C/R)
POKE 1013,76 (C/R)
POKE 1014,0 (C/R)
POKE 1015,142 (C/R)

For A 32K System:

HIMEM: 19968 (C/R)
POKE 1013,76 (C/R)
POKE 1014,00 (C/R)
POKE 1015,78 (C/R)

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 a-gley. 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(I) by B(X%(I)).

```

62986 REM *****
62987 REM *
62988 REM * SEARCH/CHANGE *
62989 REM * J. D. CHILDRESS *
62990 REM *
62991 REM * SEARCH/CHANGE *
62992 REM *
62993 REM * COPYRIGHT (C) 1981 *
62994 REM * MICRO INK, INC. *
62995 REM * CHELMSFORD, MA 01824 *
62996 REM * ALL RIGHTS RESERVED *
62997 REM *
62998 REM *****
62999 END ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
63000 DIM SK(100),NT(100),L(100):START = 256 * PEEK (104) + PEEK (103)
:FINI = 256 * PEEK (106) + PEEK (105)
63010 HM = 256 * PEEK (116) + PEEK (115): POKE 834,160: POKE 835,0: POKE
836,76: POKE 837,44: POKE 838,254
63020 FOR WW = FINI - 540 TO START STEP - 1: IF X = 0 AND PEEK (WW) =
188 THEN W = WW:X = 1:WW = WW - 1730
63030 IF 256 * PEEK (WW + 1) + PEEK (WW) < > 62986 THEN NEXT
63040 NL = 256 * PEEK (WW - 1) + PEEK (WW - 2):CO = NL - WW - 10: HIMEM:
HM - WW - 100
63050 IF 256 * PEEK (START + 3) + PEEK (START + 2) < > 1 THEN PRINT
" YOU MUST ENTER YOUR SEARCH ITEM AS LINE": PRINT "1 BEFORE YOU RUN
63000.": HIMEM: HM: END
63060 LIST 1,2: PRINT : PRINT "PLEASE VERIFY IF THE COMPUTER TAK
ES": PRINT "THIS AS YOU INTENDED. DO YOU WANT": INPUT "TO CONTINUE (
YES OR NO)? ";Y$: PRINT : IF Y$ < > "YES" THEN HIMEM: HM: END
63070 PRINT "THE CHANGE ENTERED WILL BE MADE IN THE": PRINT "RANGE OF LI
NES CHOSEN ENTER": INPUT " THE BEGINNING LINE ";BL: INPUT " THE
ENDING LINE ";EL: PRINT
63080 NF = 256 * PEEK (START + 1) + PEEK (START)
63090 FOR I = 0 TO 255:SK(I) = PEEK (START + 4 + I): IF SK(I) < > 0 THEN
NEXT
63100 N = I - 1
63110 IF SK(0) = 34 OR SK(0) = 64 THEN SQ = (SK(0) = 34): FOR I = 1 TO N
:SK(I - 1) = SK(I): NEXT :N = N - 1
63120 M = START + N + 6 + SQ:INC = 0:CH = 0: IF 256 * PEEK (M + 3) + PEEK
(M + 2) < > 2 THEN CH = 1: GOTO 63170
63130 FOR I = 0 TO 255:NT(I) = PEEK (M + 4 + I): IF NT(I) < > 0 THEN NEXT
63140 NN = I - 1:ADD = NN - N:M = M + NN + 6:WW = WW - ADD(ADD < 0) + 6
63150 IF NT(0) = 34 THEN FOR I = 1 TO NN:NT(I - 1) = NT(I): NEXT :NN =
NN - 1:ADD = ADD - 1
63160 X = INT (NL / 256):Y = NL - 256 * X: POKE START + 1,X: POKE START,
Y
63170 LM = 256 * PEEK (M + 3) + PEEK (M + 2):NA = 256 * PEEK (M + 1) +
PEEK (M): IF LM > 62986 THEN 63230

```

```

63180 FOR I = M + 4 TO M + 255: IF PEEK (I) < > 0 AND PEEK (I) < > S
      K(0) AND PEEK (I) < > 34 THEN NEXT
63190 IF PEEK (I) = 34 THEN SQ = SQ + 1 - 2 * (SQ = 1)
63200 IF PEEK (I) = SK(0) AND SQ < > 1 THEN GOSUB 63330
63210 IF PEEK (I) = 0 THEN NA = NA + INC: X = INT (NA / 256): Y = NA - 2
      56 * X: POKE M + 1, X: POKE M, Y: M = I + 1: GOTO 63170
63220 NEXT
63230 X = INT (NF / 256): Y = NF - 256 * X: POKE START + 1, X: POKE START,
      Y
63240 PRINT : PRINT : PRINT "THE ITEM": PRINT " "; LIST 1: PRINT : PRINT
      "IS FOUND IN THE FOLLOWING LINES:": PRINT : IF L(1) = 0 THEN PRINT
      " NONE.": HIMEM: HM: END
63250 FOR I = 1 TO K: PRINT L(I),: NEXT : PRINT
63260 PRINT : INPUT "DO YOU WANT THESE LINES LISTED (Y/N)? "; Y$: IF Y$ =
      "N" THEN HIMEM: HM: END
63270 PRINT : PRINT "THERE WILL BE A WAIT AFTER EACH LINE": PRINT "UNTIL
      YOU HIT 'RETURN' TO CONTINUE.": PRINT
63280 FOR I = 1 TO K: IF L(I) = L(I - 1) THEN 63320
63290 L$ = "0000" + STR$ (L(I)): L$ = RIGHT$ (L$, 5)
63300 FOR J = 1 TO 5: POKE W + J, 48 + VAL ( MID$ (L$, J, 1)): NEXT
63310 LIST 00000::: INPUT " "; Y$
63320 NEXT : HIMEM: HM: END
63330 FOR J = 0 TO N: IF PEEK (I + J) < > SK(J) THEN RETURN
63340 NEXT
63350 IF CO - INC < ADD AND CO% = 0 THEN CH = 1: CO% = 1: PRINT "THE SUPP
      LY OF COLONS IN LINE 62999 IS": PRINT "DEPLETED. THE CHANGE HAS BEE
      N MADE": PRINT "THROUGH PART OR ALL OF LINE "; LM: PRINT
63360 K = K + 1: L(K) = LM: IF CH < > 0 OR LM < BL OR LM > EL THEN RETURN
63370 ZS = I + N: X = INT (ZS / 256): Y = ZS - 256 * X: POKE 61, X: POKE 60
      , Y: X = INT (WW / 256): Y = WW - 256 * X: POKE 63, X: POKE 62, Y: ZH = H
      M - 100 - WW + ZS: X = INT (ZH / 256): Y = ZH - 256 * X: POKE 67, X: POKE
      66, Y: CALL 834
63380 POKE 61, X: POKE 60, Y: X = INT ((HM - 100) / 256): Y = HM - 100 - 25
      6 * X: POKE 63, X: POKE 62, Y: ZS = I + NN: X = INT (ZS / 256): Y = ZS -
      256 * X: POKE 67, X: POKE 66, Y: CALL 834
63390 WW = WW + ADD: INC = INC + AD
63400 FOR J = 0 TO NN: POKE I + J, NT(J): NEXT : I = I + NN
63410 RETURN

```

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 *
32582 REM * PROGRAM EDIT AID *
32583 REM * ALAN G. HILL *
32584 REM *
32585 REM * EDIT *
32586 REM *
32587 REM * COPYRIGHT (C) 1981 *
32588 REM * MICRO INK, INC. *
32589 REM * CHELMSFORD, MA 01824 *
32590 REM * ALL RIGHTS RESERVED *
32591 REM *
32592 REM *****
32593 REM
32600 DIM A$(30)
32610 INPUT "FIND?",A$: CALL -936: IF A$(1,1)="@" THEN 32630:KK= LEN(A$):
FOR I=1 TO KK: POKE 911+I, ASC(A$(I,I)): NEXT I
32620 POKE 12,KK-1: GOTO 32650
32630 A$=A$(2, LEN(A$)):KK= LEN(A$): FOR I=1 TO KK STEP 2:J= ASC(A$(I,I))
-176:JJ= ASC(A$(I+1,I+1))-176
32640 IF J>9 THEN J=J-7: IF JJ>9 THEN JJ=JJ-7: POKE 912+I/2,J*16+JJ: NEXT
I: POKE 12,KK/2-1
32650 POKE 10,912 MOD 256: POKE 11,912/256
32660 CALL 768
32670 IF PEEK (18)>127 THEN 32730:LINE= PEEK (17)+ PEEK (18)*256
32680 LIST LINE
32690 IF PEEK (37)>18 THEN 32730
32700 CALL 846
32710 GOTO 32670
32720 CALL -936: GOTO 32700
32730 END

```



```

0800      1 ;*****
0800      2 ;*   FIND ROUTINE   *
0800      3 ;*       BY       *
0800      4 ;*   ALAN G. HILL *
0800      5 ;*
0800      6 ;* COPYRIGHT(C) 1979 *
0800      7 ;* MICRO INK, INC. *
0800      8 ;*ALL RIGHTS RESERVED*
0800      9 ;*
0800     10 ;* ROUTINE TO SEARCH *
0800     11 ;* INTEGER BASIC PRO-*
0800     12 ;* GRAMS FOR ALL OCC-*
0800     13 ;* URRENCES OF SPECI-*
0800     14 ;* FIED ITEMS..... *
0800     15 ;*****
0800     16 ;
0800     17 HILO   EPZ $03      ;HIMEM LO BYTE
0800     18 HIHI   EPZ $04      ;HIMEM HI BYTE
0800     19 BSL    EPZ $06      ;BASIC STATEMENT LO
0800     20 BSH    EPZ $07      ;BASIC STATEMENT HI
0800     21 SEAL   EPZ $08      ;STATEMENT ENDING ADDRESS LO
0800     22 SEAH   EPZ $09      ;STATEMENT ENDING ADDRESS HI
0800     23 STRL   EPZ $0A      ;STRING LO
0800     24 LNL    EPZ $11      ;LINE NUMBER LO
0800     25 LNH    EPZ $12      ;LINE NUMBER HI
0800     26 ;
0800     27 ;*****
0800     28 ;* MAIN PROGRAM *
0800     29 ;*****
0800     30 ;
0300     31          ORG $300
0300     32          OBJ $800
0300     33 ;
0300 A5CA     34 START  LDA $00CA      ;SET UP ADDRESS OF FIRST
0302 8506     35          STA BSL      ;BASIC STATEMENT IN
0304 A5CB     36          LDA $CB      ;LOCS 6 AND 7
0306 8507     37          STA BSH
0308 A54C     38          LDA $4C      ;SET UP TO STOP SEARCH
030A 8503     39          STA HILO     ;AT HIMEM. COULD BE
030C A54D     40          LDA $4D      ;CHANGED TO LIMIT SEARCH
030E 8504     41          STA HIHI     ;AT END OF PROGRAM BEING EDITED
0310 A000     42 LENGTH LDY #$00      ;GET STATEMENT LENGTH
0312 B106     43          LDA (BSL),Y
0314 38       44          SEC
0315 E902     45          SBC #$02      ;MINUS 2 TO POINT TO
0317 18       46          CLC
0318 6506     47          ADC BSL
031A 8508     48          STA SEAL      ;SET UP STATEMENT ENDING
031C A507     49          LDA BSH      ;ADDRESS IN 8 AND 9
031E 6900     50          ADC #$00      ;ADD IN CARRY IF ANY
0320 8509     51          STA SEAH
0322 A001     52          LDY #$01      ;SAVE LINE NUMBER IN
0324 B106     53          LDA (BSL),Y     ;IN 11 AND 12
0326 8511     54          STA LNL
0328 C8       55          INY
0329 B106     56          LDA (BSL),Y
032B 8512     57          STA LNH
032D A200     58          LDX #$00      ;ADJUST BSL TO POINT
032F A903     59          LDA #$03      ;TO FIRST TOKEN
0331 206403   60          JSR INCPNT
0334 A000     61          LDY #$00      ;COMPARE TOKEN TO
0336 B106     62 TTOKEN LDA (BSL),Y     ;FIRST CHARACTERIN
0338 D10A     63          CMP (STRL),Y     ;STRING
033A D003     64          BNE NXTOKN     ;IF NOT EQUAL POINT TO NEXT
033C 207F03   65          JSR COMPAR     ;IFEQUAL COMPARE REMAINING CHARS
033F 207003   66 NXTOKN JSR INCTOK      ;POINT TO NEXT TOKEN
0342 90F2     67          BCC TTOKEN     ;CARRY CLEAR THEN LOOK AT NEXT
0344 A508     68          LDA SEAL      ;AT END OF STATEMENT.
0346 C503     69          CMP HILO     ;CHECK TO SEE IF AT END OF
0348 A509     70          LDA SEAH     ;SEARCH LIMIT
034A E504     71          SBC HIHI
034C B011     72          BCS LIMIT
034E A508     73          LDA SEAL      ;CARRY SET = LIMIT OF SEARCH
;SET UP BSL AND BSH TO POINT

```

```

0350 8506      74          STA BSL          ;TO NEXT STATEMENT
0352 A509      75          LDA SEAH
0354 8507      76          STA BSH
0356 A200      77          LDX #$00          ;POINT TO LENGTH OF
0358 A902      78          LDA #$02          ;STATEMENT BYTE
035A 206403    79          JSR INCPNT
035D D0B1      80          BNE LENGTH        ;ALWAYS BRANCH
035F A9FF      81  LIMIT  LDA #$FF          ;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  INCPNT CLC              ;ROUTINE TO INCREMENT
0365 7506      90          ADC BSL,X        ;POINTERS. ENTER WITH
0367 9506      91          STA BSL,X        ;XREG = DISPLACEMENT
0369 B507      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 SEAH        ;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 88        120  COMPX  DEY
038B 10F4      121         BPL COMPY      ;FOUND A MATCH! POP STACK ADDRESS
038D 68        122         PLA
038E 68        123         PLA
038F 60        124         RTS
038F           125         END

```

```

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

```

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

** ZERO PAGE VARIABLES:

```

HILO 0003 HIHI 0004 BSL 0006 BSH 0007 SEAL 0008 SEAH 0009
STRL 000A LNL 0011 LNH 0012

```

** ABSOLUTE VARIABLES/LABELS

```

START 0300 LENGTH 0310 TTOKEN 0336
NXTOKN 033F LIMIT 035F INCPNT 0364 INCTOK 0370 REXIT 037E COMPAR 037F
COMPY 0381 COMPX 038A

```

SYMBOL TABLE STARTING ADDRESS:6000
SYMBOL TABLE LENGTH:00B2

2

I/O ENHANCEMENTS

Introduction	24
A Little Plus for Your Apple II <i>Craig Peterson</i>	25
Zoom and Squeeze <i>Gary B. Little</i>	29
A Slow List for Apple BASIC <i>R.B. Sander-Cederlof</i>	33
Alarming Apple <i>Paul Irwin</i>	37

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 \$AA5B 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.

```

0800      1  ;*****
0800      2  ;*
0800      3  ;*          EDIT PLUS          *
0800      4  ;*          CRAIG PETERSON      *
0800      5  ;*
0800      6  ;*          EDIT PLUS          *
0800      7  ;*
0800      8  ;*          COPYRIGHT (C) 1981    *
0800      9  ;*          MICRO INK, INC.     *
0800     10  ;*          ALL RIGHTS RESERVED *
0800     11  ;*
0800     12  ;*A PROGRAM TO GIVE THE STANDARD*
0800     13  ;* APPLE II THE ENHANCED CURSOR *
0800     14  ;* EDITING CAPABILITIES OF THE  *
0800     15  ;*          APPLE II PLUS     *
0800     16  ;*
0800     17  ;*****
0800     18  ;
0800     19  WNWD   EPZ $21
0800     20  CH    EPZ $24
0800     21  BASL  EPZ $28
0800     22  YSAV  EPZ $35
0800     23  CSWL  EPZ $36
0800     24  CSWH  EPZ $37
0800     25  KSWL  EPZ $38
0800     26  KSWH  EPZ $39
0800     27  DOS   EQU $A851
0800     28  YDOS  EQU $AA5B
0800     29  KBRD  EQU $C000
0800     30  STRB  EQU $C010
0800     31  ESC1  EQU $FC2C
0800     32  CEOP  EQU $FC42
0800     33  RKEY  EQU $FDOC
0800     34  OUT1  EQU $FDF0
0800     35  KEYN  EQU $FD1B
0800     36  MNTR  EQU $FF65
0800     37  ;
0300     38      ORG $300
0300     39      OBJ $800
0300     40  ;
0300 A913   41  BGIN  LDA #$13          ;CHANGE
0302 8538   42      STA KSWL          ; OUTPUT POINTERS
0304 A903   43      LDA #$03          ; TO NEW ROUTINE
0306 8539   44      STA KSWH          ; AT 'SKEY' AND
0308 A969   45      LDA #$69          ; 'SVID' RESP.
030A 8536   46      STA CSWL
030C A903   47      LDA #$03
030E 8537   48      STA CSWH
0310 4C51A8 49      JMP DOS              ;CHANGE DOS POINTERS & RETURN
0313 201BFD 50  SKEY  JSR KEYN          ;GET NEXT CHARACTER
0316 C99B    51      CMP #$9B          ;IS CHARACTER = 'ESC'?
0318 F00B    52      BEQ ESC2          ;IF SO, GO TO ESC2
031A 60      53      RTS              ;IF NOT, RETURN
031B 38      54  SPCL  SEC              ;PREPARE A POINTER AND
031C E9C9    55      SBC #$C9          ; TURN I,J,K, AND M
031E A8      56      TAY              ; INTO A,B,C, AND D,
031F B98C03 57      LDA TABL,Y          ; RESPECTIVELY..
0322 202CFC 58      JSR ESC1          ;DO STANDARD ESCAPE

```

```

0325 A424      59  ESC2  LDY CH          ;GET THE NEXT INPUT
0327 B128      60          LDA (BASL),Y    ; CHARACTER.....
0329 48        61          PHA
032A 293F      62          AND #$3F
032C 0940      63          ORA #$40
032E 9128      64          STA (BASL),Y
0330 68        65          PLA
0331 201BFD    66          JSR KEYN
0334 C9C8      67          CMP #$C8          ;IS CHAR = 'H'?
0336 D002      68          BNE SKIP          ;IF NOT=H, SKIP
0338 A9C0      69  HOME  LDA #$C0          ;IF IS, LOAD '@'
033A C9D0      70  SKIP  CMP #$D0          ;IS CHAR = 'P'?
033C F018      71          BEQ POKE          ;IF IS, --> POKE
033E C9CE      72          CMP #$CE          ;IS CHARACTER >= 'N'?
0340 F010      73          BEQ NRML          ;IF='N', GOTO NRML
0342 B019      74          BCS RTRN          ;IF>'N', GOTO RETRN
0344 C9C9      75          CMP #$C9          ;IS CHARACTER < 'I'?
0346 9015      76          BCC RTRN          ; IF SO, RETURN.
0348 C9CC      77          CMP #$CC          ;IS CHARACTER = 'L'?
034A D0CF      78          BNE SPCL          ;IF <> 'L', DO SPCL.
034C 2051A8    79          JSR DOS           ;IF = 'L', RESET DOS
034F 4C65FF    80          JMP MNTR          ; POINTERS AND JUMP TO MNTR.
0352 A928      81  NRML  LDA #$28          ;NORMAL SCREEN
0354 D005      82          BNE CONT          ; WIDTH=$28 (40)
0356 2042FC    83  POKE  JSR CEOP          ;CLEAR TO EOP &
0359 A921      84          LDA #$21          ; POKE 33,33
035B 8521      85  CONT  STA WNWD          ;STORE->WINWIDTH
035D 38        86  RTRN  SEC           ;CHAR. IS NOT I,J,K, OR M,
035E 202CFC    87          JSR ESC1          ; SO DO A STANDARD ESC.
0361 A424      88          LDY CH          ;CORRECT YSAVE REGISTER IN
0363 8C5BAA    89          STY YDOS          ; DOS AND RETURN...
0366 4C0CFD    90          JMP RKEY
0369 8435      91  SVID  STY YSAV          ;SAVE Y.
036B C98D      92          CMP #$8D          ;IS CHARACTER A CR?
036D D018      93          BNE RETN          ;IF NOT, RETURN.
036F AC00C0    94          LDY KBRD          ;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 2C10C0    98          BIT STRB          ;CLEAR KEYBOARD STROBE
037B AC00C0    99  AGIN  LDY KBRD          ;IS KEY PRESSED?
037E 10FB     100          BPL AGIN          ;IF NOT, TRY AGAIN!
0380 C083     101          CPY #$83          ;IS IT CTRL 'C'?
0382 F003     102          BEQ RETN          ;IF SO, LEAVE STROBE
0384 2C10C0   103          BIT STRB          ;CLEAR KEYBOARD STROBE
0387 A435     104  RETN  LDY YSAV          ;RESTORE Y AND REJOIN
0389 4CF0FD   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 KBRD  C000 STRB  C010
ESC1  FC2C CEOP  FC42 RKEY  FDOC 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 F**
****ORMATTED 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 'ORMATTED' 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.

```

0800      1  ;*****
0800      2  ;*  ZOOM AND SQUEEZE *
0800      3  ;*    GARY LITTLE  *
0800      4  ;*                    *
0800      5  ;*          ZOOM    *
0800      6  ;*                    *
0800      7  ;* COPYRIGHT(C) 1980 *
0800      8  ;* MICRO INK, INC.  *
0800      9  ;*ALL RIGHTS RESERVED*
0800     10  ;*                    *
0800     11  ;* PROGRAM EXTENDING *
0800     12  ;* EDITING FEATURES  *
0800     13  ;* OF THE APPLE...  *
0800     14  ;*****
0800     15  ;
0800     16  ;
0800     17  WIDTH  EPZ $21          ;WINDOW WIDTH
0800     18  CH    EPZ $24          ;HORIZONTAL CURSOR POSITION
0800     19  BASL  EPZ $28          ;SCREEN BASE ADDRESS POINTER
0800     20  KSWL  EPZ $38          ;INPUT HOOK (LO)
0800     21  ;
0800     22  IN    EQU $0200        ;INPUT BUFFER
0800     23  KEYIN EQU $FD1B        ;KEYPRESS ROUTINE
0800     24  ;
0800     25  ;
0300     26  START ORG $300        ;MAIN PROGRAM
0300     27  OBJ  $800
0300     28  ;
0300 A909     29  LDA #INHK          ;SET INPUT HOOK
0302 8538     30  STA KSWL          ; TO INHK ($309)
0304 A903     31  LDA /INHK
0306 8539     32  STA KSWL+1
0308 60       33  RTS
0309         34  ;

```

```

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 #$21           ;CHANGE WINDOW WIDTH
0312 8521    43        STA WIDTH        ; TO 33
0314 A9A0    44        LDA #$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 E8      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 RTS1  RTS           ; RETURN
          68        END

```

***** END OF ASSEMBLY

```

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

```

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

** ZERO PAGE VARIABLES:

WIDTH 0021 CH 0024 BASL 0028 KSWL 0038

** ABSOLUTE VARIABLES/LABELS

IN 0200 KEYIN FD1B
START 0800 INHK 0309 CTRLZ 0317 LOOP 031B FIN 0335 RTS1 033A

SYMBOL TABLE STARTING ADDRESS:6000
SYMBOL TABLE LENGTH:0072

!

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:


```

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  ;
0340     20          ORG $340          ;MAIN PROGRAM
0340     21          OBJ $800
0340     22  ;
0340 C98D  23  SLOW  CMP #$8D          ;CHECK IF CHAR IS CARRIAGE RETURN
0342 D01A  24          BNE CHR0UT      ;NO, SO GO BACK TO COUT
0344 48    25          PHA              ;SAVE CHARACTER ON STACK
0345 2C00C0 26          BIT $C000       ;TEST KEYBOARD STROBE
0348 100E  27          BPL WAIT        ;NOTHING TYPED YET
034A AD00C0 28          LDA $C000       ;GET CHARACTER FROM KEYBOARD
034D 2C10C0 29          BIT $C010       ;CLEAR KEYBOARD STROBE
0350 C9A0  30          CMP #$A0        ;CHECK IF CHAR IS A SPACE
0352 F010  31          BEQ STOP        ;YES - STOP LISTING
0354 C98D  32          CMP #$8D        ;CHECK IF CHAR IS CARRIAGE RETURN
0356 F009  33          BEQ ABORT       ;YES - ABORT LISTING
0358 A900  34  WAIT   LDA #$00        ;MAKE A LONG DELAY
035A 20A8FC 35          JSR $FCA8       ;CALL MONITOR DELAY SUBROUTINE
035D 68    36          PLA              ;GET CHARACTER FROM STACK
035E 4CF0FD 37  CHR0UT JMP $FDF0       ;REJOIN COUT SUBROUTINE
0361 4C03E0 38  ABORT JMP $E003       ;SOFT ENTRY INTO APPLE BASIC
0364 2C00C0 39  STOP   BIT $C000       ;WAIT UNTIL KEYBOARD STROBE
0367 10FB  40          BPL STOP        ;APPEARS ON THE SCENE
0369 8D10C0 41          STA $C010       ;CLEAR THE STROBE
036C 30EA  42          BMI WAIT        ;UNCONDITIONAL BRANCH
036E      43  ;
036E      44  ;*****
036E      45  ;* SUBROUTINE TO DE- *
036E      46  ;* ACTIVATE SLOW LIST *
036E      47  ;*****
036E      48  ;
036E A9F0  49  OFF   LDA #$F0          ;RESTORE $FDF0 TO
0370 8536  50          STA $36          ;LOCATIONS $36 AND $37
0372 A9FD  51          LDA #$FD
0374 8537  52          STA $37
0376 60    53          RTS
0377      54  ;

```

36 I/O Enhancements

```
0377      55 ;*****
0377      56 ;* SUBROUTINE TO *
0377      57 ;* ACTIVATE SLOW LIST *
0377      58 ;*****
0377      59 ;
0377 A940   60 ON      LDA #$40      ;SET $340 INTO
0379 8536   61      STA $36      ;LOCATIONS $36 AND $37
037B A903   62      LDA #$03
037D 8537   63      STA $37
037F 60     64      RTS
0380      65 ;
          66      END
```

***** END OF ASSEMBLY

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

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

** ZERO PAGE VARIABLES:

** ABSOLUTE VARIABLES/LABELS

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

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 \$DD0, 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.

```

0800      1  ;*****
0800      2  ;*
0800      3  ;*  ALARMING  APPLE  *
0800      4  ;*                BY      *
0800      5  ;*                PAUL IRWIN *
0800      6  ;*
0800      7  ;*                ALARM    *
0800      8  ;*
0800      9  ;*  COPYRIGHT (C) 1981  *
0800     10 ;*  MICRO INK, INC.    *
0800     11 ;*  ALL RIGHTS RESERVED *
0800     12 ;*
0800     13 ;*****
0800     14 ;
0D83     15 ;                ORC $0D83
0D83     16 ;                OBJ $800
0D83     17 ;
0D83     18 ;
0D83 AD30C0 19 ;                LDA $C030
0D86 8E     20 ;                DEY
0D87 D005   21 ;                BNE $0D8E
0D89 CE820D 22 ;                DEC $0D82
0D8C F009   23 ;                BEQ $0D97

```

0D8E	CA	24	DEX
0D8F	D0F5	25	BNE \$0D86
0D91	AE810D	26	LDX \$0D81
0D94	4C830D	27	JMP \$0D83
0D97	60	28	RTS
0D98	A000	29	LDY #\$00
0D9A	A200	30	LDX #\$00
0D9C	A947	31	LDA #\$47
0D9E	8D810D	32	STA \$0D81
0DA1	A9A0	33	LDA #\$A0
0DA3	8D820D	34	STA \$0D82
0DA6	20830D	35	JSR \$0D83
0DA9	2C00C0	36	BIT \$C000
0DAC	10D7	37	BPL \$0D85
0DAE	AD00C0	38	LDA \$C000
0DE1	2C10C0	39	BIT \$C010
0DB4	60	40	RTS
0DB5	A000	41	LDY #\$00
0DB7	A200	42	LDX #\$00
0DB9	A960	43	LDA #\$60
0DBB	8D810D	44	STA \$0D81
0DBE	A9A0	45	LDA #\$A0
0DC0	8D820D	46	STA \$0D82
0DC3	20830D	47	JSR \$0D83
0DC6	4C980D	48	JMP \$0D98
0DC9	20980D	49	JSR \$0D98
0DCC	C983	50	CMP #\$83
0DCE	D0F9	51	BNE \$0DC9
0DD0	60	52	RTS
		53	END

```

1 REM *****
3 REM * SAMPLE BASIC CALL SEQUENCE *
4 REM * FOR ALARM PROMPT ROUTINE *
6 REM *****
7 REM
10 TASK=3000:OFF=0:TASK=200:TRAP=300:ALARM=3529
95 REM
96 REM *****
97 REM MAIN LINE SEQUENCE
98 REM *****
99 REM
100 ERR=OFF:GOSUB TASK:IF ERR THEN GOSUB TRAP
110 GOTO 32767:REM *** BOGUS LINE # ***
120 REM
200 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

```


3

RUNTIME UTILITIES

Introduction	42
Data Statement Generator <i>Virginia Lee Brady</i>	43
An Edit Mask Routine in Applesoft BASIC <i>Lee Reynolds</i>	47
Business Dollars and Sense in Applesoft <i>Barton M. Bauers, Jr.</i>	55
Lower Case and Punctuation in Applesoft <i>James D. Childress</i>	62

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.BO (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 AS" '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 + 1 + 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.BO 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.

Original Last Line			First Added Line			New Last Line		
POINT LOW	08	1000	PSN - 2	0A	2000	PSN - 2	40	1234
POINT HIGH	10	1001	PSN - 1	20	2001	PSN - 1	12	1235
LINE LOW	64	1002	PSN	65	2002	PSN	66	1236
LINE HIGH	00	1003	PSN + 1	00	2003	PSN + 1	00	1237
"DATA"	83	1004	PSN + 2	83	2004	PSN + 2	83	1238
data	XX	1005	PSN + 3	XX	2005	PSN + 3	XX	1239
	XX	1006	PSN + I + 3	XX	2006	PSN + I + 3	XX	123A
"END"	00	1007		XX	2007		XX	123B
NEXT LOW	00/02	1008		XX	2008		XX	123C
NEXT HIGH	00/02	1009	"END"	00	2009		XX	123D
Orig. End		100A	NEXT LOW	36	200A		XX	123E
			NEXT					
			HIGH	12	200B	"END"	00	123F
Note: Original Last Line						NEXT LOW	00	1240
NEXT LOW/HIGH change from 0000						NEXT		
						HIGH	00	1241
to 2002.				(AF.B0)		New End		1242

Figure 1: "MAP" of Two New DATA Statements being Added

```

10 REM *****
12 REM *
14 REM * DATA STATEMENT *
16 REM * GENERATOR *
18 REM * VIRGINIA LEE BRADY *
20 REM *
22 REM * DATA-GEN *
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 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 ";F$(1)
1016 INPUT "INPUT THE PATTERN ";F$(2)
1017 INPUT "INPUT THE YARDS IN DECIMAL ";F$(3)
1018 INPUT "INPUT THE FABRIC TYPE ";F$(4)
1020 REM ALLOW CHANGES
1035 FOR I = 1 TO 4: PRINT I; TAB( 5)F$(I): NEXT I
1040 INPUT "ANY CHANGES ? ";Y$: IF LEFT$(Y$,1) = "N" THEN 1050
1045 INPUT "WHICH ONE ? ";W: PRINT "CHANGE PART ";W;" TO ";: INPUT F$(W)
: GOTO 1035
1050 F$ = "": FOR I = 1 TO 3:F$ = F$ + F$(I) + ",": NEXT I:F$ = F$ + F$(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 (F$): POKE PSN + I + 2, ASC ( MID$ (F$,I,I)): NEXT
I
1070 POKE PSN + I + 2,0:NUMBER = PSN + I + 3: GOSUB 100
1075 POKE PSN - 2,LO: POKE PSN - 1,HI
1080 INPUT "ADD MORE ? ";Y$: 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$ = "$99,999.99"` and `NUM = 11.45`, then the result of editing is `OUT$ = " $11.45"` (note that there are four blanks preceding the dollar sign in the result field). And if `NUM = 2321`, then we have this result: `OUT$ = " $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 insertion 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 leftmost 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 character 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 I-1 = II AND MID$ (MASK$,I-1,1) = "9" then 480
500 IF N$ = " " 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 digits 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\$


```

10 REM *****
11 REM *
12 REM * EDIT MASK *
14 REM * LEE REYNOLDS *
15 REM *
16 REM * EDIT MASK *
17 REM *
18 REM * COPYRIGHT (C) 1981 *
20 REM * MICRO INK, INC. *
21 REM * CHELMSFORD, MA 01824 *
22 REM * ALL RIGHTS RESERVED *
23 REM *
24 REM *****
25 HOME : PRINT "EDIT MASK ROUTINE": PRINT : PRINT " THE EDIT MASK CAN
CONTAIN ANY INSERTION CHARACTERS, PLUS FOLLOWING SPECIAL
"
30 PRINT " CHARACTERS:": PRINT " IF $ IS FIRST CHAR., IT IS TREATED AS"
: PRINT " A FLOATING DOLLAR SIGN"
40 PRINT " IF - IS LAST CHAR., IT WILL BE OUTPUT": PRINT "IF NUMBER TO
EDIT IS NEGATIVE, OR RE-": PRINT "PLACED BY BLANK IF POSITIVE"
50 PRINT " 9 CORRESPONDS TO A DIGIT TO PLACE IN": PRINT "THAT POSITION
OF THE MASK": PRINT " 0 CORRESPONDS TO A NONZERO DIGIT TO"
60 PRINT "PLACE IN THAT POSITION. IF YOU WANT A": PRINT "COMMA OR COLON
IN THE MASK, ENCLOSE THE"
65 PRINT " ENTIRE MASK IN QUOTES TO INPUT IT.": PRINT
70 INPUT "EDIT MASK? ";MASK$
75 INPUT "NUMBER TO EDIT?";NUM: GOSUB 100: PRINT "EDITED NUMBER:";OUT$
80 PRINT : INPUT "1=NEW NUMBER, 2=NEW MASK AND NUMBER?";N
85 ON N GOTO 75,70
90 GOTO 80
100 NUM$ = STR$(NUM):LN = LEN (NUM$):LM = LEN (MASK$):QM = 0:QN = 0:R
M = 0:RN = 0:PN = 0:PM = 0:NF% = 0:MF% = 0:FD% = 0:DF% = 0
110 OUT$ = "": IF NUM < 0 THEN NF% = 1: REM SET FLAG TELLING WHETHER INPU
T NUMBER ISNEGATIVE
120 IF RIGHT$(MASK$,1) = "-" THEN MF% = 1: REM SET FLAG TELLING WHETH
ER INPUT MASK HAS TRAILING MINUS SIGN
130 IF LEFT$(MASK$,1) = "$" THEN FD% = 1: REM SET FLAG TELLING WHETHE
R INPUT MASK HAS FLOATING DOLLAR SIGN
140 FOR I = 1 TO LM: REM FIND POSITION OFDECIMAL POINT IN MASK
150 IF MID$(MASK$,I,1) = "." THEN PM = I
160 NEXT : IF FD% = 0 THEN DF% = 1: REM IF NO FLOATING DOLLAR SIGN IN M
ASK, SET FLAG SAYING "$" ALREADY OUTPUT TO EDITED FIELD
170 FOR I = 1 TO LN: REM FIND POSITION OF DECIMAL POINT IN NUMBER TO ED
IT
180 IF MID$(NUM$,I,1) = "." THEN PN = I
190 NEXT
200 IF PN THEN RN = LN - PN: REM IF DECIMAL POINT IN NUMBER, COMPUTE #
DIGITS RIGHT OF DECIMAL POINT
210 IF PM = 0 THEN 250: REM IF DECIMAL PT. IN MASK, FIND # DIGIT POSITI
ONS RIGHT OF IT
220 FOR I = LM TO PM STEP - 1
230 IF MID$(MASK$,I,1) = "0" OR MID$(MASK$,I,1) = "9" THEN RM = RM +
1
240 NEXT
250 IF PN = 0 AND PM = 0 THEN 300
260 IF RM = RN THEN 300
270 IF RM < RN THEN 290
280 FOR I = RN TO RM - 1:NUM$ = NUM$ + "0": NEXT : GOTO 300: REM ZERO-F
ILL RIGHTMOST DECIMAL POSITIONS OF NUM$
290 I = LN - RN + RM - 1:NUM$ = LEFT$(NUM$,I) + "": REM TRUNCATE NUM$
TO MATCH MASK, PUT "" IN RIGHTMOST DIGIT
300 QN = LEN (NUM$) - RM: IF PN THEN QN = QN - 1: REM GET # DIGITS LEFT
OF DEC. PT. IN NUMBER, IGNORING DEC. PT. IF ANY
310 IF NF% AND MF% THEN QN = QN - 1: REM IGNORE MINUS SIGN IN NUMBER I
F TRAILING MINUS IN MASK
320 FOR I = 1 TO LM: IF I = PM THEN 350: REM FIND # DIGITS IN MASK LEFT
OF DEC. POINT
330 IF MID$(MASK$,I,1) = "0" OR MID$(MASK$,I,1) = "9" THEN QM = QM +
1
340 NEXT
350 IF QM > = QN THEN 370: REM TRUNCATE NUMBER ON LEFT, MAKING LEFTMOS
T DIGIT ""
360 I = LEN (NUM$) - QN + QM - 1: IF NF% AND MF% THEN I = I - 1: REM DR
OP MINUS SIGN ALSO IF IGNORED BEFORE
365 NUM$ = "" + RIGHT$(NUM$,I):QN = QM

```

```

370 I1 = 1: IF FD% THEN I1 = 2: REM WILL IGNORE ANY FLOATING DOLLAR SIGN
380 I2 = LM: IF MF% THEN I2 = LM - 1: REM WILL IGNORE ANY TRAILING MINUS
    IN MASK
385 IF NF% AND MF% AND LEFT$ (NUM$,1) = "-" THEN QN = QN + 1: REM IF N
    UMBER'S MINUS SIGN WAS IGNORED BEFORE, PUT IT BACK IN
389 DUM$ = " ": IF QN THEN DUM$ = LEFT$ (NUM$,QN)
390 IF PN THEN NUM$ = DUM$ + RIGHT$ (NUM$,RM): REM DROP DECIMAL POINT
    FROM FROM NUMBER STRING
400 IF NF% AND MF% AND LEFT$ (NUM$,1) = "-" THEN NUM$ = RIGHT$ (NUM$, LEN
    (NUM$) - 1): REM DROP MINUS SIGN IF TRAILING MINUS IN MASK
410 J = LEN (NUM$): FOR I = I2 TO I1 STEP - 1:M$ = MID$ (MASK$,I,1):N$
    = " ": IF J > 0 THEN N$ = MID$ (NUM$,J,1)
420 IF M$ < > " ," THEN 490
430 IF N$ < > "-" THEN 450
440 OUT$ = N$ + 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
    ERT BLANK
470 DF% = 1:OUT$ = "$" + OUT$: GOTO 550
480 OUT$ = M$ + OUT$: GOTO 550
490 IF M$ < > "9" 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$ < > "0" THEN 480: REM GO OUTPUT CURRENT CHARACTER IN MASK
530 IF N$ < > "0" THEN 500: 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.

Number Entered	Value Returned
1.00	100
-2.99	-299
300.89	30089

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.

Value	Value * .01
-299	-2.99
-100	-1
180	1.8

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$ = STR$(CA)
90 GOSUB 15000
100 CA$ = 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 D$ = CHR$ (4): INPUT "NAME OF TEXTFILE IS - ";AA$: PRINT D$"OPEN "
;AA$: PRINT D$;"WRITE ";AA$: LIST 1,63998: PRINT D$;"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 ID ZZ$
15002 REM RESPONSE IS XW$
15005 :
15010 M% = LEN (ZZ$)
15020 XV$ = "$":XX$ = "":XY$ = "."
15030 XZ$ = RIGHT$ (ZZ$,2)
15040 ON M% GOTO 15060,15070,15100,15100,15100,15100,15100,15100,15100
15050 PRINT "ERROR ON INPUT VALUE ": GOTO 15120
15060 XZ$ = "0" + XZ$: GOTO 15110
15070 IF LEFT$ (ZZ$,1) = "-" GOTO 15090
15080 GOTO 15110
15090 XZ$ = "0" + RIGHT$ (XZ$,1):XX$ = "-": GOTO 15110
15100 XX$ = 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 :
15500 REM ARGUMENT IS ZZ$
15505 REM RESPONSE IS XW$
15507 :
15510 IF LEFT$ (ZZ$,1) = "-" GOTO 15560
15520 M% = LEN (ZZ$)
15530 XV$ = "$":XY$ = "."
15540 XZ$ = RIGHT$ (ZZ$,2)
15550 ON M% GOTO 15570,15600,15580,15580,15580,15580,15580,15580,15580
15560 PRINT "ERROR ON INPUT VALUE":XW$ = "": GOTO 15660
15570 XZ$ = "0" + XZ$: GOTO 15600
15580 XX$ = LEFT$ (ZZ$, (M% - 2))
15590 GOTO 15620
15600 XW$ = XV$ + XY$ + XZ$
15610 GOTO 15630
15620 XW$ = XV$ + XX$ + XY$ + XZ$
15630 XT$ = "*****":B = 30 - LEN (XW$)
15640 XS$ = RIGHT$ (XT$,B)
15650 XW$ = XS$ + XW$
15660 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.

Figure 1

```

62980 REM *****
62981 REM *
62982 REM * LOWER CASE INSERT *
62983 REM * JAMES D. CHILDRESS *
62984 REM *
62985 REM * LOWER CASE INSERT *
62986 REM *
62990 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
62999 END
63000 HOME : VTAB (3): PRINT "LOWER CASE INSERTION PROGRAM": PRINT :
63010 LMAX = 62999: PRINT "NUMBER OF FIRST LINE TO BE RE-": INPUT "WRITTE
N ";LT: PRINT
63020 PRINT :M = 256 * PEEK (104) + PEEK (103) + 2
63030 LN = 256 * PEEK (M + 1) + PEEK (M): IF LN > = LMAX OR LN > LT THEN
63320
63040 IF LN < > LT THEN M = 256 * PEEK (M - 1) + PEEK (M - 2) + 2: GOTO
63030
63050 K = 0:LL = 0:UL = 0
63060 FOR J = M + 2 TO M + 255:TST = PEEK (J): IF TST = 0 THEN M = J +
3: GOTO 63030
63070 IF TST = 58 THEN K = 0
63080 IF TST = 186 OR TST = 132 THEN K = 1
63090 IF K = 1 AND LL > 0 AND TST = 34 THEN UL = J - 1: GOTO 63120
63100 IF K = 1 AND LL = 0 AND TST = 34 THEN LL = J + 1
63110 NEXT
63120 BB$ = "":BZ$ = "":BB = 0:ZZ = 0
63130 FOR I = LL TO UL: PRINT CHR$ ( PEEK (I));: NEXT : PRINT "###"
63140 GET AA$:AZ$ = AA$: IF ASC (AA$) = 13 THEN NORMAL : GOTO 63260
63150 IF ASC (AA$) = 1 THEN ZZ = 1: INVERSE :BB = 0: GOTO 63140
63160 IF ASC (AA$) = 3 THEN BB = 1: INVERSE : GOTO 63140
63170 IF ASC (AA$) = 19 THEN BB = 0: NORMAL : GOTO 63140
63180 IF ZZ = 1 OR BB = 1 THEN ZZ = 0: GOTO 63210
63190 IF ASC (AA$) < 65 OR ASC (AA$) > 90 THEN 63210
63200 AA$ = CHR$ ( ASC (AA$) + 32)
63210 BZ$ = BZ$ + AZ$: PRINT AZ$;: IF BB = 0 THEN NORMAL
63220 BB$ = BB$ + AA$: IF ASC (BB$) = 8 AND ASC (AA$) = 8 THEN PRINT "
";
63230 IF LEN (BB$) < = 2 AND ASC (AA$) = 8 THEN BB$ = "":BZ$ = "": GOTO
63140:
63240 IF ASC (AA$) = 8 THEN BB$ = LEFT$ (BB$, LEN (BB$) - 2)
63250 GOTO 63140
63260 IF BB$ = "" THEN 63310
63270 PRINT : FOR I = LL TO UL
63280 DD$ = MID$ (BB$,I - LL + 1,1):MM = ASC (DD$)
63290 POKE I,MM
63300 NEXT
63310 UL = 0:LL = 0: PRINT : GOTO 63110
63320 PRINT : PRINT " NUMBER OF NEXT LINE TO BE REWRITTEN": INPUT "(ENTE
R 0 TO END PROGRAM ";LT
63330 IF LT = 0 THEN END
63340 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 *
62990 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$) = 8 THEN BB$ = "":BZ$ = "": GOTO
63010
63110 IF ASC (AA$) = 8 THEN BB$ = LEFT$ (BB$, LEN (BB$) - 2)
63120 GOTO 63010
63130 PRINT : RETURN
63140 END

```

4

GRAPHICS

Introduction	68
Graphing Rational Functions <i>Ron Carlson</i>	69
A Hi-Res Graph Plotting Subroutine in Integer BASIC for the Apple II <i>Richard Fam</i>	75
How to Do a Shape Table Easily and Correctly <i>John Figueras</i>	78
Define Hi-Res Characters for the Apple II <i>Robert F. Zant</i>	96
Apple II High Resolution Graphics Memory Organization <i>Andrew H. Eliason</i>	99

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:

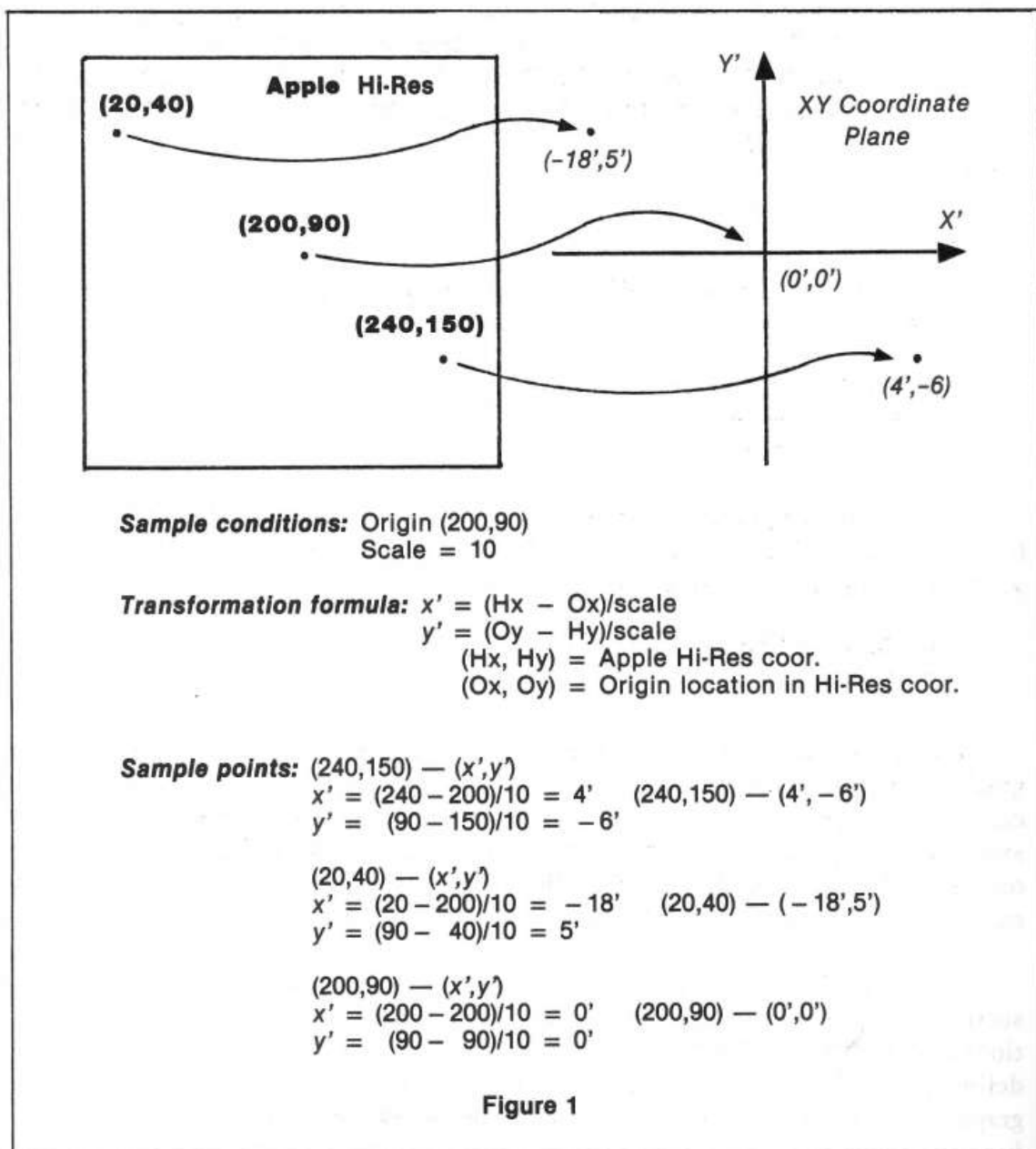
```
60 DEF FN NUM(X) = X*SIN(X)
70 DEF FN DEN(X) = 1
```

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 280×192 .

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.



A mathematical transformation formula will change the HGR coordinates to x and y or x and y to HGR coordinates.

$$(\text{real } x \text{ coord.}) = (\text{HGR } x \text{ coord.}) - (x \text{ coord. of origin})$$

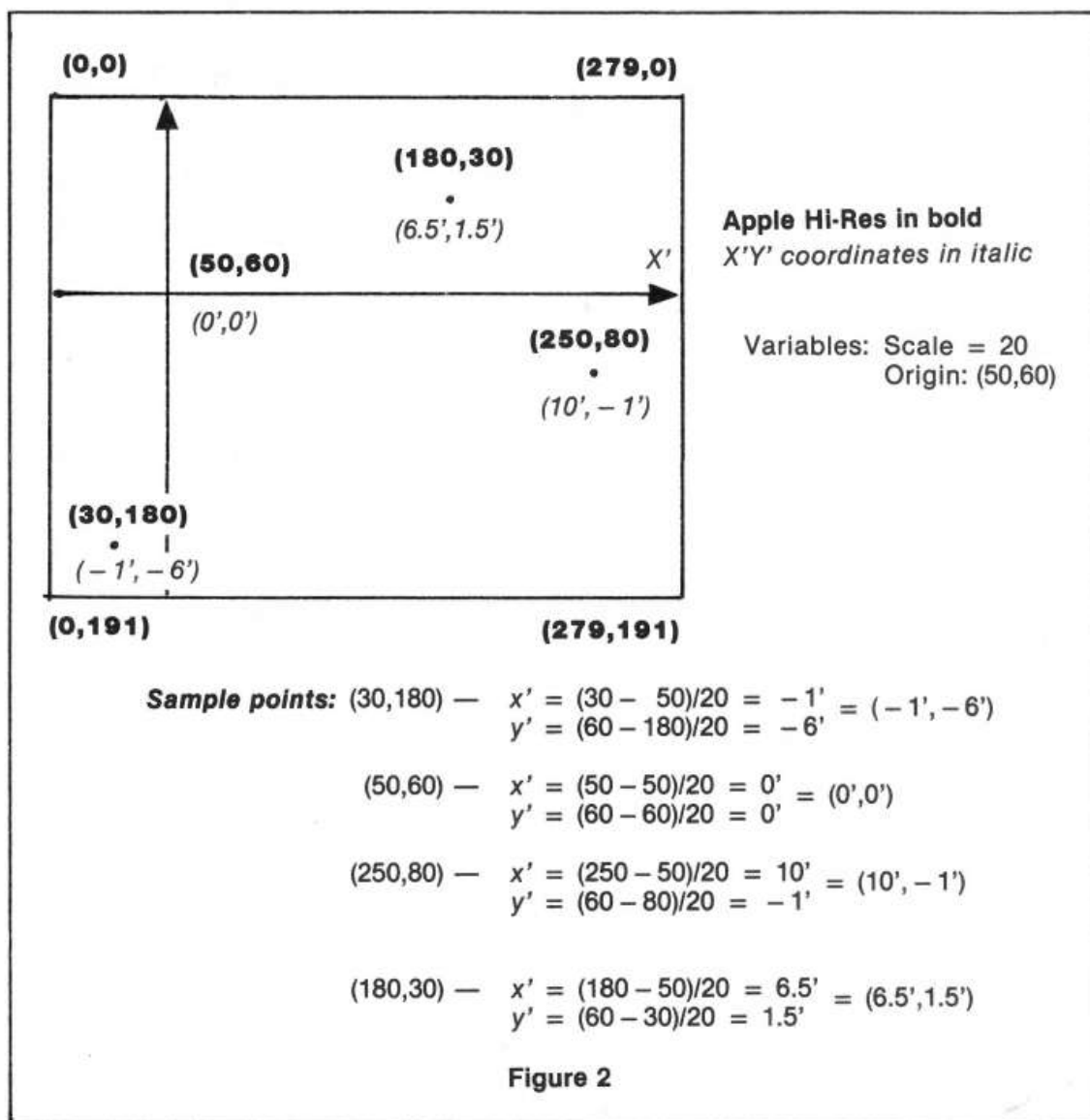
$$X = H - A$$

$$(\text{real } y \text{ coord.}) = (y \text{ coord. of origin}) - (\text{HGR } y \text{ coord.}) \text{ and } Y = B - V.$$

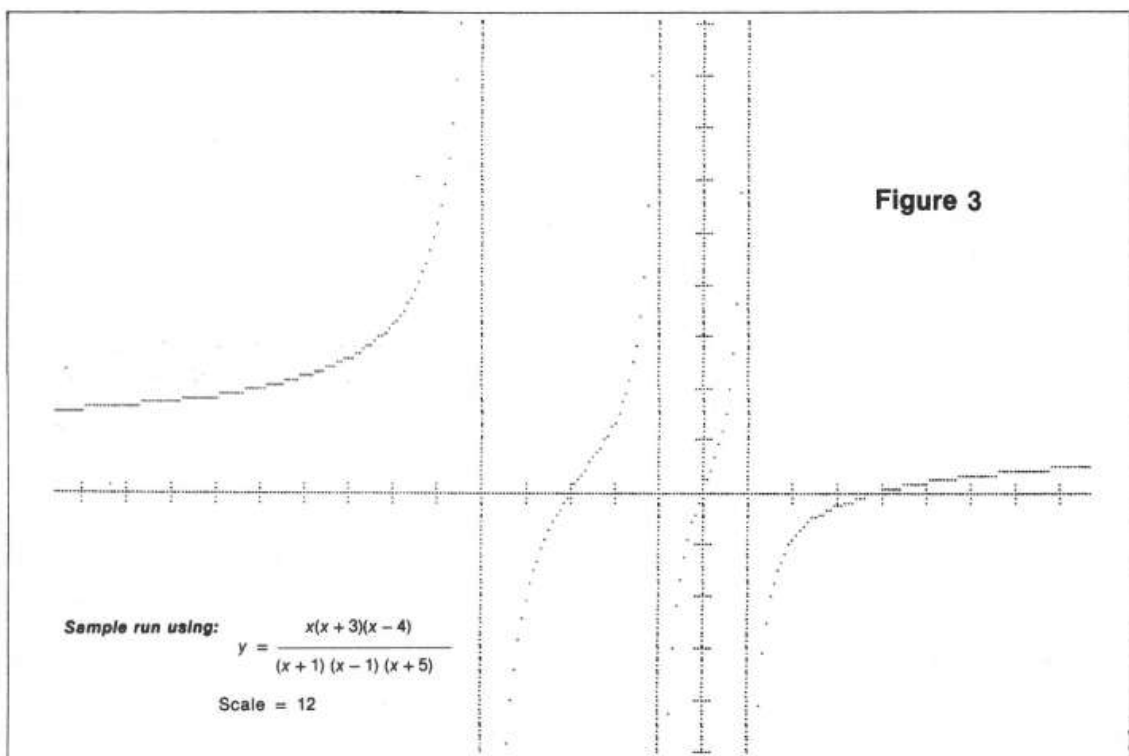
When the scale factor, S , is considered, then the transformation formulas look like:

$$X = (H - A)/S$$

$$V = B - Y * S$$



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.



```

10 REM *****
11 REM *
12 REM * GRAPHING RATIONAL *
14 REM * FUNCTIONS *
16 REM * RON CARLSON *
18 REM *
20 REM * FUNCTION GRAPH *
21 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 *****
50 :
60 DEF FN NUM(X) = X
70 DEF FN DEN(X) = (X + 2)
75 :
80 REM DEF FN DEN(X)=1>> IF YOU HAVE A NON-RATIONAL GRAPH
85 :
90 HOME : INPUT "THERE ARE 280 HORIZONTAL DOTS. HOW MANY DOTS/UNIT DO YOU WANT?";S
100 VTAB 21: PRINT "INDICATE THE INTENDED LOCATION OF THE ORIGIN BY MOVING THE CURSOR WITH THE L R U D KEYS. F=FINISHED"
110 REM THIS ALLOWS THE USER TO SELECT WHICH AREA OF THE GRAPH TO VIEW
120 GOSUB 620: REM TO POSITION THE ORIGIN
130 REM S WILL BE THE SCALE
140 REM DETAIL INCREASES AS S INCREASES
150 VTAB 21: PRINT "AFTER THE BOTTOM HALF OF THE GRAPH IS FINISHED, HIT ANY KEY"
160 PRINT "THERE IS A HASH MARK (/) ON THE AXIS FOR EACH UNIT"
170 HGR : HCOLOR= 7
180 REM AXIS, WITH THE REAL AXIS AT (A,B)
190 HPLOT 0,B TO 279,B: HPLOT A,0 TO A,191
200 REM HASH MARKS EVERY UNIT ON THE AXIS
210 FOR H = A TO 279 STEP S: HPLOT H,B - 2 TO H,B + 2: NEXT
220 FOR H = A TO 0 STEP - S: HPLOT H,B - 2 TO H,B + 2: NEXT
230 FOR V = B TO 191 STEP S: HPLOT A - 2,V TO A + 2,V: NEXT
240 FOR V = B TO 0 STEP - S: HPLOT A - 2,V TO A + 2,V: NEXT
250 REM ACTUAL GRAPHING
260 FOR H = 0 TO 279
270 REM TRANSFER THE HGR COOR TO THE REAL VALUE
280 X = (H - A) / S: D = FN DEN(X)
290 REM DRAW THE VERTICAL ASYMPTOTES IF NECESSARY
300 IF D = 0 THEN HCOLOR= 3: HPLOT H,0 TO H,191: HCOLOR= 7: GOTO 350
310 Y = FN NUM(X) / D: V = B - Y * S
320 REM TRANSFORM THE REAL Y VALUE TO HGR AND SEE IF IT STILL ON THE SCREEN
330 IF V > 191 OR V < 0 THEN 350
340 HPLT H,V
350 NEXT H
370 REM THIS POKE WILL DISPLAY THE BOTTOM QUARTER OF THE GRAPH
380 POKE - 16302,0: GET A$
390 TEXT : HOME
400 INPUT "DO YOU WANT TO SHIFT THE ORIGIN AND CHANGE SCALE?";A$
410 IF A$ = "Y" OR A$ = "YES" THEN 90
420 GOTO 830
440 HOME : PRINT " DIRECTIONS FOR RATIONAL FUNCTIONS"
450 PRINT " YOU MUST DEFINE YOUR FUNCTION IN TERMS OF NUMERATOR AND DENOMINATOR"
460 PRINT " FOR EXAMPLE IF YOU WISH TO GRAPH THE FOLLOWING:"
470 PRINT " (X-1)(X+2)"
480 PRINT " Y = -----"
490 PRINT " X(X-7)"
500 PRINT : PRINT " YOU WOULD TYPE THE FOLLOWING"
510 PRINT "60 DEF FNNUM(X)=(X-1)*(X+2)"
520 PRINT "70 DEF FNDEN(X)=X*(X-7)"
530 PRINT "RUN"
540 PRINT : FLASH : PRINT "REMEMBER :"
550 PRINT "60 DEF FNNUM(X)=";: NORMAL : PRINT "LEGAL BASIC EXPRESSION"
560 FLASH : PRINT "70 DEF FNDEN(X)=";: NORMAL : PRINT "LEGAL BASIC EXPRESSION"
570 PRINT "RUN"
580 GOTO 830

```

```
600 REM POSITIONING THE ORIGIN OF THE SCREEN (40,40)
610 REM USING L R U D AND F
620 GR : COLOR= 3: PLOT 20,20:A = 20:B = 20
630 GET A$
640 A1 = A:B1 = B
650 IF A$ = "U" THEN B = B - 1: GOTO 710
660 IF A$ = "D" THEN B = B + 1: GOTO 710
670 IF A$ = "L" THEN A = A - 1: GOTO 710
680 IF A$ = "R" THEN A = A + 1: GOTO 710
690 IF A$ = "F" THEN 800
700 REM KEEP ON THE LO RES SCREEN
710 IF B < 1 THEN B = 1
720 IF B > 39 THEN B = 39
730 IF A < 1 THEN A = 1
740 IF A > 39 THEN A = 39
750 REM BLANK OLD POSITION
760 COLOR= 0: PLOT A1,B1: COLOR= 3
770 REM PLOT NEW POSITION
780 PLOT A,B
790 GOTO 630
800 A = 7 * A:B = B * 192 / 40
810 REM CHANGE SCALE TO REFLECT HGR (280 BY 192)
820 TEXT : HOME : RETURN
830 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 $0 \leq \text{GPH}(X) \leq 191$ 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
```

```

10 REM *****
11 REM *
12 REM * HI-RES GRAPH PLOTTING *
14 REM * RICHARD FAM *
15 REM *
16 REM * GRAPH-PLOT *
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))
9231 NEXT MK: GOTO 9242
9240 POKE LH/7+16336,255
9242 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+B*2 ^
(LH 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.

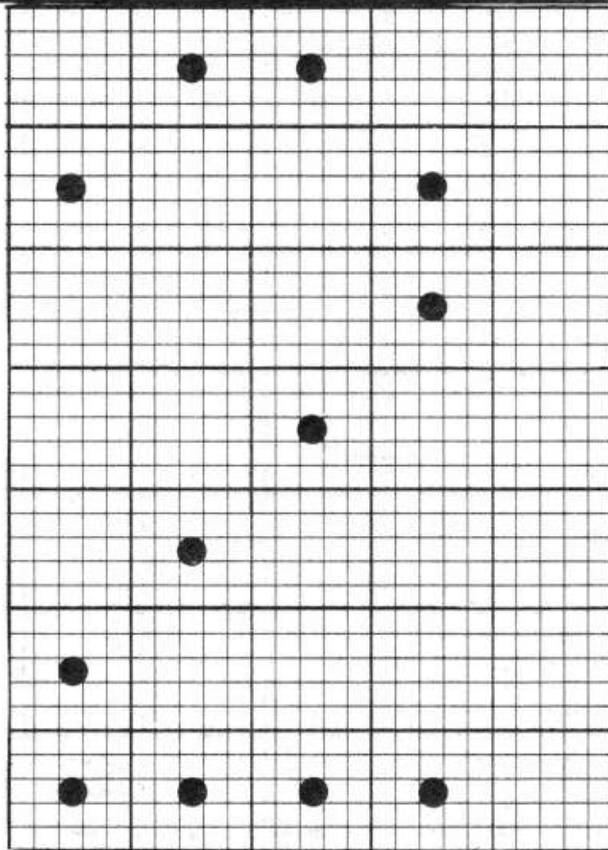
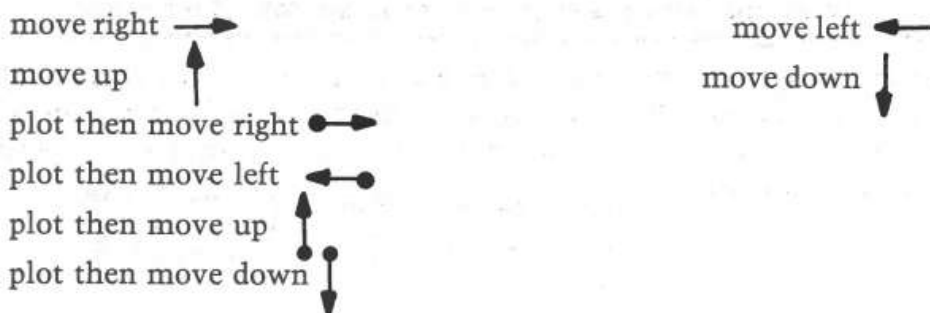


Figure 1: Shape to be coded



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 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 *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.

←• ←	00 111 011 0011 1011	3B
←• ←•	00 111 111 0011 1111	3F
↑• ↑	00 100 100 0010 0100	24
↑• →	00 100 001 0010 0001	21
↑• →	00 100 001 0010 0001	21
↑• →	00 100 001 0010 0001	21
←• ↑	00 011 100 0001 1100	1C
←• ←•	00 111 111 0011 1111	37
↓• ↓	00 110 010 0011 0010	32

(a) (b) (c) (d)

Figure 3: Translation of shape vectors to Hexidecimal Code.

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 15×15 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 × 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 GET KEY\$ will load the character accessed by the next keystroke into the variable KEY\$. We may examine KEY\$ 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 KSVE\$) is examined. If 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 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

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:

$$\text{BYTE} = \text{BYTE} + 8 * V_2 \quad \begin{array}{cc} V_2 & V_1 \\ 00\text{YYYYXX} \end{array}$$

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 $64 (= 2^6)$:

$$\text{BYTE} = \text{BYTE} + 64 * V_3 \quad \begin{array}{ccc} V_3 & V_2 & V_1 \\ \text{ZZYYYYXX} \end{array}$$









Plotting Vectors	3-Bit Codes	Decimal Equivalents
	000	0
	001	1
	010	2
	011	3
	100	4
	101	5
	110	6
	111	7

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 left-most 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.

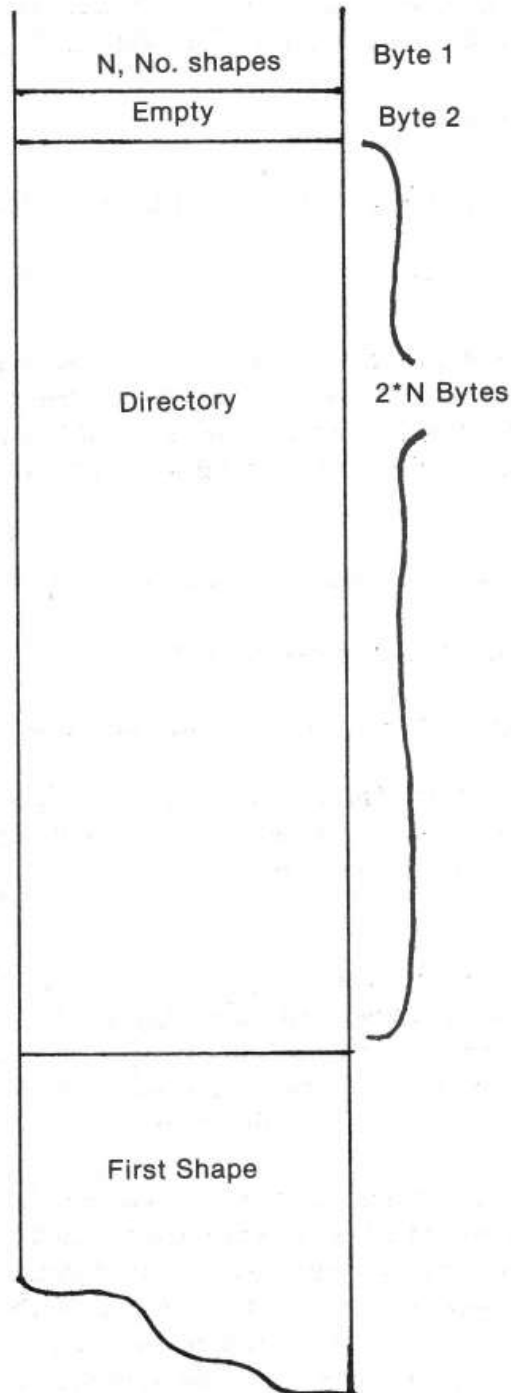


Fig. 5: Memory Map for Shape Table

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 15×15 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 15×15 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 × 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 15×15 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.

```

10 REM *****
11 REM * HOW TO DO A SHAPE TABLE *
12 REM *       JOHN FIGUERAS       *
14 REM *                               *
15 REM *       SHAPE1              *
16 REM *                               *
18 REM *   COPYRIGHT (C) 1981     *
20 REM *   MICRO INK, INC.       *
22 REM *   CHELMSFORD, MA 01824  *
24 REM *   ALL RIGHTS RESERVED   *
25 REM *                               *
26 REM *****
28 REM
29 REM
30 INPUT "NAME OF SHAPE TABLE ";NAMES$
35 INPUT "STARTING ADDRESS, DECIMAL ";ADDR
40 INPUT "NO. OF SHAPES TO BE STORED ";N
50 REM ZERO DIRECTORY
60 FOR I = 0 TO 2 * N + 1
70 POKE ADDR + I,0: NEXT
80 REM CALCULATE INDEX TO CURSOR
90 N = 2 * N + 2
100 REM PUT CURSOR INDEX INTO DIRECTORY
110 POKE ADDR + 2,N - 256 * INT (N / 256)
120 POKE ADDR + 3, INT (N / 256)
130 REM CALC INITIAL ADDRESS TO CURSOR
140 INIT = ADDR + N
150 REM ENTER CURSOR SHAPE VECTORS
160 DATA 62,36,45,54,04,00
170 FOR I = 0 TO 5
180 READ A: POKE INIT + I,A: NEXT
190 REM GET INDEX TO NEXT SHAPE
200 N = N + 6
210 REM STORE NEW INDEX IN DIRECTORY
220 POKE ADDR + 4,N - 256 * INT (N / 256)
230 POKE ADDR + 5, INT (N / 256)
240 REM UPDATE SHAPE COUNTER
250 POKE ADDR,1
260 REM STORE INITIALIZED FILE ON DISK
270 D$ = CHR$(4)
280 PRINT D$;"NOMON C,I,0"
290 PRINT D$;"BSAVE" + NAMES$ + ",A" + STR$(ADDR) + ",L" + STR$(N) +
    ",V0,D2"
300 END

```

```

10 REM *****
12 REM *
14 REM * HOW TO DO A SHAPE TABLE *
16 REM * JOHN FIGUERAS *
18 REM *
20 REM * SHAPE2 *
21 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 PRINT TAB( 6);"*****CREATE A SHAPE TABLE*****"
32 PRINT
35 PRINT TAB( 5);"J. FIGUERAS, ROCHESTER, N.Y.": PRINT
40 PRINT TAB( 16)"9/12/79": PRINT
50 PRINT TAB( 17)"*****": PRINT
60 REM INPUT TABLE NAME AND LOCATION
70 INPUT "SHAPE TABLE NAME ";NAME$
80 INPUT "STARTING ADDRESS, DECIMAL ";ASVE
90 REM DISK ACCESSES USE DISK D2
100 D$ = CHR$( 4): PRINT D$;"NOMON C,I,O"
110 PRINT D$;"BLOAD " + NAME$ + ",A" + STR$( ASVE) + ",V0,D2"
120 REM GET CAPACITY MAX OF FILE
130 MAX = PEEK (ASVE + 2) + 256 * PEEK (ASVE + 3)
140 MAX = (MAX - 2) / 2
150 REM GET NO. OF SHAPES IN TABLE
160 N = PEEK (ASVE)
170 REM GET FILE LENGTH
180 INDEX = PEEK (ASVE + 2 * N + 2) + 256 * PEEK (ASVE + 2 * N + 3)
190 REM COMPUTE ADDRESS OF NEXT FREE BYTE
200 ADDR = ASVE + INDEX
210 REM SEE IF FILE IS FULL
220 IF MAX > N THEN 260
230 PRINT "SHAPE TABLE FULL. NEXT FREE BYTE AT ";ADDR
240 GOTO 1370
250 REM SET UP ADDRESS POINTERS TO TABLE
260 POKE 232,ASVE - 256 * INT (ASVE / 256): POKE 233, INT (ASVE / 256)
270 REM UPDATE SHAPE COUNTER
280 N = N + 1: POKE ASVE,N
290 REM DISPLAY PLOTTING GRID. INITIALIZE COUNTER, CYCLE
300 HCOLOR= 3: SCALE= 1: ROT= 0:CYCYLE = 0
310 HGR
320 FOR X = 0 TO 150 STEP 10: HPLOT X,0 TO X,150: NEXT
330 FOR Y = 0 TO 150 STEP 10: HPLOT 0, Y TO 150,Y: NEXT
340 REM CLEAR TEXT AND GET INITIAL PLOT COORDS
350 PRINT : PRINT : PRINT : PRINT
360 PRINT "ENTER STARTING COORDS"
370 INPUT "X ";X:X = 10 * X - 5
380 INPUT "Y ";Y:Y = 10 * Y - 5
390 DRAW 1 AT X,Y:XS = X:YS = Y
400 REM CLEAR TEXT. DISPLAY INSTRUCTIONS
410 PRINT : PRINT : PRINT : PRINT
420 PRINT "MOVE PLOT CURSOR WITH KEYS"
430 PRINT "L-LEFT R-RIGHT U-UP D-DOWN"
440 PRINT "P TO PLOT Q TO QUIT"
450 REM INITIALIZE KEY$. PLOT CURSOR
460 KEY$ = "":KSVE$ = "": GOTO 570
470 REM FLAG RE-ENABLES CURSOR AFTER A PLOT DISABLE
480 IF FLAG = 1 THEN 520
490 REM ERASE CURSOR IN PREVIOUS SQUARE
500 XDRAW 1 AT X1,Y1
510 REM PLOT CURSOR AT NEW X,Y. SAVE X,Y
520 X1 = X:Y1 = Y:FLAG = 0
530 XDRAW 1 AT X,Y

```

```

540 REM SAVE LAST TWO KEYSTROKES.  KI$ IS NEEDED FOR ERASE R OUTLINE
550 KI$ = KSVE$:KSVE$ = KEY$
560 REM GET NEW KEYSTROKE
570 GET KEY$
580 REM GO TO SIEVE TO GET 3-BIT PLOT VECTOR FROM KEY$ AND KSVE$
590 IF KEY$ < > "U" THEN 610
600 SYMBOL = 0:Y = Y - 10: GOTO 760
610 IF KEY$ < > "R" THEN 630
620 SYMBOL = 1:X = X + 10: GOTO 760
630 IF KEY$ < > "D" THEN 650
640 SYMBOL = 2:Y = Y + 10: GOTO 760
650 IF KEY$ < > "L" THEN 670
660 SYMBOL = 3:X = X - 10: GOTO 760
670 IF KEY$ < > "P" THEN 690
680 FLAG = 1: GOSUB 1000: GOTO 530
690 IF KEY$ = "Q" THEN 1080
700 REM NEXT STATEMENT PROTECTS FROM KEYING ERROR
710 IF KEY$ < > "E" THEN 570
720 HCOLOR= 0:FLAG = 0: GOSUB 1000
730 REM SET UP PRE-PLOT STATUS
740 KSVE$ = KI$: HCOLOR= 3: GOTO 500
750 REM ADJUST 3-BIT VECTOR FOR PLOT
760 IF KSVE$ = "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
TE INTO TABLE AND STORE 3-BIT VECTOR IN NEXT BYTE.
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 KSVE$ < > "P" 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 ";KI$
1190 IF KI$ = "Y" THEN 1220
1200 N = N - 1: GOTO 180
1210 REM GET INDEX FOR NEXT FREE BYTE
1220 N = N + 1:ADDR = ADDR - ASVE
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

```

```

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 D$;"BSAVE" + NAME$ + ",A" + STR$ (ASVE) + ",L" + STR$ (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 "HIT ANY KEY FOR EACH PAGE OF TABLE"
40 D$ = CHR$ (4): PRINT D$;"NOMON C,I,O"
50 PRINT D$;"BLOAD 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 / 256):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= 3: 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 KEY$
220 REM SCLEAR SCREEN AND CREATE GRID
230 REM GRID WILL HOLD 36 SHAPES
240 CALL 62450
250 HPLOT 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 HPLOT L,J
290 NEXT J: NEXT L
300 FOR L = 30 TO 180 STEP 30
310 FOR J = 0 TO 269 STEP 45
320 HPLOT 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)

```

```
390 C2 = I - 100 * C1
400 C2 = INT (C2 / 10)
410 C3 = I - 10 * INT (I / 10)
420 POKE 232,NLO: POKE 233,NHI
430 C1 = C1 + 2:C2 = C2 + 2:C3 = C3 + 2
440 IF C1 = 2 THEN 460
450 DRAW C1 AT 45 * COL + 5,30 * ROW + 7
460 IF C2 = 2 AND C1 = 2 THEN 480
470 DRAW C2 AT 45 * COL + 10,30 * ROW + 7
480 DRAW C3 AT 45 * COL + 15,30 * ROW + 7
490 REM NOW GET SHAPES
500 POKE 232,ALO: POKE 233,AHI
510 DRAW I AT 45 * COL + 30,30 * ROW + 15
520 NEXT I
530 GET KEYS$
540 TEXT
550 END
```

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

```
02,02,02,02,02,42,7E,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 × 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 × 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).

BYTE	8192						8193							
SCREEN POSITION	0	1	2	3	4	5	6	7	8	9	10	11	12	13
BIT	0	1	2	3	4	5	6	0	1	2	3	4	5	6
	V	G	V	G	V	G	V	G	V	G	V	G	V	G
(Bit 7 not used)							7							7

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 \times 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 $C = \text{SCRN}(X,Y)$ 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.
PAGE 1 = 8192 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 = $LV \text{ MOD } 8 * 1024 + (LV/8) \text{ MOD } 8 * 128 + (LV/64) * 40 +$
FB
BY = $LH/7 + BV$
BI = $2(LH \text{ MOD } 7)$

To Plot a Point (Without Hi-Res Plot Routine):

```
LH  = X MOD 280 : LV = Y MOD 192 (OR)
      LV = 192-Y MOD 192

FB  = 8192
BV  = LV MOD 8 * 1024 + (LV/8) MOD 8 * 128 + (LV/64) * 40 + FB
BY  = LH/7 + BV
BI  = 2 (LH MOD 7)
WO  = PEEK (BY)
IF (WO/BI) MOD 2 THEN RETURN
POKE BY, BI + WO
RETURN
```

To Remove a Point, Substitute:

```
IF (WO/BI) MOD 2 = 0 THEN RETURN
POKE BY, WO-BI
```

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.

```
10 REM *****
12 REM *
14 REM * HI-RES GRAPHICS MEMORY *
16 REM * ORGANIZATION *
18 REM * ANDREW H. ELIASON *
20 REM *
22 REM * GRAPHICS-ORG *
24 REM *
25 REM * COPYRIGHT (C) 1981 *
26 REM * MICRO INK, INC. *
27 REM * CHELMSFORD, MA 01824 *
28 REM * ALL RIGHTS RESERVED *
29 REM *
30 REM *****
96 REM
100 REM SET HIMEM:8191
110 REM HIRES GRAPICS LEARNING AID
120 POKE -16304,0: REM SET GRAPHICS MODE
130 POKE -16297,0: REM SET HIRES 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
```

5

EDUCATION

Introduction	104
Apple Pi <i>Robert J. Bishop</i>	105
Sorting Revealed <i>Richard C. Vile, Jr.</i>	109
Solar System Simulation with or without an Apple II <i>David A. Partyka</i>	134
Programming with Pascal <i>John P. Mulligan</i>	143

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:

$$\pi = \sum_{K=1}^{\infty} \frac{16(-1)^{K+1}}{(2K-1)5^{2K-1}} - \sum_{K=1}^{\infty} \frac{4(-1)^{K+1}}{(2K-1)239^{2K-1}}$$

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 1000, 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 *      APPLE PI      *
3 REM *    ROBERT J. BISHOP  *
4 REM *
5 REM *      APPLE PI      *
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 PEEK (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=DIGIT+ PEEK (PLACE)
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
3990 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
9986280348253421170679821480865132823066
4709384460955058223172535940812848111745
0284102701938521105559644622948954930381
9644288109756659334461284756482337867831
6527120190914564856692346034861045432664
8213393607260249141273724587006606315588
1748815209209628292540917153643678925903
6001133053054882046652138414695194151160
9433057270365759591953092186117381932611
7931051185480744623799627495673518857527
2489122793818301194912983367336244065664
3086021394946395224737190702179860943702
7705392171762931767523846748184676694051
3200056812714526356082778577134275778960
9173637178721468440901224953430146549585
3710507922796892589235420199561121290219
6086403441815981362977477130996051870721
1349999998372978049951059731732816096318
5950244594553469083026425223082533446850
3526193118817101000313783875288658753320
8381420617177669147303598253490428755468
7311595628638823537875937519577818577805
3217122680661300192787661119590921642019
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:

$\log_2 x$ = base 2 logarithm of x
 $[x]$ = floor of x

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

$[\log_2 N]$

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:

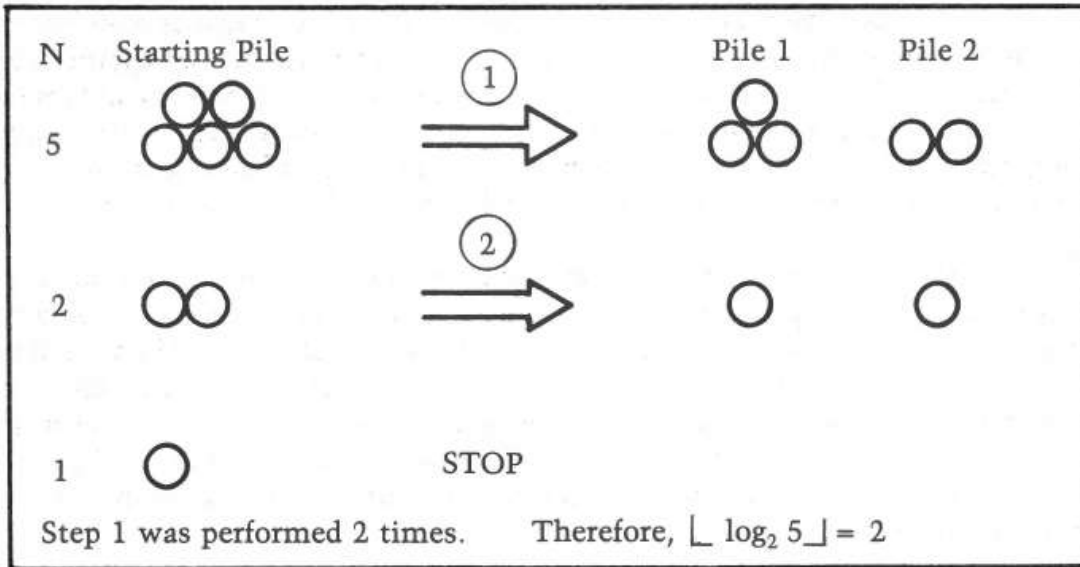
1. Subdivide the pile into two piles which are as nearly equal in size as possible.
2. Take the smaller of the two piles from step 1. If it consists of one coconut, then stop. Otherwise, repeat from step 1.

Now how many times did you do step 1? The answer is the value of $[\log_2 N]$! 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.

Figure 1



- 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.
- 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:

```
TEMP = A(I)
A(I) = A(J)
A(J) = 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:

$$N * (\lceil \log_2 N \rceil)$$

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 , $\lceil \log_2 N \rceil$, and $N * \lceil \log_2 N \rceil$. 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 $\lceil \log_2 N \rceil$). 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¢ 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.

N	N^2	$\log N$	$N \log N$
64	4096	6	384
128	16,384	7	896
256	65,536	8	2,048
512	262,144	9	4,608
1,024	1,048,576	10	10,240
2,048	4,194,304	11	22,528
4,096	16,777,216	12	49,152
8,192	67,108,864	13	106,496
16,384	268,435,456	14	229,376
32,768	1,073,741,824	15	491,520
65,536	4,294,967,296	16	1,048,576
131,072	17,179,869,184	17	2,228,224
262,144	68,719,476,736	18	4,718,592
524,288	274,877,906,944	19	9,961,472
1,048,576	1,099,511,626,776	20	20,971,520

Table 1

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 $A(1), A(2), \dots, A(N)$, 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 $A(J)$, then what other array entries will it be swapped with?"

```

                                BASIC
100      IF A(I) <= A(I+1) THEN 140
110          TEMP = A(I)
120          A(I) = A(I+1)
130          A(I+1) = TEMP
140      ...

                                Pascal
if A[I] > A[I+1] then
begin

    Temp := A[I];
    A[I] := A[I+1];
    A[I+1] := Temp;

end;

```

Figure 2: The "Classic Interchange"

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 $A(1)$, $A(2)$, ... $A(N-2)$, 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.

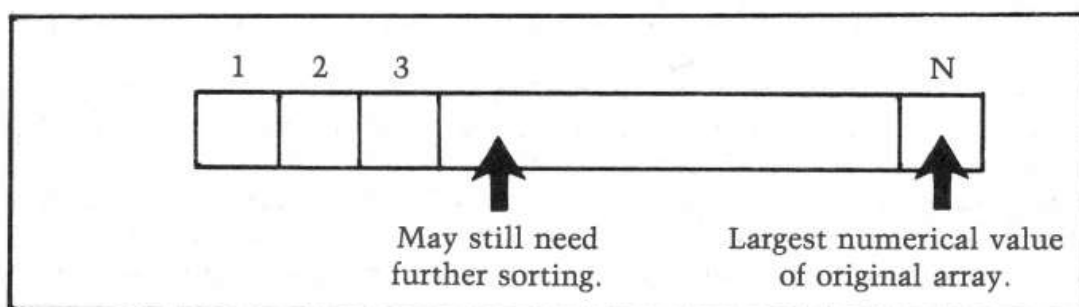


Figure 3: Array after sweep of Bubble sort

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
    if A[J] > A[J+1] then
      begin

          Temp := A[J];
          A[J] := A[J+1];
          A[J+1] := 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 $N-I$. 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:

$$(N-1) + (N-2) + \dots + 3 + 2 + 1$$

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 $1/2 (N^2 - N)$. 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 $N \log N$ 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 $A(P)$, $A(P+1)$, ..., $A(Q)$ 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 $A(P)$, $A(P+1)$, ..., $A(Q)$ 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

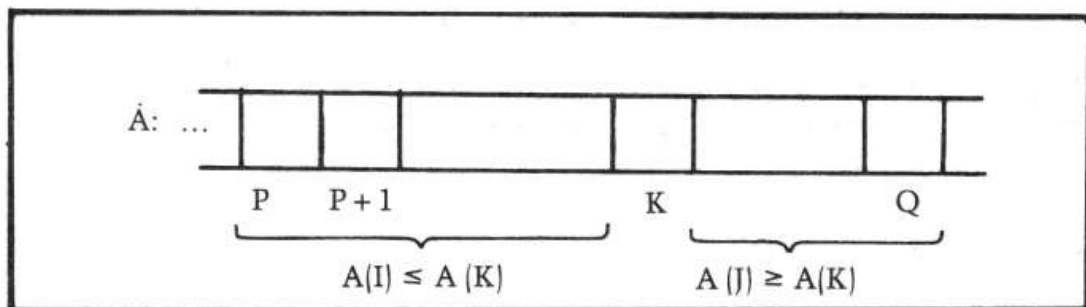


Figure 5: Partitioning $A(P)$ - $A(Q)$

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.

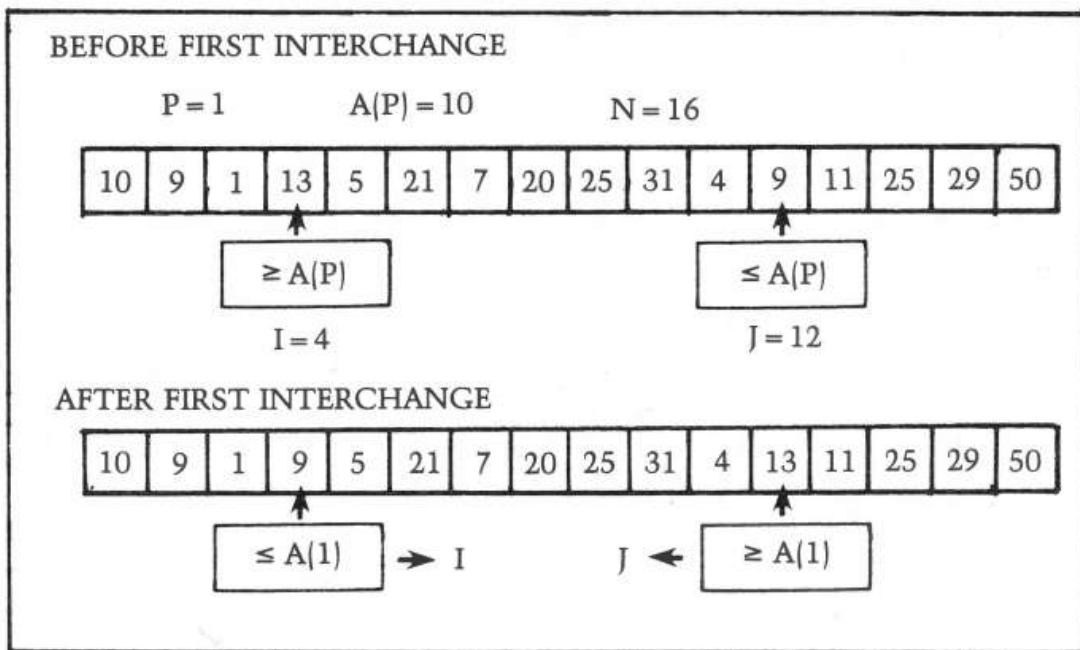


Figure 6: Partition of A: P = 1

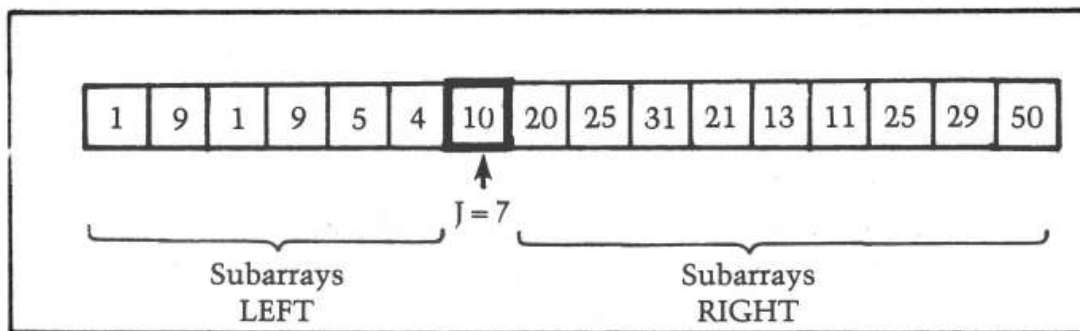


Figure 7: Partition step complete A(7) in correct position.

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 $A(1), A(2), \dots, A(N)$ 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 $I, I = 2, 3, \dots, N$. The repeat statement:

```
repeat
  J := J - 1
until A[J] ≤ 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[Lower] := A[J];
A[J] := 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 `Lower` and `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 *voilà!*, 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);    {Partition A between      }
                                   {A(Lower) and A(Upper)   }
    Sort(A,Lower,J-1);             {Sort the "left" subarray }
    Sort(A,J+1,Upper);             {Sort the "right" subarray }
  end {of Procedure Sort};

```

Figure 9

A								Call
---								-----
10	9	1	13	5	21	7	20	Partition(A,1,8);
10	9	1	7	5	21	13	20	
5	9	1	7	10	21	13	20	
5	9	1	7	10	21	13	20	Partition(A,1,4);
5	1	9	7	10	21	13	20	
1	5	9	7	10	21	13	20	
1	5	9	7	10	21	13	20	Partition(A,1,1);
1	5	9	7	10	21	13	20	
1	5	9	7	10	21	13	20	Partition(A,3,4);
1	5	7	9	10	21	13	20	
1	5	7	9	10	21	13	20	Partition(A,3,3);
1	5	7	9	10	21	13	20	Partition(A,5,4);
1	5	7	9	10	21	13	20	Partition(A,6,8);
1	5	7	9	10	20	13	21	
1	5	7	9	10	20	13	21	Partition(A,6,7);
1	5	7	9	10	13	20	21	
1	5	7	9	10	13	20	21	Partition(A,6,6);
1	5	7	9	10	13	20	21	Partition(A,8,7);
1	5	7	9	10	13	20	21	Partition(A,9,8);

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.

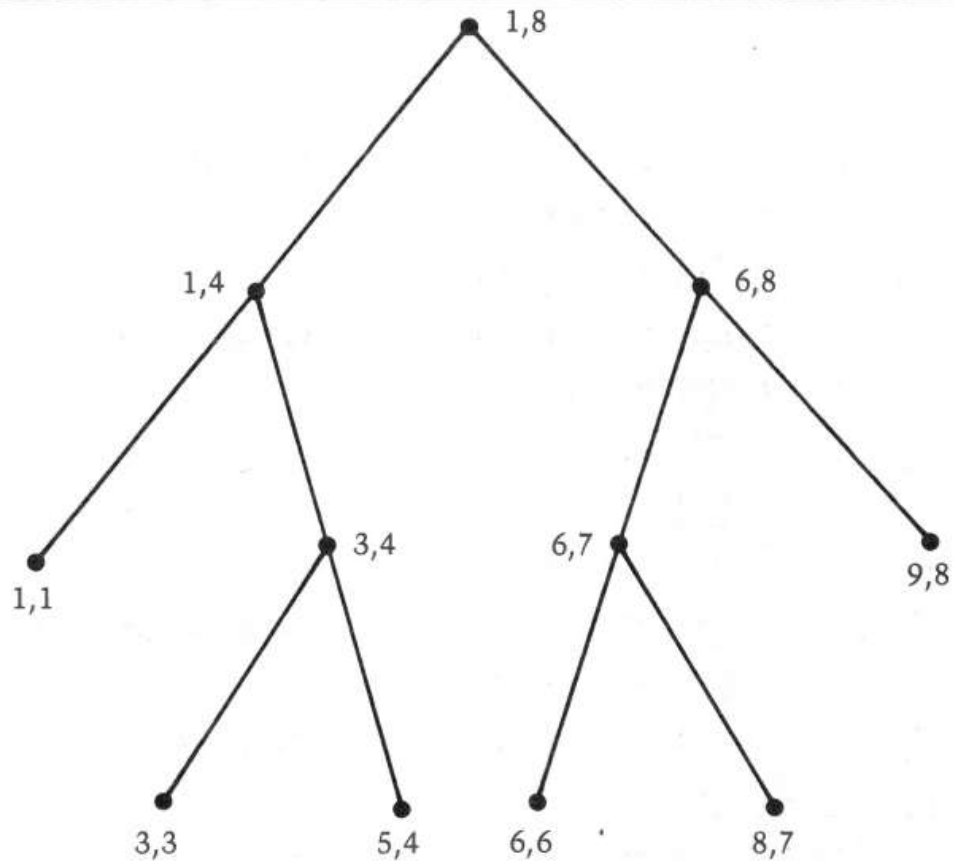


Figure 11: Call tree for figure 10. Each node is labelled with the values of Lower, Upper for the corresponding call. The levels of the tree correspond to the depth of the recursion.

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(I) = RND * 100: NEXT I
90 FOR I = 0 to N
100 L = RND * (N + 1): IF A(L) = 0 THEN 100
105 A(L) = I: 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.

	0	1	2	3	4	5	6	7	8	9
0!	12	72	14	68	54	23	32	3	56	24
1!	44	26	41	0	87	67	8	81	39	39
2!	3	26	60	64	35	20	39	78	65	26
3!	16	17	99	69	81	88	65	32	5	68
4!	37	44	32	89	65	37	20	38	84	77
5!										
6!										
7!										
8!										
9!										

SHELL SORT
SPAN = 10

J = 10
A(J) = 44

Figure 13: Just before the start of the shell sort. Fifty elements are being sorted.

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 *****
12 REM *
14 REM *   SORTING REVEALED   *
16 REM * RICHARD C. VILE, JR. *
18 REM *
20 REM *           BUBBLE     *
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(100)
42 KBD=-16384:CLR=-16368:TITLE=500:INTRO=1000
44 DISPLAY=600:WAIT=800:COMPARE=900:INTERCHANGE=950
46 MUSIC=10473:TIME=766:TIMBRE=765:PITCH=767
48 GOSUB INTRO
50 GOSUB TITLE
90 FOR R=0 TO 100:A(R)=32767: NEXT R
100 FOR I=0 TO N
105 A(I)= RND (100):X=I: GOSUB DISPLAY
108 IF N=0 THEN 150
110 NEXT I
150 FOR I=1 TO NUM-1
152 FLAG=0
155 FOR J=0 TO N-I
158 FOR T=0 TO PDL (1): NEXT T
159 GOSUB COMPARE
160 IF A(J)<=A(J+1) THEN 200
163 X=100: POKE 50,127:A(100)=A(J): GOSUB DISPLAY
165 KEEP=A(J): GOSUB INTERCHANGE:X=X
170 POKE 50,63
173 A(J)=A(J+1): GOSUB DISPLAY: GOSUB INTERCHANGE: POKE 50,255
175 GOSUB DISPLAY:X=X+1: POKE 50,63
180 A(J+1)=KEEP: GOSUB DISPLAY: GOSUB INTERCHANGE: POKE 50,255
185 GOSUB DISPLAY
190 FLAG=1
195 KEY= PEEK (KBD): IF KEY<128 THEN 200
196 POKE CLR,0: GOSUB WAIT
200 NEXT J
202 IF FLAG=0 THEN 208
205 NEXT I
208 VTAB 24: TAB 21: 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 1: PRINT "TEMP=";: TAB 20
528 PRINT "BUBBLE 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("Q") THEN 810
805 TEXT : CALL -936: END
810 VTAB 2*ROW+3: TAB 6+3*COL: PRINT ">";
815 KEY= PEEK (KBD): IF KEY<128 THEN 810
817 VTAB 2*ROW+3: TAB 6+3*COL: PRINT " ";
820 POKE CLR,0: RETURN
900 REM *** TO REMOVE SOUND, 901 IS ADDED ***
901 RETURN
902 POKE PITCH,10: 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

```

```

0 I=J=Y=N
10 REM *****
12 REM *
14 REM * SORTING REVEALED *
16 REM * RICHARD C. VILE, JR. *
18 REM *
20 REM * INSERT *
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:COMPARE=900:INTERCHANGE=950
45 MUSIC=-10473:TIME=766:TIMBRE=765:PITCH=767
46 DELAY=975:ERASE=650
48 GOSUB INTRO
50 GOSUB TITLE
90 FOR R=0 TO 99:A(R)=32767: NEXT R
100 FOR I=0 TO N
105 A(I)= RND (100):X=I: GOSUB DISPLAY
108 IF N=0 THEN 150
110 NEXT I
150 FOR I=1 TO N
151 IF I>N THEN 206:Y=A(I)
152 VTAB 23: TAB 32: PRINT "I=";: IF I<10 THEN PRINT " ";: PRINT I;
153 VTAB 24: TAB 32: PRINT "Y=";: IF Y<10 THEN PRINT " ";: POKE 50,127:
PRINT Y;: POKE 50,255
154 GOSUB INTERCHANGE
155 FOR J=I-1 TO 0 STEP -1
156 GOSUB DELAY:KEY= PEEK (KBD): IF KEY<128 THEN 159
158 POKE CLR,0: GOSUB WAIT
159 GOSUB COMPARE
160 IF Y>A(J) THEN 202
163 A(J+1)=A(J)
166 GOSUB INTERCHANGE
168 POKE 50,63
175 X=J: GOSUB DISPLAY: GOSUB DELAY
178 X=J+1: GOSUB DISPLAY: GOSUB DELAY
180 POKE 50,255: GOSUB DISPLAY: GOSUB DELAY
185 X=J: GOSUB ERASE
200 NEXT J
202 A(J+1)=Y
203 POKE 50,63:X=J+1: GOSUB DISPLAY
204 GOSUB INTERCHANGE
205 POKE 50,255: GOSUB DISPLAY

```

```

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 " ";
670 RETURN
800 IF KEY<> ASC("Q") 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 "I WILL SORT UP TO 100"
1001 TAB 5: PRINT "INTEGERS INTO ASCENDING"
1002 TAB 5: PRINT "ORDER USING THE INSERTION SORT."
1008 VTAB 15: TAB 10: INPUT "VALUE OF N PLEASE",NUM:N=NUM-1
1010 IF N>=0 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 * SELECT *
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
203 A(MAX)=A(N-I):X=MAX: POKE 50,63: GOSUB DISPLAY: GOSUB INT: POKE 50,
255: GOSUB DISPLAY
204 A(N-I)=TEMP:X=N-I: POKE 50,63: GOSUB DISPLAY: GOSUB INT: POKE 50,255
: GOSUB DISPLAY
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("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 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

```

```

10 REM *****
12 REM *
14 REM *   SORTING REVEALED   *
16 REM * RICHARD C. VILE, JR. *
18 REM *
20 REM *   SHELL   *
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 DIM A(99), INCS(5)
105 MUSIC=-10473:PITCH=767:TIME=766:TIMBRE=765: POKE TIMBRE,32
110 KBD=-16384:CLR=-16368:TITLE=400:INTRO=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
260 POKE 50,63
265 GOSUB INT
270 A(K+SPAN)=A(K)
280 X=K+SPAN: GOSUB DISPLAY
285 KEY= PEEK (KBD): IF KEY<128 THEN 290
287 POKE CLR,0: GOSUB WAIT
290 GOSUB DELAY
300 POKE 50,255: GOSUB DISPLAY
305 X=K: GOSUB ERASE
310 NEXT K
320 POKE E50,63
325 GOSUB INT
330 A(K+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 PEEK (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

```

```

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:CMP=6500:DELAY=6600
38 MUSIC=10473:TIME=766:TIMBRE=765:PITCH=767
40 TEXT : CALL -936
42 PRINT "MICRO APPLE, VOLUME 1"
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)= RND (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 "Q= ";: IF K<100 THEN PRINT " ";: IF K<10 THEN PRINT
" ";: PRINT K;
139 GOSUB 1145
140 IF J-P<Q-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)=Q
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";
5050 VTAB 5: TAB 39: PRINT "S": TAB 39: PRINT "T": TAB 39: PRINT "A": TAB
39: PRINT "C": TAB 39: PRINT "K"
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

```

```
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"
10020 TAB 5: PRINT "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
10080 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

$$1.) \text{ Area} = \frac{ab}{2} (H - e \sin H).$$

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, (total area = πab), by the number of days to complete the orbit we have the area swept out per day. Rearranging equation (1), we get

$$2). \quad H - e \sin H = \frac{\text{area} * 2}{ab}$$

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

$$3.) \quad RV = 2a - (P/1 + e(\cos(180-A)))$$

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

$$4.) \quad \cos V1 = \frac{R - RV}{RV * e}$$

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.

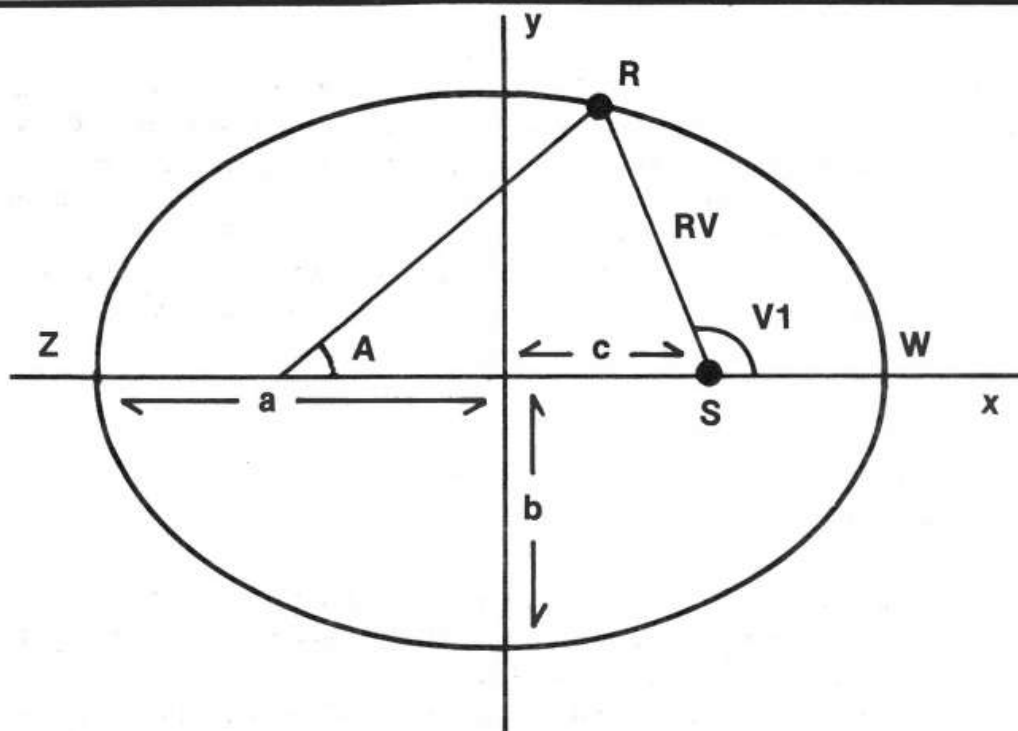


Figure 1

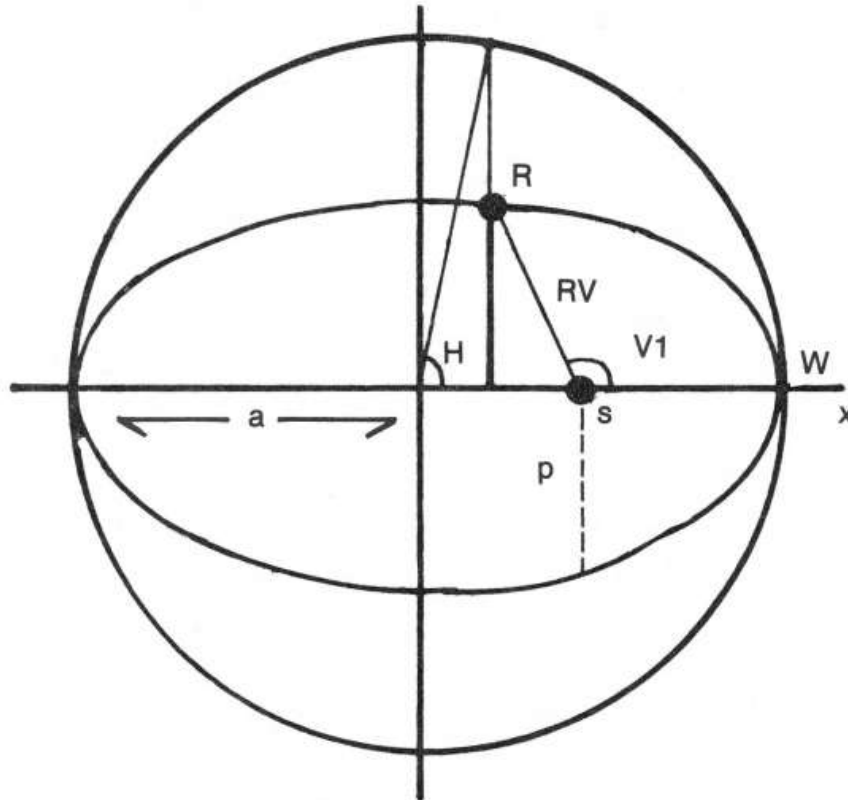


Figure 2

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 $\frac{1}{4}$ 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,

$$5). \quad \cos H = \frac{a-RV}{ae}$$

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.

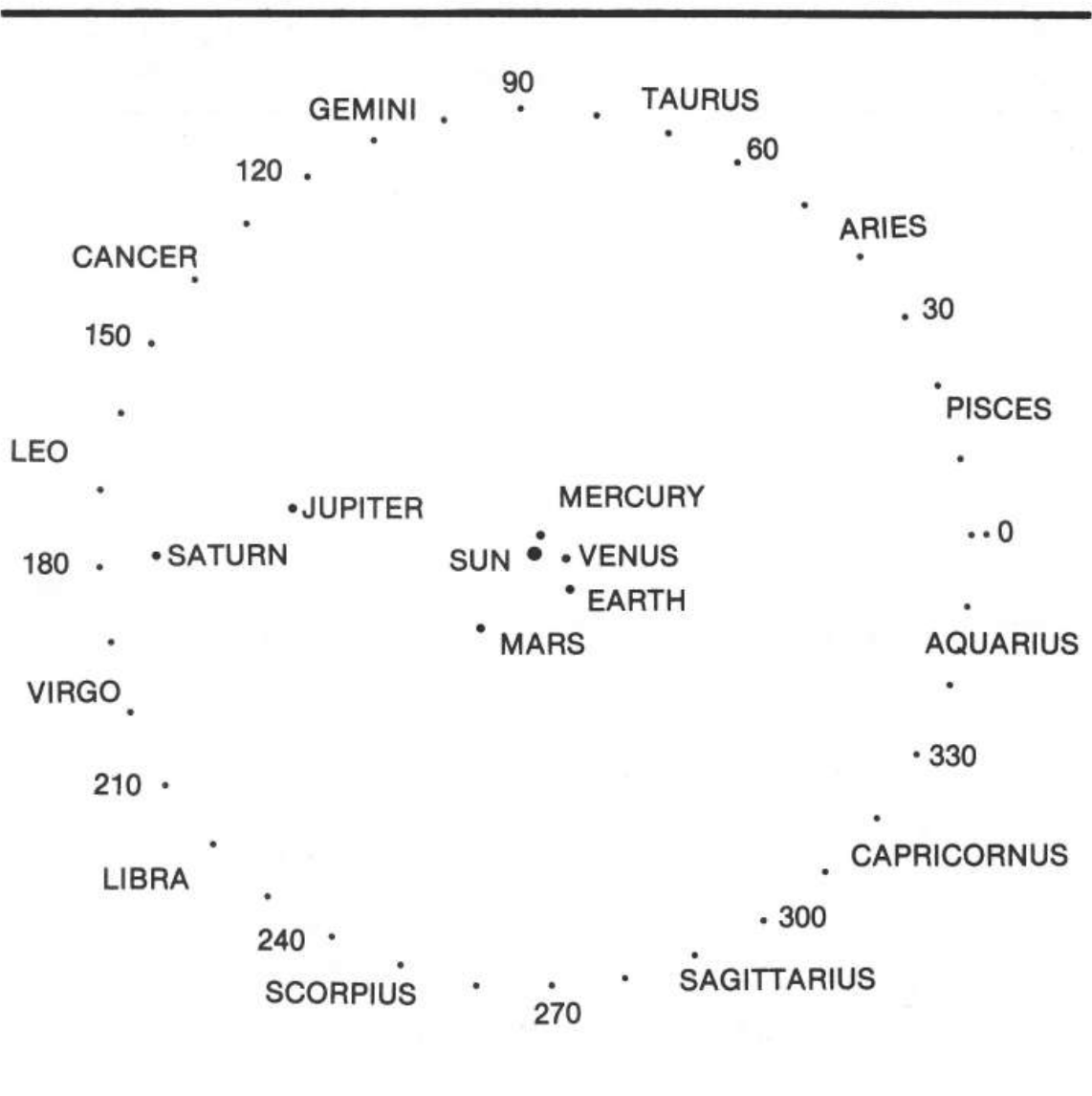


Figure 3: This is an example of the display for all six planets for Aug. 11, 1980 (224 days from Jan. 0).

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.

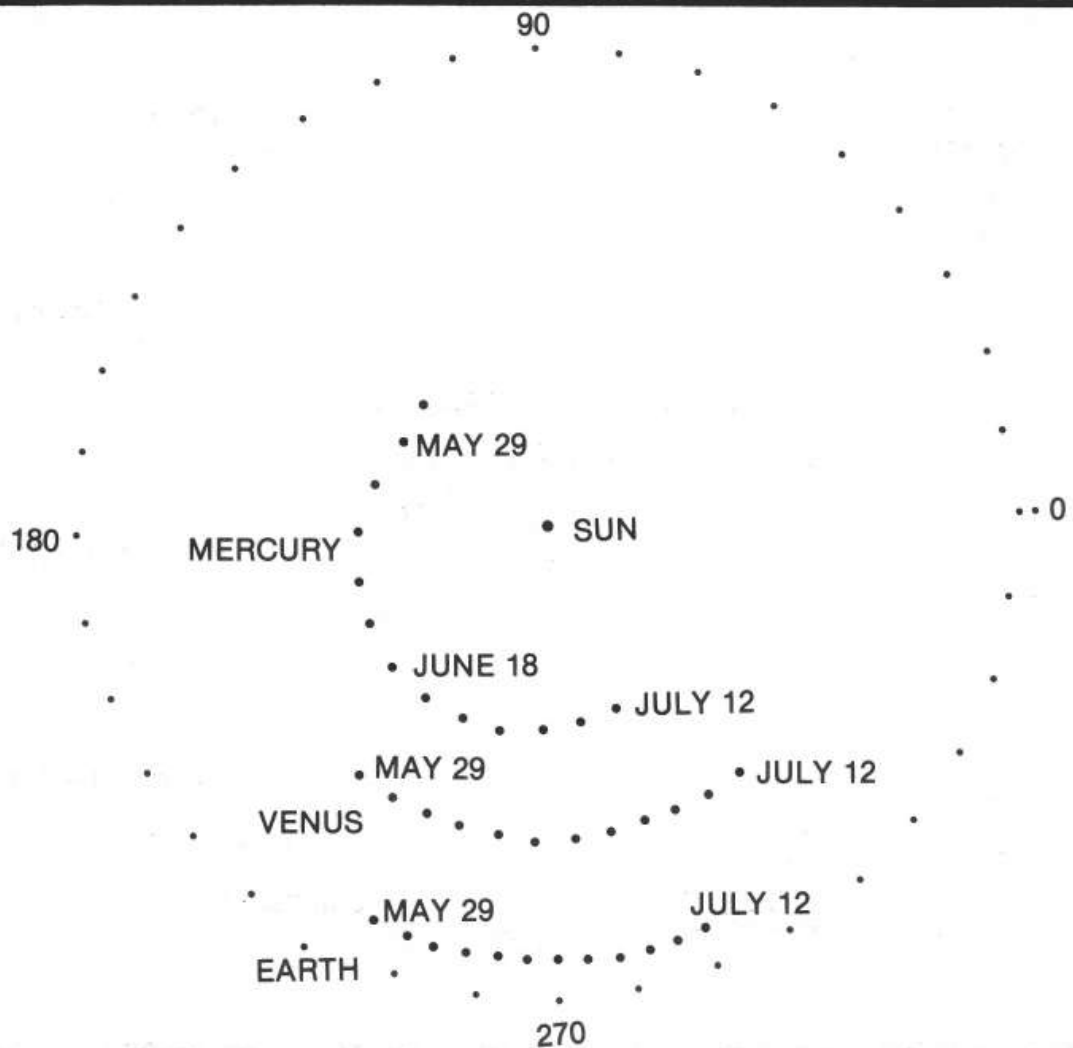


Figure 4: This is a display of the planets Mercury, Venus, and the Earth. This example is for continuous plots starting May 29th (day 150), for 44 days with 4 days between plots.

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

	Sidereal revolution in days	Distance from Sun in million miles		Longitude of perhelion in degrees	Eccentricity
		max.	min.		
Mercury	87.969	43.403	28.597	77.1	.2056
Venus	224.701	67.726	66.813	131.3	.0068
Earth	365.256	94.555	91.445	102.6	.0167
Mars	686.980	154.936	128.471	335.7	.0934
Jupiter	4332.125	507.046	460.595	13.6	.0478
Saturn	10825.863	937.541	838.425	95.5	.0555
Uranus	30676.15	1859.748	1699.331	172.9	.0503
Neptune	59911.13	2821.686	2760.386	58.5	.0066
Pluto	90824.2	4551.386	2756.427	223.0	.2548

Listing 2

Address	Old	New	
C01	20	40	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.
C65	0B	3B	
C7E	0C	3C	
CE3	0D	3D	
D0A	0D	3D	
D62	0D	3D	
D6B	0D	3D	
D93	0D	3D	
D9F	0D	3D	
DCD	0E	3E	
DD5	0E	3E	The value in location C01 was changed to use page 2 (4000-6000) instead of page 1 (2000-4000).
DF6	0D	3D	
E02	0D	3D	
E3D	0D	3D	
EBF	0C	3C	
EC6	0E	3E	
EC9	0C	3C	
ED8	0C	3C	
EF1	0D	3D	

	Old Values	High-Res	New Values
	Dec. Hex.	Commands	Dec. Hex
	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

```

10 REM *****
15 REM *
20 REM * SOLAR SYSTEM SIMULATION *
30 REM * DAVID A. PARTYKA *
35 REM *
40 REM * SOLAR *
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 (100-110) PLOT X AND Y VALUES
100 HPLOT 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 - EZ
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
    OF AN OVAL
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 "; A$
1820 PRINT : PRINT
1830 IF A$ = "N" THEN 2000
1840 IF A$ < > "Y" THEN 1800
1850 IF S1 < > 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 S1=ORBITAL PERIOD: P1=A1*(1-E1*E1)/2
2014 REM E1=ECCENTRICITY: U1=P1/E1;K1=1/E1
2015 REM A1= 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 PLANETS
    ARE DISPLAYED
2020 INPUT "DISPLAY MERCURY "; ME
2021 S1 = 87.969
2022 E1 = .2056
2023 A1 = 43.403 + 28.597
2024 P1 = A1 * (1 - E1 * E1) / 2
2025 K1 = 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 = - 3
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 = 335.7 * 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 = 10825.863
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 : INPUT "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 HGR2
4802 REM (4805-4860) PLOT REFERENCE POINTS AND OUTER 10 DEGREE CIRCLE

```

```

4805 HCOLOR= 3
4810 X = 140:Y = 96: GOSUB 100
4811 X = 141:Y = 96: GOSUB 100
4815 X = 248:Y = 96: GOSUB 100
4820 FOR L1 = 0 TO Q2 STEP 1 / 36 * Q2
4830 X = X1 + COS (L1) * 105.9
4840 Y = Y1 - SIN (L1) * 105.9 * FA
4850 GOSUB 100
4860 NEXT L1
4900 REM (5100-5140) SET UP VALUES FOR MERCURY AND PLOT
5100 IF ME = 0 THEN 5200
5110 A = A1:P = P1:E = E1:PE = U1:EZ = K1:SRD = S1:J = J1:W = W1:Z = Z1 +
W
5120 GOSUB 200:F1 = X:G1 = Y
5125 IF TY = 1 THEN 5140
5130 X = M1:Y = N1: HCOLOR= 0: GOSUB 100
5140 X = F1:Y = G1:M1 = X:N1 = Y: HCOLOR= 3: GOSUB 100
5190 REM (5200-5240) SET UP VALUE FOR VENUS AND PLOT
5200 IF VE = 0 THEN 5300
5210 A = A2:P = P2:E = E2:PE = U2:EZ = K2:SRD = S2:J = J2:W = W2:Z = Z1 +
W
5220 GOSUB 200:F2 = X:G2 = Y
5225 IF TY = 1 THEN 5240
5230 X = M2:Y = N2: HCOLOR= 0: GOSUB 100
5240 X = F2:Y = G2:M2 = X:N2 = Y: HCOLOR= 3: GOSUB 100
5290 REM (5300-5240) SET UP VALUES FOR EARTH AND PLOT
5300 IF EA = 0 THEN 5400
5310 A = A3:P = P3:E = E3:PE = U3:EZ = K3:SRD = S3:J = J3:W = W3:Z = Z1 +
W
5320 GOSUB 200:F3 = X:G3 = Y
5325 IF TY = 1 THEN 5340
5330 X = M3:Y = N3: HCOLOR= 0: GOSUB 100
5340 X = F3:Y = G3:M3 = X:N3 = Y: HCOLOR= 3: GOSUB 100
5390 REM (5400-5440) SET UP VALUES FOR MARS AND PLOT
5400 IF MA = 0 THEN 5500
5410 A = A4:P = P4::E = E4:PE = U4:EZ = K4:SRD = S4:J = J4:W = W4:Z = Z1 +
W
5420 GOSUB 200:F4 = X:G4 = Y
5425 IF TY = 1 THEN 5440
5430 X = M4:Y = N4: HCOLOR= 0: GOSUB 100
5440 X = F4:Y = G4:M4 = X:N4 = Y: HCOLOR= 3: GOSUB 100
5490 REM (5500-5540) SET UP VALUES FOR JUPITER AND PLOT
5500 IF JU = 0 THEN 5600
5510 A = A5:P = P5:E = E5:PE = U5:EZ = K5:SRD = S5:J = J5:W = W5:Z = Z1 +
W
5520 GOSUB 200:F5 = X:G5 = Y
5525 IF TY = 1 THEN 5540
5530 X = M5:Y = N5: HCOLOR= 0: GOSUB 100
5540 X = F5:Y = G5:M5 = X:N5 = Y: HCOLOR= 3: GOSUB 100
5590 REM (5600-5640) SET UP VALUES FOR SATURN
5600 IF SA = 0 THEN 6000
5610 A = A6:P = P6:E = E6:PE = U6:EZ = K6:SRD = S6:J = J6:W = W6:Z = Z1 +
W
5620 GOSUB 200:F6 = X:G6 = Y
5625 IF TY = 1 THEN 5640
5630 X = M6:Y = N6: HCOLOR= 0: GOSUB 100
5640 X = F6:Y = G6:M6 = X:N6 = Y: HCOLOR= 3: GOSUB 100
6000 Z1 = Z1 + DA
6100 IF Z1 > DE + DN THEN 7000
6200 GOTO 5100
7000 X = 279:Y = 190: GOSUB 100: INPUT AS
7050 REM (7000) PLOT POINT 297 190 TO INDICATE END OF SIMULATION THEN W
AIT FOR INPUT OF ANY CHARACTER TO START AGAIN
7100 Z1 = 0:DE = 0
7200 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 recursion 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 1 1:D 1 (*$L PRINTER:*)
2 1 1:D 1 PROGRAM TESTSORT;
3 1 1:D 3 (*****
4 1 1:D 3 (* *)
5 1 1:D 3 (* QUICKSORT ARRAY OF INTEGER *)
6 1 1:D 3 (* *)
7 1 1:D 3 (*****
8 22 1:D 3 ($ )
9 22 1:D 3
10 22 1:D 3
11 22 2:D 3 FUNCTION PADDLE(SELECT: INTEGER): INTEGER;
12 22 3:D 3 FUNCTION BUTTON(SELECT: INTEGER): BOOLEAN;
13 22 4:D 1 PROCEDURE TTLOUT(SELECT: INTEGER; DATA: BOOLEAN);
14 22 5:D 3 FUNCTION KEYPRESS: BOOLEAN;
15 22 6:D 3 FUNCTION RANDOM: INTEGER;
16 22 7:D 1 PROCEDURE RANDOMIZE;
17 22 8:D 1 PROCEDURE NOTE(PITCH,DURATION: INTEGER);
18 22 8:D 3
19 22 1:D 3 IMPLEMENTATION
20 22 1:D 1
21 1 1:D 1 USES APPLESTUFF;
22 1 1:D 3
23 1 1:D 3 CONST MAXMEMS = 284;
24 1 1:D 3
25 1 1:D 3 VAR OUT : INTERACTIVE;
26 1 1:D 304 CON : INTERACTIVE;
27 1 1:D 605 NUM : ARRAY[0..MAXMEMS] OF INTEGER;
28 1 1:D 890 ACTMEMS, IY : INTEGER;
29 1 1:D 892 P, D, IX : INTEGER;
30 1 1:D 895 CHRCTR : CHAR;
31 1 1:D 896 (*$P*)
32 1 2:D 1 PROCEDURE PRINT(TEXT:STRING);
33 1 2:D 43 (*****
34 1 2:D 43 (* *)
35 1 2:D 43 (* PRINT THE INTEGER ARRAY *)
36 1 2:D 43 (* *)
37 1 2:D 43 (*****
38 1 2:D 43 VAR IX, CTR : INTEGER;
39 1 2:D 45
40 1 2:0 0 BEGIN
41 1 2:1 0 PAGE(OUT);
42 1 2:1 14 WRITELN(OUT,TEXT);
43 1 2:1 31 WRITELN(OUT);
44 1 2:1 38 WRITELN(OUT);
45 1 2:1 45 IX := 0;
46 1 2:1 48 CTR := 0;
47 1 2:1 51 REPEAT
48 1 2:2 51 WRITE(OUT, ' ':4, NUM[IX]:6);
49 1 2:2 81 IX := IX + 1;
50 1 2:2 87 CTR := CTR + 1;
51 1 2:2 93 IF CTR = 12 THEN
52 1 2:3 99 BEGIN
53 1 2:4 99 CTR := 0;
54 1 2:4 102 WRITELN(OUT)
55 1 2:3 109 END;
56 1 2:1 109 UNTIL IX > ACTMEMS;
57 1 2:1 117 WRITELN(OUT);
58 1 2:0 124 END;

```

```

59 1 2:0 138 (*P*)
60 1 3:D 1 PROCEDURE SPLIT(X,Y:INTEGER);
61 1 3:D 3 (*****
62 1 3:D 3 (*
63 1 3:D 3 (* SPLIT IS A PROCEDURE WHICH
64 1 3:D 3 (* ACTUALLY DOES THE SORTING.
65 1 3:D 3 (* THE SORT ALGORITHM USED IS
66 1 3:D 3 (* THE QUICKSORT METHOD.
67 1 3:D 3 (*
68 1 3:D 3 (*****
69 1 3:D 3 VAR F,L,MID : INTEGER;
70 1 3:D 6 ODDPASS : BOOLEAN;
71 1 3:D 7
72 1 4:D 1 PROCEDURE SWITCH(SW1,SW2:INTEGER);
73 1 4:D 3 VAR HOLD : INTEGER;
74 1 4:D 4
75 1 4:0 0 BEGIN
76 1 4:1 0 HOLD := NUM[SW1];
77 1 4:1 14 NUM[SW1] := NUM[SW2];
78 1 4:1 38 NUM[SW2] := HOLD
79 1 4:0 49 END;
80 1 4:0 64
81 1 5:D 1 PROCEDURE BUBBLE(BB1,BB2:INTEGER);
82 1 5:D 3 VAR Z,X : INTEGER;
83 1 5:D 5
84 1 5:0 0 BEGIN
85 1 5:1 0 FOR Z := BB1 TO (BB2 - 1) DO
86 1 5:2 13 BEGIN
87 1 5:3 13 FOR X := (Z + 1) TO BB2 DO
88 1 5:4 26 BEGIN
89 1 5:5 26 IF NUM[Z] > NUM[X] THEN SWITCH(Z,X);
90 1 5:4 57 END;
91 1 5:2 64 END;
92 1 5:0 71 END;
93 1 5:0 88
94 1 6:D 1 PROCEDURE MOVEUP;
95 1 6:0 0 BEGIN
96 1 6:1 0 ODDPASS := FALSE;
97 1 6:1 4 REPEAT
98 1 6:2 4 IF NUM[F] >= NUM[L] THEN
99 1 6:3 35 BEGIN
100 1 6:4 35 SWITCH(F,L);
101 1 6:4 43 F := F + 1;
102 1 6:4 51 MID := L;
103 1 6:4 57 EXIT(MOVEUP)
104 1 6:3 61 END
105 1 6:2 61 ELSE
106 1 6:3 63 L := L - 1;
107 1 6:1 71 UNTIL NOT (L > F);
108 1 6:0 81 END;
109 1 6:0 96
110 1 7:D 1 PROCEDURE MOVEDN;
111 1 7:0 0 BEGIN
112 1 7:1 0 ODDPASS := TRUE;
113 1 7:1 4 REPEAT
114 1 7:2 4 IF NUM[L] < NUM[F] THEN
115 1 7:3 35 BEGIN
116 1 7:4 35 SWITCH(F,L);
117 1 7:4 43 L := L - 1;
118 1 7:4 51 MID := F;

```

```

119 1 7:4 57          EXIT(MOVEDN)
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 (*#P*)
126 1 7:0 96 (*****
127 1 7:0 96 (*          *)
128 1 7:0 96 (*  MAIN LOGIC FOR SPLIT  *)
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 83 IF ODDPASS THEN
142 1 3:3 86 MOVEUP
143 1 3:2 86 ELSE
144 1 3:3 90 MOVEDN;
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 (*#P*)
155 1 3:0 142 (*****
156 1 3:0 142 (*          *)
157 1 3:0 142 (*  PROGRAM TESTSORT LOGIC  *)
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 UNITCLEAR(1);
172 1 1:1 368 READLN(ACTMEMS);
173 1 1:1 387 RANDOMIZE;
174 1 1:1 390 FOR IY := 0 TO ACTMEMS 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 UNITCLEAR(1);
177 1 1:1 494 READ(CHRCTR);
178 1 1:1 505 IF CHRCTR = 'Y' THEN

```

```
179 1 1:2 512 BEGIN
180 1 1:3 512 GOTOXY(06,14); WRITE('START PRINTER AND HIT ANY KEY ');
181 1 1:3 559 UNITCLEAR(1);
182 1 1:3 562 READ(CHRCTR);
183 1 1:3 573 GOTOXY(00,00);
184 1 1:3 578 PRINT('BEFORE THE SORT -')
185 1 1:2 598 END;
186 1 1:1 600 GOTOXY(12,16); WRITE('SORT INITIATED ');
187 1 1:1 632 P := 18;
188 1 1:1 636 D := 100;
189 1 1:1 640 NOTE(P,D);
190 1 1:1 649 SPLIT(0,ACTMEMS);
191 1 1:1 655 P := 18;
192 1 1:1 659 NOTE(P,D);
193 1 1:1 668 PAGE(CON);
194 1 1:1 678 GOTOXY(05,14); WRITE('START PRINTER AND HIT ANY KEY ');
195 1 1:1 725 UNITCLEAR(1);
196 1 1:1 728 READ(CHRCTR);
197 1 1:1 739 PRINT('AFTER THE SORT -')
198 1 1:0 758 END.
```

BEFORE THE SORT -

213	3303	2154	2406	1897	1348	248	1919	492	2580	23	3433
1786	1291	3451	1394	3244	2128	453	1139	1610	2982	317	3034
1813	2632	2593	2907	575	2310	1815	1938	1246	986	1506	2786
1160	3053	1433	286	1881	1820	1481	2394	2076	3004	519	1051
422	2612	1918	1708	715	1970	2371	3157	880	2612	3121	1445
929	2442	1161	2602	2043	711	3262	1640	2433	1151	1805	600
1781	3351	2234	2257	3526	2301	1320	972	1400	2658	423	383
112	2869	179	2360	2239	1770	2238	886	1168	1059	1167	404
3314	648	2967	670	2471	1920	2401	3420	2313	1246	1445	2854
1025	2014	2824	657	196	1574	1540	2854	1085	1261	1156	2703
2574	2909	1959	419	919	3212	3208	3258	2971	855	849	559
3527	1064	2566	1270	663	585	1333	989	1103	1201	1314	3220
2746	272	2456	1415	1062	303	900	1206	676	2903	1133	3210
3154	308	3573	2034	3173	2308	3482	2711	854	1817	3502	3390
582	553	2911	3056	1505	1845	1087	683	3003	3258	3317	1010
1209	1877	606	2338	785	2241	605	3221	2876	2665	830	2164
1563	3476	1433	1167	1542	1073	3005	1791	1477	3391	653	3043
951	1454	592	3326	1323	1421	2581	3609	1426	1214	1259	1836
3462	1592	1248	347	738	2298	2774	2458	2954	3116	991	2545
644	3243	2061	1381	1841	2171	1352	1568	398	2834	1764	3345
1750	1634	3661	3164	753	3690	1756	712	1019	1201	2603	1630
3486	1601	2211	3279	1122	531	2760	3020	348	302	522	874
2205	3427	1907	1857	2243	2691	3134	1570	2394	1725	713	2393
1199	1158	3477	1904	1177	318	1675	3354	2541			

AFTER THE SORT -

23	112	179	196	213	248	272	286	302	303	308	317
318	347	348	383	398	404	419	422	423	453	492	519
522	531	553	559	575	582	585	592	600	605	606	644
648	653	657	663	670	676	683	711	712	713	715	738
753	785	830	849	854	855	874	880	886	900	919	922
929	951	986	989	991	1010	1019	1025	1051	1059	1062	1064
1073	1085	1087	1103	1122	1133	1139	1151	1156	1158	1160	1161
1167	1167	1168	1177	1199	1201	1201	1206	1209	1214	1246	1246
1248	1259	1261	1270	1291	1314	1320	1323	1333	1348	1352	1381
1394	1400	1415	1421	1426	1433	1433	1445	1445	1454	1477	1481
1505	1506	1540	1542	1563	1568	1570	1574	1592	1601	1610	1630
1634	1640	1675	1681	1708	1725	1750	1756	1764	1770	1781	1786
1791	1805	1813	1815	1817	1820	1836	1841	1845	1857	1877	1892
1904	1907	1918	1919	1920	1938	1959	1970	2014	2034	2043	2061
2076	2128	2154	2164	2171	2205	2211	2234	2238	2239	2241	2243
2257	2298	2301	2308	2310	2313	2338	2360	2371	2393	2394	2394
2401	2406	2433	2442	2456	2458	2471	2541	2545	2566	2574	2580
2581	2593	2602	2603	2612	2612	2632	2658	2665	2691	2703	2711
2746	2760	2774	2786	2824	2834	2854	2854	2869	2876	2903	2907
2909	2911	2954	2967	2971	2982	3003	3004	3005	3020	3034	3043
3053	3056	3116	3121	3134	3154	3157	3164	3173	3208	3210	3212
3220	3221	3243	3244	3258	3258	3262	3279	3303	3314	3317	3326
3345	3351	3354	3390	3391	3420	3427	3433	3451	3462	3476	3477
3482	3486	3502	3526	3527	3573	3609	3661	3690			

6 GAMES

Introduction	154
Spelunker <i>Thomas R. Mimlitch</i>	155
Life for Your Apple <i>Richard F. Sutor</i>	168
Apple II Speed Typing Test with Input Time Clock <i>John Broderick, CPA</i>	173
Ludwig Von Apple II <i>Marc Schwartz and Chuck Carpenter</i>	175

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!

Spelunker

by Thomas R. Mimlitch

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 stalagmite 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

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

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

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

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.

Room	Room Table				Notes
	Tunnel Connects				
	N	E	S	W	
1 Mouth	50		19	2	Truck Tent Knife Light
2 Tree		1	3		
3 Writing	2		10	20	Bomb
4 Pit		20			Ax Use rope to go down
5 South Lake Shore	-18				Raft-north Rope-up
6 West Lake Shore		-18		12	Raft-east
7 North Lake Shore	9		-18		Raft-south
8 Maze	8	9	8	20	All 45's return to Maze
9 Frozen River	7	1		8	Rope
10 Swift River Room	3		-11		Raft-south
11 Hub	13	14	-49	21	-15 22 12 (NE SE SW NW)
12 Ice			11	6	Ice Lamp
13 Chimney			11		Fire Rope-up
14 Gold				11	Gold Ogre
15 Bones				11	Curse Bones
16 Bats	22				Chest Bats
17 Ghost		21			Ghost
18 Misty Lake	7		5	6	Pearl Clam
19 Swift River	1				
20 Intersect 1	8	3		4	
21 Intersect 2		11	22	17	
22 Intersect 3	11		16	21	
49 Falls (over)					Death
50 Home					End game

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.

Monster Table

Monster name	Ogre	Bats	Ghost	Clam	Ice	Bones
Monster number	24	21	23	20	36	22
Reward	Gold	Chest		Pearl	Lamp	
Reward number	26	25	0	27	28	0
Move delay	0	0	0	1	1	1
Move increment	2	4	6	0	0	0
Attack count	0	0	0	0	0	0
Kill probability	60	60	0	90	0	0
Steal all probability	30	40	0	60	60	0
Steal own probability	55	90	0	65	0	0
Home room number	14	16	17	18	12	15
Life force quotient	100	40	50	60	25	75

Weapon Table

Weapon name	Ax	Bomb	Fire	Knife	Light	Ice
Weapon number	15	16	18	19	33	36
Power	100	150	30	50	30	40
Attacks Monster No. 1	24	24	21	24	23	21
Attacks Monster No. 2		22	22	20		
Attacks Monster No. 3		36	36			

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.

```

10 REM *****
15 REM *
20 REM * SPELUNKER *
25 REM * THOMAS R. MIMLITCH *
30 REM *
35 REM * SPELUNKER *
40 REM *
45 REM * COPYRIGHT (C) 1981 *
50 REM * MICRO INK, INC. *
55 REM * CHELMSFORD, MA 01824 *
60 REM * ALL RIGHTS RESERVED *
65 REM *
70 REM *****
75 REM
80 REM
100 GOTO 30000: REM TO INITIALIZE
1000 PRINT "?";: INPUT IN$:IN$( LEN(IN$)+1)=" GO N * ":I=1
1005 NOUN=0:VERB=0
1010 GOSUB 1500: GOSUB 1600: GOSUB 1700
1020 IF W3$#"*" THEN 1010
1050 NTYP=NTYP(NOUN):VTYP=VTYP(VERB)
1060 IF (VTYP MOD (NTYP*2))>=NTYP THEN 2000
1070 PRINT "ICH VERSTEHE NICHT"
1080 GOTO 3000
1200 GOTO 2000
1500 W3$="":S=0: FOR I=1 TO LEN(IN$): IF S=0 THEN 1520: IF IN$(I,I)=" " THEN
1580: IF S=5 THEN 1560: GOTO 1540
1520 IF IN$(I,I)=" " THEN 1560
1540 S=S+1:W3$(S)=IN$(I,I)
1560 NEXT I
1580 IF S<5 THEN W3$(S+1)=SPC$(S+1)
1590 RETURN
1600 IF NOUN#0 THEN RETURN : FOR J=1 TO NUMN: IF W3$#NOUN$(J*5-4,J*5) THEN
NEXT J: IF J>=NUMN THEN RETURN :NOUN=J:W2$=W3$
1610 RETURN
1700 IF VERB#0 THEN RETURN : FOR J=1 TO NUMV: IF W3$#VERB$(J*5-4,J*5) THEN
NEXT J: IF J>=NUMV THEN RETURN :VERB=J:W1$=W3$
1710 RETURN
2000 REM MOVE
2010 NLOC=0
2020 IF NOUN>8 THEN 2200
2030 IF (NOUN MOD 2)=1 THEN 2100
2040 IF LOC#11 AND LOC#8 THEN 1070
2100 GOTO 2100+NOUN*10
2110 NLOC=N(LOC): GOTO 2190
2120 NLOC=0: GOTO 2190
2130 NLOC=E(LOC): GOTO 2190
2140 NLOC=15: IF LOC=8 THEN NLOC=8: GOTO 2190
2150 NLOC=S(LOC): GOTO 2190
2160 NLOC=22: IF LOC=8 THEN NLOC=8: GOTO 2190
2170 NLOC=W(LOC): GOTO 2190
2180 NLOC=12: IF LOC=8 THEN NLOC=8: GOTO 2190
2190 IF RAFT=1 THEN NLOC= ABS (NLOC)
2191 RAFT=0:PLOC=LOC
2192 IF NLOC>0 THEN LOC=NLOC
2193 IF NLOC#12 THEN 2900
2194 IF M(50)<5 THEN 2900
2195 IF PLOC=6 THEN S(12)=0
2196 IF PLOC=11 THEN W(12)=0
2197 GOTO 2900
2200 IF (NOUN=9 OR NOUN=10) AND ROPE=0 THEN GOTO 1070
2205 IF NOUN#9 THEN 2250
2210 IF LOC#5 AND LOC#13 THEN 1070
2220 IF LOC=5 THEN LOC=4
2230 IF LOC=13 THEN LOC=50
2240 GOTO 3000
2250 IF NOUN#10 THEN 2300
2260 IF LOC#4 THEN 1070
2270 LOC=5: GOTO 3000
2300 IF VERB=8 OR VERB=11 THEN 2320: GOTO 2350
2320 IF NUMP=8 THEN 1070
2325 IF NOUN=34 AND (LOC=19 OR LOC=10 OR LOC=5 OR LOC=18 OR LOC=7 OR LOC=
6 OR LOC=11) THEN 2345
2330 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
2383 IF NOUN#33 THEN 2420
2385 IF VERB#10 THEN STA(33)=0
2387 LIGHT=0
2390 GOTO 2420
2400 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
2449 IF M(M+4)=0 THEN M(M+4)=1
2450 PRINT "ASSAULT ON ";NOUN$(M(M)*5-4,M(M)*5);", ",HT;" UNITS"
2452 PRINT "ITS LIFE FORCE IS NOW ";M(M+9);"%"
2455 IF M(M+9)>0 THEN 2460
2456 PRINT NOUN$(M(M)*5-4,M(M)*5);" HAS BEEN ELIMINATED"
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(LCC)=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:RAFT=1: GOTO 2900
2530 IF NOUN#30 THEN 2540:ROPE=1: GOTO 2900
2540 REM
2900 IF NOUN<11 THEN ROPE=0
2910 IF STA(30)=LOC THEN ROPE=1
2920 IF LOC=12 THEN 3000
2930 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 STA(35)=2
3090 GOTO 4000
3800 REM MONS SUB
3802 MRM=STA(M(M)) MOD 100
3810 IF (STA(M(M+1)) MOD 100)=MRM AND M(M+4)=0 THEN 3900
3820 IF MRM=LOC THEN 3860
3830 M(M+2)=(M(M+2)+M(M+3)) MOD 6
3840 IF M(M+2)#0 THEN RETURN
3845 GOTO 3850+ RND (4)
3850 NLOC=N(MRM): GOTO 3855
3851 NLOC=E(MRM): GOTO 3855
3852 NLOC=S(MRM): GOTO 3855
3853 NLOC=W(MRM): GOTO 3855
3855 IF NLOC<1 THEN RETURN
3858 STA(M(M))=NLOC+STA(M(M))-MRM: RETURN
3860 M(M+4)=M(M+4)+1
3865 KP=(M(M+5)-(STA(M(M+1))=-1)*40+9*(M(M+4)-2))*M(M+9)/100+CURSE
3866 IF KP>60 THEN KP=60
3870 SAP=(M(M+6)+9*(M(M+4)-2))*M(M+9)/100+CURSE
3871 IF SAP>70 THEN SAP=70
3875 SRP=(M(M+7)+9*(M(M+4)-2))*M(M+9)/100+CURSE
3876 IF SRP>80 THEN SRP=80
3877 PRINT "ATTACK BY ";NOUN$(M(M)-1)*5+1,M(M)*5)
3879 R1= RND (100):R2= RND (100):R3= RND (100)
3880 IF KP>R1 THEN 3920
3885 IF SAP>R2 THEN 3940
3887 IF STA(M(M+1))=-1 THEN RETURN
3890 IF SRP>R3 THEN 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";NOUNSS((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
4060 GOTO 4000+100*LOC
4070 POKE 50,63: VTAB 3: TAB 2: PRINT LOC$;: POKE 50,255: PRINT " ";: RETURN
4090 VTAB 23: TAB 1
4095 IF LIGHT=1 OR LOC<3 OR LOC=19 THEN 9100
4097 PRINT "IT IS VERY DARK"
4099 GOTO 9100
4100 LOC$="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 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$="INTERSECTION": GOSUB 4070
6099 GOTO 4090
6100 GOTO 6000
6200 GOTO 6000
6999 GOTO 4090
8900 LOC$="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 THEN AMT=AMT+13
9020 IF STA(26)=-1 THEN AMT=AMT+22
9030 IF STA(27)=-1 THEN AMT=AMT+8

```

```

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 $";AMT;","; RND (900)+100;" IN TREASURES"
9080 IF AMT>13 THEN PRINT "NICE SPELUNKING!"
9090 PRINT "GOOD-BYE"
9099 END
9100 FOR I=2 TO 10: VTAB I: TAB 30: PRINT "      ": NEXT I
9105 IF LIGHT=0 AND LOC>2 AND LOC#19 THEN 9290
9110 VTAB 5: TAB 33: PRINT "^": TAB 33: PRINT "+": POKE 50,63
9140 IF N(LOC)=0 OR (N(LOC)<0 AND RAFT=0) THEN 9150: VTAB 3: TAB 33: PRINT
"N": TAB 33: PRINT " "
9150 IF S(LOC)=0 OR (S(LOC)<0 AND RAFT=0) THEN 9160: VTAB 8: TAB 33: PRINT
" ": TAB 33: PRINT "S"
9160 IF E(LOC)=0 OR (E(LOC)<0 AND RAFT=0) THEN 9170: VTAB 6: TAB 35: PRINT
" E"
9170 IF W(LOC)=0 OR (W(LOC)<0 AND RAFT=0) THEN 9180: VTAB 6: TAB 30: PRINT
"W "
9180 IF (LOC=5 OR LOC=13) AND ROPE=1 THEN 9185: GOTO 9190
9185 VTAB 2: TAB 33: PRINT "UP"
9190 IF LOC#4 OR ROPE=0 THEN 9200
9195 VTAB 10: TAB 33: PRINT "DOWN"
9200 IF LOC=11 OR LOC=8 THEN 9210: GOTO 9290
9210 VTAB 3: TAB 30: PRINT "N ": TAB 30: PRINT " W"
9215 IF LOC#8 THEN 9220: VTAB 3: TAB 35: PRINT " E": TAB 35: PRINT "N "
9220 VTAB 8: TAB 30: PRINT " W";: TAB 35: PRINT "S ": TAB 30: PRINT "S "
;: TAB 35: PRINT " E"
9290 POKE 50,255
9300 IF LIGHT=0 AND LOC>2 AND LOC#19 THEN 9400
9305 VTAB 5: TAB 2: J=0
9310 FOR I=1 TO NUMN-1
9320 IF (STA(1) MOD 100)#LOC THEN 9360
9330 PRINT NOUNS$( (I-1)*5+1, I*5); " ";
9340 J=(J+1) MOD 4: IF J#0 THEN 9360
9350 PRINT "": TAB 2
9360 NEXT I
9400 VTAB 13: TAB 2: FOR I=1 TO 12: PRINT "      ";: NEXT I
9410 VTAB 13: TAB 2: PRINT "POSSESSIONS ";: NUMP=0
9420 FOR I=1 TO NUMN-1
9430 IF STA(I)>=0 THEN 9480
9440 PRINT NOUNS$( (I-1)*5+1, I*5); " ";
9450 NUMP=NUMP+1: IF NUMP=4 THEN TAB 14
9480 NEXT I
9900 VTAB 23: TAB 1: GOTO 1000
30000 REM INITIALIZE ROUTINE
30010 DIM IN$(40), NOUNS$(255), VERBS$(255), W1$(5), W2$(5), W3$(5), NTYP(50), VTYP(
50), STA(50)
30020 DIM N(50), E(50), S(50), W(50)
30030 TEXT : CALL -936
30040 DIM LOC$(26), SPC$(5), M(6*10)
30050 SPC$=" "
30060 NUMW=6
30065 DIM WT(5*NUMW)
30070 LOC=1
30100 REM INITIALIZE VARIABLES
30101 REM SHOULD BE READ AND DATA STMTS
30110 NOUNS$( LEN(NOUNS$)+1)="N NE E SE S SW W NW UP DOW
N "
30120 NOUNS$( LEN(NOUNS$)+1)="CAVE LAKE RIVERTREE "
30130 NOUNS$( LEN(NOUNS$)+1)="AX BOMB CURSEFIRE KNIFE"
30140 NOUNS$( LEN(NOUNS$)+1)="CLAM BATS BONESGHOSTOGRE "
30150 NOUNS$( LEN(NOUNS$)+1)="CHESTGOLD PEARLLAMP "
30160 NOUNS$( LEN(NOUNS$)+1)="RAFT ROPE TENT TRUCKLIGHT"
30170 NOUNS$( LEN(NOUNS$)+1)="WATERAPPLEICE "
30195 NOUNS$( LEN(NOUNS$)+1)="*****"
30199 NUMN=37
30210 VERBS$( LEN(VERBS$)+1)="GO JUMP RUN WALK DRIVECLIMB"
30220 VERBS$( LEN(VERBS$)+1)="DIG "
30230 VERBS$( LEN(VERBS$)+1)="CARRYDROP PUT TAKE USE HIT FIGHT"
30240 VERBS$( LEN(VERBS$)+1)="HELP KILL STOP HIT FIGHT"
30250 VERBS$( LEN(VERBS$)+1)="RUB "
30260 VERBS$( LEN(VERBS$)+1)="STARTDRIVE"
30270 VERBS$( LEN(VERBS$)+1)="DRINKEAT BITE "
30295 VERBS$( LEN(VERBS$)+1)="*****"
30299 NUMV=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)=116: 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:S(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
30620 N(10)=3:N(11)=13:N(16)=22:N(18)=7
30630 N(19)=1:N(20)=8:N(22)=11
30640 E(2)=1:E(4)=20:E(6)=-18:E(8)=9:E(9)=1:E(11)=14:E(17)=21:E(20)=3
30650 E(21)=11
30660 S(1)=19:S(2)=3:S(3)=10:S(7)=-18:S(8)=8:S(10)=-11:S(11)=-49:S(12)=11
:S(13)=11:S(18)=5
30670 S(21)=22:S(22)=16
30680 W(1)=2:W(3)=20:W(6)=12:W(8)=20:W(9)=8:W(11)=21
30690 W(12)=6:W(14)=11:W(15)=11:W(18)=6:W(20)=4:W(21)=17:W(22)=21
30700 POKE 50,63
30710 VTAB 24: GOSUB 31999: VTAB 1: GOSUB 31999: VTAB 11: 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
30740 POKE 50,255: POKE 32,1: POKE 33,37: POKE 34,16: POKE 35,23: VTAB 17
: TAB 2
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
30820 M(11)=21:M(12)=25:M(14)=4:M(16)=60:M(17)=40:M(18)=90:M(19)=16:M(20)
=40
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 NUMM=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
30920 WT(11)=18:WT(12)=30:WT(13)=21:WT(14)=22:WT(15)=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)=0:WT(25)=0
30950 WT(26)=36:WT(27)=40:WT(28)=21:WT(29)=0:WT(30)=0
30999 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. Sutor

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 Sutor'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 × 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"
16 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 AS(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=I1 TO I2
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
2590 GOTO 100
9000 END

```

```

0800      1 ;*****
0800      2 ;*
0800      3 ;* GAME OF LIFE FOR APPLE II *
0800      4 ;*           BY           *
0800      5 ;*       RICHARD SUITOR     *
0800      6 ;*
0800      7 ;*           LIFE           *
0800      8 ;*
0800      9 ;*   COPYRIGHT (C) 1981     *
0800     10 ;*   MICRO INK, INC.      *
0800     11 ;*   ALL RIGHTS RESERVED   *
0800     12 ;*
0800     13 ;*****
0800     14 ;
0800     15 ;
0800     16           ORG $800
0800     17           OBJ $800
0800     18 ;LIFE ROUTINES
0800     19 ;ENTER AT GEN0 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        ;40 VERT. OCC. #'S
0800     26 BF1P   EQU $0942
0800     27 BF1M   EQU $093F
0800     28 BUF2   EQU $0970
0800     29 BF2P   EQU $0972
0800     30 BF2M   EQU $096F
0800 A505     31 NXLN   LDA NWLN+01
0802 8503     32       STA OLLN+01
0804 A504     33       LDA NWLN
0806 8502     34       STA OLLN
0808 18       35       CLC
0809 6980     36       ADC 80
080B 8504     37       STA NWLN
080D A505     38       LDA NWLN+01
080F 6900     39       ADC 00
0811 C908     40       CMP 08
0813 D00C     41       BNE SAME
0815 A504     42       LDA NWLN
0817 6927     43       ADC 27
0819 C952     44       CMP 52
081B 1008     45       BPL LAST
081D 8504     46       STA NWLN
081F A904     47       LDA 04
0821 8505     48 SAME   STA NWLN+01
0823 18       49       CLC
0824 60       50 RTS1   RTS
0825 38       51 LAST   SEC
0826 B0FC     52       BCS RTS1
0828         53 ;GENERATE BIRTHS (COLOR=3) & DEATHS (COL=8)
0828 20CA08   54 GEN0   JSR INIT
082B 200008   55 GEN1   JSR NXLN
082E 9001     56       BCC GEN2
0830         57 ;ALL DONE IF CARRY SET
0830 60       58       RTS
0831 A027     59 GEN2   LDY 27
0833 98       60       TYA
0834 AA       61       TAX
0835         62 ;COMP VERT OCC #S
0835 A900     63 GEN6   LDA 00
0837 994009   64       STA BUF1,Y
083A 997009   65       STA BUF2,Y
083D B102     66       LDA (OLLN),Y
083F F00F     67       BEQ GEN3
0841 1006     68       BPL GEN7
0843 FE4009   69       INC BUF1,X
0846 FE7009   70       INC BUF2,X
0849 2908     71 GEN7   AND 08
084B F003     72       BEQ GEN3
084D FE4009   73       INC BUF1,X
0850 B104     74 GEN3   LDA (NWLN),Y
0852 F00F     75       BEQ GEN5

```

```

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 AD5B09   87          LDA BUF1+27
086D 6D5A09   88          ADC BUF1+26
0870 8506     89          STA SUM1
0872 AD8B09   90          LDA BUF2+27
0875 6D8A09   91          ADC BUF2+26
0878 8507     92          STA SUM2
087A          93      ;COMP OCC #S IN 3X3 & CHANGE COLOR
087A 18       94      GNLP   CLC
087B A506     95          LDA SUM1
087D 793F09   96          ADC BF1M,Y
0880 38       97          SEC
0881 F94209   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 GN10
0890 B102    105      GEN8   LDA (OLLN),Y
0892 F00A    106         BEQ GN10
0894 298F    107         AND 8F
0896 5004    108         BVC GN16
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
08A8 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 29F8    125         AND 0F8
08BA 5004    126         BVC GN15
08BC B104    127      GN12   LDA (NWLN),Y
08BE 0903    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 8D6809  138         STA BF1P+$26
08D5 8D9809  139         STA BF2P+$26
08D8 60      140         RTS
08D9          141      ;MOP UP, IF COLOR AND 3=0, REMOVE (COL=0)
08D9          142      ;OTHERWISE, ALIVE (COL=11)
08D9 20CA08  143      MOP0   JSR INIT
08DC 200008  144      MOP1   JSR NXLN
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

```

```

08EA C910      151      CMP 10
08EC 3002      152      BMI MOP4
08EE 0980      153      ORA 80
08F0 9102      154 MOP4    STA (OLLN),Y
08F2 B104      155 MOP5    LDA (NWLN),Y
08F4 F00A      156      BEQ MOP7
08F6 29F7      157      AND 0F7
08F8 6A        158      ROR
08F9 9002      159      BCC MOP6
08FB 0904      160      ORA 04
08FD 2A        161 MOP6    ROL
08FE 9104      162      STA (NWLN),Y
0900 88        163 MOP7    DEY
0901 F0D9      164      BEQ MOP1
0903 10DF      165      BPL MOP3
                166      END

```

***** END OF ASSEMBLY

```

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

```

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

** ZERO PAGE VARIABLES:

```

OLLN  0002  NWLN  0004  SUM1  0006  SUM2  0007

```

** ABSOLUTE VARIABLES/LABELS

```

BUF1  0940  BF1P  0942
BF1M  093F  BUF2  0970  BF2P  0972  BF2M  096F  NXLN  0800  SAME  0821
RTS1  0824  LAST  0825  GEN0  0828  GEN1  082B  GEN2  0831  GEN6  0835
GEN7  0849  GEN3  0850  GEN4  0859  GEN5  0863  GNLP  087A  GEN8  0890
GEN9  0898  GN16  089C  GN10  089E  GN11  08B4  GN12  08BC  GN15  08C0
GN13  08C2  GN14  08C7  INIT  08CA  MOP0  08D9  MOP1  08DC  MOP2  08E2
MOP3  08E4  MOP4  08F0  MOP5  08F2  MOP6  08FD  MOP7  0900

```

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 Appleliars, 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 VV = 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.


```

1 REM *****
2 REM *
3 REM * SPEED TYPING TEST *
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
16 REM DEFINE VV= VTAB & TT= TAB
17 REM THEN GOSUB 8400- THIS DOES THE
18 REM SAME AS AN ORDINARY INPUT W$
20 TEXT : CALL -936
21 REM CAN BE GIVEN AWAY
22 PRINT "MICRO/APPLE VOLUME 1"
24 VTAB 3: PRINT "SEE 'SPEED TYPING TEST"
26 VTAB 4: PRINT " WITH INPUT TIME CLOCK'"
28 VTAB 6: PRINT "BY JOHN BRODERICK"
30 IF PEEK (-16384)<>160 THEN 30
40 DIM TYPE$(250): POKE 33,36
45 CALL -936
80 INPUT "DO YOU WISH TO MAKE UP YOUR OWN TEST SENTENCE Y/N ?",TYPE$
84 IF TYPE$# "Y" THEN 90: PRINT : PRINT "ENTER TEST SENTENCE NOW": PRINT
: PRINT : INPUT TYPE$: GOTO 100
90 TYPE$="NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR COUN
TRY"
100 CALL -936: PRINT :ERR=0: PRINT "YOU ARE TAKING A SPEED TYPING TEST!"
120 PRINT "TYPE THE NEXT SENTENCE APPEARING ON THE SCREEN AS FAST AS YOU CAN
"
130 FOR I=1 TO 4000: NEXT I: REM
135 REM ---BODY OF PROGRAM---
140 CALL -936:ERR=0
150 VV=13: REM SET SUBPOINT VTAB
160 TT=1: REM SET SUBROUTINE TYAB
170 VTAB (9): TAB 1: PRINT TYPE$: GOSUB 8400
180 VTAB (16): TAB 1
200 IF W$=TYPE$ THEN 510: REM
204 REM COMPUTE ERRORS 210- 410
210 FOR I= LEN(W$) TO LEN(TYPE$):W$(I+1)=B$(1,1): NEXT I
220 FOR I=1 TO LEN(TYPE$): IF I> LEN(W$) THEN ERR=ERR+1: IF I> LEN(W$) THEN
NEXT I
230 IF W$(I,I)#TYPE$(I,I) THEN ERR=ERR+1: NEXT I
400 PRINT : PRINT : CALL -198: PRINT " ";ERR;" ERRORS HIT RETURN": GOTO
520
410 CALL -198: PRINT " ";ERR;" ERRORS";" HIT RETURN"
500 REM - COMPUTE WPM
501 T=(X*23)+J:L= LEN(TYPE$): IF L<1 THEN 520
502 L=L-(ERR*6): IF L<2 THEN GOTO 506
503 WPM=(L*12*20)/T
506 VTAB (24): TAB 30: PRINT WPM;"WPM": VTAB (16): TAB 1: RETURN
510 PRINT "CORRECT- HIT RETURN": PRINT : PRINT : PRINT
520 GOSUB 500: INPUT W$:WPM=0: GOTO 140: REM
8400 REM - SUBROUTINE TO INPUT VIA KEYBOARD TO RETAIN AND INPUT WORD IN W$
8405 IF SWITCH=1 THEN 8407:SWITCH=1: DIM W$(255),A$(70),B$(2):B$=" "
8406 A$=" ! # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z "
8407 Y=1: POKE -16336,0:W$=" ":X=0:J=0
8410 FOR U=1 TO 250
8412 REM USER AREA HERE X= SECONDS SO USER CAN TEST X LIKE IF X=12 THEN RET
URN
8430 J=J+1: IF J<23 THEN 8434:X=X+1:J=0
8431 FOR BB=1 TO 3:KK= PEEK (-16336)- PEEK (-16336): NEXT BB: GOTO 8434
8434 VTAB (24): TAB 13:U=U-1: PRINT X;".";J*10/23;" SECONDS";:K= PEEK (-
16384)
8437 IF K#136 THEN 8444:Y=Y-1
8438 VTAB (VV): TAB TT+Y-1: PRINT B$(1,1)
8440 W$(1)=W$(1, LEN(W$)-1)
8441 VTAB (13): TAB 1: PRINT W$
8442 POKE -16368,0: NEXT U
8444 IF K=141 THEN 8540: IF K<160 THEN NEXT U
8447 K=K-159:W$(Y)=A$(K,K)
8461 POKE -16368,0: VTAB (VV): TAB TT: PRINT W$:Y=Y+1: NEXT U
8540 Y=1: CALL -756: 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 * LUDWIG *
7 REM * *
8 REM * COPYRIGHT (C) 1981 *
9 REM * MICRO INK, INC. *
10 REM * CHELMSFORD, MA 01824 *
11 REM * ALL RIGHTS RESERVED *
12 REM * *
14 REM *****
16 CALL -936
18 VTAB 5: PRINT "LUDWIG VON APPLE II"
19 IF PEEK (-16384)<>160 THEN 19
20 DIM A$(255)
30 POKE 2,173
40 POKE 3,48
50 POKE 4,192
60 POKE 5,165
70 POKE 6,0
80 POKE 7,32
90 POKE 8,168
100 POKE 9,252
110 POKE 10,165
120 POKE 11,1
130 POKE 12,208
140 POKE 13,4
150 POKE 14,198
160 POKE 15,24
170 POKE 16,240
180 POKE 17,5
190 POKE 18,198
200 POKE 19,1
210 POKE 20,76
220 POKE 21,2
230 POKE 22,0
240 POKE 23,96
300 A$="001100715211720172017201"
310 A$(25)="5211521152110071521100710012"
400 FOR Z=1 TO LEN(A$)-3 STEP 4
410 Z1=( ASC(A$(Z))-176)*( ASC(A$(Z+1))-176)+ ASC(A$(Z+2))-176
420 Z2=40-Z1
430 POKE 0,Z2
440 POKE 24, ASC(A$(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.)

0000-	0F	???	
0001-	00	BRK	
0002-	AD 30 C0	LDA	\$C030
0005-	A5 00	LDA	\$00
0007-	20 A8 FC	JSR	\$FCA8
000A-	A5 01	LDA	\$01
000C-	D0 04	BNE	\$0012
000E-	C6 18	DEC	\$18
0010-	F0 05	BEQ	\$0017
0012-	C6 01	DEC	\$01
0014-	4C 02 00	JMP	\$00002
0017-	60	RTS	
0018-	00	BRK	

```
>LIST
  10 POKE 0,5
  99 END
```

```
>CALL 2
```

```
>10 POKE 0,10
>RUN
```

```
>CALL 2
```

```
>10 POKE 0,15
>RUN
```

```
>CALL 2
```

7

REFERENCE

Introduction	180
An Apple II Programmer's Guide <i>Rich Auricchio</i>	181
Exploring the Apple II DOS <i>Andy Hertzfeld</i>	186
Applesoft II Shorthand <i>Allen J. Lacy</i>	191
The Integer BASIC Token System in the Apple II <i>Frank D. Kirschner</i>	198
Creating an Applesoft BASIC Subroutine Library <i>N.R. McBurny</i>	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.

An Apple II Programmer's Guide (You Can Get There from Here!)

by Rick Auricchio

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

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

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

Name	Loc.	Steps On	Function
PLOT	F800	AC	Plot a point. COLOR contains color in both halves of byte (\$00-\$FF). AC: y-coord, Y: x-coord.
CLRSCR	F832	AC,Y	Clear screen - graphics mode.
SCRN	F871	AC	Get screen color. AC: y-coord, Y: x-coord.
INSTDSP	F8D0	ALL	Disassemble instruction at PCH/PCL.
PRNTYX	F940	AC	Print contents of Y and X as 4 hex digits.
PRBL2	F94C	AC,X	Print blanks: X is number to print.
PREAD	FB1E	AC,Y	Read paddle. X: paddle number 0-3.
SETTXT	FB39	AC	Set TEXT mode.
SETGR	FB40	AC	Set GRAPHIC mode (GR).
VTAB	FC22	AC	VTAB to row in AC (0-\$17).
CLREOP	FC42	AC,Y	Clear to end-of-page.
HOME	FC58	AC,Y	Home cursor and clear screen.
SCROLL	FC70	AC,Y	Scroll up one line.
CLREOL	FC9C	AC,Y	Clear to end-of-line.
NXTA4	FCB4	AC	Increment A4 (16 bits), then do NXTA1.
NXTA1	FCBA	AC	Increment A1 (16 bits). Set carry if result >= A2.
RDKEY	FD0C	AC,Y	Get a key from the keyboard.
RDCHAR	FD35	AC,Y	Get a key, also handles ESCAPE functions.
GETLN	FD6A	ALL	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.
CROUT	FD8E	AC,Y	Print a carriage return.
PRBYTE	FDDA	AC	Print contents of AC as 2 hex digits.
COUT	FDED	AC,Y	Print character in AC; also works for CR, BS, etc.
PRERR	FF2D	AC,Y	Print "ERR" and bell.
BELL	FF3A	AC,Y	Print bell.
RESET	FF59	--	RESET entry to Monitor - initialize.
MON	FF65	--	Normal entry to 'top' of Monitor when running.
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.

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

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 \$AA53 - \$AA54 and the input hook immediately follows at \$AA55 - \$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 2: Important addresses in the Apple II DOS

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

(continued)

no.*	address to power up the disk
\$C088 + \$S0, S = slot	
no.*	address to power down the disk
\$BD00	routine which reads in the directory off the disk. It is called by virutally 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
7F51.7FFFR
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
HIMIM: 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:B0 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 key board, use the command IN#0. 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.

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

Control U	—
Control H	—
Control M	RETURN
Control J	Line feed
Control G	BELL
Control X	Kill input line
Control C	Stops a running program
Control D	Is used by DOS

Table 2

ROM VERSION: Add \$C800

8D0 END	8D3 FOR	8D6 NEXT	8DA DATA
8DE INPUT	8E3 DEL	8E6 DIM	8E9 READ
8ED GR	8EF TEXT	901 HLIN	905 VLIN
909 HGR2	90D HGR	910 HCOLOR =	917 H PLOT
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)

9EF THEN	9F3 AT	9F5 NOT	9F8 STEP
9FC +	9FD -	9FE *	9FF /
A00 ▲	A01 AND	A04 OR	A06 >
A07 =	A08 >	A09 SGN	A0C INT
A0F ABS	A12 USR	A15 FRE	A18 SCRN(
A1D PDL	A20 POS	A23 SQR	A26 RND
A29 LOG	A2C EXP	A2F COS	A32 SIN
A35 TAN	A38 ATN	A3B PEEK	A3F LEN
A42 STR\$	A46 VAL	A49 ASC	A4C CHR\$
A50 LEFT\$	A55 RIGHT\$	A5B MID\$	

```

0800      1  ;*****
0800      2  ;*
0800      3  ;* APPLESOFT SHORTHAND *
0800      4  ;*           BY           *
0800      5  ;*       ALAN LACY       *
0800      6  ;*
0800      7  ;*       SHORTHAND       *
0800      8  ;*
0800      9  ;* COPYRIGHT (C) 1981 *
0800     10  ;*   MICRO INK, INC.   *
0800     11  ;* ALL RIGHTS RESERVED *
0800     12  ;*
0800     13  ;*****
0800     14  ;
0800     15  ZP      EPZ $1E           ;R15 OF SWEET 16
0800     16  ;
0800     17  ;*****
0800     18  ;* LOCATIONS 478-47F NOT *
0800     19  ;* USED BY SCREEN DISPLAY *
0800     20  ;*****
0800     21  ;
0800     22  SW      EQU $0478         ;SWITCH
0800     23  CT      EQU $0479         ;CHAR COUNT
0800     24  XSAV    EQU $047A
0800     25  YSAV    EQU $047B
0800     26  POIN    EQU $047C         ;POINTER
0800     27  ZPS     EQU $047E
0800     28  ;*
0800     29  RKEY    EQU $FD1B         ;KEY READ CODE
0800     30  SW16    EQU $F689         ;SWEET 16
0300     31  ORG     $300
0300     32  OBJ     $800
0300     33  ;
0300     34  ;*****
0300     35  ;*
0300     36  ;* START LOCATION OF SHORTHAND *
0300     37  ;*
0300     38  ;*****
0300     39  ;
0300     40  SH      STX XSAV           ;SAVE X REG
0303     41  STY YSAV           ;SAVE Y REG
0306     42  PHA             ;SAVE ACC
0307     43  LDA ZP            ;SAVE ZERO PAGE
0309     44  STA ZPS           ; LOCATIONS
030C     45  LDA ZP+1
030E     46  STA ZPS+1
0311     47  ;

```

```

0311          48 ;*****
0311          49 ;* SWEET 16 IS USED TO STORE *
0311          50 ;* KP PROGRAM COUNTER IN $1E *
0311          51 ;* $1F AND THIS IS USED TO *
0311          52 ;* FIND THE LOCATION OF THE *
0311          53 ;* TABLE IN SHORTHAND..... *
0311          54 ;*****
0311          55 ;
0311 2089F6   56          JSR SW16          ;ENTER SWEET16
0314 00      57          HEX 00          ;LEAVE SWEET16
0315          58 KP      EQU *          ;KNOWN POINT
0315 AD7804   59          LDA SW
0318 D03B     60          BNE NBYT        ;CHECK SW
031A 68      61          PLA          ;RESTORE ACC
031B 201BFD   62          JSR RKEY        ;READ A KEY
031E 48      63          PHA          ;STORE KEY VAL
031F C99B     64          CMP #$9B        ;CONTROL KEY?
0321 9014     65          BCC CTR
0323          66 ;*
0323          67 ;*****
0323          68 ;*
0323          69 ;* IF NOT A CONTROL, JUST RETURN *
0323          70 ;*
0323          71 ;*****
0323          72 ;*
0323 68      73 RET      PLA          ;RESTORE KEY
0324 AE7A04   74 RT      LDX XSAV        ;RESTORE X REG
0327 AC7B04   75          LDY YSAV        ;RESTORE Y REG
032A 48      76          PHA
032B AD7E04   77          LDA ZPS          ;RESTORE ZERO
032E 851E     78          STA ZP          ;PAGE LOCATIONS
0330 AD7F04   79          LDA ZPS+1
0333 851F     80          STA ZP+1
0335 68      81          PLA
0336 60      82          RTS
0337 297F     83 CTR     AND #$7F        ;WHICH KEY
0339 0A      84          ASL          ;TIMES 2
033A 6964     85          ADC #TAB-KP    ;OFFSET FROM KP
033C A8      86          TAY
033D C8      87          INY
033E B11E     88          LDA (ZP),Y      ;LOAD ENTRY
0340          89 ;
0340          90 ;*****
0340          91 ;*
0340          92 ;* IF VALUE OF THE HIGH BYTE IN *
0340          93 ;* TABLE IS 0 THEN RETURN THE *
0340          94 ;* CONTROL CHAR ELSE SET UP TO *
0340          95 ;* RETURN THE CHARACTERS FROM *
0340          96 ;* APPLESOFT'S INTERNAL TABLE. *
0340          97 ;*
0340          98 ;*****
0340          99 ;
0340 F0E1     100         BEQ RET          ;IF 0 RETURN
0342 8D7D04   101        STA POIN+1        ;STORE IN POIN
0345 88      102        DEY
0346 B11E     103        LDA (ZP),Y
0348 8D7C04   104        STA POIN
034B A9FF     105        LDA #$FF        ;SET SW
034D 8D7804   106        STA SW
0350 A900     107        LDA #0          ;SET CT TO 0
0352 8D7904   108        STA CT
0355          109 ;*
0355          110 ;*****
0355          111 ;*
0355          112 ;* NEYT IS USED TO PASS THE CHAR-*
0355          113 ;* ACTERS FROM THE TABLE IN APPL-*
0355          114 ;* SOFT AS IF THEY WERE TYPED IN *
0355          115 ;*
0355          116 ;*****
0355          117 ;*
0355 68      118 NEYT   PLA
0356 AC7904   119        LDY CT          ;LOAD CHAR CT
0359 AD7C04   120        LDA POIN        ;STORE POIN IN
035C 851E     121        STA ZP          ;ZERO PAGE

```

```

035E AD7D04 122 LDA POIN+1
0361 851F 123 STA ZP+1
0363 B11E 124 LDA (ZP),Y ;LOAD NEXT CHAR
0365 C980 125 CMP #\$80 ;LAST CHAR?
0367 B007 126 BCS END
0369 0980 127 ORA #\$80
036B EE7904 128 INC CT ;INCREMENT CT
036E D0B4 129 BNE RT ;RETURN CHAR
0370 48 130 END PHA ;SAVE CHAR
0371 A900 131 LDA #0 ;RESET SW
0373 8D7804 132 STA SW
0376 68 133 PLA ;RESTORE CHAR
0377 D0AB 134 BNE RT ;RETURN CHAR
0379 135 ;*
0379 136 ;*****
0379 137 ;* *
0379 138 ;* TABLE TO STORE ADDRESSES OF *
0379 139 ;* COMMANDS IN APPLESOFT II *
0379 140 ;* *
0379 141 ;* WILL HAVE TO BE CHANGED FOR *
0379 142 ;* ROM VERSION *
0379 143 ;* *
0379 144 ;*****
0379 145 ;*
0379 0000 146 TAB ADR \$000 ;@
037B F908 147 ADR \$8F9 ;A CALL
037D 3B0A 148 ADR \$A3B ;B PEEK
037F 0000 149 ADR \$000 ;C
0381 0000 150 ADR \$000 ;D
0383 EF08 151 ADR \$8EF ;E TEXT
0385 D308 152 ADR \$8D3 ;F FOR
0387 0000 153 ADR \$000 ;G
0389 0000 154 ADR \$000 ;H
038B DE08 155 ADR \$8DE ;I INPUT
038D 0000 156 ADR \$000 ;J
038F D009 157 ADR \$9D0 ;K CONT
0391 D409 158 ADR \$9D4 ;L LIST
0393 0000 159 ADR \$000 ;M
0395 D608 160 ADR \$8D6 ;N NEXT
0397 EF09 161 ADR \$9EF ;O THEN
0399 FD08 162 ADR \$8FD ;P PLOT
039B 0109 163 ADR \$901 ;Q HLIN
039D 5B09 164 ADR \$95B ;R COLOR=
039F A409 165 ADR \$9A4 ;S GOSUB
03A1 9309 166 ADR \$993 ;T GOTO
03A3 0000 167 ADR \$000 ;U
03A5 6409 168 ADR \$964 ;V VTAB
03A7 0509 169 ADR \$905 ;W VLIN
03A9 0000 170 ADR \$000 ;X
03AB 2509 171 ADR \$925 ;Y HTAB
03AD C709 172 ADR \$9C7 ;Z POKE
173 LS END

```

***** END OF ASSEMBLY

```

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

```

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

** ZERO PAGE VARIABLES:

ZP 001E

** ABSOLUTE VARIABLES/LABELS

SW	0478	CT	0479	XSAV	047A	YSAV	047B	POIN	047C				
ZPS	047E	RKEY	FD1B	SW16	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

EXAMPLES FOR
16K Apple

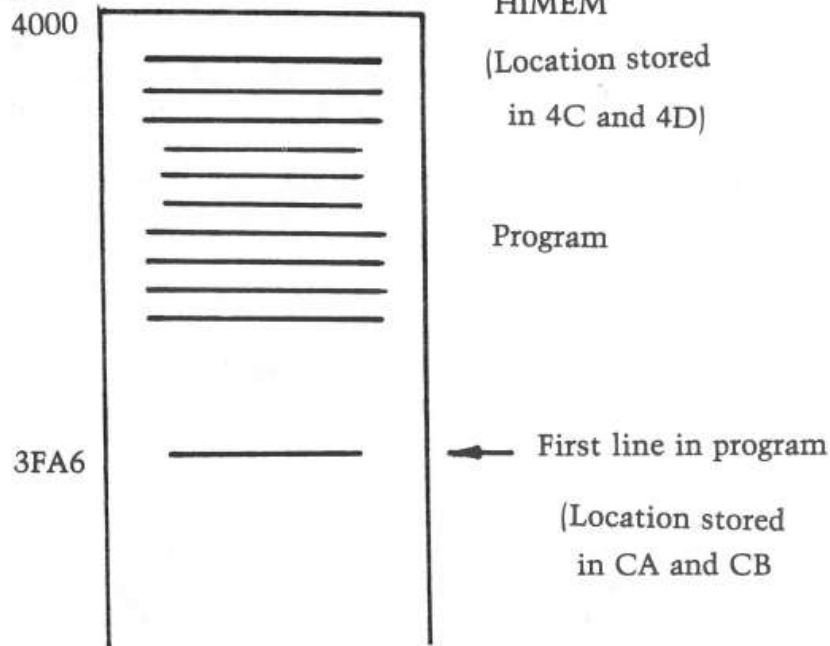


Figure 1: Memory Map for Program Storage

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

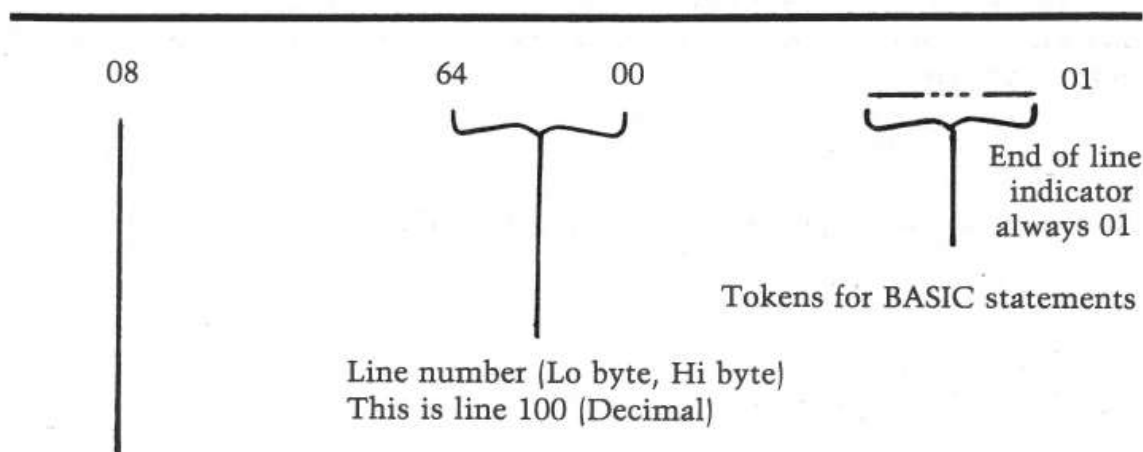
To demonstrate the use of this information, return to BASIC and try to enter the following BASIC line:

```
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.



Number of bytes in BASIC line (also, one less than the number of bytes from the beginning of the next line).

Figure 2

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: IF
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 = 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 FF)
    PEEK (I + 2) GOSUB 65535
(FE 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

BASIC Command or Function	Hex Token	BASIC Command (Continued)	Hex Token
ABS	31	LOAD	04
(3F	MAN	0F
)	72	NEW	0B
ASC (3C Includes left parenthesis	NEXT	59
)	72	,	5A
"	28 First quote	NO DSP	79
"	29 Second quote	NO TRACE	7A
AUTO	0D	PDL	32
,	0A	(3F
CALL	4D)	72
CLR	0C	PEEK	2E
COLOR =	66 Includes =	3F	(
CON	60	72)
DEL	09	PLOT	67
,	0A	,	68
DIM	4F Numeric arrays	POKE	64

(continued)

(34	,	65
)	72	POP	77
DIM	4E String array	PRINT	63 If used alone
(22	PRINT	62 Numeric variable
)	72	:	46
\$	40	,	49
DSP	7C Numeric variable	PRINT	61 String variable
DSP	7B String variable	"	28 First
END	51	"	29 Second
FOR	55	PR#	7E Includes #
=	56	REM	5D
TO	57	RETURN	5B
STEP	58	RND	2F
GOSUB	5C	(3F
GOTO	5F)	72
GR	4C	-	36
HIMEM:	10 Includes :	SAVE	05
HLIN	69	SCRN (3D Includes(
,	6A	,	3E
AT	6B)	72
IF	60	SGN	30
THEN	24 When followed by a line no.	(3F
)	72
THEN	25 When followed by GOSUB or a BASIC operation	TAB	50
		TEXT	4B
INPUT	54 Numeric variable	TRACE	7D
INPUT	52 String variable	VLIN	6C
INPUT	53 Input if followed by ...	,	6D
	27	AT	6E
,	27	VTAB	6F
"	28 First	:	03
"	29 Second	=	71 In assignment
IN #	7F Includes #	AND	1D
LEN (3B Includes (OR	1E
LET	5E	MOD	1F
LIST	74	NOR	DE
	75		

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

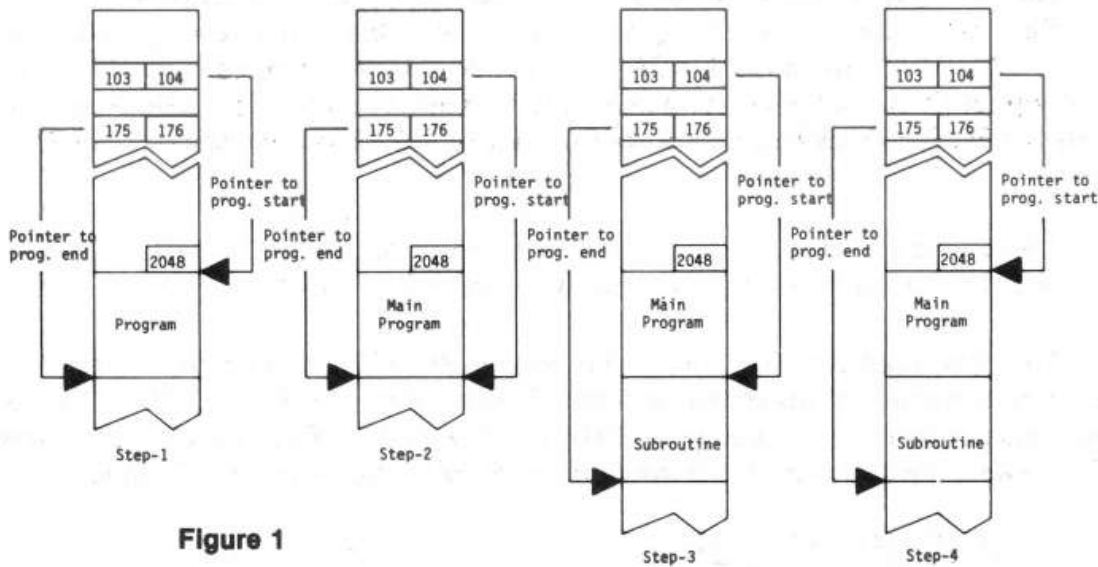


Figure 1

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,I-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):

```
] 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:

```
] 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 (\$300-\$ 33D). 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.

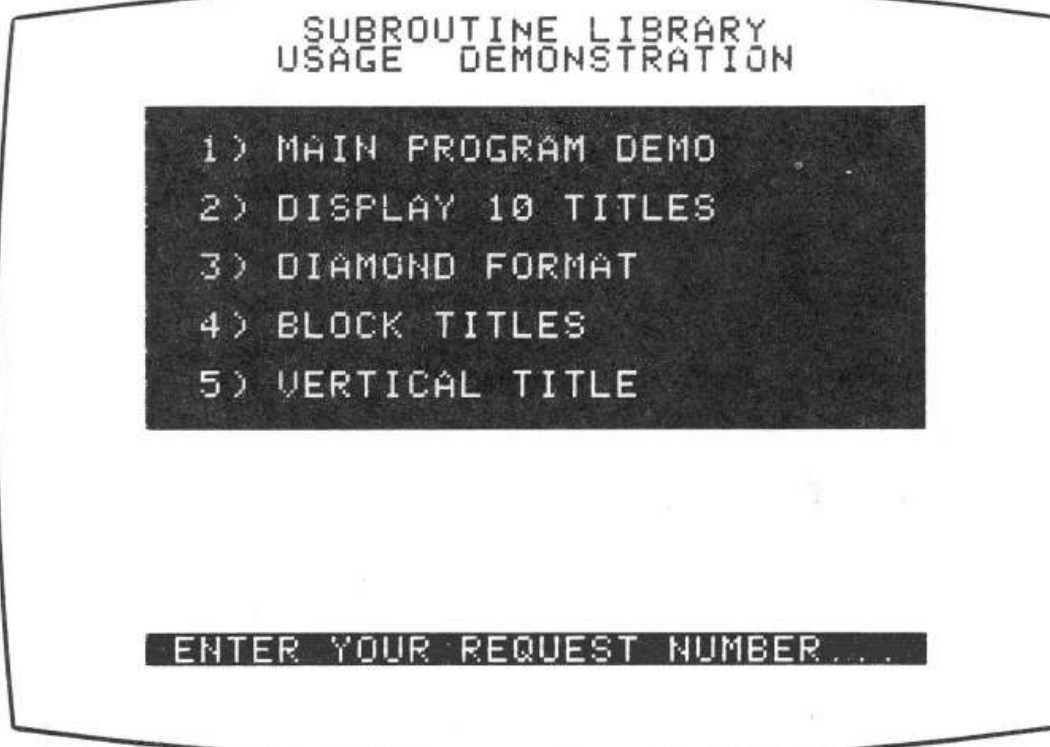


Figure 2

Listing 1: Generalized EXEC File Writer Program

```

1000 REM *****
1001 REM *
1002 REM * CREATING AN APPLESOFT *
1003 REM * SUBROUTINE LIBRARY *
1004 REM * N.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 *
1012 REM * ALL RIGHTS RESERVED *
1013 REM *
1014 REM *****
1016 REM GENERALIZED EXEC FILE WRITER:
1020 REM QUOTES IN THE DATA STATEMENTS ARE USED ONLY AS DELIMITERS AND
WILL NOT
1030 REM APPEAR ON THE EXEC FILE. APOSTROPHES IN THE DATA STATEMENTS WI
LL APPEAR
1050 REM AS QUOTATION MARKS IN THE EXEC FILE.
1060 REM
1070 D$ = CHR$(4)
1080 HOME : PRINT CHR$(7)
1090 INPUT "NAME FOR EXEC FILE?";FILES$
1100 HOME
1110 PRINT D$;"MON O"
1120 PRINT D$;"OPEN";FILES$
1130 PRINT D$;"DELETE";FILES$
1140 PRINT D$;"OPEN";FILES$
1150 PRINT D$;"WRITE";FILES$
1160 ONERR GOTO 1250
1161 REM
1162 REM READ IN LINE AND REPLACE
1163 REM APOSTROPHES WITH QUOTES
1164 REM
1170 READ S$
1180 A$ = ""
1190 FOR I = 1 TO LEN (S$)
1200 IF MID$(S$,I,1) < > "'" THEN A$ = A$ + MID$(S$,I,1)
1210 IF MID$(S$,I,1) = "'" THEN A$ = A$ + CHR$(34)
1220 NEXT
1230 PRINT A$
1240 GOTO 1170
1241 REM
1245 REM CHECK FOR CORRECT ERROR CODE
1246 REM (#42=OUT OF DATA)
1247 REM
1250 IF PEEK (222) = 42 THEN GOTO 1255
1251 PRINT "ERROR #"; PEEK (222)
1252 PRINT "IN LINE #" + PEEK (218) + PEEK (219) * 2556
1253 STOP
1255 POKE 216,0
1259 PRINT D$;"NOMON O"
1260 PRINT D$;"CLOSE";FILES$
1270 REM
1280 REM BEGIN DATA STATEMENTS DEFINING
1290 REM TEXT TO BE PLACED IN EXEC FILE
1300 REM
1310 DATA "I=PEEK(176)*256+PEEK(175)-2:POKE 104,INT(I/256):POKE 103,I-IN
T(I/256)*256"
1320 DATA "LOAD SUBROUTINES"
1330 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 103,1: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

```

MAINS$="":FORI=1TO30:MAINS$=MAINS$+CHR$(PEE
K(767+I)):NEXT:PRINT CHR$(4);"LOAD ";MAI
NS$
I=PEEK(176)*256+PEEK(175)-2:POKE 104,INT
(I/256):POKE 103,I-INT(I/256)*256
SUBR$="":FORI=1TO30:SUBR$=SUBR$+CHR$(PEE
K(798+I)):NEXT:PRINT CHR$(4);"LOAD ";SUB
R$
POKE 103,I:POKE 104,8
RUN

```

Listing 5: MAIN Program

```

100 REM
110 REM DEMONSTARTION MAIN PROGRAM
120 REM
130 HOME
140 TITLE$ = "FIRST LINE OF TITLE"
150 GOSUB 10000
160 TITLE$ = "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 (TITLE$)
10050 PRINT TAB( 20 - L / 2);TITLE$
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);"USEAGE 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 A$ = " ENTER YOUR REQUEST NUMBER... "
1240 VTAB 22
1250 HTAB 5
1260 A$ = MID$( A$,2) + LEFT$( A$,1)
1270 PRINT A$
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 CHR$( 7): GOTO 1240
1460 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-$33D.
1510 REM
1520 K1 = 767:K2 = 798
1530 SUBR$ = "SUBROUTINES
1540 MAIN$ = LEFT$( MAIN$ + "
1550 FOR I = 1 TO 30
1560 POKE K1 + I, ASC ( MID$( MAIN$,I,1))
1570 POKE K2 + I, ASC ( MID$( SUBR$,I,1))
1580 NEXT
1590 HOME
1600 PRINT CHR$( 7)
1610 PRINT CHR$( 4);"EXEC MASTER MERGE"

```


Language Index

APPLESOFT BASIC

RENUMBER	Applesoft Renumbering, Childress	8
SEARCH/CHANGE	Search/Change, Childress	15
DATA-GEN	Data Statement, Brady	46
EDIT MASK	Edit Mask, Reynolds	53
DOLLAR MASK	Business Dollars, Bauers	61
CHECK PROTECT	Business Dollars, Bauers	61
LC INSERT	Lower Case, Childress	65
LC ENTRY	Lower Case, Childress	66
FUNCTION GRAPH	Graphing, Rational Functions, Carlson	73
SHAPE 1	How to Do a Shape Table, Figueras	91
SHAPE 2	How to Do a Shape Table, Figueras	92
SHAPE 3	How to Do a Shape Table, Figueras	94
CHARACTERS	Define Hi-Res Characters, Zant	98
SOLAR	Solar System, Partyka	140
EXEC FILE WRITER	Creating a Subroutine Library, McBurney	208

INTEGER BASIC

EDIT	Program Edit Aid, Hill	19
ALARM PROMPT	Alarming Apple, Irwin	39
GRAPH PLOT	Hi-Res Graph Plot, Fam	77
GRAPHICS-ORG	Hi-Res Graphics Memory, Eliason	102
APPLE PI	Apple Pi, Bishop	107
BUBBLE	Sorting Revealed, Vile	126
INSERT	Sorting Revealed, Vile	127
SELECT	Sorting Revealed, Vile	128
SHELL	Sorting Revealed, Vile	130
QUICK	Sorting Revealed, Vile	131
SPELUNKER	Spelunker, Mimlitch	153
APPLE LIFE	Life for Your Apple, Suitor	169
TYPE TEST	Speed Typing Test, Broderick	174
LUDWIG	Ludwig Von Apple, Schwartz	176

MACHINE LANGUAGE

FIND	Program Edit Aid, Hill	20
EDIT PLUS	A Little Plus, Peterson	27
ZOOM	Zoom and Squeeze, Little	32
SLOW LIST	Slow List, Sander-Cederlöf	35
ALARM	Alarming Apple, Irwin	38
LIFE	Life for Your Apple, Suitor	170
SHORTHAND	Applesoft Shorthand, Lacy	195

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.....105
 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

MICRO/APPLE

Volume 1

Recorded in 16 sector DOS 3.3 format.

Copyright © 1981 by MICRO INK, Inc.
P.O. Box 6502
Chelmsford, MA 01824
All rights reserved

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.

diskette
includes

SPELUNKER

the hit
game

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!**

With this volume, Micro Ink, Inc., publisher of MICRO, The 6502 Journal, inaugurates a series of books entitled MICRO/Apple.

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)**

ISSN 0275-3537
ISBN 0-938222-05-8

**MICRO INK, Inc.
P.O. Box 6502
Chelmsford, Massachusetts 01824**