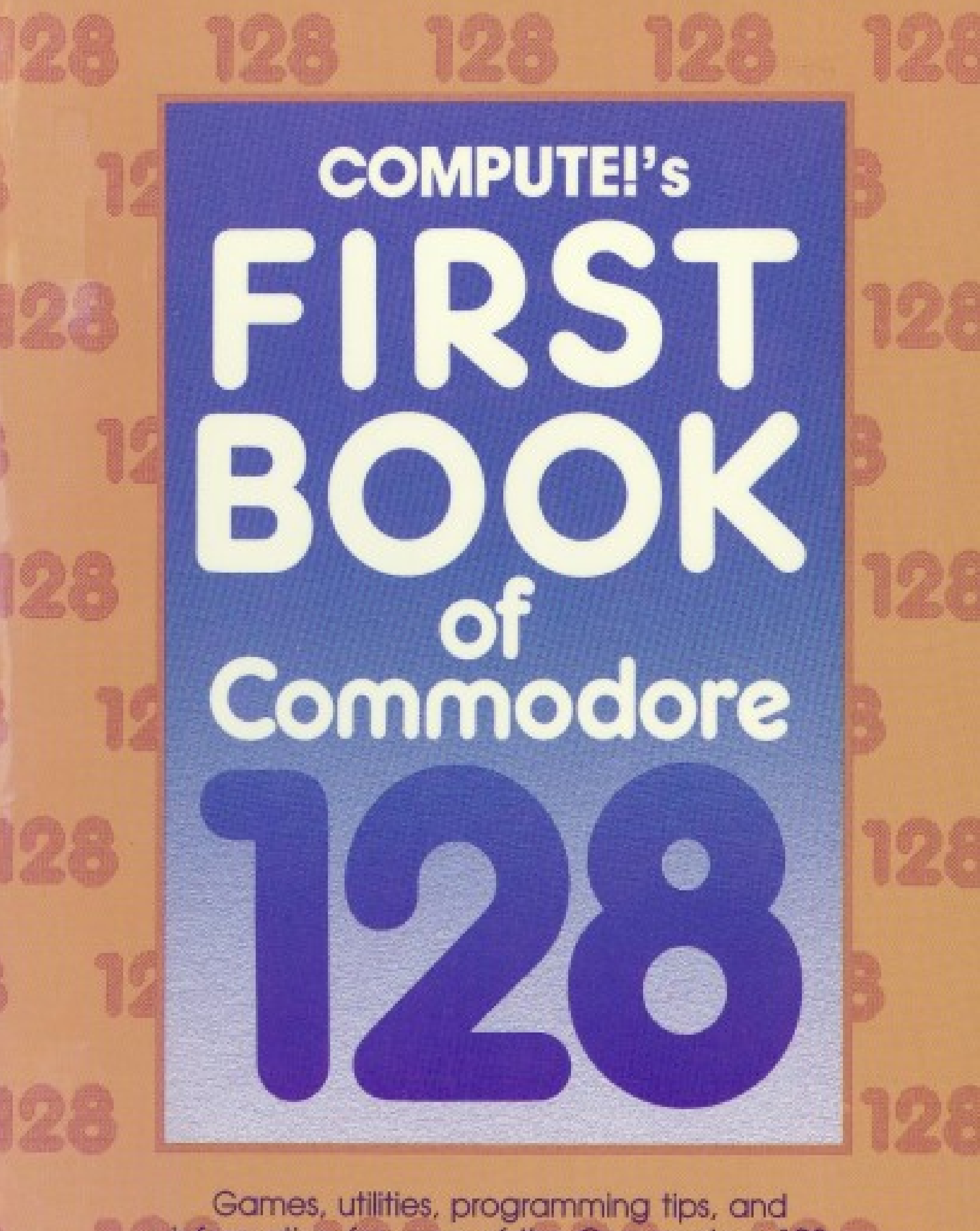
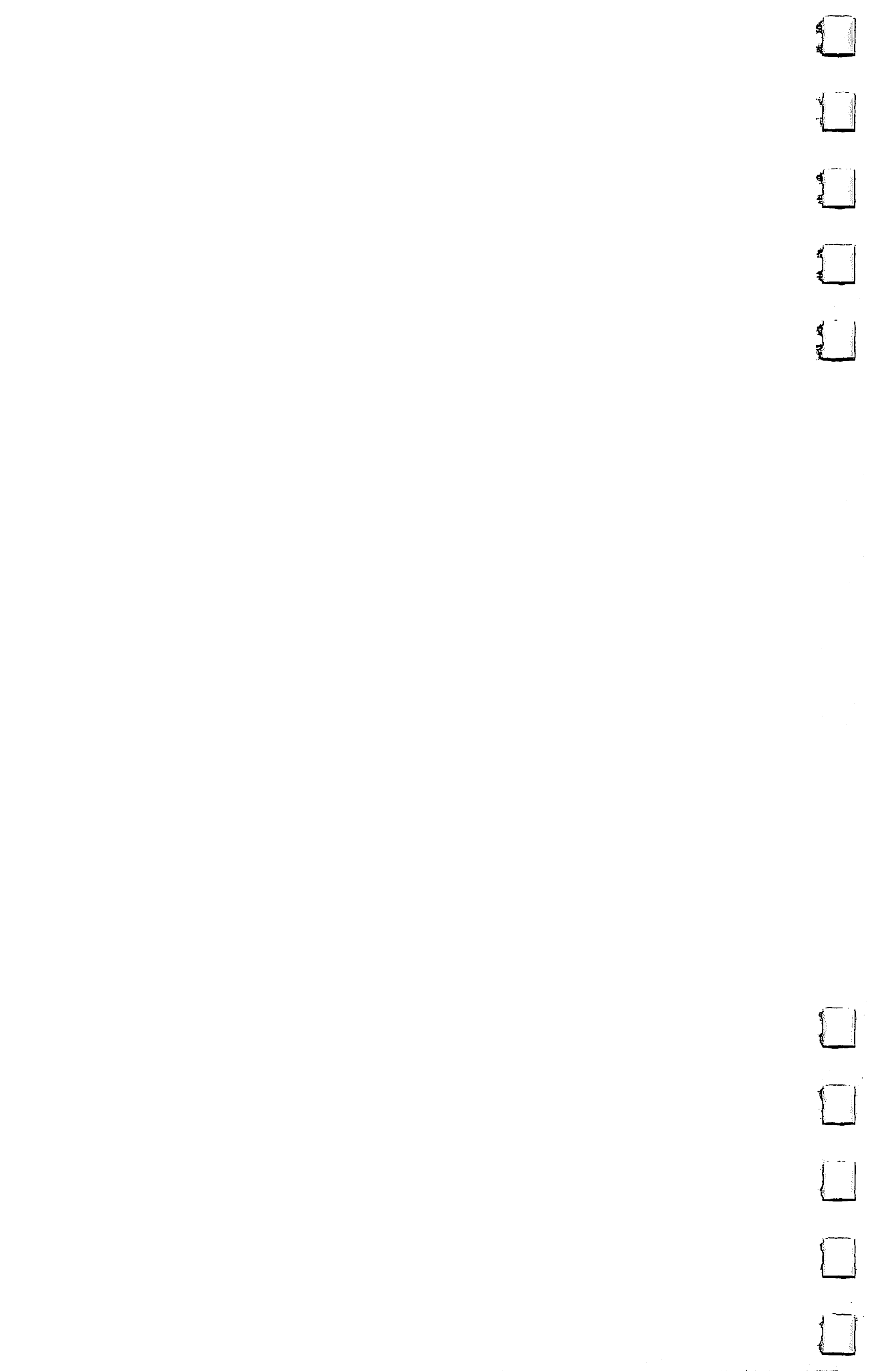


Games, utilities, programming tips, and information for users of the Commodore 128 personal computer in 128 mode.

A **COMPUTE! Books** Publication $14.95



COMPUTE!'s FIRST BOOK OF THE COMMODORE 128

COMPUTE!™ Publications, Inc. abc
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

The following articles were originally published in *COMPUTE!* magazine, copyright 1985, COMPUTE! Publications, Inc.:
"Sound and Music," originally titled "Sound and Music on the Commodore 128" (August and September); "Jump Search" (September); "Word Search" (September); "Save-with-Replace: Debugged at Last" (October and November); "Dynamic Keyboard," originally titled "Dynamic Keyboard for Commodore Machines" (October, November, and December); "Advanced Commodore 128 Video" (December).

The following articles were originally published in *COMPUTE!'s Gazette* magazine, copyright 1985, COMPUTE! Publications, Inc.:
"Litter Patrol" (September); "Exploring BASIC 7.0," originally titled "Exploring 128 BASIC: An End to PEEKs and POKEs" (November).

The following article was originally published in *COMPUTE!* magazine, copyright 1986, COMPUTE! Publications, Inc.:
"Switchbox" (March).

The following articles were originally published in *COMPUTE!'s Gazette* magazine, copyright 1986, COMPUTE! Publications, Inc.:
"Programming Music and Sound," originally titled "Programmng Music and Sound on the 128" (January); "REM Highlighter" (January); "Exploring the 128's Monitor" (February); "Important 128 Memory Locations," originally titled 'Commodore 128 Memory Map Important Locations" (February); "Lexitron" (February); "Disk Commands," originally titled "Disk Commands on the 128" (February); "Autoboot," originally titled "128 Autoboot" (March); "Cataloger" (March); "Storage and Display," originally titled "Storage and Display: Using Peripherals with the 128" (March); "All About CP/M on the 128" (April); "Windows on the 128" (April); "Word Counter" (May); "Blick," originally titled "Power BASIC: Blick" (May); "Coder-Decoder" (May).

Copyright 1986, COMPUTE! Publications, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-059-9

The authors and publisher have made every effort in the preparation of this book to insure the accuracy of the programs and information. However, the information and programs in this book are sold without warranty, either express or implied. Neither the authors nor COMPUTE! Publications, Inc., will be liable for any damages caused or alleged to be caused directly, indirectly, incidentally, or consequentially by the programs or information in this book.

The opinions expressed in this book are solely those of the authors and are not necessarily those of COMPUTE! Publications, Inc.

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Commodore 64, Commodore 128, and VIC-20 are trademarks of Commodore Electronics Limited. CP/M is a registered trademark of Digital Research, Inc.

Contents

Foreword v

Chapter 1. Programming 1
Exploring BASIC 7.0
Todd Heimarck 3
Save-with-Replace: Debugged at Last
P. A. Slaymaker 17
Dynamic Keyboard
Jim Butterfield 28
Jump Search
Jerry Sturdivant 37
Coder-Decoder
W. M. Shockley 40
Exploring the 128's Monitor
Richard Mansfield 43
Important 128 Memory Locations
Jim Butterfield 46
All About CP/M on the 128
Howard Golk 51

Chapter 2. Sound and Graphics 61
Windows on the 128
Jim Vaughan 63
Advanced Commodore 128 Video
Jim Butterfield 69
Programming Music and Sound
Philip I. Nelson 74
Sound and Music
Philip I. Nelson 81

Chapter 3. Games 99
Orbitron
Mark Tuttle and Kevin Mykytyn 101
Litter Patrol
Charles Brannon 110
Word Search
Michael B. Williams
128 Version by Patrick Parrish 120

Switchbox
Todd Heimarck 126
Lexitron
Ron Wilson 135

Chapter 4. Utilities 141
MetaBASIC: Programmer's Problem Solver
Kevin Mykytyn 143
REM Highlighter
Don A. Ellis 154
Blick
Plummer Hensley 157
Word Counter
Thomas K. Tucker 159

Chapter 5. Peripherals 163
Storage and Display 165
Disk Commands
Todd Heimarck 170
Cataloger
Kevin Mykytyn 180
Autoboot
Steve Stiglich 186

Appendices 195
A. How to Type In Programs 197
B. The Automatic Proofreader
Philip I. Nelson 199
C. Machine Language Editor, MLX
Ottis R. Cowper 203

Index 213

Disk Coupon 217

Foreword

COMPUTE!'s First Book of the Commodore 128 is packed full of information and programs ready to type in and run in 128 mode.

COMPUTE! Publications is the leading publisher of type-in programs and articles for Commodore users. In *COMPUTE!'s First Book of the Commodore 128* we've collected our best programs and articles for the 128 from *COMPUTE!* magazine and *COMPUTE!'s Gazette* and added some never before published programs.

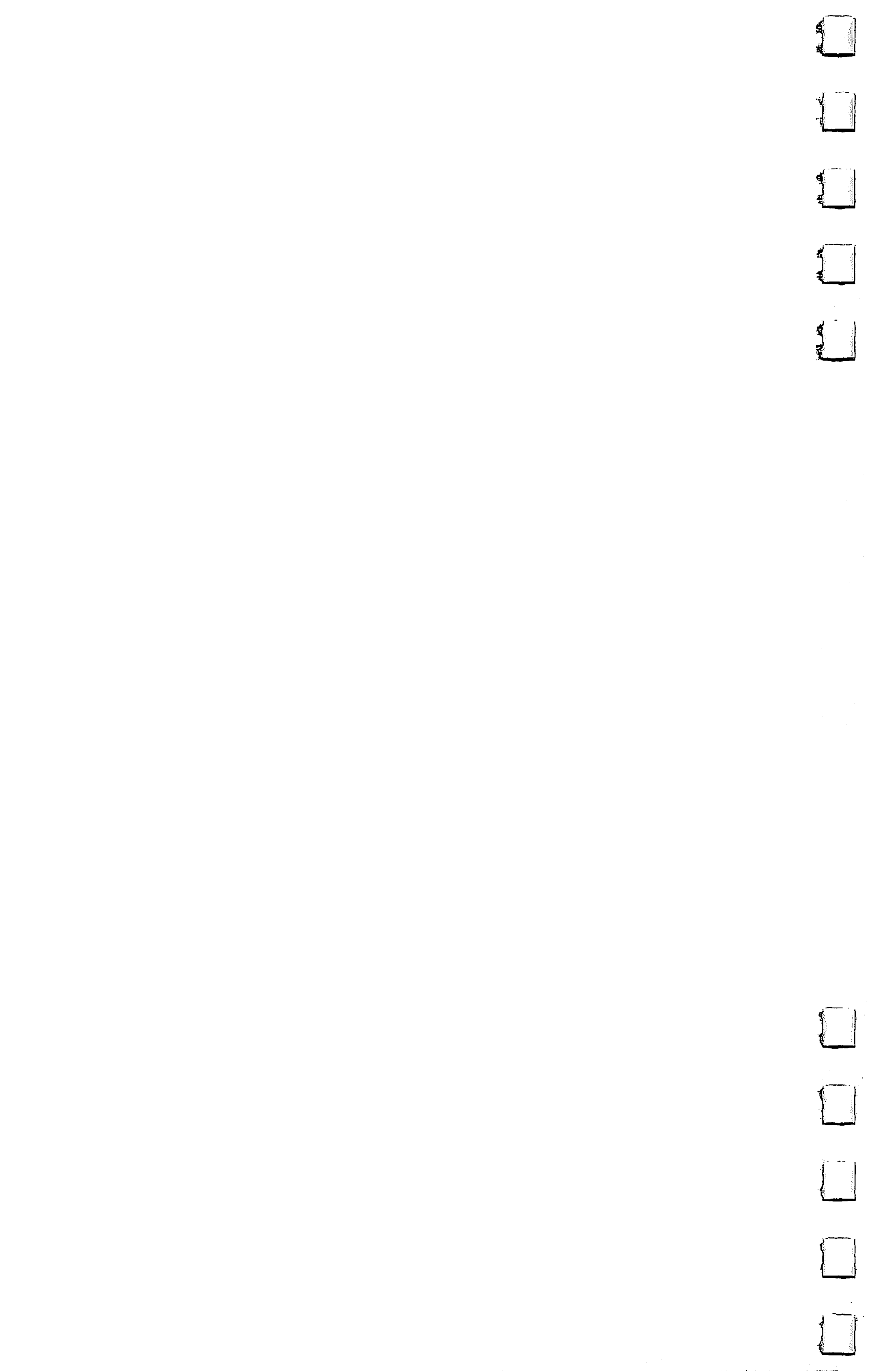
Game players will enjoy the challenge and thrill of playing "Orbitron," a fast-paced game for two players that requires careful planning, and "Switchbox," a strategy game for one or two players.

Programmers will enjoy the added convenience of "MetaBASIC," a utility that adds 11 new commands to BASIC, such as Find, Merge, Dlist, and Resave. They'll also enjoy articles and programs that illustratate how to add windows, sound, and music to programs.

Also included are easy-to-use programs to catalog your disk library, make any program on a disk automatically load and run when you turn on your 128, and much more.

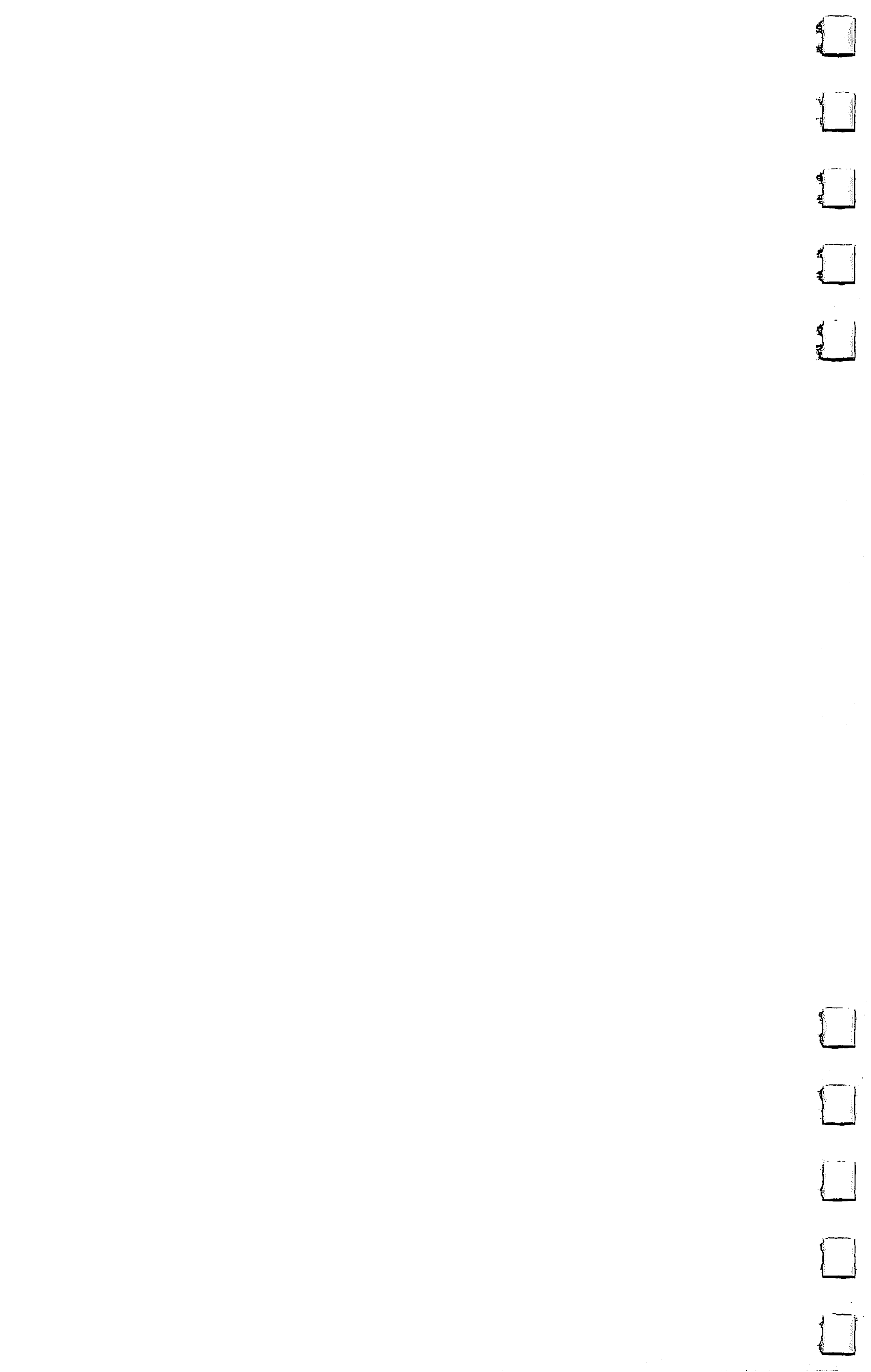
The articles are clearly and concisely written. And we've included "The Automatic Proofreader" and "MLX, the Machine Language Editor" to help you avoid errors when typing in the programs. As with all COMPUTE! publications, each program has been carefully and thoroughly tested.

All the programs in *COMPUTE!'s First Book of the Commodore 128* are ready to type in and run. If you prefer not to type in the programs, however, you can order a disk which includes all the programs in this book by calling toll-free 800-346-6767 (in New York, 212-887-8525), or by using the coupon found in the back of the book.



Chapter 1

Programming



Exploring BASIC 7.0

Todd Heimarck

BASIC 7.0 is the most powerful Commodore BASIC to date. If you learned programming on a VIC or 64, you'll appreciate the many new commands which greatly simplify programming. In a line or two, you can accomplish what might have taken five or ten on a VIC or 64.

The Commodore 128 has two things going for it. The first is 122,365 *available* bytes of memory, twice as much memory as a Plus/4, three times as much as a 64, and 34 times as much as an unexpanded VIC. If you wish, you can add still more memory—the 128 is expandable to 512K.

The second and even more impressive feature is the new BASIC 7.0, which virtually eliminates the need for PEEKs and POKEs. BASIC programmers can forget about looking up the memory locations for high-resolution graphics or trying to remember how to define a sprite. (Machine language programmers, of course, will still be concerned with how PEEKs and POKEs affect the computer.)

The 128's BASIC includes all commands from BASIC 2.0 (the language in the VIC and 64), all commands except one (RLUM) from BASIC 3.5 (found in the Plus/4 and 16), the disk commands from the Commodore PET, plus many brand new ones.

If you're experienced at programming the VIC or 64, you'll enjoy exploring the new BASIC. Let's look at what you can do with the 128.

Sprites and Music Without POKEs

The Plus/4 has more memory and a better BASIC than the 64. But Commodore eliminated two very popular features of the 64: sprites and the SID chip. Sprites, objects which can be moved independently around the screen, are often used as characters in games. The Sound Interface Device (SID chip), best described as a minisynthesizer, can produce sounds and music that would be impossible on computers with simple tone generators.

Creating sprites on the 64 can be difficult, even if you know what you're doing. First, you have to convert the shape to DATA statements, either on graph paper or with a sprite editor. Next, each sprite needs several POKEs to set x and y positions, colors, expansion, priorities, and so on.

The 128 makes sprites easy. A sprite editor program is built in—just enter SPRDEF to turn it on. Draw the sprite on the screen, and the shape can be immediately saved to memory. You then use the SPRITE command to turn it on, give it a color, and set priority, expansion, and multicolor mode. MOVSPR moves it to a specific position on the screen. You can also include a speed and direction—the sprite will move automatically, until you tell it to stop.

Another option is to draw a shape on the hi-res screen, save it into a variable with SSHAPE (Save SHAPE), and put that shape into a sprite with SAVSPR. The shape can be stored to disk through BSAVE or by putting the SSHAPEd variable into a sequential file.

While the sprite is moving around the screen, you can check for collisions with COLLISION. This command works like a conditional GOSUB. When a sprite hits an object on the screen, the program automatically goes to a subroutine. Within the subroutine, BUMP tells you which sprites are involved. There are also commands for reading sprite colors and positions. It's all done without a single POKE.

Music is just as easy to program. The new statements give you much easier access to the capabilities of the SID chip. PLAY "DEF", for example, plays the notes D, E, and F. There are three voices, six octaves, and ten envelopes (including piano, accordion, calliope, drum, flute, guitar, harpsichord, organ, trumpet, and xylophone). The notes of the melody can range from sixteenths to whole notes. You can define your own instruments with ENVELOPE and FILTER. TEMPO speeds up or slows down the melody being played. There's also a SOUND command for explosions, blips, and other sound effects.

More Control over Programming

Numerous commands to help the programmer are included. AUTO enables automatic line numbering, and RENUMBER renumbers lines. DELETE removes a range of lines from the program. KEY allows you to put commonly used strings into

function key definitions. HELP is very useful when you're debugging a program; it lists the line where the error occurred and indicates where the problem is (there's also a HELP key, which does the same thing). TRON turns on the trace function, so you can follow a program line by line. TRAP is like GOTO; when it's enabled, an error does not halt the program—it causes the program to jump to an error-handling routine you've written.

New reserved variables include ER (the error number if something goes wrong in a program), EL (the line number where an error occurred), and ERR$ (the name of the error). RESUME makes the program continue after an error has stopped it. And if the light on the disk drive starts blinking because of a disk error, you no longer have to open the error channel and input the error information. Just type PRINT DS,DS$ to find out what went wrong.

Other disk commands, most of which are self-explanatory, include APPEND, BACKUP, BLOAD, BOOT, BSAVE, CATALOG, COLLECT, CONCAT, COPY, DCLEAR, DCLOSE, DIRECTORY, DLOAD, DOPEN, DSAVE, DVERIFY, HEADER, RECORD, RENAME, and SCRATCH.

You can experiment with machine language by using the built-in ML monitor, entered via MONITOR. It's similar to *Micromon* or *Supermon.* Once you've written an ML program, start it up with the new version of the SYS command, which allows passing of values to the A, X, Y, and P registers. Once you exit to BASIC, you can look at the last values in these registers with RREG (Read REGister). DEC and HEX assist in making conversions between decimal and hexadecimal. Commodore has added XOR (eXclusive OR) to complement AND and OR. There are also ways of handling data that may be in the external memory expansion module: FETCH, STASH, SWAP.

Enhancements and Improvements

In addition to the normal PRINT statement, there's PRINT USING, which allows formatting of strings and numbers before they're printed. This is especially useful when you're dealing with financial information and want dollars and cents printed. PUDEF allows prior definition of which characters will appear in PRINT USING statements. When you print out

a check, for example, you might want leading asterisks instead of spaces.

The CHAR statement is a variation on PRINT, but it works on both text and hi-res screens. Since you can include the x and y positions, it works like PRINT AT, which is found on other computers.

IF-THEN now includes ELSE and BEGIN–BEND. BEGIN and BEND mark off a section of the program that will be executed only if the previous IF condition is true. FOR-NEXT loops can be replaced with DO-LOOP (see the example below).

MID$ has a new feature: It can assign a string to the middle of another string. So A$="HELLO": MID$(A$,2)= "IPP" would make A$ into "HIPPO". INSTR finds the position of one string inside another. It could tell you, for example, that "DEF" is inside "ABCDEFGHI", starting at the fourth position.

The RESTORE command can be followed by a line number to set the READ-DATA pointer to a specific line.

PRINT FRE(0) now displays how much memory is available in the first bank of memory, where programs are stored, while FRE(1) displays how much memory is left in the bank containing variables. And since the variables are kept separate from the program, adding a line to a program does not destroy variable definitions. You can stop a program, make some changes, and then GOTO a line to resume the program with previously defined variables.

The Slowest Commodore—And the Fastest

How fast is the 128? As a simple benchmark, we ran a FOR-NEXT loop that counted from 1 to 10,000 and printed the number of jiffies (a jiffy is 1/60 second). The 128 was the fastest *and* the slowest Commodore computer (it was tested twice, once with the FAST command, once with SLOW):

Speed in Jiffies	Computer
929	128 (SLOW)
895	+4/16
653	64
612	VIC
440	128 (FAST)

There's a good reason why the 128 can be both the fastest and the slowest. All Commodore BASICs are *interpreted,* which means the computer figures out (interprets) what the program should be doing as the program is running. Some other languages—even other BASICs—are *compiled.* Compiled programs are written with an editor program (similar to a word processor) and then compiled into object code that's closer to machine language than an interpreted program.

When an interpreted BASIC gets to a command like PRINT, it has to look through a table listing the location of the routine that makes PRINT work. Adding more commands to the Plus/4, and even more to the 128, makes the list longer. So the computer has to spend more time searching for command definitions. In addition, TO has several meanings in BASIC 7.0. You can loop FOR D=1 TO 10000, or DRAW from one point TO another, or copy files from one disk drive TO another.

The 128, with the biggest and best Commodore BASIC, is also the slowest of the bunch, as you can see from the timings above. To adjust for this, Commodore has added two new commands: FAST and SLOW.

The clock that drives the main processing chip runs at 1 megahertz (MHz), or 1,000,000 cycles per second. The FAST command doubles the speed of the clock to 2 MHz. It's especially good in programs that require a lot of calculations. There's a tradeoff, though. The 40-column screen goes blank while FAST is in effect. But if you want speed, the screen display probably won't matter. If you own a 64 and tape drive, you're probably used to seeing the screen blank while programs are loaded.

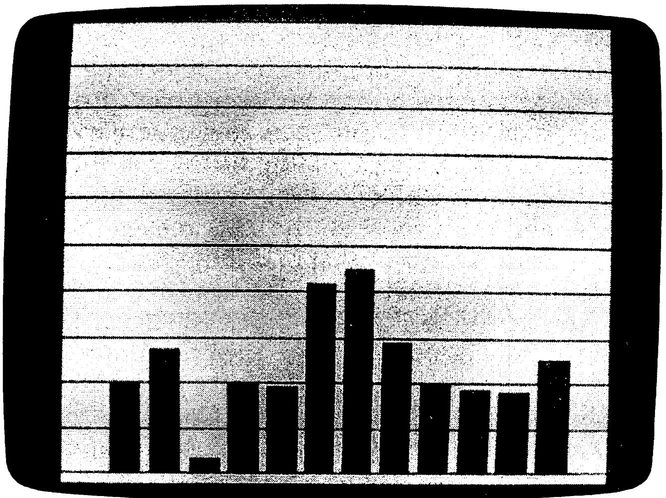
The VIC-20 has been the fastest Commodore computer for the last few years. Now there's a faster one.

Hi-Res Graphics

The sound, programming, disk access, and other commands are great improvements. But the new hi-res graphics commands are the most fun to play with. Rather than trudging through a long list of what the commands do, let's look at three short graphics programs.

Program 1-1 creates a bar chart. It reads values from DATA statements, figures out a scale, and draws the chart.

The number of values is limited only by the horizontal resolution of the screen (320 pixels). The values can be any positive number; the graph will be scaled accordingly.



First, lines 20–30 set up a loop to READ through the DATA statements. Normally, you'd expect an OUT OF DATA error as soon as there are no more items to read. But line 10 prevents the program from stopping. TRAP40 tells the computer to go to line 40 if an error occurs. Forcing an error to happen is not particularly good programming practice, but it illustrates one use of TRAP.

By the time we get to line 40, the variable MAX holds the highest value, and TL is the total number of bars to be plotted. We change the TRAP target to line 100, which switches to the text screen, GRAPHIC0, and prints the line number with the error, EL, and the type of error, ERR$(ER). If you type the program correctly, this error routine shouldn't be necessary.

The business with the logarithms is part of the scaling. Dividing the LOG of a number by the LOG of 10 and taking the INTeger value gives you the number of digits to the left of the decimal place. Another way to do this is SC= LEN(STR$(INT(MAX*1.2)))−1.

We'll be using the screen as if it were graph paper 500 squares across by 1024 squares deep. The screen is really only 320 by 200, and it's not possible to get better resolution than that, but the SCALE command in line 50 allows you to treat the screen as if it had more points.

A single command, GRAPHIC1 in line 50, turns on hi-res mode. Other options would be to split the screen between hi-res and text or to go into multicolor hi-res (with or without text split). SCNCLR clears whichever screen (hi-res or text) is currently being displayed. COLOR0, the background, is set to white. The foreground (COLOR1) and border (COLOR4) are painted purple. The 128 uses color numbers 1–16 rather than 0–15, so you can look at the keyboard and find the color number from the colors printed on the numeric keys. Blue would be color 7, for example.

We can now start the graph. In line 60, ten lines are drawn across the screen as background for the bars. The first number after DRAW is the color (color 1 was set to purple in the previous line). It's followed by the x and y coordinates of the beginning and the end of the line. RESTORE resets the DATA statements so the values can be read again.

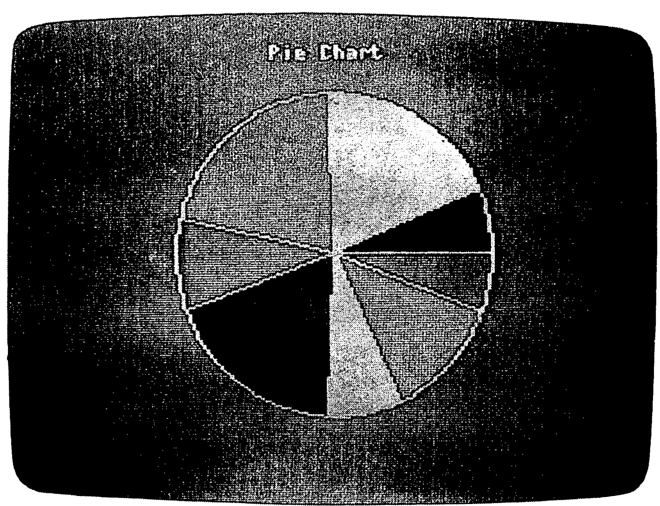
All of the bars are drawn in the next loop. The values are taken from DATA statements and the coordinates (XC and YC) are calculated for the top left corner of the bar. BOX then plots a rectangle, based on the x and y positions of two opposite corners. To make the bars fatter, increase the .75 to a value of 1. Decrease it to make thinner bars. The second to the last number after BOX means the rectangle should not be rotated. The last number 1 after BOX fills in the rectangle after it's drawn.

Finally, we wait for a keypress. GETKEY works like GET, except it stops the program until a key is pressed. GRAPHIC0 sends us back to the text screen, and the program ends.

Everything has been done in ten lines, without a single PEEK, POKE, AND, or OR. If you wrote such a program on the 64, it would be at least twice as long. And you'd need a reference guide to find all the POKEs and formulas for turning pixels on and off.

The Pie Chart

The next example program, Program 1-2, reads a list of values from DATA statements and creates a pie chart.



The first line may look strange to VIC and 64 owners, with the DO–UNTIL–LOOP structure. The typical FOR-NEXT loop starts counting at one number and ends when it reaches a specified value. But this loop starts at DO and repeats over and over when it reaches the LOOP statement. Three things can terminate the loop. It can loop WHILE an expression is true (when it's false, the loop ends). Or DO-LOOP can continue while some expression is false, UNTIL it's true. Or, the EXIT command can get you out of the loop.

In this case, the loop reads a variable A$ from DATA statements. It continues UNTIL it finds the word END. TL is the total number of items on the list—the numbers which will be translated to slices of the pie graph. SUM is the total of the values. We don't need a maximum in this program.

Line 20 enables the multicolor hi-res screen and clears it (adding ,1 after GRAPHIC3 is another way of doing SCNCLR). The different colors are set, and SCALE is turned on (to the default value of 1024 by 1024). WIDTH1 makes the graphics routines use the thinnest lines available.

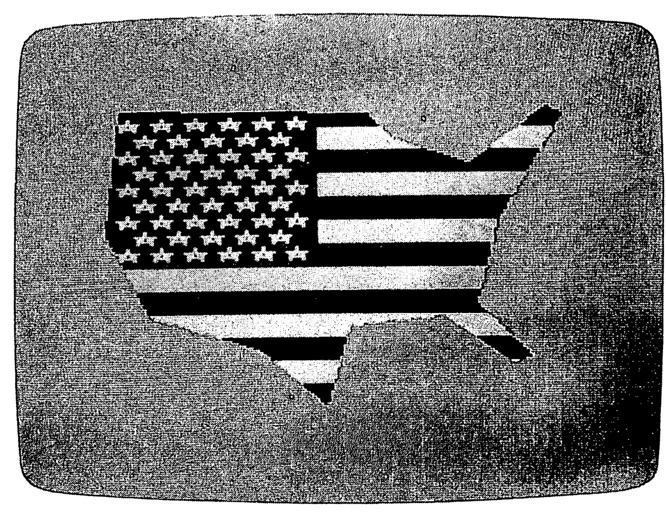
Next, the CIRCLE command draws a circle in color 2, centered on the screen. CIRCLE could have been called POLYGON, because it can also create triangles, squares, pentagons, hexagons, and other regular shapes. The label "Pie Chart" is then printed in uppercase/lowercase above the circle. The CHAR command, as noted above, prints at a specified location on any text or hi-res screen. In this case, we're printing in color 2, starting at the fourteenth column and first row.

The rest of the program is math, to convert the numbers from the DATA statements first into percentages of the total and then into angles (which will add up to 360 degrees). Note the underlined ↑ at the end of line 50. That means you should type a SHIFTed ↑, the pi character (π).

We don't need to go into the details of trigonometric functions like sine and cosine. They're necessary to calculate the angles for the sides of the wedges. After the wedge is on the screen, we also have to find a point in the middle of the angle and use PAINT to fill it with color (line 90).

The U.S. on a Flag

The final program, Program 1-3, best illustrates the power of the new graphics commands. It's just nine lines long, plus three for DATA statements. The result is an American flag superimposed on a map of the United States.



Line 10 jumps right into the multicolor hi-res screen. Colors 0–3 are set to blue, red, white, and light gray. Line 20 draws the 13 stripes, using the BOX command, alternating between red and white. The expression (J>96) will equal 0 if false and −1 if true. It's needed to make the stripes shorter at the top of the flag, so there will be room for the stars.

A single white star is created in line 30. There should be 50 stars on the flag, but it's necessary to draw only one to begin with. The SSHAPE (Save SHAPE) in line 40 copies the star (ten pixels wide and six deep) into the variable SA$. Once it's saved, we can use SA$ like a rubber stamp and quickly make all 50 stars appear.

Line 50 calculates x and y positions for each of the 50 stars and GSHAPEs them into place. The flag is done.

The coordinates of the roughly drawn U.S. map shape have been put into DATA statements to be read in line 60. DRAW puts the edges of the map on the screen, on top of the flag. Finally, line 80 fills the area outside the map with light gray.

The Question of Compatibility

VIC and 64 owners may wonder about hardware and software compatibility between their systems and the 128. As far as we can tell, tape drives, disk drives, monitors, modems, joysticks, and other peripherals will work with the 128 in either 64 mode or 128 mode. In addition, the 64 mode is completely compatible with all 64 software (on disk, tape, or cartridge) we have tested. The CP/M option, however, requires the newer, faster 1571 (single double-sided drive) or 1572 (dual double-sided drive). And the tape drives and joysticks for the Plus/4 and 16 will not work on the 128 because the connectors are of different sizes.

There are two video outputs available in 128 and CP/M mode: 40-column composite video and 80-column RGB (the 64 mode has no 80-column option). You can have two screens containing completely different text. Early reports on the 128 noted that the 80-column option is available only with an RGB (Red, Green, Blue) monitor. That's true if you want 80 columns and color; we've hooked up the 128 to an IBM RGB monitor using a standard cable. But pin 7 of the IBM cable is not used, and Commodore has put 80-column monochrome output on that pin. We've wired up a cable that allows 80 columns (black-and-white only) on a 1701 or 1702 monitor.

(Cardco recently announced such a cable, at a suggested price of $9.95.) You *can* have 80 columns, but no color, on your Commodore monitor with only a slight sacrifice of resolution.

BASIC 7.0 Keywords

All Commodore BASICs contain the commands of version 2.0, found on the VIC and 64. Version 3.5, from the Plus/4 and 16, included many useful new graphics commands as well as some new commands and functions for program control. The new BASIC 7.0 has all of the previous commands (except RLUM, used to read a color's luminance in BASIC 3.5), and much more, giving 128 programmers the most powerful BASIC yet available on a Commodore computer.

Table 1-1. BASIC 7.0 Keywords

ABS
AND
APPEND
ASC
ATN
AUTO
BACKUP
BANK
BEGIN–BEND
BLOAD
BOOT
BOX
BSAVE
BUMP
CATALOG
CHAR
CHR$
CIRCLE
CLOSE
CLR
CMD
COLLECT
COLLISION
COLOR
CONCAT
CONT
COPY
COS
DATA
DCLEAR
DCLOSE
DEC
DEFFN
DELETE
DIM
DIRECTORY
DLOAD
DO–LOOP–WHILE–UNTIL–EXIT
DOPEN
DRAW
DSAVE
DS$
DVERIFY
ELSE
END
ENVELOPE
ERR$
EXP
FAST
FETCH
FILTER
FN
FOR-NEXT
FRE
GET
GET#
GETKEY
GO 64
GOSUB
GOTO/GO TO
GRAPHIC
GSHAPE
HEADER
HELP

HEX$
IF-GOSUB
IF-GOTO
IF-THEN
IF-THEN-ELSE
INPUT
INPUT#
INSTR
INT
JOY
KEY
LEFT$
LEN
LET
LIST
LOAD
LOCATE
LOG
LOOP
MID$
MONITOR
MOVSPR
NEW
ON-GOSUB
ON-GOTO
OPEN
PAINT
PEEK
PEN
PLAY
POINTER
POKE
POS
POT
PRINT
PRINT#
PRINT USING
PRINT# USING
PUDEF
RCLR
RDOT
READ
RECORD
REM
RENAME
RENUMBER
RESTORE
RESUME
RETURN
RGR
RIGHT$
RND
RREG
RSPCOLOR
RSPPOS
RSPRITE
RUN
RWINDOW
SAVE
SCALE
SCNCLR
SCRATCH
SGN
SIN
SLEEP
SLOW
SOUND
SPC
SPRCOLOR
SPRDEF
SPRITE
SPRSAV
SQR
SSHAPE
ST
STASH
STEP
STOP
STR$
SWAP
SYS
TAB
TAN
TEMPO
TI
TI$
TO
TRAP
TROFF
TRON
UNTIL
USR
VAL
VERIFY
VOL
WAIT
WHILE
WIDTH
WINDOW
XOR

Program 1-1. Bar Chart

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
JX 1Ø  TRAP4Ø
PF 2Ø  READN:IFN>MAXTHENMAX=N
GK 3Ø  TL=TL+1:GOTO2Ø
CG 4Ø  TRAP1ØØ:SC=INT(1+LOG(MAX*1.1)/LOG(1Ø)):YY=1Ø
       ↑(3-SC):XX=5ØØ/(TL+2)
DC 5Ø  GRAPHIC1:SCNCLR:SCALE1,5ØØ,1Ø23:COLORØ,2:COL
       OR1,5:COLOR4,5
EM 6Ø  FORJ=1TO1Ø:DRAW1,Ø,J*1ØØTO499,J*1ØØ:NEXT:RES
       TORE
FE 7Ø  FORJ=1TOTL:READN:XC=J*XX:YC=1ØØØ-(N*YY):BOX1
       ,XC,YC,XC+XX*.75,1ØØØ,Ø,1:NEXT
EC 8Ø  GETKEYA$:GRAPHICØ:END
RH 9Ø  DATA252,183,185,2Ø4,289,446,418,193,2Ø4,34,2
       72,2Ø3
QA 1ØØ  GRAPHICØ:PRINT"LINE"EL"SEEMS TO HAVE AN ERR
        OR: ":PRINTERR$(ER)
```

Program 1-2. Pie Graph

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
XX 1Ø  DO UNTIL A$="END":READA$:TL=TL+1:SUM=SUM+VAL
       (A$):LOOP:TL=TL-1
HJ 2Ø  GRAPHIC3,1:COLORØ,13:COLOR1,3:COLOR2,2:COLOR
       3,15:COLOR4,13:SCALE1:WIDTH1
GA 3Ø  CIRCLE2,512,512,3ØØ,36Ø:CHAR2,14,1,CHR$(14)+
       " PIE CHART ",Ø
ES 4Ø  XS=812:YS=512:PC=Ø:AN=Ø:RESTORE
CX 5Ø  FORJ=1TOTL:DRAW2,512,512TOXS,YS:READN:TN=AN+
       (N/SUM)*2*↑
RP 6Ø  YS=512-SIN(TN)*36Ø:XS=512+COS(TN)*3ØØ
PK 7Ø  MA=PN+(AN-PN)/2
JD 8Ø  AT=512-SIN(MA)*27Ø:XT=512+COS(MA)*255
BF 9Ø  PAINTPC,XT,AT,1
GG 1ØØ  PN=AN:AN=TN:PC=(PC+1)AND3:NEXT
SG 11Ø  GETKEYA$:GRAPHICØ,1:COLORØ,12:LIST
SH 12Ø  DATA1235,3679,4168,1718,3696,1467,2375,1137
        ,END
```

Program 1-3. Map with Flag

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
CS 1Ø  GRAPHIC3,1:COLORØ,7:COLOR1,3:COLOR2,2:COLOR3
       ,16:COLOR4,16:WIDTH1:C=1
HK 2Ø  FORJ=35TO16ØSTEP1Ø:HX=64*(J>96)+75:BOXC,HX,J
       ,15Ø,J+9,Ø,1:C=3-C:NEXT
```

```
DD 30 DRAW2,19,40 TO 21,45 TO 16,42 TO 22,42 TO 17
      ,45 TO 19,40
QA 40 SSHAPE SA$,15,40,24,45:NS=5:IN=0
GS 50 FORJ=0TO8:SY=J*7+40:FORK=0TONS:SX=K*10+14+IN
      :GSHAPESA$,SX,SY:NEXTK:NS=9-P-NS:IN=5-IN:NEX
      TJ
EH 60 X=15:Y=36:FORJ=1TO15:READNX,NY:DRAW3,X,YTONX
      ,NY:X=NX:Y=NY:NEXT
MH 70 DRAW3,X,YTO15,36
PJ 80 PAINT3,80,1,0
FE 90 GETKEYA$:GRAPHIC0
EC 100 DATA90,38,98,50,120,60,143,35,148,39
CD 110 DATA 128,86,122,120,138,143,132,146,112,126
SP 120 DATA 84,130,77,165,56,140,24,126,11,9?0
```

Save-with-Replace: Debugged at Last

P. A. Slaymaker

Since the early days of the Commodore PET in the late 1970s, a controversy has raged over one particular disk command—Save-with-Replace. This convenient command automatically replaces an existing disk file with a new file of the same name, combining SCRATCH and SAVE in one operation. But for years, many Commodore users have shunned Save-with-Replace like poison, swearing that it contains a mysterious bug which unpredictably scrambles disks. And just as many other users contend the bug doesn't exist at all. Now, finally, there's proof: The bug does exist in the 1541 and 1571 drives, it can be demonstrated, and, most importantly, it can be avoided. This is the first full explanation of why the bug happens and how you can circumvent it.

It's time to settle something once and for all: There *is* a Save-with-Replace bug! It afflicts the disk operating system (DOS) built into every 1541 and 1571 disk drive, potentially threatening every disk on which you use the Save-with-Replace command. We'll review what the Save-with-Replace bug typically does, list a program which demonstrates the bug beyond doubt, explain *why* it happens, and, finally, recommend a procedure for avoiding the bug.

The Save-with-Replace command (typed as SAVE@) has been accused of scrambling, swapping, duplicating, or overwriting disk files and of messing up block allocation maps (a BAM is a map on a disk which keeps track of which blocks are storing files and which are free). Many computer magazines and other authorities in the Commodore community have warned against using SAVE@. Yet other Commodore experts have never experienced problems with SAVE@ and swear the bug is an old hacker's tale. There are many anecdotes about when the bug strikes, which files are affected, and when the files or BAM will be garbled. The mystery has persisted for so long because usually the bug is not repeatable. But this article shows how to replicate the bug and explains

why it is related primarily to the file length and the distribution of free blocks on the disk as determined by the BAM.

Recently some new evidence surfaced about SAVE@. In an article published in the July 1985 issue of *The Transactor,* "SAVE with Replace Exposed!!," author Charles H. Whittern showed that the bug exists under some conditions. This article made some observations on files likely to be affected and listed a program which repeatedly loaded and saved files using SAVE@. Afterward, an examination of the disk showed some files to be scrambled. Unfortunately, no details of the file configurations were given, and the editors admitted that the bug had them baffled—but at least the problem was recognized, a first step.

Our investigation shows that the bug usually occurs when the drive number has not been specified on previous drive operations, such as loading a file or listing a directory. In other words, typing LOAD"filename",8 or LOAD"$",8 instead of DLOAD"filename" or LOAD"0:filename",8 or LOAD"0:$",8 sets up conditions for the bug. The drive number 0 should be specified in disk commands because, as we'll explain later, *the SAVE@ bug is related to the phantom software drive 1 in the 1541 and 1571.* (Note that BASIC 7.0 adds the 0: to all DOS commands.) In addition, the bug tends to bite disks on which many files have been scratched and rewritten. This leaves gaps on the disk so that a file is scattered over many tracks. These gaps do not normally cause a problem if you specify the drive number in disk commands.

Therefore, the key to avoiding the SAVE@ bug is to always specify drive 0 when performing any disk drive function, to always reset the drive before any SAVE@ operation, or to use DLOAD, DSAVE, and F3 for directory. Resetting the drive requires either turning the drive off and then on, or sending a reset command (OPEN15, 8,15,"UJ").

Demonstrating the Bug

At this point, some of you might be skeptical that the SAVE@ bug really exists. To prove that it does, the accompanying program formats a new disk with the single file "SAVE@ DEMO" and alters the BAM to simulate a partially used disk with a gap due to scratched files.

For this demonstration you'll need two disks—one to save the original program on and a second disk that will be formatted by the accompanying program. Remember, *formatting a disk erases everything from it, so be sure to use a new disk for this demonstration.*

Follow these instructions carefully (be sure to use the LOAD and SAVE commands, *not* DLOAD and DSAVE, for this demonstration):

1. Type the program *exactly* as listed—including all uppercase REM statements. It's important to type the program as listed because it must be at least nine blocks long on the test disk to insure proper results.
2. Save the program on a disk before running it.
3. Set the computer for 40 columns. Put a blank test disk in the drive and run the program. It will format the disk (erasing anything that might be on the disk) and save a file called SAVE@ DEMO on the disk. Type LOAD"$",8 to list the directory and notice that 254 blocks are free if you have a 1541 drive and that 918 blocks are free if you have a 1571 drive.
4. Reset the drive by turning it off, then on. Load the file by typing LOAD"SAVE@ DEMO",8.
5. Save the file three times using the SAVE@ command (SAVE"@0: SAVE@ DEMO",8). *Do not* list the directory or perform any other operation between SAVE@ commands.
6. Initialize the drive with OPEN 1,8,15,"I0:":CLOSE 1.
7. List the directory by typing LOAD"$",8. What's this? There were 254 (1541) or 918 (1571) blocks free before, but now there are 258 (1541) or 922 (1571)—a discrepancy of four blocks. (If you don't get this result, it probably means that you haven't followed the directions exactly. Start again at step 3.) If you examine the BAM with a disk utility, you'll see that the first four sectors of the file are marked as free. If you executed a fourth SAVE@ command, it would overwrite the beginning of the file, and the disk would be corrupted even worse.
8. Now rerun the program to make a new test disk. Reset the drive and run the above test again, but specify the drive number for the load (LOAD"0:SAVE@ DEMO",8). The SAVE@ bug does *not* occur!

Always Specify Drive 0

This demonstration provides a powerful lesson: All DOS commands should include the drive number 0 (remember, BASIC 7.0 DOS commands add drive number 0 for you):

Load file	**LOAD"0:filename",8**
	DLOAD"filename"
Save file	**SAVE"0:filename",8**
	DSAVE"filename"
Save with replace	**SAVE"@0:filename",8**
	DSAVE"@filename"
LOAD directory	**LOAD"$0:",8**
	DIRECTORY
	F3
Initialize drive 0	**OPEN15,8,15,"I0":CLOSE15**
Validate	**OPEN15,8,15,"V0":CLOSE15**
	COLLECT

Similarly, all disk file commands should specify the drive number.

Most Commodore users do not specify the drive number when loading the directory or files. The *1541 User's Manual* examples for the LOAD command don't specify the drive, and neither do most magazine articles. If the drive number is not specified, the disk drive is supposed to default to drive 0. What actually happens very often causes an error message such as 74,DRIVE NOT READY,00,00.

The Missing Drive

The early Commodore PETs were available with dual disk drives—two drives in one unit. The drives were addressed as 0: and 1: when using disk commands. But on later Commodore computers designed to use the 1540/1541, multiple drives are addressed by changing the *device number,* not the *drive number.* The device number for a single drive is 8. That's why you type a command like LOAD "filename",8. On dual-drive systems, the second drive is usually addressed as device 9, as in LOAD"filename",9. Therefore, most people stopped (or never started) specifying the drive number, which is 0:, for all 1541 and 1571 disk drives. Drive 1: simply doesn't exist with the 1541.

What happens when the drive number is not specified for a LOAD or SAVE? DOS first checks for a drive number. If

none is specified, it assumes drive 0. Okay so far. Then DOS attempts to read the disk. If no disk is found, DOS automatically switches to the nonexistent drive 1. A DRIVE NOT READY error then results whether or not a drive number was specified. If a disk is found, DOS searches its internal directory for the specified file. If the default drive was used, DOS switches to drive 1 to continue searching. This also causes the DRIVE NOT READY error, since there is no drive 1. Furthermore, drive 1 remains the default drive as long as there are directory searches to be done. The internal drive pointers must be reset to recover from this error condition.

SAVE@ always works properly in our tests if the drive number is specified on all operations and no direct access buffers are allocated. We are not aware of anyone who has documented a failure under these conditions (assuming a closed file was specified, sufficient room was present on the disk, and no read or write errors occurred). Thus, Commodore experts who claim there is no bug are partially correct. We have also found that if the drive number is not always specified during loads and directory listings, as is common practice, the SAVE@ bug can occur even though the drive number is specified in the SAVE@ command.

Files stored on just one or two tracks—such as short files on a fresh disk—are not prone to the SAVE@ bug. Files stored over many tracks on disks on which many files have been saved and scratched are the most susceptible, as are files saved with some utilities intended to speed up the 1541 disk drive.

DOS Thievery

First, we should note that although the SAVE@ command deletes a disk file and saves a replacement in a single operation, it works differently than if you issued separate SCRATCH and SAVE commands. SAVE@ calls entirely different DOS routines—the SCRATCH and SAVE are executed as part of a continuous procedure, and the SAVE@ command therefore requires that more drive buffers be available.

DOS V2.6 has five internal buffers, numbered 0–4. These buffers start at memory pages $300, $400, $500, $600, and $700, respectively. Normally, an image of the disk's BAM (Block Availability Map) is stored in the page at $700, an image of the directory sector in use is stored at $600, and the

other three buffers are available for file use. As long as a buffer is active, it cannot be used for anything else. If DOS has assigned an internal channel to the BAM at $700, then trying to open a direct channel to buffer 4 (from BASIC: OPEN 2,8,2,"#4") will produce a 70,NO CHANNEL,00,00 error.

Similarly, DOS assigns channels and buffers to the directory sector and file sectors which are being read or written. Normally, DOS assigns two read or two write channels and uses only three of the five buffers. The SAVE@ command, however, requires all five buffers—two read, two write, and the BAM. If DOS can't find a free buffer, it tries to steal an assigned but inactive buffer. This thievery causes the SAVE@ command to fail occasionally—for reasons which will be discussed shortly.

Why does omitting the drive number in disk commands cause DOS to steal a buffer? When a file is opened or loaded via the OPEN routine ($D7B4), DOS searches the internal directory to look for the specified filename (DOS routine names and addresses in this article conform to those listed in *Inside Commodore DOS*, Datamost, 1984). ONEDRV ($C312) determines whether a drive was specified. OPTSCH ($C3CA) assigns a default or specified drive for each file in the command, and also calls AUTOI ($C63D). AUTOI reads the BAM of the disk in the specified drive, and also tries to initialize drive 1 if no drive was specified. Usually buffer 3 ($600) is allocated for the phantom drive 1 BAM, and a B1 SEEK command is issued to the disk controller. This results in an internal DRIVE NOT READY error in the disk controller. The error is trapped by AUTOI, but not reported outside the disk drive. This leaves buffer 3 allocated but inactive. FFST ($C49D) then reads the directory and tries to find the file.

The reason this inactive buffer assignment is important is that the SAVE@ command requires all five buffers, but only four are now available. Whenever DOS needs to allocate a buffer, it calls GETBUF ($D28E). If one is not free, GETBUF tries to steal an inactive one by calling STLBUF ($D339). If the drive number is always specified and no direct access buffers are allocated, STLBUF is never called. We verified this by modifying GETBUF after copying DOS onto an EPROM (Erasable-Programmable Read Only Memory). If a channel can't be stolen, then a NO CHANNEL error occurs. But if STLBUF is called, the SAVE@ bug sometimes occurs.

Stealing the Wrong Buffer

STLBUF can be called several times during a SAVE@ command. The result is that the BAM and directory sectors can be reassigned to different buffers during a single SAVE@. We have found the BAM and directory sectors in every drive buffer after different SAVE@ commands. We have found copies of the current directory sector in two different buffers, one an old sector and one properly updated, but the wrong one had been written to the disk. Somehow, the pointers to the BAM and directory sectors are not properly accounted for. Which buffer is stolen by STLBUL depends on prior buffer usage and the values stored in LRUTBL,Y ($FA,Y), the least recently used table. It appears that STLBUF updates all pointers except LRUTBL,Y. This means that multiple calls to STLBUF may steal the wrong buffer—in this case, the wrong buffer to steal is the BAM.

The BAM is stored in the drive in one of the buffers. STLBUF should not steal the drive 0 BAM, but should instead take back the unused buffer incorrectly assigned to drive 1. It never steals the drive 1 BAM, buffer 3 at $600, because STLBUF cannot take a buffer which encountered a drive error. Remember that an internal DRIVE NOT READY error did occur, because there is no drive 1.

To test this, we copied into EPROM an altered version of DOS with STLBUF modified to allow stealing a buffer with this error. This allowed the phantom drive 1 BAM buffer to be freed, and the SAVE@ bug did not strike during tests with this modified DOS.

If this buffer stealing occurs, why does SAVE@ work most of the time? We must dig deeper into DOS to answer this question. When a file is opened and blocks (or sectors) are written to a disk, the BAM is *not* directly updated in the drive memory. Instead, a BAM image for each of two tracks is stored at BAM ($2A1–$2B0). Each time a new block is allocated by WUSED ($EF90), it is recorded in the BAM image. When a new track is tested for free sectors, DOS checks whether it has a BAM image for it. If not, it calls SWAP ($F05B), which first updates the BAM with the BAM image from the next-to-last track, copies the new track's BAM map into the BAM image, and then zeros that track in the BAM. This all works perfectly—most of the time.

After the last file sector is written to the disk, the BAM still has not been written to the disk. In fact, the BAM in the drive is wrong because it has not yet been updated from the BAM images. When a file is closed, the disk directory is closed, CLSDIR ($DBA5), by reading in the file's directory sector, testing for a replace file type, and then rewriting it to the disk. MAPOUT ($EEF4) is called to read the BAM off the disk, if necessary, and then to update it from the BAM images by calling PUTBAM ($F0A5). The updated BAM is then written back to the disk.

During a SAVE@ command, DOS performs an additional step after reading the directory sector. The file type is designated as replace, so DELFIL ($C87D) is called to delete the original version of the file from the BAM. It reads in the BAM if necessary when freeing the first sector, FRETS ($EF5F), and then proceeds to trace through the file and delete sectors in the BAM images. The BAM is then written to the disk.

Bungled BAM

Normally, this procedure works correctly. But havoc results if the BAM buffer is stolen while the file is being closed. This can happen during a SAVE@ command because DELFIL requires two additional buffers. The BAM can be stolen at different points during the procedure, depending on which buffers were previously used—which, in turn, depends on the number of sectors in the file and the tracks on which it is stored.

After the BAM is stolen, it is read back in when needed and updated from the BAM images. Only *two* tracks can be updated, however, since there are only two images. If more than two tracks have been accessed by SAVE@, the BAM may *not* be correctly updated. A track could be updated correctly, left unchanged, or fully allocated, depending on when the BAM was stolen.

If extra sectors are allocated, the BAM is incorrect, but no permanent harm is done. A validate command will cure the problem. If sectors are not allocated, then a new file will be saved on top of the old file's sectors. In Program 1-4, a fourth SAVE@ command would result in the file being written on top of the old file's first four sectors, and then the whole new file would be scratched—a tragic result, indeed.

Based on these findings, we recommend that you avoid

the SAVE@ command when direct access channels to the drive are open or if you don't always specify the drive number in disk commands. You should also avoid SAVE@ when using programs or cartridges intended to speed up access on the 1541 disk drive. These programs often reserve internal drive buffers and may cause problems even if the drive number is specified. If you're using the DOS Wedge, we recommend issuing a >UI or >UJ command before each SAVE@ command to be sure all the buffer pointers are reset. Many word processors also allow you to send these commands to the drive. Otherwise, the drive should be turned off and then on before using SAVE@. (On the SX-64, Commodore's portable 64, press the drive reset button.)

During our studies we found several other minor bugs in DOS V2.6, including the subroutine which puts the value 2 at the drive memory location $197. This bug does no harm since it affects a normally unused section of drive memory. However, we have found it can affect DOS routines downloaded into the drive. There may be other bugs or quirks which we have not found, so the Commodore DOS controversy may never be fully closed.

Program 1-4. SAVE@ Bug Demonstration

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
KQ 10 COLOR 0,1:COLOR 4,1
FQ 20 PRINT"{BLK}{CLR}{YEL}{2 DOWN}{RIGHT}{RVS}
      {TAB}"CHR$(14)"{H}SAVE@ BUG DEMONSTRATION"
CC 40 PRINT"{CYN}{DOWN}{RIGHT}THE SAVE@ DEMO PROGR
      AM FORMATS A BLANK"
GD 50 PRINT"{RIGHT}DISK, ALTERS THE BAM, SAVES ITS
      ELF, AND";
RS 60 PRINT"{RIGHT}THEN ALTERS THE BAM AGAIN.
      {2 SPACES}SAVE@ WILL"
PM 70 PRINT"{RIGHT}FAIL THE THIRD TIME ON THIS DIS
      K."
RD 80 PRINT"{DOWN}{RIGHT}{GRN}INSERT DISK TO FORMA
      T - PRESS {RVS} SPACE {OFF}."
FD 90 GOSUB 5000: REM{2 SPACES}GET KEY PRESS
BP 100 IF K<>60 THEN 90:REM WAIT FOR SPACE
BC 110 PRINT"{DOWN}{RIGHT}{RED}WARNING! THE DISK W
       ILL BE ERASED."
HR 120 PRINT"{RIGHT}{YEL}ARE YOU SURE? TO CONTINUE
        PRESS <Y>."
RF 130 FOR T=0 TO 100:NEXT: REM TIME DELAY
HM 140 GOSUB 5000: REM{2 SPACES}GET KEY PRESS
```

```
GH 15Ø  IF K=25 THEN 17Ø: REM CONTINUE IF <Y>
JA 16Ø  PRINT"{DOWN}{RIGHT}{YEL}PROGRAM ABORTED.":G
        OTO 33Ø
PD 17Ø  CLOSE2:CLOSE15: REM{2 SPACES}CLOSE CHANNELS
KQ 18Ø  OPEN15,8,15: REM OPEN COMMAND CHANNEL
SM 19Ø  PRINT"{DOWN}{RIGHT}{CYN}NOW FORMATTING DISK
         - PLEASE WAIT."
QA 2ØØ  PRINT#15,"NØ:SAVE@ TEST"CHR$(44)"PS":
        {3 SPACES}REM{2 SPACES}FORMAT DISK
XA 21Ø  GOSUB 3ØØØ: REM CHECK ERROR CHANNEL
JH 22Ø  PRINT"{UP}{RIGHT}{PUR}FORMATTING HAS BEEN C
        OMPLETED.{4 SPACES}"
KB 23Ø  PRINT"{DOWN}{RIGHT}E73ALTERING BAM."
FX 24Ø  GOSUB 4Ø1Ø: REM OPEN DIRECT CHANNEL AND CHE
        CK ERROR CHANNEL
XE 25Ø  GOSUB 1Ø1Ø: REM{2 SPACES}ALTER BAM
CF 26Ø  CLOSE2:CLOSE15: REM{2 SPACES}CLOSE CHANNELS
JH 27Ø  PRINT"{RIGHT}{RED}SAVING SAVE@ DEMO."
BG 28Ø  SAVE"Ø:SAVE@ DEMO",8
EG 29Ø  PRINT"{RIGHT}{YEL}ALTERING BAM."
PB 3ØØ  GOSUB 4ØØØ: REM OPEN DIRECT CHANNEL AND CHE
        CK ERROR CHANNEL
KA 31Ø  GOSUB 2Ø1Ø: REM{2 SPACES}ALTER BAM
HM 32Ø  PRINT"{DOWN}{RIGHT}{CYN}DISK IS FINISHED! N
        OW REFER TO TEXT."
SK 33Ø  CLOSE2:CLOSE15: REM{2 SPACES}CLOSE CHANNELS
JX 34Ø  POKE 2Ø8,Ø: REM{2 SPACES}CLEAR KEYBOARD
PP 35Ø  END
EA 1ØØØ REM * MODIFY BAM SECTOR FOR SAVE
FP 1Ø1Ø PRINT#15,"U1:2 Ø 18 Ø" : GOSUB 3ØØØ:
        {4 SPACES}REM READ BAM SECTOR
BG 1Ø2Ø PRINT#15,"B-P:2 52" : GOSUB 3ØØØ:
        {6 SPACES}REM POSITION BUFFER POINTER TRAC
        K 13
BA 1Ø3Ø FOR I=1 TO 2Ø: PRINT#2,CHR$(Ø);:{8 SPACES}
        NEXT:REM{2 SPACES}FILL BAM WITH ZEROS
BA 1Ø4Ø PRINT#15,"B-P:2 76":GOSUB 3ØØØ:{6 SPACES}R
        EM POSITION BUFFER POINTER TRACK 19
DQ 1Ø5Ø FOR I=25 TO 92: PRINT#2,CHR$(Ø);:
        {8 SPACES}NEXT:REM{2 SPACES}FILL BAM WITH
        {SPACE}ZEROS
EG 1Ø6Ø PRINT#15,"U2:2 Ø 18 Ø" : GOSUB 3ØØØ:
        {4 SPACES}REM{2 SPACES}WRITE TO BAM SECTOR
PC 1Ø7Ø PRINT#15,"IØ" : GOSUB 3ØØØ:{13 SPACES}REM
        {SPACE}INITIALIZE BAM
XF 1Ø8Ø RETURN
AF 2ØØØ REM * MODIFY BAM SECTOR AFTER SAVE
AP 2Ø1Ø PRINT#15,"U1:2 Ø 18 Ø" : GOSUB 3ØØØ:
        {4 SPACES}REM READ BAM SECTOR
```

```
BD 2020 PRINT#15,"B-P:2 60" : GOSUB 3000:
        {7 SPACES}REM POSITION BUFFER POINTER TRAC
        K 15
JE 2030 REM{2 SPACES}FREE UP 12 SECTORS ON TRACKS
        {2 SPACES}15 TO 17
BS 2040 PRINT#2,CHR$(4)CHR$(15)CHR$(0)CHR$(0);
CX 2050 PRINT#2,CHR$(4)CHR$(15)CHR$(0)CHR$(0);
XJ 2060 PRINT#2,CHR$(4)CHR$(15)CHR$(0)CHR$(0);
SG 2070 PRINT#15,"U2:2 0 18 0" : GOSUB 3000:
        {4 SPACES}REM{2 SPACES}WRITE TO BAM SECTOR
KB 2080 PRINT#15,"I0" : GOSUB 3000:{13 SPACES}REM
        {SPACE}INITIALIZE BAM
KF 2090 RETURN
KA 3000 INPUT#15,EN,E$,ET,ES
MA 3010 IF EN=0 OR EN=73 THEN RETURN
FC 3020 PRINT"{2 DOWN}{RIGHT}"EN;E$;ET;ES
SM 3030 CLOSE2:CLOSE15:END
JA 4000 OPEN15,8,15:GOSUB3000:REM{2 SPACES}OPEN CO
        MMAND CHANNEL AND CHECK ERROR
HD 4010 OPEN2,8,2,"#":GOSUB3000: REM OPEN DIRECT C
        HANNEL AND CHECK ERROR CHANNEL
HM 4020 RETURN
AR 5000 POKE208,0:POKE212,88: REM CLEAR KEY
XS 5010 K=PEEK(212)
SA 5020 IF K=88 THEN 5010
RJ 5030 RETURN
```

Dynamic Keyboard

Jim Butterfield

Dynamic keyboard techniques let you perform tasks that would otherwise be difficult or impossible in BASIC.

Many BASIC commands can be used in either *direct mode* (typed directly on the keyboard without a line number) or *program mode* (as part of a program). Certain statements and commands, however, work only in direct mode. Using them in a program requires the *dynamic keyboard* technique, which lets a program act like it's you—typing on the keyboard. This method is especially effective on Commodore machines because of their full-screen editing. The term *dynamic keyboard* was first used by Mike Louder in 1978, though the technique had been used previously by Larry Tessler to merge programs.

Direct Versus Programmed

A direct-mode command doesn't have a line number and is executed as soon as you press RETURN. An example is PRINT "HELLO". In program mode, the command does have a line number and is executed only when you type RUN and then press RETURN. An example is 10 PRINT "HELLO". Most BASIC commands work in both direct and program mode.

A few BASIC commands cannot be used in direct mode, however; they may appear only in a program. GET, INPUT, GET#, and INPUT# are the best-known of these. Usually, these commands use a segment of memory called the *input buffer* to store data as it arrives, and they won't work in direct mode because the same input buffer is used to hold the command itself. Thus, the incoming data might overwrite the command you typed in.

On the other hand, some BASIC commands can be used only in direct mode—not in a program. CONT, for example, causes an indefinite pause when used in a program. LIST works in program mode, but on most Commodore computers the program ends after executing LIST. In direct mode, you can enter a program line to add to the program or change it. You can't do this while running a program. Again, there's a

difference between programs and direct commands—they have different powers.

A very important difference is found in the LOAD command. If typed as a direct command, LOAD fills memory with a new program from tape or disk. If there was already a program in memory, it vanishes and its variables are thrown away. But a LOAD command executed within a program is quite different. The new program comes in, but existing variables are not scrapped—they are preserved so that the new program can use them. This is a powerful programming technique called *chaining,* which lets one program continue processing data that was generated by a previous program.

Invisible Fingers

Direct keyboard statements can perform certain tasks that programs can't (at least, not in the usual way). For example, if we want a program to invite a student to type in a formula, BASIC doesn't allow the formula to be evaluated (an INPUT statement won't evalute the formula 2 + 2 as 4).

Similarly, suppose we want one program—perhaps a main menu program—to load and run another program. That's hard to do because BASIC wants to chain the new program to the old one. Instead of starting the next program fresh, it tries to make it a continuation of the previous program. On rare occasions, there may be a real need to allow a program to change itself, although this is tricky because every time you change a program (by editing a line, for example), its variables go away. It's hard for any program to continue running after its variable values disappear.

We can accomplish these things, however, by using a startling technique: making the computer *type on its own keyboard.* How can a computer do this? It doesn't even have any fingers.

Here's how it works. When you strike a key, the information always goes first to a memory area called the *keyboard buffer.* After it gets there, it is picked up and used by the computer. If we can put a character in the keyboard buffer without actually pressing any keys, it will appear to have been typed, and the computer responds exactly as if the corresponding key was pressed.

Self-Keying

Let's try a quick example to see how it works. We'll ask the 128 to self-type the letter X:

POKE 208,1:POKE 842,88

The first POKE tells the computer how many characters are waiting in the keyboard buffer. The second puts the character X in the first slot of the buffer. After you type the line and press RETURN, the computer reports READY and acts as if you had pressed the X key. The letter X appears on the screen and the cursor flashes to its right. It would be easier just to type the X, of course, but we've established a new capability. A program can now, in effect, type on the keyboard.

Using the Screen

With this technique alone, you're limited to pretty short commands. The keyboard buffer usually has a size limit of about nine characters. Also, it's cumbersome for a program to put characters into the buffer one at a time. But on Commodore machines we can take advantage of *screen editing* to process longer direct commands.

Whenever you press the RETURN key, the computer reads the screen. Whatever it finds there, it does—perform a command, enter a line, or whatever. To make a program execute a long direct-mode command, follow these steps:

1. PRINT the command on the screen in a known place.
2. Position the cursor three lines above the command to be executed.
3. Enter POKE 208,1 so the operating system will know that there is one keypress stored in the keyboard buffer.
4. Put a carriage return in the keyboard buffer by entering POKE 842,13.
5. Terminate execution with an END command.

When the program reaches END, here's what happens. The desired command is on the screen and the RETURN is in the keyboard buffer. The program terminates, and the computer prints READY. Although the program has ended, the computer receives the RETURN as if you had just pressed that key, so it executes the line on the screen. Among other things, that line might contain a GOTO or CONT that would continue the program.

A Simple Example

Here's a simple program that uses the dynamic keyboard method to do something normally forbidden by BASIC: a computed GOTO. In most cases, a straightforward ON-GOTO command does the same job better, but let's use this example for the sake of simplicity.

Enter the following lines:

```
JM 120 PRINT "PICK A NUMBER 3 TO 5"
FR 130 INPUT "NUMBER";L
KG 140 IF L<3 OR L>5 THEN 130
QH 150 PRINT CHR$(147)
EB 160 PRINT
AC 170 PRINT
GB 180 PRINT "GOTO";L*100
BG 190 PRINT CHR$(19)
```

The program isn't finished, but you might like to see what we have so far. If you run it and enter 3 in response to the prompt, you'll find the program stopped with the cursor blinking over a line that says GOTO 300. To execute that direct command, all you'd need to do is press RETURN. When we complete the program, it will press RETURN by itself. Finish the program by entering these lines:

```
PS 200 POKE 208,1
BM 210 POKE 842,13
CD 220 END
JX 300 PRINT "THIS IS LINE 300"
DE 310 GOTO 120
DF 400 PRINT "HERE'S 400"
QP 410 GOTO 120
KS 500 PRINT "LINE 500 IS THE END"
```

It's as easy as that. Once you grasp the basic method, all sorts of interesting applications come to mind.

You may print more than one command on a screen line. Just as in a program line, separate the multiple direct-mode commands with colons. You can use more than one screen line of direct-mode commands as well. However, you must be careful to put the commands in exactly the right place, and make sure the cursor flashes directly over the line to be executed when the program stops.

Here are some applications for the dynamic keyboard technique:

- Allow a user to enter a formula that the program will use;
- Allow a program to load another program;
- Allow a program to modify itself (tricky);
- Run test programs to determine, for instance, how the computer responds to certain direct commands and calculations.

Keyboard Buffer Locations

The keyboard buffer counter on the 128 is at location 208, and the start of the buffer is 842. Usually, your program must POKE a value of 1 into the counter, and a value of 13 (the character code for RETURN) into the buffer. That tells the computer there's one RETURN character in the buffer waiting to be processed. If there's more than one line of direct-mode commands on the screen to be performed, you need a higher count and more characters.

Entering a Formula

Let's write a brief program that allows a student to enter a formula and then generates a table of values based on the formula. More complex versions of the program might solve an equation or draw a graph, but we'll keep the example simple. In practice, it would be wise for your program to check for valid syntax before evaluating the formula. Again, for the sake of brevity, we'll do only the dynamic keyboard portion.

```
MR 100 PRINT"{CLR}{DOWN}FORMULA EVALUATION.":PRINT
       "INPUT A FORMULA"
JC 110 PRINT"BASED ON{SHIFT-SPACE}VARIABLE X":PRIN
       T"SUCH AS:":PRINT"{DOWN}{2 SPACES}Y= X*7-SQ
       R(X)":PRINT
RK 120 PRINT"YOUR FORMULA:":INPUT"{DOWN}{2 SPACES}
       Y=";F$:PRINTCHR$(147):PRINT:PRINT
CE 130 PRINT"Y=";F$;":GOTO150":DIMV(10):FORX=1TO10
       :PRINTCHR$(19)
HA 140 POKE 208,1:POKE842,13:END
FJ 150 V(X)=Y:NEXT X:FOR X=1 TO 10:PRINT X,V(X):NE
       XT X
```

Notice how this program does a task which would be difficult or impossible without using the dynamic keyboard technique.

Loading Another Program

If you put a LOAD command in a program, the new program doesn't load in the usual way. Instead, it's *chained* to the old program. The new program retains the variables and arrays (if certain rules are observed), and the effect is that of two successive programs working continuously on a single job. That's not always what is wanted. Especially with menu programs or bootstraps (program-loading programs), your goal may be simply to start the new program without preserving variables or data from the old one. That's what happens when you perform LOAD as a direct command. With the dynamic keyboard technique, we can simulate this from within a program.

Let's write a simple dynamic keyboard loading sequence.

```
MQ 100 PRINT"{CLR}{DOWN}PROGRAM LOADING":PRINT"PRO
       GRAM{2 DOWN}":PRINT"PROGRAM NAME":INPUTP$
BR 110 PRINT"{CLR}":PRINT:PRINT:PRINT"LOAD";CHR$(3
       4);P$;CHR$(34);",8":PRINT:PRINT
RQ 120 PRINT:PRINT:PRINT"RUN":PRINTCHR$(19):POKE20
       8,2:POKE842,13:POKE843,13
```

Note that there are two separate command lines: one for LOAD and one for RUN. Of course, it's important to position the lines correctly, but that's not hard to work out when you set up the program. You see everything happening on the screen, and, if you've placed your command a line too high or low, the problem is easy to spot.

Tricks and Advanced Points

On computers with color capabilities, you can hide your dynamic keyboard tricks if you wish. If you print the direct-mode commands in the same character color as the screen background, they won't be visible to you, but the computer can still see and execute them. Your program can even change colors as it runs so that some parts of the commands are visible and some are not.

Occasionally, you'll want to use the dynamic keyboard technique to change a program as it runs. That's always tricky, and if you are in 64 mode, anytime you add or change a program line, the values of all variables are lost. It's hard to run a

program when its variables disappear, but it can be done if handled carefully. The critical variables can be reentered using the dynamic keyboard technique, using lines such as X=7:L=120: GOTO 580. Another, somewhat more cumbersome method is to POKE the value of each variable into spare memory and PEEK the value later when needed. In 128 mode, variables are not a problem as variables are not disturbed by changing the program.

Why would a program need to change itself? The most usual situation involves converting an ASCII program listing into tokenized BASIC format. It's common to list programs in ASCII (untokenized) form when translating from one computer to another. This is especially true when you transfer programs over the phone line with a modem. As each line of the ASCII listing arrives, it must be entered as if it were being typed, to store it in tokenized format. While it's possible to do the whole job by hand (by printing each line on the screen and pressing RETURN), the dynamic keyboard technique lets the computer do this busywork for you.

Self-Editing Programs

The usual way to change a program is to type in a new line and press RETURN. The line is either added to the program or it replaces an existing line with the same line number. A program can do this, too, using the dynamic keyboard technique.

You might be wondering why you'd ever need to design a program that modifies itself, anyway. Here's an example. Suppose you have something in a special part of memory—a machine language program, a screen picture, or a data table. Whatever it is, you want to take the information and build it into a series of DATA statements so that it can be reconstituted by a BASIC program when needed. Perhaps you'd like to publish a small machine language program in a newsletter or magazine, and want readers to be able to type it in as DATA statements rather than the more complex hexadecimal code. How to do it?

First, let's write some data into memory so that you'll have something to convert to DATA statements. Here's a quick program to put a series of prime numbers into memory locations 3072–3125:

```
HD 100 POKE 3072,2
DF 110 POKE 3073,3
RD 120 N=3
GC 130 FOR A=3074 TO 3125
HM 140 N=N+2
RE 150 FOR M=3 TO SQR(N)+.1 STEP2
MR 160 T=N/M
AF 170 IF T=INT(T) THEN 140
BD 180 NEXT M
GP 190 PRINT N,
EX 200 POKE A,N
BS 210 NEXT A
```

That's not the most efficient prime number generator, but it does put the numbers into memory. The last number should be 251. Now, suppose you want these values in DATA statements so that a different program will be able to POKE them back at the start of the run.

Frenzied Activity

Type NEW to make space for the new program.

```
BH 10 L=100:A=3074
MF 15 PRINT CHR$(147):N=0
DG 20 PRINT
CH 25 PRINT
EK 30 PRINT L;"DATA";
SB 35 D$=STR$(PEEK(A))
SA 40 IF N>0 THEN D$=","+MID$(D$,2)
QE 45 PRINT D$;
BC 50 A=A+1:N=N+1:IF N<10 AND A<3126 THEN GOTO 35
DK 55 PRINT
JX 60 L=L+10
EA 65 PRINT "GOTO 15"
XX 70 PRINT CHR$(19)
JQ 75 POKE 208,1:IF A<3126 THEN POKE 208,2
XJ 80 POKE 842,13:POKE 843,13
GF 85 END
```

Be sure to type the semicolon at the ends of lines 30 and 45. When you run the program, you'll see a frenzy of activity on the screen for a few moments. Then the action stops with the cursor over a line which says GOTO 15. Don't execute this line. Instead, move the cursor down, type LIST, and press RETURN. You'll find that the program contains six new lines of DATA statements.

Start the new DATA lines at line 100 (variable L). Since the data maker program ends at a lower line number, there's no danger of replacing existing lines with new ones. The Commodore 128 in 128 mode, unlike the Commodore 64 and the 128 running in 64 mode, does not reset the variables back to zero when a program line is added to a program. Therefore, it's necessary to set the counter, N, back to zero on line 15 and to simply increment L, line number, by ten.

After the DATA lines have been created—you've generated only a few—you might want to get rid of the program that made them. You could do this manually by clearing the screen and giving the direct command:

FOR J=10 TO 85 STEP 5:PRINT J:
NEXT J

This prints the line numbers on a blank screen. You could then move the cursor back and strike RETURN 16 times, eliminating the lines. It would take a little ingenuity, but you could even cause the program to wipe itself out using the dynamic keyboard. (Hint: Crunch the program into fewer than ten lines—then stuff the keyboard buffer with the same number of RETURN characters.)

It's been a long voyage. If you've stayed with it, you can probably see how the dynamic keyboard technique expands what you can do with the computer. Though it requires care, it also creates new possibilities. "Dynamic keyboard" is not just a buzzword, although you may add it proudly to your vocabulary. It's a new resource.

Jump Search

Jerry Sturdivant

Learn how the binary search method can speed up data handling. The short demonstration program listed here runs in either 64 or 128 mode.

Searching for a specific item in a collection of data is a fundamental computing task. Word processors, databases, and address book programs all need to locate data quickly and accurately. This article shows how to use the simple binary search method in BASIC programs for efficient data handling.

For a demonstration, type in, save, and run "Jump Search." Program 1-5 will operate in either 64 or 128 mode.

The demo program creates a list of ten city names in alphabetic order, with population figures for each city (of course, an actual program would contain much more data). Lines 100–140 store the city names in a string array and the population figures in a matching numeric array. Once this is done, you can find the population of any city in the list by searching for its name. For example, if your search finds that AKRON is stored in array element S$(2), then the population for Akron can be found in the numeric array element PP(2).

The city names are stored in the array in alphabetical order because *this search technique works only on data that has been arranged in alphabetic or numeric order.* If you consider the situation for a moment, you'll realize that no organized searching method can speed up the hunt for a particular item in a randomly arranged set of data. If you can't tell whether a word you've found should come before or after the word you're looking for, then you'll have to examine every word in the list until you find an exact match. Arranging the data into alphabetic or numeric order, called *sorting,* is a separate problem. Just remember that only ordered data can be searched efficiently.

The simplest way to find a word in an alphabetical list is to start at the A's and hunt forward through the alphabet until you find a match. A sequential search of this type is very easy to program (all you need is a FOR-NEXT loop), but it's also slow and inefficient. When the target word is toward the end

of the alphabet, sequential searching wastes a lot of time looking through all the preceding words.

Jump to the Center

The binary search method (called *binary* because it repeatedly divides the data list in half) is much faster. Rather than starting at the beginning of the alphabet, it jumps in at the center. Let's look at the example program to see how this works.

The variable B stands for the beginning of the word list, E stands for the end, and C represents the center. Say that your target word is ATLANTA. When the search begins, line 200 finds the center of the ten-word list and jumps to that position (in this case finding the sixth word, ANAHEIM). Since ANAHEIM doesn't match ATLANTA, the program skips to line 250 for a critical test.

At this point the database is divided into two blocks, lower and higher. The program first decides which block holds the target word, then jumps to the center of that block to continue the search. Since ATLANTA comes after ANAHEIM in the alphabet, it must be stored in the higher block of words. Note that in just one step, you've eliminated the need to look at anything in the first half of the database. A sequential search (which compares ATLANTA to ABILENE, then to AKRON, then to ALBANY, and so forth) takes six steps to accomplish the same result.

Now it's time for the second jump. Lines 260–270 set a new beginning point just above the center (B = C + 1) and go back to line 200. The program finds the center of the new list (which consists of four words, ANCHORAGE to AUSTIN) and jumps to that position. This time the target word matches the found word. While the binary method found the target word with only two comparisons, a sequential search would require nine (eight comparisons to eliminate ABILENE through ATHENS, and a ninth to confirm ATLANTA).

The more data you have, the more time the binary method saves. For instance, if the list contains 1,000 words, most words are found in about 8 comparisons (the sequential method usually requires hundreds). If you expand the list to 10,000 words, only about 12 comparisons are required (compared to thousands for the sequential method). The secret lies in the halving technique. By repeatedly chopping the list in

half, this method quickly eliminates large chunks of data from consideration and zeros in on the target. Of course, you're not limited to string data. With slight modifications this routine can search numeric data as well.

Program 1-5. Jump Search

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
CG 100 N=10
MQ 110 DIM S$(N),PP(N)
HD 120 FOR I=1 TO N
KJ 130 READ S$(I),PP(I)
AS 140 NEXT I
XR 150 E=N
KC 160 B=1
XG 170 P=0
SE 180 PRINT "ENTER CITY"
SD 190 INPUT C$
HF 200 C=INT((E+1-B)/2)+B
QE 210 IF E-B<3 THEN 300
AJ 220 IF C$<>S$(C) THEN 250
HB 230 P=C
AC 240 GOTO 340
GC 250 IF C$<S$(C) THEN 280
MP 260 B=C+1
JC 270 GOTO 200
QS 280 E=C-1
AD 290 GOTO 200
HB 300 FOR I=B TO E
XH 310 IF C$<>S$(I) THEN 330
JJ 320 P=I
DM 330 NEXT I
JK 340 IF P<>0 THEN 370
RQ 350 PRINT "DATA NOT FOUND."
MM 360 GOTO 150
XG 370 PRINT S$(P),PP(P)
CP 380 GOTO 150
EP 999 REM CITY & POPULATION DATA
PC 1000 DATA ABILENE,89000
QH 1010 DATA AKRON,237000
KR 1020 DATA ALBANY,250000
CH 1030 DATA ALBUQUERQUE,332000
XR 1040 DATA ALVERINA,29000
HH 1050 DATA ANAHEIM,219000
RK 1060 DATA ANCHORAGE,174500
KP 1070 DATA ATHENS,150000
PC 1080 DATA ATLANTA,425000
XX 1090 DATA AUSTIN,346000
```

Coder-Decoder

W. M. Shockley

Protect the privacy of your DATA statements with this short routine that scrambles and restores any text. It's useful in almost any program that keeps information in DATA statements.

Probably the most convenient way to store lists of information in BASIC programs is to use DATA statements. A word game like Hangman, for instance, might have 50–100 words in DATA. The questions and answers in a trivia game would fit nicely in DATA statements. An adventure game would contain lists of rooms and their treasures. A history quiz would contain names and dates. There are many possibilities.

But DATA statements aren't very secure. Someone can easily list the program, where the words, questions, rooms, history facts, and so on, are right there for the user to read or memorize. In other situations—a personal diary, say—you want the information kept secret from anyone but yourself.

Scrambling Characters

"Coder-Decoder" is a short utility program which transforms normal DATA inputs into seeming gibberish. If the program is listed, the DATA statements are almost impossible to read. A second part of the program (lines 63210 on) retranslates the gibberish into the original DATA statements.

Type in the program and save a copy. Coder-Decoder allows DATA statements to be typed in directly, without line numbers or the word DATA. It uses the dynamic keyboard technique to add DATA statements to memory. The Coder section (lines 63010–63130) can be used as a subroutine to generate statements for a program already in memory. It can be added as is. Once it's in memory with the program, just type **RUN 63010**. It will continue until the word END is typed at the prompt.

Adding It to a Program

The two routines are short enough so that they can be listed on the screen (after being loaded) and added to a program on

40-column computers. Load the Coder portion of the program and list it on the screen. Load the program to which it is to be appended. Then go to the top of the screen and press RETURN enough times to enter the lines of the Coder routine into the program in memory.

The Decoder section (lines 63210–63300) can be added in the same way. This routine turns the DATA statements back into what you originally typed in. The DATA statements are read into the variable A$. After decoding, an unscrambled word is returned to the program as B$. When you have more than one DATA statement, use a FOR-NEXT loop to retrieve the coded words.

There are a couple of restrictions which must be observed. Commas, colons, and semicolons cannot be used in the inputs. The letters and numbers and extra characters which can be used are listed in S$, defined in line 63220. Others can be added by extending S$ and S1$ and the 41 in the R loop (line 63080 in the Coder routine and line 63260 in the Decoder routine).

In addition, each input must be no longer than 116 characters on the 128, and up to 70 characters on the 64, VIC, Plus/4, and 16.

Program 1-6. Coder-Decoder

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
MH 63000 REM **** CODER ****
DC 63010 X=10000
CX 63020 Y=842:Z=208:B$=""
XK 63030 PRINT"{WHT}{CLR}"
DP 63040 S$=".ABCDEFGHIJKLMNOPQRSTUVWXYZ?!12345678
         90' "
EG 63050 S1$=" !?ZYXWVUTSRQPONMLKJIHGFEDCBA.098765
         4321'"
HX 63060 PRINT"ENTER DATA TO BE CODED ('END' TO QU
         IT):":INPUTA$:IFA$="END"THEN END
HC 63070 FORN=1TOLEN(A$)
SR 63080 FORR=1TO41
AM 63090 IFMID$(A$,N,1)=MID$(S$,R,1)THENC$=MID$(S1
         $,R,1):R=41
XX 63100 NEXTR
KR 63110 B$=B$+C$:NEXTN
RB 63120 PRINT"{CLR}{BLK}"X"DA"CHR$(34)B$CHR$(34)"
         {2 DOWN}X="X+1":GOTO63020"
SX 63130 POKEY,19:POKEY+1,13:POKEY+2,13:POKEY+3,13
         :POKEZ,4:END
```

```
AP 63200 REM **** DECODER ****
BM 63210 S1$=".ABCDEFGHIJKLMNOPQRSTUVWXYZ?!1234567
         890' "
BD 63220 S$=" !?ZYXWVUTSRQPONMLKJIHGFEDCBA.0987654
         321'"
BK 63230 PRINT"{CLR}ENTER # OF DATA STATEMENTS TO
         {SPACE}DECODE:":INPUTN:FORL=1TON:B$=""
RF 63240 READA$:REM GET FROM DATA STATEMENT
KS 63250 FORN=1TOLEN(A$)
KG 63260 FORR=1TO41
BB 63270 IFMID$(A$,N,1)=MID$(S$,R,1)THENC$=MID$(S1
         $,R,1):R=41
HR 63280 NEXTR
FX 63290 B$=B$+C$:NEXTN
KP 63300 PRINTB$:NEXTL:REM USE B$ IN YOUR PROGRAM,
          DELETE PRINTB$ IF NECESSARY
```

Exploring the 128's Monitor

Richard Mansfield

Unlike the VIC and 64, the 128 has a built-in monitor. A monitor can be a valuable tool for machine language programmers. It can be especially useful for debugging ML routines.

Let's take a look at a special feature built into the new Commodore 128 which makes life easier for ML programmers. The several early Commodore machines—the PET, 8032, and SuperPet which preceded the VIC and 64—all had a doorway to ML called a *monitor*. Open this door and you go down below BASIC, down into the computer's engine room where you can get close to the microprocessor and the computer's memory.

The VIC and 64 did not come with a monitor (though you can buy add-on monitors or find monitor programs like *Supermon* and *Micromon* in books). Nonetheless, it's nice to have a resident monitor, a permanent, easy passageway into the heart of the machine.

On the 128, you just type MONITOR or press F8, and you're in a different world, with new prompts and new commands. Here's a rundown of the various commands:

- *Compare* takes a look at two separate sections of memory and prints the address of any locations which differ. You might use this to see whether two programs differ or to quickly determine which version is the more recent. (Each command is a single letter, and all numbers are hexadecimal (base 16). Compare, for example, is activated with C 1000 2000 8000 which prints out differences in memory between $1000–$2000 and $8000–$9000.)
- *Fill* fills a range of memory with a specific value. It's sometimes used to lay down a zone of zeros in, say, the second cassette buffer, prior to running a commercial program. Then, you can go back and look at the blanket of zeros and quickly see which locations are being used by the commercial program for storage.

- *Go,* like SYS, starts the execution of an ML routine, which should end with a BRK if you want to reenter the monitor.
- *Hunt* is particularly useful for tracking down ROM locations that you might want to use. If, for example, you know that a 64 ROM routine to print out numbers starts with PHA:TAX, you could request a hunt for this pattern of bytes in 128 ROM and find out where to JSR. (H 1000 2000 48 AA would look for PHA:TAX.)
- *Jump,* which is undocumented in the *System Guide,* performs a JSR to an ML routine. It does the same thing as *Go,* with one exception. If you jump to a routine that ends with RTS, you return to the monitor. But if you Go to the same routine, the RTS sends you back to BASIC.
- *Load* loads a file from tape or disk.
- *Memory* goes through a specified range of memory and prints the numeric *and* character values therein. This can be used both for locating special areas of ROM like the BASIC keywords or for checking that your ML program is working correctly by looking at its buffers, pointers, and so on, after a BRK in a running program.
- *Registers* shows you what's currently in the Accumulator, X, and Y registers as well as the program counter (where you were in memory when BRK took effect) and the status register. The registers are automatically printed onscreen when you enter the monitor. It's quite useful when debugging to see where you hit a BRK and what's going on with the registers. This is the equivalent of inserting STOP in your BASIC programs and then asking for variable values with ? VARIABLENAME so you can try to locate where things are going awry.
- *Save* is very valuable. You can save any section of memory to disk or tape, even the screen. If you've tried to save machine language programs on the VIC or 64 without a monitor or assembler, you know how useful a built-in ML Save command is (S "NAME",08, 1000,2000).
- *Transfer* sounds better than it is. It allows you to move any section of memory to another location. Unfortunately, most ML isn't relocatable (JSRs, and so forth, still target their old addresses). The best way to relocate ML is to use an efficient assembler where you can simply change the start address and reassemble at the new location.

The monitor also allows you to change the values in the registers (to set up a test); to directly modify the bytes in memory (not too useful unless you're typing in a "hex dump" type listing from a book or magazine); and to directly type in mnemonics (very useful when you want to insert a BRK, NOP, or test values). Possibly the most valuable tool in the monitor is the disassembler. Like BASIC's LIST command, a disassembler will display the fundamental source code of any ML, and then you can directly modify it and test it again. There's also a mini-assembler, but unless you're creating a very short ML routine, it's far better to stick with more effective assemblers.

Some of the facilities of the monitor are more valuable than others, and it would have been nice to have a single-step trace function. But when you're trying to hunt down those elusive bugs in an ML program, there's nothing like having a good, built-in monitor only a function key away. In fact, on some computers the monitor is called, simply, *the debugger.*

Important 128 Memory Locations

Jim Butterfield

This abridged memory map shows key locations of the Commodore 128 in 128 mode. Included are decimal and hexadecimal addresses, and brief descriptions of the functions of each location.

This memory map applies to the Commodore 128 when used in the 128 mode. In 64 mode, the machine's map is identical to that of the Commodore 64.

There are 28 pages (256 bytes each) of overhead before the start of BASIC. The following list shows some of the more important locations.

Architecture

Bank numbers as used in the BASIC BANK command and the built-in machine language monitor's addressing scheme are misleading. The banks do not represent different physical blocks of memory, but rather different arrangements of the RAM and ROM in the computer; configuration numbers might be a more appropriate term. Bank 0 shows RAM level 0, which contains work areas and the user's BASIC program. Bank 1 also shows RAM—this time (for addresses above hexadecimal $0400) level 1, which contains variables, arrays, and strings. Other "banks" are really configurations, with various types of ROM or I/O chip registers overlaying RAM. Thus, bank 15 is BASIC and Kernal ROM and I/O chip registers covering part of RAM bank 0. Bank 14 is BASIC and Kernal ROM and the character generator ROM overlaying part of RAM bank 0. Architecture is set so that addresses below $0400 in all banks reference bank 0 only. Memory configurations other than the 16 predefined banks can be achieved by storing a mask value in address $FF00, or calling up prestored masks by writing to $FF01–$FF04.

Memory Map

All Banks:

Hex	**Decimal**	**Description**
0000–0001	0–1	8502 on-chip I/O port, similar to 64
000F	15	Type: $FF=string; $00=numeric
0010	16	Type: $80=integer; $00=floating point
0015	21	Current I/O prompt flag
0016–0017	22–23	Integer value
002D–002E	45–46	Pointer: start-of-BASIC (bank 0)
002F–0030	47–48	Pointer: start-of-variables (bank 1)
0031–0032	49–50	Pointer: start-of-arrays (bank 1)
0033–0034	51–52	Pointer: end-of-arrays (bank 1)
0035–0036	53–54	Pointer: string storage (bank 1)
0039–003A	57–58	Pointer: limit-of-memory (bank 1)
003B–003C	59–60	Current BASIC line number
003D–003E	61–62	Pointer: current character in BASIC text
0041–0042	65–66	Current DATA line number
0043–0044	67–68	Pointer: current DATA item.
0047–0048	71–72	Pointer: current BASIC variable name
0049–004A	73–74	Pointer: current variable address
0063	99	Floating point accumulator 1: exponent
0064–0067	100–103	Floating point accumulator 1: mantissa
0068	104	Floating point accumulator 1: sign
006A–006F	106–111	Floating point accumulator 2: exponent, and so on
0070	112	Sign comparison, accumulator 1 versus 2
0071	113	Floating point accumulator 1 low-order byte (for rounding)
007D–007E	125–126	Pointer: BASIC runtime stack
0090	144	Status word ST for serial/tape operations
0091	145	STOP and RVS flags
0098	152	Number of open files
0099	153	Current input device, normally 0
009A	154	Current output CMD device, normally 3
009D	157	I/O messages: 192=all, 64=errors, 0=none
00A0–00A2	160–162	Jiffy clock high/medium/low
00AE–00AF	174–175	Ending address for LOAD, SAVE, and VERIFY
00B7	183	Number of characters in current filename
00B8	184	Current logical file number
00B9	185	Current secondary address
00BA	186	Current device number
00BB–00BC	187–188	Address of current filename
00C0	192	Tape motor interlock

Hex	Decimal	Description
00C8–00CB	200–203	RS-232 input/output buffer addresses
00CC–00CD	204–205	Keyboard decode pointer (bank 15)
00D0	208	Number of characters in keyboard buffer
00D1	209	Number of programmed characters waiting
00D3	211	Key shift flag: 0=no shift
00D5	213	Matrix coordinate of last key pressed (88 if no key)
00D6	214	Input from screen or keyboard
00D7	215	40/80 columns: 0=40 columns
00D9	217	Character base: 0=ROM, 4=RAM
00E0–00E1	224–225	Pointer to current text screen line
00E2–00E3	226–227	Pointer to current color line
00E4–00E7	228–231	Screen margins: bottom, top, left, right
00E8–00E9	232–233	Input cursor log (row, column)
00EB	235	Current cursor line
00EC	236	Current cursor column
00FB -00FE	251–254	Unused
0100–01FF	256–511	Processor stack area
0100–013E	256–318	Tape error log
0100–0124	256–292	DOS work area
0125–0138	293–312	PRINT USING work area
0200–02A1	512–673	BASIC input buffer
02A2–02AE	674–686	Routine to get a character from any bank
02AF–02BD	687–701	Routine to store a character in any bank
02BE–02CC	702–716	Routine to compare a character in any bank
02CD–02E2	717–738	JSR to another bank
02E3–02FB	739–763	JMP to another bank
02FC–02FD	764–765	Function execute hook
0300–0311	768–785	BASIC indirect vectors
0312–0313	786–787	Unused
0314–0315	788–789	IRQ vector
0316–0317	790–791	Break interrupt vector
0318–0319	792–793	NMI interrupt vector
031A–032D	794–813	Kernal vectors
032E–033D	814–829	Kernal links
033E–0349	830–841	Keyboard matrix shift vectors
034A–0353	842–851	Keyboard buffer
0354–035D	852–861	Tab stop bits
035E–0361	862–865	Line wrap bits

Hex	Decimal	Description
0362–036B	866–875	Logical file number table
036C–0375	876–885	Device number table
0376–037F	886–895	Secondary address table
0380–039E	896–926	CHRGET subroutine
0386	902	CHRGOT entry
039F–03D1	927–938	Subroutines to fetch from RAM banks
03DF	991	Floating point accumulator 1: overflow
FF00	65280	MMU configuration register
FF01–FF04	65281–65284	MMU load configuration registers
Bank 0 (BASIC programs):		
0400–07FF	1024–2047	40-column text screen memory
07F8–07FF	2040–2047	Sprite pointers
0800–09FF	2048–2559	BASIC runtime stack
0A00–0A01	2560–2561	Vector: BASIC restart
0A05–0A06	2565–2566	Bottom-of-memory pointer (bank 0)
0A07–0A08	2562–2563	Top-of-memory pointer (bank 1)
0A18	2584	Index to last character in the RS-232 input buffer
0A19	2585	Index to first character in the RS-232 input buffer
0A1A	2586	Index to last character in the RS-232 output buffer
0A1B	2587	Index to first character in the RS-232 output buffer
0A20	2592	Maximum number of characters in the keyboard buffer
0A22	2594	Key repeat flag: 128=all, 64=none
0B00–0BBF	2816–3007	Cassette buffer, also used by disk autoboot programs (CP/M or otherwise)
0C00–0DFF	3072–3583	RS-232 input, output buffers
0E00–0FFF	3584–4095	Sprite definition area (56–63)
1000–10FF	4096–4351	Programmed key lengths and definitions
117A–117B	4474–4475	Float-fixed vector
117C–117D	4476–4477	Fixed-float vector
11E9–11EA	4585–4586	Light pen values, x and y
1200–1201	4608–4609	Previous BASIC line number
1202–1203	4610–4611	Pointer: BASIC statement for CONT
1204–1207	4612–4615	PRINT USING characters (, . $)
1208	4616	Error type (ER) of last error
1209–120A	4617–4618	Line number of last error (EL)
1210–1211	4624–4625	Pointer: End of BASIC (bank 0)
1212–1213	4626–4627	Pointer: BASIC program limit
1218–121A	4632–4634	USR jump vector

Hex	Decimal	Description
121B–121F	4635–4639	RND seed value
1C00–FEFF	7168–65279	BASIC program text area
1C00–1FFF	7168–8191	Color memory (if hi-res screen is used)
2000–3FFF	8192–16383	Screen bitmap memory (if hi-res screen is used)
4000–FEFF	16384–65279	BASIC program text area (if hi-res screen is used)
Bank 1 (BASIC variables):		
0400–FEFF	1024–65279	BASIC variables, arrays, strings
Bank 14 Same as bank 15, below, except:		
D000–DFFF	53248–57343	Character generator ROM
Bank 15:		
4000–CFFF	16384–53247	ROM: BASIC
D000–D030	53248–53296	40-column video chip (8564)
D400–D41C	54272–54300	SID sound chip (6581)
D500–D50A	54528–54538	MMU memory configuration chip (8722)
D600–D601	54784–54785	80-column video chip (8563)
	R18–19	Video address, low/high
	R31	Video data, read/write
D800–D8E7	55296–56295	Color nybbles (for 40 column video)
DC00–DC0F	56320–56336	CIA 1 (6526)
DD00–DD0F	56576–56591	CIA 2 (6526)
DF00–DF0A	57088–57098	DMA controller
E000–FEFF	57344–65279	ROM: Kernal
FF47–FFFF	65351–65535	ROM: Jump table to important Kernal routines

All About CP/M on the 128

Howard Golk

CP/M is one of the oldest operating systems—but one of the newest available for Commodore users. We've included lots of practical examples, useful tips, and helpful notes on available CP/M software.

The Commodore 128 brings something very new to Commodore users: CP/M (Control Program for Microcomputers). Although CP/M was briefly available for the 64, it was a poor version which conformed to only a few of the standards for truly compatible CP/M software. With the 128 and 1571 disk drive, a 100 percent compatible version of CP/M has arrived. You're probably aware of the thousands of programs that run under CP/M. But before you invest a lot of time and money, there are a few things you should know about what CP/M is—and more importantly, what it is not.

A Fundamental Difference

All computers have an operating system (OS). The OS handles all the primary input, output, and housekeeping operations. When you type LOAD and press RETURN on a Commodore 64, you're instructing the OS to locate and read in a program from tape or disk. The OS is responsible for all communication between your programs and peripherals, such as the disk drive and display screen.

There are many different kinds of operating systems. Commodore computers have always had *dedicated* operating systems; that is, each model (PET, CBM, 64, VIC-20, and so on) contains its own customized operating system. Because software written for dedicated operating systems is not transportable from one machine to another, each model requires its own library of software. PET programs didn't run correctly (if at all) on 8032s or VICs and vice versa. Many other popular computers also have dedicated operating systems: Apple II, Atari, and Timex/Sinclair to name a few.

CP/M, however, is a transportable operating system. It was not written for any one particular computer. The idea is that programs written for an Osborne can theoretically be run on a Kaypro, Sanyo, Heathkit, or any other computer with CP/M. The early CP/M machines employed a standard eight-inch disk format (IBM-34). (Incidentally, the first CP/M machines were very expensive because they required 64K of memory, a massive amount at the time, to operate.)

Commodore operating systems are ROM based. The entire OS (which is mostly a collection of small machine language programs) is stored on chips inside the machine. This method of storage has many advantages. The computer generally performs fast, and all system commands are available at all times. However, ROM-based operating systems have a few disadvantages. Because the OS is stored on chips, it must be relatively small. ROM-based systems also are more difficult to upgrade or expand since this requires adding or replacing chips in the computer.

An alternative is a RAM-based operating system like CP/M. Rather than putting the machine code that makes up the operating system on chips, the code is on disk instead. RAM-based operating systems can generally be much larger than those which are ROM based—which means you can have many more commands and utilities. Upgrading RAM-based operating systems is much easier—you simply add or replace files on the system disk. The disadvantage to this kind of OS is that you amass a lot of system disks. These can sometimes be a source of frustration when you're doing routine jobs like copying files. Disk capacity is also a problem, since the OS can easily use up half the space on a disk. But once a program is running, the system disk is no longer required and can be removed. In fact, the system disk is useless while an applications program is running.

CP/M Versus Commodore

CP/M has many interesting advantages over dedicated operating systems like Commodore's. Not all of these advantages are features of CP/M particularly. Many are the results of the efforts toward compatibility with dozens of different computers by clever hardware and software developers.

Commodore users are accustomed to software that needs no installation. You just insert the program disk, type something like LOAD"*",8,1, and you're on your way. This is rarely the case with CP/M programs.

Software packages for CP/M computers must be installed for the particular hardware they're to run on. Since this process is required for compatibility, it gives every program the capability for a large degree of customization. For example, you can generally run CP/M software with any combination of disk drives. You tell the software which drives to use and how to access them. This eliminates a problem common to dedicated operating systems. All too often, your software expects specific hardware devices to be used. If your hardware is unusual in any way, you may be stuck. For example, some Commodore programs are designed to operate with a dual disk drive, but not two single drives.

When you install software, you provide it with the codes and parameters used by your hardware to do such things as clear the screen and move the cursor. Business software is usually without color, yet color can be added to many CP/M business programs by the user. This is possible because CP/M programs must be installed for your terminal (screen). While you're identifying the codes to use for things like reverse, underline, and so on, you can insert a few color codes.

Software is not easy to install if you're new to computers (it's often difficult for experts, too). CP/M requires much more dealer support, especially when installing the software. This is one of the reasons CP/M software is more expensive than software for the Commodore 64. New 128 owners will quickly learn that many good CP/M programs can successfully be installed *only* by the dealer.

Since the early days of Commodore, users have often preferred non-Commodore printers. Because of this, software developers for Commodore computers began providing several versions of their programs—each for a different printer. Eventually, these developers provided a method for users themselves to define the printer control codes for printers not on the list. With CP/M computers, everything is handled this way. The screen, printer, disk drives, memory capacity, and keyboard are all redefinable.

CP/M Command Structure

CP/M is disk based. Much of CP/M is located on the disk in the form of *COM files* (command files). When you type a command on the keyboard, the computer looks for a program on the disk by that name.

With CP/M commands, you can place a parameter after the command, and the operating system will pass that parameter to the command program. For example, if you type "DUMP MYFILE", the "DUMP.COM" program is loaded into memory and "MYFILE" (the parameter) is passed to it. In this case, the DUMP program will send the contents of "MYFILE" to the screen.

With CP/M, many applications programs depend on the operating system for part of their operation. Don't be surprised if a program you buy requires you to supply your own text editor to create and update data files. *CBASIC* from Digital Research is such a program. The CP/M disk itself includes a general-purpose text editor called "ED.COM", but reviews of this program are not exactly raves—just typing up a grocery list can be a nightmare. Nonetheless, it does allow you to manipulate text files.

One immediate use for a text editor is to create *batch files*. These are completely new to Commodore-only users—and they're extremely useful. All computers include commands for formatting disks, copying files, erasing files, loading programs, and so on. Many common housekeeping jobs require you to execute a series of these commands sequentially. Each time you perform a routine task (like making backup disks), you must type in the list of commands one at a time. With CP/M, you can put a long list of these commands into a disk file, then execute all the commands in the file by simply typing the filename (you may have to precede the filename with the word SUBMIT—depending on how your system is set up). The file that executes a series of commands is a batch file.

Batch files can even use variables as parameters. That way, the same batch file can perform a long series of system functions on different groups of files. In a sense, then, CP/M is both an operating system and a simple programming language. Under CP/M you can write programs that run other programs. As an example, suppose you have a batch file on your system disk called "PURGE.SUB" that contains:

```
PIP B:$2 = A:$1
ERA A:$1
```

The $1 and $2 are variables. When you type the batch filename followed by one or more parameters, the parameters will take the place of the variables. If you type PURGE SOMEFILE ANYFILE, the result would be the same as if you had typed

```
PIP B:ANYFILE = A:SOMEFILE
ERASE A:SOMEFILE
```

PIP will copy SOMEFILE from drive A to drive B and rename it as ANYFILE. ERA erases SOMEFILE on drive A. One of the nice features of CP/M is that you can rename commands. Try this:

```
RENAME COPY.COM = PIP.COM
```

Now you can use COPY instead of PIP. All other aspects of the command remain the same. Of course, if you used the PIP command in any batch files (like the one above), they would have to be changed. Alternatively, you can have *both* by making a copy of "PIP.COM" instead of renaming it (that is, PIP COPY.COM = PIP.COM).

The Transition

CP/M's design seems rather alien if you learned on a Commodore system. The disk system will no doubt be frustrating, especially with only one disk drive. Since CP/M is disk based, your disks are cluttered with "system utilities." To execute most CP/M commands, a COM file must be on the disk you're using. This can be maddening—often a Catch-22 situation. You place utilities (COM files) on disks, execute them, then erase them to free up disk space. You could, of course, just leave all your COM files on all your disks, but there would be little or no room left for your programs and data.

The CP/M operating system takes disk drives very seriously. Commodore's disk operating system (DOS) stores only a few items of information about files on a disk. Only the name, type, and size of the file are stored in the disk's directory. CP/M disks have a much more sophisticated directory. Commodore users will find a lot of new features with CP/M directories: Here's a sample:

Directory for Drive A: User 0

Name		Bytes	Recs	Attributes	Prot	Update	Access
DITS	BAK	1K	1	Dir RW	read	09/01/82 13:04	09/01/82 13:07
SETDEF	COM	4K	29	Sys RO	none	08/25/82 13:07	09/01/82 03:30
PURGE	SUB	1K	1	Dir RO	none	10/02/85 14:50	10/02/85 14:50

Total Bytes = 6K   Total Records = 31   Files Found = 3
Total 1K Blocks = 6   Used/Max Dir Entries for Drive A: 11/64

You can mark individual files as "read only" and prevent them from being altered or erased. You can hide files so that they do not show up in the disk's directory. You can even give files a password. CP/M will tell you the date and time a file was created and last updated (or the last time it was read). CP/M even knows if a file has been altered since the last time it was copied, which is a handy feature when updating backup disks.

CP/M computers often employ hard disk drives. To help organize the potential thousands of files on one disk, CP/M allows you to break up the directory into 16 "user areas." Essentially, the computer treats each directory as a different disk. To change user areas, type USER *n,* where *n* is a number from 0 to 15. User areas can be troublesome. When reformatting an old disk, you might erase important files because they're listed in another user area. To see the entire directory, type

DIR [USERS=ALL]

Mountains of Software

Why use CP/M anyway? Software—and lots of it, thousands of programs that do a multitude of things. If you need a program that calculates the net capacity of an oval salad bowl, or the number of toothpicks required to build a full-size boat, chances are there's a CP/M program out there to do it. Before you begin digging through the heap of available CP/M programs, let's look at a few items which might be of interest.

WordStar from MicroPro. Nearly every CP/M computer system contains a copy of this extremely powerful word processing program. It's so popular that it's almost become part of the CP/M standard. There is a close copy of *WordStar* called *NewWord* (from NewStar Software). It has some interesting advantages over *WordStar,* especially for systems with advanced features like those found on the new 128. When properly installed, *NewWord* shows bold and underlining on the screen. It's a true "what you see is what you get" word processor.

MBASIC-80 from Microsoft. There are thousands of programs written in *MBASIC*. Commodore users will quickly notice the lack of a screen editor. Many programmers use *WordStar* (or another text editor) to enter and edit BASIC programs. This is possible because *MBASIC* can optionally read and write program files in text form (nontokenized). This also makes it easy for BASIC programs to write other BASIC programs.

Turbo Pascal from Borland International. Many 128 owners will have purchased their machines specifically to run this fast and powerful language. It has many outstanding features and sells for under $50. If you write large programs, consider Pascal as an alternative to BASIC. Many consider that *Turbo Pascal* is fast becoming *the* definitive language for CP/M (and MS-DOS) computers. *Turbo* is even suitable for developing advanced programs like word processors and spreadsheets.

SuperCalc from Sorcim. An outstanding spreadsheet, powerful enough to be used even to work out math routines in your BASIC or Pascal programs. Like *NewWord, SuperCalc* is an "enhanced" version of another program, *VisiCalc* (from VisiCorp). *SuperCalc*'s documentation is built into the program itself. You can press the ? key anytime for instructions.

dBASE from Ashton-Tate. This is a simplified programming language designed specifically for database applications. You can learn to program *dBASE* in a fraction of the time required to learn an actual computer language.

What is CP/M best for? Business. Word processing and database programs run especially well under CP/M. The 80-column screen is considered a *must* for business applications. You won't find a lot of arcade-style games for CP/M, but you will find some excellent and lengthy adventure games (by John O'Hare). In general, graphics programs are few and far between.

Although we've mentioned BASIC and Pascal, you can get almost *any* language for CP/M, including Forth, C, PILOT, Logo, COBOL, FORTRAN, and many more. There are hundreds of user groups for CP/M also. Most offer free advice, technical information, and public domain software.

Hands On

Let's switch on your Commodore 128 with the CP/M disk in the drive. The computer will automatically come up in CP/M+ mode (also known as CP/M 3.0).

If you do not have an RGB monitor connected to your 128, something is rather odd from the start. The 40-column screen shows only half the computer's screen. The other half is sitting invisibly off to the right. If you move the cursor more than 39 characters to the right, the screen will shift over for you (to move more quickly, hold down the CONTROL key and press the cursor-right or cursor-left key on the top row of the keyboard). Why only half a screen? Because most CP/M computers have 80-column screens. Also, many CP/M programs format their output for an 80-column screen. This strange compromise was the result. It's best either to buy an RGB monitor or to connect the 80-column output to a monochrome monitor. (See your Commodore dealer for a special cable. The 80-column cables are available from at least three sources: Batteries Included, Cardco, and Microvation.) If you already own a color monitor, you can get 80 columns (in black-and-white only) with such a cable.

CP/M filenames contain three parts:

D: (DRIVE): Each disk drive is identified by a letter. The first drive is drive A, the second is B, and so on. The drive letter is always followed by a colon. In filenames, the drive letter identifies the location of the file.

FILENAME The filename can be from one to eight letters long. It can contain the letters *A–Z*, the numbers 0–9, and a few punctuation marks. To be safe, do not use punctuation marks in filenames. Usually, case is not important. CP/M translates lowercase to uppercase for all CP/M utilities. However, some programs (like *MBASIC-80*) allow upper- and lowercase filenames, but if used, CP/M utilities will not be able to access them.

.EXT A three-letter extension is optional (with a few exceptions). It usually identifies the type of the file. For example, all word processing files could have an extension of .TXT (for text). Or .DAT for data files, .BAS for BASIC programs, .LTR for letters, and so on. You can make up all the extensions you need. A few are reserved for the system (like .COM), and others are used by applications programs.

If you do not specify a drive letter, the default drive is used. This is the drive identified in the system prompt:

A> means A is the default drive.
B> means B is the default drive.

You can change the default drive by typing the desired drive letter followed by a colon (you would type the B: in this example):

```
A>B:
B>
```

Now the system will assume drive B whenever a drive letter is not specified for a file.

All the CP/M commands outlined in the 128 manual follow certain file naming guidelines. The system also contains a standard ambiguous file naming system that allows you to specify a group of files that have something in common.

The asterisk is a wildcard. As the name implies, anything will match it. Suppose your disk contains the following files:

```
LETTER.TXT   BOB.TXT    BUDGET.CAL   MAIL.DAT
SPOOL.PRN    DEBI.TXT   MARY.TXT     MLPGM.ASM
SID.COM
```

If we type

```
DIR *.TXT
```

the computer will respond with

```
LETTER.TXT   BOB.TXT    DEBI.TXT     MARY.TXT
```

The asterisk can be used along with letters:

```
DIR M*.*     MAIL.DAT   MARY.TXT     MLPGM.ASM
```

Another wildcard is the question mark. The asterisk matches items of any length. The question mark will match only one letter. In other words, *.* is the same as ????????.???. Here's an example using wildcards:

```
DIR *.?A?
BUDGET.CAL   MAIL.DAT
```

Only those files with an A in the second position of the extension are displayed.

Running Programs

The first programs you'll probably run are those found on the CP/M disk. You might spend hours trying to load programs in order to run them. If you're used to a Commodore, you'll see dozens of strange error messages if you try typing things like:

LOAD "PROGRAM"
or **LOAD PROGRAM**
or **LOAD PROGRAM.COM**
or **EXECUTE PROGRAM**
or **EXECUTE PROGRAM.COM**
or **RUN PROGRAM.COM**
or **ACCESS PROGRAM**

None of these work. CP/M automatically loads and runs a program when you type its name. Your CP/M disk contains a program called "HELP.COM" . To run this program, you need only type its name (excluding .COM):

HELP

The "HELP" program will then load and run. To exit, press RETURN.

If the program you want to run is not in machine language (or compiled), the proper language interpreter must be loaded first. A program written in BASIC will generally have an extension of .BAS. But you must first load a program such as *MBASIC*. You can do it all at once by typing

MBASIC ***PROGRAM***

where *PROGRAM* is the name of the BASIC program you wish to load. *MBASIC* will be loaded, then the BASIC program. The BASIC program will then run automatically. To exit *MBASIC*, type SYSTEM.

The Bottom Line

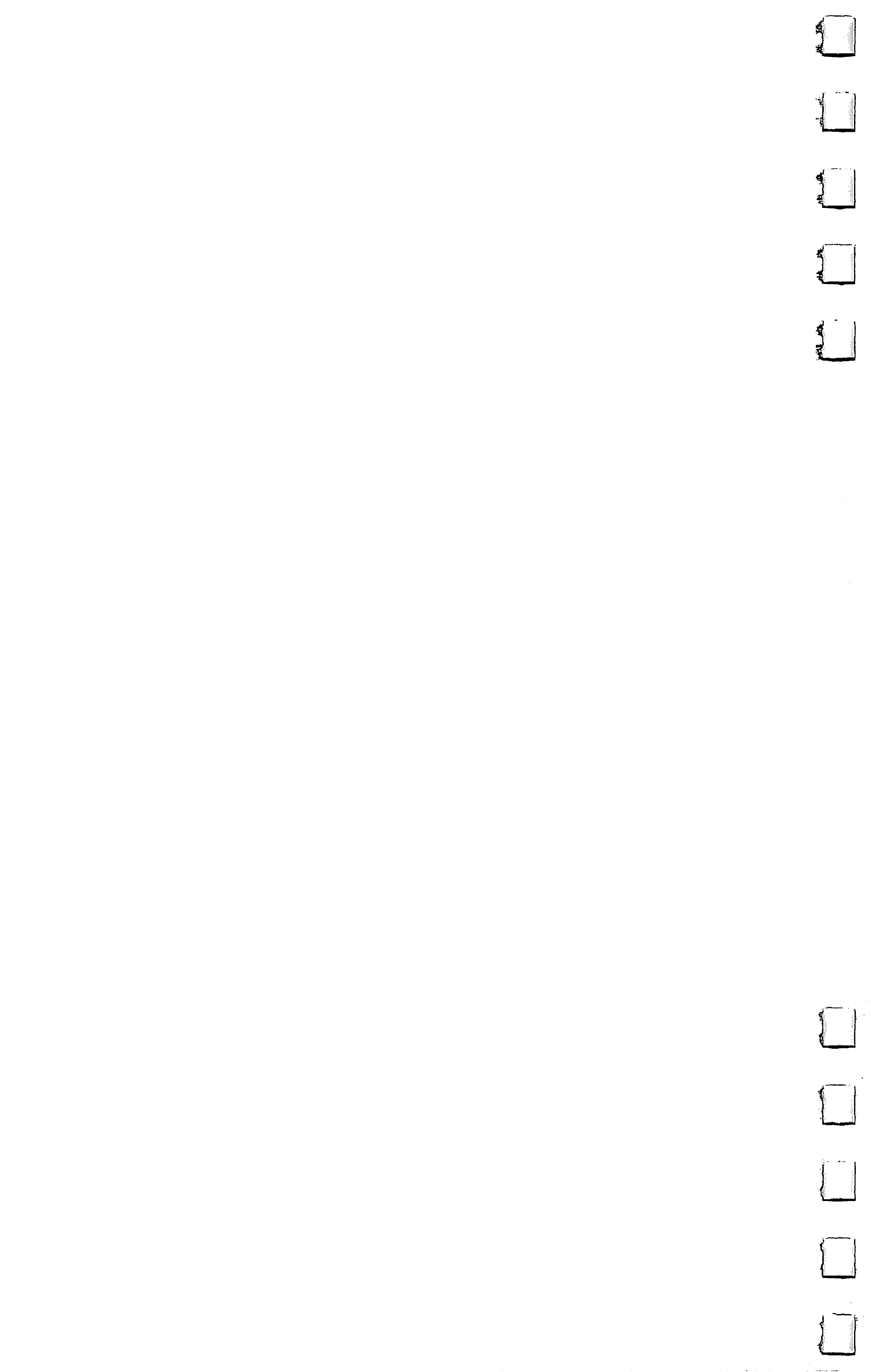
CP/M is a little cranky, somewhat sluggish, and rather unforgiving. But it has endured the test of time. The CP/M world is very complete: Every imaginable program, gadget, and utility is available in one or more forms for CP/M.

Commodore's 128 version of CP/M conforms to all the CP/M standards if it's run with the 1571 disk drive. However, if you run this version with a 1541 disk drive, be sure to bring a lunch. This configuration is very, very slow. Even a simple directory listing is extraordinarily slow.

Speed is not the only factor. The 1541 cannot read the disks from other CP/M computers. Without this capability, CP/M is practically useless. But with the 1571 and a 128, all the speed and versatility of CP/M are available.

Chapter 2

Sound and Graphics



Windows on the 128

Jim Vaughan

Creating windows is fast and easy on the 128. This tutorial covers the basics—what windows are and how to use them. Also included is a program that allows you to save the text area beneath a window.

The Commodore 128 is a powerful and versatile machine. Besides having 128K of user memory, 80- or 40-column screen output, and a powerful BASIC (7.0), it also has a built-in Commodore 64 and full CP/M capability.

While new programs for the 128 mode are beginning to emerge, it's still mainly up to the owner to explore the new horizons opened by BASIC 7.0. One of the most fascinating new commands added to the BASIC language is WINDOW. Windows have become increasingly popular within the personal computer industry in the past few years. Some word processors now use pull-down menus for help while preserving your text onscreen. Some windowing allows two separate programs to be run on the two halves of the screen.

Creating Your First Window

A window is simply a section of the screen that you partition off for your exclusive use. When you're in a window, the computer acts as if that portion of the screen is all there is. A program listing, a disk directory, or even a running program will be displayed in just one section of the screen. In this way, you can perform calculations or list programs in one section without disturbing the work you're doing elsewhere on the screen. The 128 offers two ways in which you can implement windows. Try this simple experiment. First, type in this line in direct mode (no line number), and press RETURN:

```
FOR I=1 TO 640:PRINT"*=";:NEXT
```

This fills your screen with a jumble of garbage, but it's sufficient to illustrate our example. Now, move the cursor to any point in the upper left part of the screen, press ESC and then T (ESC is the first gray key on the top row of the keyboard). Don't hold down the ESC key; press it once and re-

lease it, then press T. Now move the cursor to any point in the lower right side of the screen, and press ESC and then B. You've just created your first window—but it doesn't look like much, right? Now, press SHIFT-CLR/HOME. *Voilà!* Type in a few commands (DIRECTORY, for example) and see how the window keeps the screen output within the borders that you give it. It's easy to remember the keys: ESC-T (T for Top) sets the top left corner of the window, and ESC-B (Bottom) sets the bottom right corner.

This simple example illustrates the first method of windowing using direct mode. You can create a window anywhere on the screen with this technique. To restore your screen to its full format (80 × 25 or 40 × 25), just press the CLR/HOME key twice. This clears the window settings and resets your screen to normal. The direct method (ESC-T and ESC-B) is useful for quick calculations or program debugging. For example, I often wish to do some simple calculations while debugging a program, but I want to see the program listing also. It's easy. I just move my cursor off to the side of the listing, use the above sequence to create a window in direct mode, and calculate. The listing doesn't scroll off the screen while I'm trying to do some simple calculations. You can also use the window in direct mode to test out a program line to see its effect on the screen.

Adding Windows to a Program

Once you start playing with the above windowing technique, you'll no doubt think of many programming applications where windowing could be used. The ESCape key has an ASCII value of 27, so within a program you could position the cursor to the top left corner and then **PRINT CHR$(27); "T"** for the top of the window and then cursor down and right to **PRINT CHR$(27); "B"** for the bottom. But BASIC 7.0 provides an easier means to create a window: with the WINDOW command. This allows easy access to windowing from within your BASIC programs. The format for the command is

WINDOW ***X1,Y1,X2,Y2,CLEAR***

The variables X1 and Y1 are the screen coordinates of the upper left corner of the window, and the variables X2 and Y2 are the screen coordinates of the lower right corner of the window. CLEAR is an optional flag. If CLEAR is set to 1, it clears

the window area after it's created, and if CLEAR is 0 (or omitted altogether), any text on the screen remains there. The x values for the WINDOW command must be between 0 and 79 for the 80-column screen. The y values must be between 0 and 24.

Program 2-1, "Window Demo," will work either in 40- or 80-column mode. The program's purpose is to illustrate the use of windows in a program, but it also creates an interesting screen display while running. The program listing provides the basics for creating a general subroutine to handle windowing. Given four values (X1, Y1, X2, Y2), this routine will create the window, clear it of any text, and then create a border around the new window to set it off from the rest of the screen. It should be noted that this routine will create a window slightly larger than the one requested so that it can accommodate the border around the window. Program 2-1 is fine if you don't care about the text (or graphics) that will be written over when the window is created. But what about that pull-down menu that comes down onto the screen of your word processor or database? Surely, you don't want to lose any of that valuable data. The programming solution is to read in the data that lies beneath the window, save it in some buffer area, create the window, and then when you're done with it, restore the previous contents of the screen.

Your first instinct might be to go in and start PEEKing the appropriate screen locations and saving the data. This would work fine for the 40-column screen (memory locations between \$0400 and \$0800), but 80-column output is handled a bit differently. If you take a look at the abbreviated memory map in the back of your *128 System Guide,* you'll note that there are no memory locations listed for the 80-column screen. This is because the 80-column screen is stored internally in a 16K memory area which is not directly accessible to the user, and therefore cannot be read or written to via any commands in BASIC.

Although the 80-column screen is not directly accessible, it can be PEEKed and POKEd in machine language. So, to save part of the screen, we'll PEEK every character from the area under the window (screen memory is found in locations \$0000–\$0800 of the internal RAM of the 80-column chip) and save them to a buffer. It's also necessary to save attribute memory (\$0800–\$1000), which is the equivalent of 40-column color memory.

The Save Routines

Program 2-2 is designed to work with the 40-column screen, while Program 2-3 is for 80 columns. Both programs POKE a machine language program into memory at 8192. (Note that this is part of the hi-res screen area, so you must avoid graphics commands while using these programs.) To add the routines in your own programs, follow these steps:

1. Be sure to include the commands GRAPHIC1:GRAPHIC0 at the beginning of your program. This sets aside 9K of memory for the hi-res screen, memory which will actually be used by the ML routine.
2. After the routine has been POKEd into memory, you can save the contents of a window with SYS 8192. This SYS must come *after* you've used the WINDOW command. You can then clear the window and print the menu (or whatever you wish to place in the window).
3. To recall the previous contents of the window, insert a SYS 8195.

The two programs create a sample screen, put a window there, and then wait for a keypress. The screen underneath the window is then restored.

Program 2-1. Window Demo

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
EK 1ØØ MODE=RGR(G)
CP 11Ø REM *CHECK TO SEE IF IT'S A 4Ø OR 8Ø
       {1Ø SPACES}COLUMN DISPLAY*
JE 12Ø IF MODE=5 THEN BEGIN
JX 13Ø :{3 SPACES}A=78:B=4Ø:C=38
QG 14Ø :{3 SPACES}FAST
BX 15Ø BEND:ELSE BEGIN
KR 16Ø :{3 SPACES}A=38:B=2Ø:C=18:BEND
SQ 17Ø REM *START THE MAIN LOOP*
GG 18Ø SCNCLR
XK 19Ø PRINTCHR$(27)"M";: REM *SET NO-SCROLL*
FP 2ØØ X1=INT(RND(Ø)*B):Y1=INT(RND(Ø)*12)
BS 21Ø X2=INT((RND(Ø)*B)+C):Y2=INT(RND(Ø)*1Ø+12)
XA 22Ø IFX1>X2 OR Y1>Y2 OR X2>A OR Y2 >22 ORX1<2 O
       R Y1<2 THEN2ØØ
RB 23Ø REM *CREATE THE LARGER WINDOW AND
       {12 SPACES}DRAW THE BORDER*
BF 24Ø WINDOW X1-1,Y1-1,X2+1,Y2+1 ,1
AD 25Ø X=RWINDOW(Ø):Y=RWINDOW(1)
QG 26Ø PRINT"O";:FORI=1TO(Y-1):PRINT"{Y}";:NEXT:PR
       INT"P"
```

```
EF 270 FORI=1TOX-1:PRINT"{H}";TAB(Y);"{M}":NEXT
MP 280 PRINT"L";:FORI=1TO(Y-1):PRINT"{P}";:NEXT:PR
       INT"@"
PX 290 REM *CREATE WINDOW AND FILL IT*
JC 300 WINDOW X1,Y1,X2,Y2
HJ 310 A1=(RND(0)*38+40):IFRND(0)<.2THENPRINTCHR$(
       15);
GF 320 IFRND(0)>.9THENPRINTCHR$(18);
HB 330 IF RND(0)>.8 THEN BEGIN
PX 340 REM *CHOOSE NORMAL OR REVERSE SCREEN*
DS 350 :{5 SPACES}IF S$="N" THEN S$="R":PRINTCHR$(
       27)S$;:ELSE PRINTCHR$(27)"N";:S$="N"
AM 360 BEND
DK 370 REM *CHOOSE COLOR FOR DISPLAY*
AE 380 PRINTCHR$(149+D);:D=D+1:IFD>7THEND=0
BH 390 IFD=3THEND=4
SK 400 FORC1=0 TO (X * Y):PRINTCHR$(A1);:NEXT:PRIN
       TCHR$(143);CHR$(146);CHR$(5)
GM 410 GOTO200
```

Program 2-2. Window Save for 40 Columns

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
MX 100 GOSUB150:GRAPHIC 1:GRAPHIC 0:COLOR 0,1
EC 110 PRINT"{CLR}";:FOR A=1 TO 24:COLOR 5,(AAND15
       )+1-(A=16):PRINT"ABCDEFGHIJKLMNOPQRSTUVWXYZ
       1234567890ASZX";:NEXT
SH 120 WINDOW 5,3,35,13:SYS 8192
AB 130 PRINT "{CLR}{5 DOWN}{3 SPACES}PRESS ANY KEY
        TO CONTINUE"
AD 140 GETKEY A$:SYS 8195:SLEEP 2:GOTO130
JH 150 C=0:FORA=8192TO8335:READB:C=C+B:POKEA,B:NEX
       T:IFC<>20215THENPRINT"{CLR}DATA ERROR":END:
       ELSE RETURN
MJ 160 DATA 169,0,44,169,1,133,143,32,100,32,169,0
       ,133,250
QS 170 DATA 169,48,133,251,165,231,56,229,230,133,
       158,230,158,165
EF 180 DATA 228,56,229,229,133,159,230,159,165,158
       ,133,254,160,0
FR 190 DATA 165,143,208,7,177,141,145,250,76,57,32
       ,177,250,145
MC 200 DATA 141,200,198,254,208,236,165,250,24,101
       ,158,133,250,165
QE 210 DATA 251,105,0,133,251,32,130,32,198,159,20
       8,210,165,142
HM 220 DATA 201,212,176,11,165,139,133,141,165,140
       ,133,142,76,18
```

```
MX 23Ø DATA 32,96,165,23Ø,133,141,169,4,133,142,16
       6,229,24Ø,6
XR 24Ø DATA 32,13Ø,32,2Ø2,2Ø8,25Ø,165,141,133,139,
       165,142,24,1Ø5
RQ 25Ø DATA 212,133,14Ø,96,165,141,24,1Ø5,4Ø,133,1
       41,165,142,1Ø5,Ø,133,142,96
```

Program 2-3. Window Save for 80 Columns

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
RE 1ØØ GRAPHIC 1:GRAPHIC 5:GOSUB15Ø:COLOR Ø,1
FC 11Ø PRINT"{CLR}";:FOR A=1 TO 48:COLOR 5,(AAND7)
       +2:PRINT"ABCDEFGHIJKLMNOPQRSTUVWXYZ12345678
       9ØASZX";:NEXT
KQ 12Ø WINDOW 1Ø,3,7Ø,13:SYS 8192
QJ 13Ø PRINT "{CLR}{5 DOWN}{16 SPACES}PRESS ANY KE
       Y TO CONTINUE"
AD 14Ø GETKEY A$:SYS 8195:SLEEP 2:GOTO13Ø
GS 15Ø C=Ø:FORA=8192TO8377:READB:C=C+B:POKEA,B:NEX
       T:IFC<>24Ø72THENPRINT"{CLR}DATA ERROR":END:
       ELSE RETURN
JQ 16Ø DATA 169,Ø,44,169,1,133,143,32,118,32,169,Ø
       ,133,25Ø
QS 17Ø DATA 169,48,133,251,165,231,56,229,23Ø,133,
       158,23Ø,158,165
RA 18Ø DATA 228,56,229,229,133,159,23Ø,159,165,158
       ,133,254,165,142
EC 19Ø DATA 162,18,32,162,32,165,141,162,19,32,162
       ,32,16Ø,Ø
FQ 2ØØ DATA 162,31,165,143,2Ø8,8,32,174,32,145,25Ø
       ,76,75,32
GB 21Ø DATA 177,25Ø,32,162.32,2ØØ,198,254,2Ø8,232,
       165,25Ø,24,1Ø1
SG 22Ø DATA 158,133,25Ø,165,251,1Ø5,Ø,133,251,32,1
       48,32,198,159
DS 23Ø DATA 2Ø8,192,165,142,2Ø1,9,176,11,165,139,1
       33,141,165,14Ø
KP 24Ø DATA 133,142,76,18,32,96,165,23Ø,133,141,16
       9,Ø,133,142
RC 25Ø DATA 166,229,24Ø,6,32,148,32,2Ø2,2Ø8,25Ø,16
       5,141,133,139
JF 26Ø DATA 165,142,24,1Ø5,8,133,14Ø,96,165,141,24
       ,1Ø5,8Ø,133
JJ 27Ø DATA 141,165,142,1Ø5,Ø,133,142,96,142,Ø,214
       ,44,Ø,214
HE 28Ø DATA 16,251,141,1,214,96,142,Ø,214,44,Ø,214
       ,16,251,173,1,214,96
```

Advanced Commodore 128 Video

Jim Butterfield

Two valuable techniques worth mastering on any computer are being able to relocate screen memory and to set up a custom character set. When you run the example program, be ready for a surprise. For intermediate and advanced BASIC programmers.

You can do a lot of graphics on the Commodore 128 with an elementary knowledge of the new BASIC: circles, squares, lines, and points appear by means of simple BASIC commands. But advanced programmers may still need to get into the mechanics of video. Here's a simple exercise for 128-mode 40-column screens that will give a little insight into the "works."

The question often arises: How can I implement a new character set? Some people want to design their own personalized codes or graphics symbols for the screen; others are interested in foreign languages. In 40 columns, the 8564 video chip is practically identical to the 6567 of the Commodore 64. With a few new rules, we can put the chip's features to work in the same way.

Because the Commodore 128 makes it easy, I'll be including some hexadecimal addresses in the following listing. If you'd rather use decimal numbers, the computer will do quick conversions for you, and you can make the substitutions in the program.

Changing Addresses

Let's build the program step by step and note points of interest.

```
100 POKE 58,DEC("C0")
110 CLR
```

I'm planning to put the screen and its new character set into memory bank 1, at addresses $C000 to $CBFF—character set at $C000, screen at $C800. [By the way, if you'd rather use

the decimal value 192 instead of DEC("C0"), be my guest. I prefer C0 because it's easier to visualize it as part of the full address $C000. Be sure to type a zero and not the letter *O*, or you'll get an error.] Bank 1 is where BASIC puts its variables; we wouldn't want these to get mixed up with our screen. So we cut down the top-of-variable-memory pointer to $C000. There's really no danger of a memory conflict with this small program, but we might as well do it right.

The CLR command makes sure the other variable pointers don't get mixed up by this change.

120 TRAP 500

This command may be unfamiliar to many Commodore programmers. It sets up an *error trap* so that if anything goes wrong in the following code, the computer hops to line 500, which will restore the screen. This saves us from the horrible prospect of watching the program stop with a syntax error while the screen is still scrambled and unreadable. The TRAP command gives us another bonus: If the computer freezes—or is just too slow—we can press RUN/STOP, and the program zips to line 500 and wraps things up.

130 BANK 15

We're about to fiddle with the insides of computer chips (registers), so this command calls for memory bank 15 to make the chips accessible. This insures that the next few POKEs will be directed to the right place.

140 POKE DEC("DD00"),148

Except for the decimal number conversion ($DD00=56576), this POKE is identical to the way it's done on the Commodore 64. Briefly, it means this: Display video out of the memory slice in the range $C000–$FFFF. We haven't specified the bank yet, but we'll get around to it in a moment.

150 POKE DEC("0A2C"),32

We're still in bank 15, but this address isn't a chip. The address $0A2C (decimal 2604) is below $4000 (16384). When we're using bank 15, all such low addresses go to RAM, bank 0. This POKE sets the position of the character set and the screen within the video slice we've selected. The calculation goes like this: We want the screen to be at $C800, which is 2K

above the start of the video slice at $C000, so multiply the 2 by 16 and add a similar value for the character set. In this case, the character set is right at the start of the slice; so we add 0 to get a value of 32.

On the Commodore 64, we'd do exactly the same calculation, but we'd put the result in address $D018 (53272). In fact, that's the same address at which our value will end up in the Commodore 128, but we must let the computer's interrupt routine deliver it there for us. So instead of POKEing the value directly into $D018, we store it at $0A2C (2604). As part of the computer's interrupt procedure, it will copy the contents of this location into $D018.

```
160 POKE DEC("D506"),68
```

This tells the computer to take video from bank 1. If we wanted video from bank 0, we'd POKE a value of 4—or just leave this line out, since that's the value that will be there in any case.

```
170 POKE 217,4
```

This POKE tells the computer to take its video from RAM, not ROM. We don't need to give this one for the addresses we have chosen since there is no conflict. This very low address has a special banking rule: All addresses below hex $400 (1024) go to RAM bank 0, regardless of the bank which has been specified.

Relocating the Screen

Now our video is set up and ready to go. We'd better put something on the screen so we can see it working. It seems sensible to copy our old screen to the new place; then we'll copy the character set. We'll make a slight change so you can see how to create a new set of characters.

First, our screen must move from bank 0, address $400, to bank 1, address $C800. We must move the whole thousand characters.

```
200 FOR J=0 TO 999
210 BANK 0:X=PEEK(1024+J)
220 BANK 1:POKE DEC("C800")+J,X
230 NEXT J
```

This moves screen memory, but since the character set is not in place, the result would look rather muddy. We can read the character set by selecting bank 14; it is found in this bank at addresses $D000–$D7FF. There are 256 characters times 8 bytes per character, which means 2048 bytes to move. Just as we moved the screen in the lines above, we must move the character bytes one at a time, flipping between banks 14 and 1.

We'll also change the characters slightly as we move them. This allows us to see that indeed we've taken control of the character set.

```
300 FOR J=DEC("C000") TO DEC("C7FF") STEP 8
310 FOR K=0 TO 7
320 BANK 14
330 X=PEEK(J+4096+7-K)
340 BANK 1
350 POKE J+K,X
360 NEXT K
370 NEXT J
```

This puts the character set in place. When you run the program (after typing in the additional lines below), you should see your original computer screen—slightly changed. We could insert a delay loop to prolong the effect, but the screen takes long enough to change that you'll have plenty of time to see what happens.

Cleaning Up

We're finished—almost. We must be neat and put everything back the way it was. This also gives you a chance to see the original values that were in the various registers and addresses.

```
500 BANK 15
510 POKE DEC("DD00"),151
520 POKE DEC("0A2C"),20
530 POKE DEC("D506"),4
540 POKE 217,0
```

These lines restore the original screen. A little study should enable you to guess at what each POKE does—or undoes.

Finally, we need two last lines to complete the job. But there's an important note: *Do not* enter these lines until you've

tested the program and found it good. If your program has a problem, you'll want to be able to look at the variables (by using commands such as PRINT J) to find out what went wrong. These final lines make it impossible for you to do so.

```
550 POKE 58,DEC("FF")
560 CLR
```

We've given back to the computer its variable storage memory. And the job is complete.

Programming Music and Sound

Philip I. Nelson

Anxious to unleash your new Commodore 128's sound and music capabilities? Here are some practical examples of how to use the powerful new BASIC 7.0 commands in working programs.

One of the Commodore 128's most welcome features is its ability to make music and sound effects with simple BASIC commands. Gone are the days when it took hours of programming and multiple POKEs to create sound on a Commodore computer. Since your *128 System Guide* explains the basics of each command, we'll look at some programs that actually put them to work.

Musical Keyboard

Program 2-4, "Musical Keyboard," is lots of fun to use and also demonstrates how arrays can simplify your programs. It defines four rows of keys on the 128's keyboard as musical keys, giving you two separate one-octave keyboards. By pressing keys 0–9 on the numeric keypad, you can switch to any of the 128's ten predefined instrument voices.

Think for a moment how you would structure a musical keyboard program like this. It requires that you read the computer's keyboard, detect the pressing of certain keys, and translate those keypresses into musical notes. One way to do this would be with a long series of individual IF tests (IF A$="X" THEN PLAY "O3C", and so on). But that would be slow and inefficient. This program takes a different approach, using arrays that store the music data and simplify the keyboard-scanning process as well.

Take a look at lines 60–90, the setup portion. Both of the arrays (P$ and T$) are dimensioned with 256 elements, enough to hold all the possible keyscan values. Line 80 stores a PLAY string (O3C, O3#C, etc.) in each element of the P$ array that corresponds to the keyscan value (23, 18, etc.) of a key that we'll use to make music. Line 90 creates a similar array for se-

lecting different instruments with the numeric keypad keys. (Actually, these two arrays could be combined into one, but we want to display the instrument data separately.)

After the setup portion is complete, the program loops continuously through lines 20–50. The statement X=PEEK(212) returns the value of the last key pressed. (Location 212 performs the same function as location 197 on the 64 and VIC-20. The statement FOR J=1 TO 1E9:PRINT PEEK(212):NEXT lets you see the keyscan value of any key.) Lines 30–40 use the keyscan value as an index into the T$ and P$ arrays. The IF statements in these lines will be true only for those array elements in which we placed data: Every other element in the arrays is empty, containing nothing but a null string (" "). Note that the arrays make it possible to use a short, efficient working loop that doesn't slow the program as a multitude of IF statements would.

Since this program uses PLAY to make the actual notes, you may wonder why there's a SOUND command in line 60. The statement FOR J=1 TO 3:SOUND J,0,0:NEXT immediately silences all SOUNDs that may be in effect from a previous program (or your own experiments). When you're setting up a sound program, it's prudent to reset sound and music parameters to a known state to avoid unwanted residual effects. If you fail to take this precaution, previous sound commands (FILTER, etc.) may prevent your sounds from working properly. Of course, pressing RUN/STOP–RESTORE resets most sound parameters, but that's not a very elegant solution. Thus, line 70 ensures that various TEMPO and PLAY parameters are set as needed in this program (filter off, maximum volume, etc.).

Although PLAY can generate as many as three notes at once, the 128's BASIC can read only one key at a time. So this keyboard is necessarily monophonic. Machine language routines are necessary to create a polyphonic (chord-playing) keyboard.

128 Soundmaker

"Soundmaker," Program 2-5, is the shortest of the example programs, but it creates the most complex effects, using all three of the 128's voices simultaneously. Type in Soundmaker and save it to disk or tape (pay close attention to the punctuation in line 60). When you run the program, it spends a few seconds in preparation, then invites you to press any key.

Whenever you press a key, the 128 executes a new SOUND command and displays it on the screen for reference. As you'll soon discover, SOUND can create a dazzling variety of effects. All three voices are used, in 1-2-3 order, so if you keep pressing keys, you'll hear as many as three different sounds at once. The duration of each sound is limited to 100. If you want to hear individual sounds, wait until the current sound is done before pressing a key.

Note the difference in the way that SOUND and PLAY handle volume. SOUND does not produce any sounds at all unless you have previously set the volume to some nonzero value with VOL (line 70). PLAY, on the other hand, sets volume for itself with the U symbol, and pays no attention to VOL commands.

The frequency of each sound is determined by the ASCII value of the key you press. Keys with high values (like Z, ASCII 90) create higher pitched sounds than those with lower values (like the space bar, ASCII 32). Pressing SHIFT pitches the entire keyboard higher. The waveform and sweep direction for each sound are selected at random, while the minimum frequency and step value are held within reasonable ranges.

Song Player

Program 2-6 demonstrates a simple way to encode and play music on the 128. "Song Player," lets you enter PLAY strings under program control, adding them to the program as DATA statements with the dynamic keyboard method. After entering your music, you can replay it at any time or resave it along with the program. Pay special attention to the punctuation in lines 60 and 190, which cause the program to modify itself.

The music entry routine permits you to enter as many as 29 PLAY symbols at one time (blank spaces are acceptable, although PLAY ignores them). Consult the *128 System Guide* for an explanation of the various PLAY symbols. Before adding the PLAY string as a DATA statement, the program checks every character in the string to make sure it is legal. If you enter a character that the PLAY command does not understand (Z, for instance), the program signals an error and lets you try again. Note that while the program can tell whether a character is a legal PLAY symbol, it does not check for correct PLAY

syntax: You are still responsible for arranging the symbols in meaningful order. For example, the string "XU$#" contains legal PLAY characters, but causes an error when you try to PLAY it. If the PLAY string is accepted, the screen flashes briefly as the program adds the string as a DATA line, then the entry prompt reappears. You can return to the main screen by entering MENU or by typing RETURN without entering any characters.

Music data is added beginning with line 1000. Successive DATA lines are entered as 1001, 1002, and so on, up through 63998. Do not delete or renumber line 63999; it contains a string that marks the end of the music data. When you exit the program by pressing Q, it automatically modifies line 10 to let you resave the program complete with the new data. The next time you load and run the program, all the data will be there. Since this program modifies itself as it runs, don't renumber it or alter any lines unless you understand exactly how the dynamic keyboard processing works.

As short as they are, these program examples demonstrate a number of handy sound and music techniques. It's often preferable to use variables rather than literal values in sound commands. SOUND VOC, FRQ, DUR is just as valid as SOUND 1, 11000, 100—and considerably easier to understand. And replacing literals with variables lets you change the sound dynamically, just by redefining the variable. Since the computer can often look up a variable faster than it can interpret a literal, variables can also speed up a program somewhat.

PLAY accepts variables, too, so PLAY A$ and PLAY A$(23) work just as well as PLAY "C D E F". You may also concatenate PLAY strings and use other string operations such as MID$, LEFT$, and so on:

```
10 PLAY "X0U9S":P$="CDEFGAB"
20 FOR J=ASC("1") TO ASC("6"):
   FOR K=1 TO 7
30 PLAY "O"+CHR$(J)+MID$(P$,K,1)
40 NEXT:NEXT
```

PLAY accepts nearly any string construction that PRINT can handle. However, you may not separate PLAY strings with a comma or semicolon. One final reason to put strings into variables is that it simplifies debugging. If you're not sure

what a PLAY statement is doing, simply PRINT the string on the screen to see what it contains.

Program 2-4. Musical Keyboard

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
QH 10 GOSUB 60
DJ 20 GETKEY A$:X=PEEK(212)
HC 30 IF T$(X)<>""THEN PLAY T$(X):PRINT "{HOME}"SP
      C(7)"TUNE:"MID$(T$(X),2)
SE 40 IF P$(X)<>""THEN PLAY P$(X):PRINT "{HOME}"P$
      (X)"{2 SPACES}"
RD 50 GOTO 20
CP 60 DIM P$(256),T$(256):FOR J=1 TO 3:SOUND J,0,0
      :NEXT
BC 70 PLAY "U9 X0 T7 S":TEMPO 15
EP 80 READ K,P$:P$(K)=P$:IF P$<>"DONE" THEN 80
ED 90 READ K,P$:T$(K)="T"+P$:IF P$<>"DONE" THEN 90
XD 100 PRINT CHR$(147)SPC(10)"{RVS}{2 DOWN}MUSICAL
        KEYBOARD{OFF}":PRINT SPC(12)"4 5 6 7 8 9"
JP 110 PRINT SPC(11)"E R T Y U I O":PRINT SPC(12)"
       D F G H J K"
BD 120 PRINT SPC(11)"X C V B N M ,":PRINT "{HOME}"
       SPC(7)"TUNE:"MID$(T$(70),2):RETURN
RE 130 DATA 23,O3C,18,O3#C,20,O3D,21,O3#D
XE 140 DATA 31,O3E,26,O3F,28,O3#F,29,O3G
GK 150 DATA 39,O3#G,34,O3A,36,O3#A,37,O3B
MM 160 DATA 47,O4C,14,O4C,11,O4#C,17,O4D
CA 170 DATA 16,O4#D,22,O4E,19,O4F,25,O4#F
RC 180 DATA 24,O4G,30,04#G,27,O4A,33,O4#A
PA 190 DATA 32,O4B,38,O5C,256,DONE
DH 200 DATA 81,0,71,1,68,2,79,3,69,4,66,5
RP 210 DATA 77,6,70,7,65,8,78,9,256,DONE
```

Program 2-5. Soundmaker

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
EH 10 GOSUB 70
KH 20 GETKEY A$:V=V+1:IF V=4 THEN V=1:PRINT
CD 30 W=INT(RND(1)*4):DI=INT(RND(1)*3):FRQ=K(T(ASC
      (A$)))
RB 40 MI=INT(FRQ/(8*(V*W+1))):S=INT((FRQ-MI)/((INT
      (RND(1)*10)+1)*(MI/100)))
HM 50 SOUND V,0,0:SOUND V,FRQ,100,DI,MI,S,W
SB 60 PRINT"SOUND"V"{LEFT},"FRQ"{LEFT},"100"{LEFT}
      ,"DI"{LEFT},"MI"{LEFT},"S"{LEFT},"W"{LEFT},"
      :GOTO 20
AK 70 FOR J=1 TO 3:SOUND J,0,0:NEXT:VOL 15:DIM K(2
      56),T(256):FOR J=1 TO 255:T(J)=J
```

```
HA 80 K(J)=J*15Ø:NEXT:POKE 2594,128:PRINT CHR$(147
      )SPC(1Ø)"{DOWN}{RVS}128 SOUNDMAKER{OFF}"
DD 9Ø PRINT SPC(1Ø)"PRESS ANY KEY":PRINT:RETURN
```

Program 2-6. Song Player

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
GK 1Ø J= 1Ø17
FD 2Ø CH$=" Ø123456789#$.XVOTUWHQISRMABCDEFG":PLAY
      "XØ U9 O4T7 I"
DS 3Ø PRINT"{CLR}{2 RIGHT}{RVS}128 SONG MAKER":PRI
      NT"{2 DOWN}{2 RIGHT}{RVS}E{OFF}NTER
      {2 SPACES}{RVS}P{OFF}LAY{2 SPACES}{RVS}Q
      {OFF}UIT"
GD 4Ø GETKEY A$:IF A$<>"E"ANDA$<>"P"ANDA$<>"Q" THE
      N 4Ø
AG 5Ø IF A$="E" THEN 12Ø
HD 6Ø IFA$="Q" THEN PRINT"{CLR}1Ø J="J:POKE 2Ø8,2:
      POKE 842,19:POKE 843,13:END
KE 7Ø REM---SONG PLAYING ROUTINE
AX 8Ø RESTORE:PRINT "{DOWN}PLAYING SONG. PRESS ANY
       KEY TO QUIT."
XX 9Ø GET A$:READ P$:IF A$=""ANDP$<>"FINI" THEN PR
      INT P$:PLAY P$:GOTO9Ø
GA 1ØØ PRINT "{DOWN}END OF SONG.
XF 11Ø REM---MUSIC ENTRY ROUTINE
SA 12Ø POKE 2Ø8,Ø:PRINT "{CLR}ENTER MUSIC DATA (29
       -CHARACTER MAXIMUM)"
GE 13Ø PRINT "TYPE 'MENU' TO EXIT"
MG 14Ø P$="":INPUT "{8 SPACES}";P$:IF P$="MENU" OR
        P$="" THEN 2Ø
DF 15Ø X=Ø:FOR M=1 TO LEN(P$):FOR K=1 TO LEN(CH$)
KE 16Ø IFMID$(P$,M,1)=MID$(CH$,K,1)THENX=X+1
KR 17Ø NEXT K,M:IF X<LEN(P$) THEN PRINT"ILLEGAL MU
       SIC DATA":PRINT P$:GOTO 13Ø
CP 18Ø PRINT "{CLR}";J;"DATA ";P$:PRINT "J="J+1":G
       OTO 12Ø"
SG 19Ø POKE 2Ø8,4:POKE 842,19:POKE 843,13:POKE 844
       ,13:END
ER 999 REM---MUSIC DATA STARTS HERE
XE 1ØØØ DATA A
JF 1ØØ1 DATA C
MG 1ØØ2 DATA D
AE 1ØØ3 DATA A
CF 1ØØ4 DATA B
EG 1ØØ5 DATA C
GH 1ØØ6 DATA D
JJ 1ØØ7 DATA E
```

```
MK 1ØØ8 DATA F
QM 1ØØ9 DATA G
DJ 1Ø1Ø DATA H
KH 1Ø11 DATA E
JF 1Ø12 DATA A
MG 1Ø13 DATA B
MJ 1Ø14 DATA E
SJ 1Ø15 DATA D
JR 1Ø16 DATA M
KR 63999 DATA FINI
```

Sound and Music

Philip I. Nelson

The Commodore 128's advanced BASIC makes it easy and fun to create music or sound effects. This chapter shows how to use the VOL, TEMPO, and ENVELOPE statements, explores the FILTER, SOUND, and PLAY commands, and includes three short tutorial programs.

If you've heard much about the new Commodore 128, you probably know that it contains a very powerful music maker: the SID (Sound Interface Device) chip, exactly as found in the Commodore 64, and still the best sound chip in any personal computer. The SID chip provides three independent voices (tone generators) for playing up to three notes at once, and four different waveforms to simulate virtually any sound.

Although both computers use the SID chip, the comparison ends there. Since Commodore 64 BASIC has no sound commands, even simple 64 sound effects require several POKE statements. The 128's BASIC eliminates the POKEs by adding six new music and sound commands: PLAY, SOUND, VOL, TEMPO, ENVELOPE, and FILTER.

Simplicity and Power

The PLAY command is both powerful and easy to use. If you have access to a 128, type in and run the following one-line program. (The spaces make the statement more readable, but are not necessary.)

```
100 PLAY "C D E F G F E D C"
```

The 128 plays nine notes, going up the scale and down again. It would take a lot more work to play the same nine notes on the 64—you'd need at least three preliminary POKEs (to set the volume and sound envelope), plus four POKEs *for each note* (one to turn on the voice, two to set the pitch, and one to turn off the voice).

Interestingly, you can control the SID chip in 128 mode with the same POKEs as on the 64. That's usually a waste of time, since the 128's BASIC commands are more convenient than POKEs. However, 128 BASIC has certain limitations

(SOUND statements can't use ring modulation or synchronization, for example). If you already know sound programming on the 64, you may still find uses for old-fashioned 64 programming techniques.

The PLAY command is so versatile that it's almost a minilanguage in itself. In addition to playing notes, you can insert rests, change octaves, choose any of ten different instrument voices, use filtering, and even play multivoice music.

VOL Means Volume

The VOL command affects all three voices at once and accepts values from 0 (silence) to 15 (maximum). Add the following line to the example program and run it again:

10 VOL 15

Since the song plays at the same volume, it seems VOL had no effect. In fact, VOL just duplicated the *default* volume setting that PLAY uses when no volume is specified. When you turn on the 128, it establishes several music and sound settings (parameters) in advance. For instance, the PLAY statement above plays the notes at maximum volume with a sound envelope and waveform that simulate a piano. Other default sound parameters, too, remain in effect until you change them.

In many cases you can set the volume at the beginning of a program and leave it alone. However, gradual changes in volume can add to the dynamics of a song. Since drastic volume changes make the SID chip "pop," don't use VOL to turn individual notes on and off. (To hear the pop, turn up the volume on your monitor or TV set, enter the following line without a line number, and press RETURN: **VOL 15:VOL 0:VOL 15.**)

Unlike PLAY statements, SOUND statements default to a volume of 0. Before using SOUND, you must always use VOL to set the volume to some nonzero value.

TEMPO

TEMPO is another command that affects all voices equally, setting the speed at which a song plays. TEMPO is followed by one number in the range 0–255. The default tempo setting is 15, a pedestrian speed. Add the following line to the example program and run it again:

20 TEMPO 50

At a tempo of 50, the song plays much faster. Try several different TEMPO values in line 20. As you'll find, the highest tempos are exceedingly fast—too speedy for playing whole songs, but handy for simulating trills and grace notes. Change the TEMPO value back to 15 when you're done experimenting with line 20.

Don't confuse tempo—the overall speed of the music—with the individual *duration* of each note (quarter note, sixteenth, etc.). In conventional music a quarter note lasts one "beat," an eighth note lasts one-half beat, and so on. Tempo defines how many beats are played in a minute. At faster tempos every note plays faster, but quarter notes still last twice as long as eighth notes. The default note duration for PLAY is a quarter note.

A Built-In Orchestra

The ENVELOPE command is more versatile than VOL or TEMPO. It is used to create customized instrument sounds for your songs. ENVELOPE takes the following general form:

ENVELOPE *i, a, d, s, r, w, p*

In the above example, *i* stands for the *instrument number, a* for *attack rate, d* for *decay rate, s* for *sustain rate, r* for *release rate, w* for *waveform,* and *p* for *pulsewidth.* Naturally, in a program these letters are replaced with appropriate numbers.

The first number in an ENVELOPE statement chooses one of the 128's *instrument* voices. There are ten predefined instruments, numbered 0–9 as shown here:

Instrument	**ENVELOPE**
Piano	0
Accordion	1
Calliope	2
Drum	3
Flute	4
Guitar	5
Harpsichord	6
Organ	7
Trumpet	8
Xylophone	9

Since PLAY commands use the same instrument numbers, you'll want to become familiar with this list. To pick an instrument within PLAY, add a T (for *tune*) followed by the desired

instrument number. For instance, **PLAY "T5 C D T3 E F"** selects instrument 5 (guitar) and plays notes C and D, then selects instrument 3 (drum) and plays notes E and F. The same numbering scheme identifies customized instruments, as you'll see in a moment. The default instrument for PLAY statements is instrument 0 (piano); if you don't specify an instrument, PLAY always produces a piano sound.

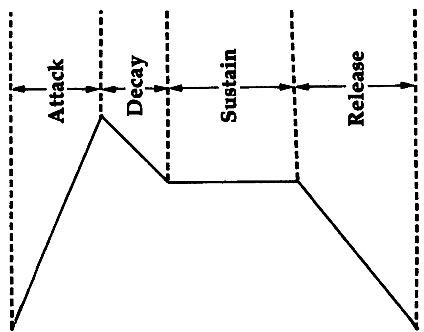
Sound Envelopes

To create new instrument sounds, you'll need to learn about sound envelopes and waveforms. Every natural sound has a distinctive *envelope,* or sound pattern. Consider the difference between a snare drum and a violin. Drum sounds begin and end very sharply. The drumhead starts vibrating the instant you strike it and fades quickly. Violin sounds start out more softly, as the string gradually picks up vibrations from the bow, and fade softly as the vibration dissipates.

The 128 defines different sound envelopes in terms of four values: attack, decay, sustain, and release (ADSR). The *attack* value defines how quickly the sound rises from silence to its peak volume. *Decay* defines how quickly the sound fades from peak volume to the volume at which it will be sustained (held). *Sustain* sets the volume level for the sound's main duration. *Release* defines how quickly the sound fades from its sustained volume back to silence again. Figure 2-1 illustrates a typical sound envelope.

In ENVELOPE statements, the four numbers after the instrument number define the ADSR envelope. ADSR numbers can range from 0 to 15.

Figure 2-1. Typical Sound Envelope



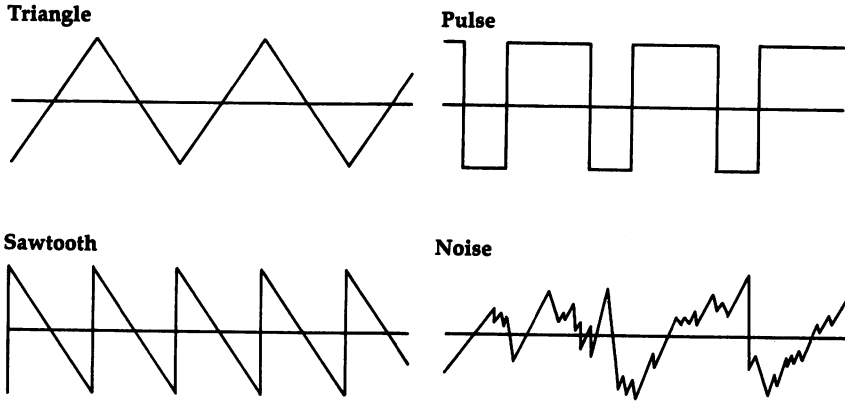
Waveforms

ENVELOPE also lets you pick different *waveforms*. Each of the SID chip's three voices can produce four different waveforms, diagrammed in Figure 2-2. The *triangle* waveform (used for the flute, instrument 4) is soft and rich. The *sawtooth* wave (used for the guitar, instrument 5) creates a louder, harsher sound.

The *pulse* waveform (used for the organ, instrument 7) is the most versatile of all. It's louder than the triangle wave and can be adjusted to make sounds that are rich and full or thin and faint. The *noise* waveform (used for the drum, instrument 3) is a random mishmash of frequencies that make a hissing or rushing sound. ENVELOPE uses the following waveform numbers:

Number	Waveform
0	Triangle
1	Sawtooth
2	Pulse
3	Noise
4	Ring modulation

Figure 2-2. Waveforms



Ring modulation is a special effect, different from the other waveforms. The SID chip creates ring modulation by combining the frequencies of two voices into one complex sound. Note that ENVELOPE cannot use *synchronization*, another SID effect familiar to 64 programmers.

Finally, ENVELOPE lets you choose different *pulsewidth* values for the pulse waveform (2). The pulsewidth number can range from 0 to 4095. Look again at the pulse wave diagram in Figure 2-2. The top portion of each wave is wider than the bottom portion. The pulsewidth value defines the ratio between these two parts of the wave. Medium pulsewidth values (roughly 1000–3000) produce fairly symmetrical waves and full, solid tones. Very small or very large pulsewidth values produce asymmetrical waves and thin, hollow tones.

ENVELOPE with PLAY

To see what ENVELOPE can do, add line 30 to the example program, and change line 100 to add T1 and a few more notes:

```
30 ENVELOPE 1, 7, 0, 0, 0, 2, 2000
100 PLAY "T1 C D E F G A G F E D C"
```

Run the program again and notice how different the new instrument sounds. Line 30 selects instrument 1; sets attack at 7; decay, sustain, and release at 0; waveform at 2 (pulse); and pulsewidth at 2000.

The T1 in line 100 might seem redundant at first: If ENVELOPE selects instrument 1, why specify instrument 1 again in the PLAY statement? This is necessary because of the default system. Until you specify otherwise *in a PLAY statement,* PLAY always uses instrument 0, the piano. Thus, whenever you define a new instrument with ENVELOPE, you must use the *same* instrument number after T in the appropriate PLAY statement. If you forget, PLAY ignores the ENVELOPE statement and uses instrument 0 or whatever instrument you last selected with T.

Redefining an instrument with ENVELOPE replaces the predefined instrument of that number. Thus, you can never have more than ten instruments at once. However, new instruments can be introduced at any time with new ENVELOPE statements.

ENVELOPE can be tricky to handle, since it gives you total control over the ADSR envelope and must be properly integrated with other sound commands. For instance, an envelope that sounds fine at slow tempos may be unsuitable at faster tempos. Don't be discouraged if your first experiments sometimes fail. Remember, ENVELOPE is necessary only for

customized instrument sounds. If you're happy with the predefined instruments, just use T in a PLAY statement to choose the one you want.

FILTER Needs PLAY

Like the ENVELOPE command, FILTER does nothing noticeable until you turn the filter on with a PLAY statement. Insert X1 inside the PLAY string wherever you want to turn the filter on, and X0 where you want to turn it off. If you leave out the X parameter, PLAY ignores preceding FILTER commands (the filter remains off). In the simplest case (a FILTER command followed by PLAY"X1"), the filter affects all three voices. However, you can also filter each voice individually:

```
FILTER 1000,1,0,0,15
PLAY "V1 X1 V2 X0 V3 X0
```

These statements turn the low-pass filter on for voice 1 and turn it off for voices 2 and 3. The 128 remembers which voice to filter when it executes subsequent PLAY statements (more about multivoice music is explained below). However, you can use only one filter *setting* at a time. For instance, you can't use a low-pass filter for voice 1 and a band-pass filter for voice 2. Whenever X1 appears in a PLAY string, the 128 uses the most recent FILTER setting. If no FILTER command has been executed, this may result in silence.

A FILTER Editor

As with other sound effects, the best way to learn is to listen and experiment; Program 2-7, "FILTER Editor," lets you do just that. It's self-prompting, so you need only type it in, save a copy, and run it. The menu screen displays all the current filter parameters and lets you change whatever you like. To select any option, press a number key from 0 to 9, and follow the prompts. The program begins with no filtering (all filters off) for comparison.

Option 9 switches you to the display screen, plays an ascending musical scale with whatever filtering you've selected, and displays the FILTER statement currently in effect. Once you find a filter setting you like, write down the FILTER statement displayed on the screen and use it in your own programs. From this screen, the number keys 1–6 select different

octaves for the scale. Press the space bar to return to the main screen.

Option 7 lets you select any of the 128's ten predefined instrument envelopes, and option 8 controls the tempo at which the scale is played. Note that some of the predefined envelopes don't work well at fast tempos: The note ends before the sound envelope can complete its natural cycle. Use a slower tempo to slow things down and study a particular effect.

The SID filter is a bit notorious. While it works fine on some machines (my old 64 has a great one), its performance may vary from one SID chip to the next. The manual for our preproduction 128 notes that filtering "cannot be counted on," suggesting that nothing was done to improve the 128's filter. With practice you should be able to achieve satisfactory effects on your own machine, though they might sound somewhat different on another computer.

The SOUND Command

SOUND is a very powerful command intended for sound effects rather than music. Unlike PLAY (which defaults to maximum volume), SOUND has a default volume setting of zero. Thus, you must turn the volume up with VOL before the first SOUND statement in a program. And whereas PLAY delays the rest of your program until it completes the current PLAY string, SOUND statements play "in the background" while the program continues. To demonstrate, enter NEW and press RUN/STOP–RESTORE (to clear the SID chip), then type in and run the following two-line program:

```
RG 10 VOL 15:SOUND 1,5000,200:SOUND 2,4000,200:SOU
      ND 3,3000,200
SX 20 FOR J=1 TO 10:PRINT"PROGRAM CONTINUING":NEXT
      :PRINT"DONE"
```

Notice how the three-voice sound continues even after this program ends and returns the computer to READY mode.

The first number in a SOUND statement (1, 2, or 3) picks one of the 128's three voices. By using different voice numbers, you can play up to three sounds at once. However, the 128 ordinarily waits until a voice has finished the current SOUND statement before starting a new SOUND statement *for that voice*. To illustrate, in line 10 of the above program,

change the 2 and 3 to 1; then run it again. Now voice 1 plays three notes in sequence.

In most cases SOUND's background-playing ability is desirable: Sound effects don't slow down the rest of your program. However, in other cases you might want to interrupt a sound immediately (if, for example, the user wants to exit the program). Fortunately, this is easy to do: SOUND statements with zero duration take effect immediately, whether or not preceding sounds have finished. Thus, SOUND 1,0,0 silences voice 1; use FOR J=1 TO 3: SOUND J,0,0: NEXT to silence all three voices.

Since variables can be used for any SOUND parameter, you can create more dynamic, integrated effects by incorporating other program variables in SOUND commands. For example, say that your game uses the variable X to represent a spaceship's screen position. To make a cruising sound, you might substitute something like X*1000 for the frequency number in a SOUND command.

A SOUND Editor

"SOUND Editor," listed below, lets you experiment with SOUND commands and design sound effects for your own programs using up to three voices at once. Type in and save Program 2-8, then run it. The first thing you'll hear are three complex, multivoice sound effects (don't worry if they're not exactly to your taste—you'll soon know enough about SOUND to replace them with your own). Next, the editing screen appears, displaying ten options and all the current SOUND parameters (your *User's Guide* explains the meaning of each parameter). To choose an option, press a number key from 0 to 9. The program instructs you how to proceed and does not let you enter inappropriate values.

Option 1 lets you switch from one voice to another. Option 9 switches you to the display screen, which plays the current sound and displays the SOUND statements that create it. It's fun to experiment with SOUND Editor, and it can save a lot of programming time. Use it to design exactly the sound you want, then copy the SOUND statements from the display screen and use them in your programs. (Though the program can play sounds with one, two, or three voices at once, it's not necessary to use multiple voices. Zero-duration SOUND statements produce no sound and may be ignored.)

The PLAY Command

Designed for real music making, PLAY is the most versatile of all the 128's sound commands. As outlined in the *User's Guide*, PLAY works much like the familiar PRINT command. Each PLAY command is followed by a string containing special control characters. The letters A–F are interpreted as notes; thus, the statement PLAY"C D E F" plays the four notes C-D-E-F. In the last example, PLAY was followed by a string of characters enclosed in quotation marks. However, PLAY can also handle string variables (A$="C D E F": PLAY A$).

To see this method at work, type in and save Program 2-9, "PLAY Demonstrator." It plays a short, Bach-like tune with several different instrument envelopes. Note that all of the music control characters are stored in DATA statements. Line 50 READs each line of data into a string named A$, and the subroutine at line 20 PRINTs each music string just before it is PLAYed.

Like other strings, PLAY strings can be concatenated (combined) with the + operator, and manipulated with any of the string-related functions: MID$, LEFT$, RIGHT$, LEN, VAL, CHR$, ASC, and STR$. Program 2-7 contains several different examples.

For complex music you might want to store PLAY strings in a string array. For instance, the following statement stores 100 elements of music data in a string array named M$( ):

FOR J=1 TO 100: READ M$(J): NEXT.

Once the music array is created, you can quickly access any string it contains: PLAY M$(3) plays the third music string held in M$( ), and so on. This is very helpful for repeating certain passages. You may also find it useful to create separate arrays for different purposes (one to store notes, another for duration characters, and so forth).

Multivoice Music

Since the SID chip has three voices, PLAY can play up to three notes simultaneously. The V control character (followed by 1, 2, or 3) determines which voice is affected. Thus, the statement

PLAY "V1 C V2 E V3 G"

plays a simple three-note chord. After processing V1 C, the

128 "looks ahead" to see whether it should play other notes at the same time; however, the computer looks ahead only *as far as the next note*. Thus, the statement

PLAY "V1 CDE V2 CDE"

does not play the notes C-D-E simultaneously with two voices. Instead, it plays two sequential notes (C-D) with voice 1, then two simultaneous notes (E and C) with voices 1 and 2, followed by two sequential notes (D-E) with voice 2.

When all voices play notes of the same duration, multivoice music is not particularly difficult to write: Insert V1 before each note for voice 1, V2 before each voice 2 note, and so forth (concatenations like A$="V1"+A$ can help condense the otherwise cumbersome code). However, when different voices play notes of different durations, you must make sure that all the durations add up.

For instance, you might want voice 1 to hold a long whole note while voice 2 plays a series of sixteenth notes. To keep the timing straight, you should not let voice 1 play another note until voice 2 has finished the equivalent of a whole note (16 sixteenths or whatever). Similarly, the timing may be thrown off if voice 2 plays *more* than 16 sixteenths before voice 1 gets back in the act. The M control character supposedly tells the 128 to wait until all voices finish the current measure before moving ahead. But M is just an adjuster. It can't magically repair music that doesn't add up in the first place.

Interactions

As noted throughout this chapter, certain 128 sound commands work with certain others. The VOL command, for instance, is needed only for SOUND statements (PLAY sets volume independently with the U control character). TEMPO, FILTER, and ENVELOPE, on the other hand, seem designed to work with PLAY. TEMPO is irrelevant to SOUND (which sets its own duration, and so on); ENVELOPE and FILTER have no effect until activated by PLAY.

However, other interactions are possible (at least on our 128, admittedly a preproduction model). For instance, though the SOUND statement provides no way to turn on the filter, SOUNDs can be affected by "leftover" filter settings. If the 128 executes a FILTER statement followed by PLAY"X1", the

filter remains on and affects subsequent SOUND statements. PLAY"X0" turns the filter off for SOUND as well as for PLAY.

This interaction can be viewed either as an advantage—filtering is otherwise unavailable with SOUND—or as a pitfall for unwary programmers. To prevent unwanted interactive effects, begin sound and music programs by setting all sound parameters at zero or default values. Commodore 64 programmers often clear the SID chip with

FOR J=54272 TO 54296: POKE J,0: NEXT

Though this statement does clear the 128's SID chip, it doesn't necessarily change the 128's sound settings, which are recorded elsewhere in memory.

Program 2-7. FILTER Editor

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
FF 100 GOSUB570:GOTO310
EG 110 FORJ=1TO3:SOUNDJ,0,0:NEXT:FILTER0,0,0,0,0:R
       ETURN
XB 120 PLAY A$:RETURN
RX 130 LP$=" OFF":IFLP=1THENLP$=" {RVS}ON {OFF}"
CS 140 RETURN
PA 150 BP$=" OFF":IFBP=1THENBP$=" {RVS}ON {OFF}"
KA 160 RETURN
GH 170 HP$=" OFF":IFHP=1THENHP$=" {RVS}ON {OFF}"
BB 180 RETURN
HM 190 PRINTD$"SET CUTOFF FREQUENCY (0-2047)"
MF 200 INPUTA:IFA<0ORA>2047THENGOSUB550:GOTO190
RF 210 FQ=A:RETURN
FM 220 LP=ABS(LP=0):RETURN
DB 230 BP=ABS(BP=0):RETURN
KX 240 HP=ABS(HP=0):RETURN
KE 250 PRINTD$"SET FILTER RESONANCE (0-15)":INPUTA
       :IFA<0ORA>15THENGOSUB550:GOTO250
RG 260 RE=A:RETURN
GF 270 PRINTD$"CHOOSE SOUND ENVELOPE (0-9)":INPUTA
       :IFA<0ORA>9THENGOSUB550:GOTO270
ER 280 WV$="T"+CHR$(A+48):RETURN
DR 290 PRINTD$"CHOOSE TEMPO (1-255)":INPUTA:IFA<1O
       RA>255THENGOSUB550:GOTO290
AM 300 TM=A:RETURN
SC 310 PRINT"{CLR}{RVS} 128 FILTER EDITOR ":PRINT
DD 320 PRINT"1 {RVS} FREQUENCY {OFF}"FQ"{LEFT}
       {4 SPACES}"
PP 330 PRINT"2 {RVS} LOW{2 SPACES}PASS {OFF}";:GOS
       UB130:PRINTLP$
```

```
ER 340 PRINT"3 {RVS} BAND PASS {OFF}";:GOSUB150:PR
       INTBP$
JR 350 PRINT"4 {RVS} HIGH PASS {OFF}";:GOSUB170:PR
       INTHP$
MR 360 PRINT"5 {RVS} RESONANCE {OFF}";RE"{LEFT} ":
       PRINT"{2 SPACES}{RVS}-----------{OFF}"
AM 370 PRINT"7 {RVS} ENVELOPE{2 SPACES}{OFF} "MID$
       (WV$,2)T$(VAL(MID$(WV$,2)))
CK 380 PRINT"8 {RVS} TEMPO{5 SPACES}{OFF}"TM"
       {LEFT}{2 SPACES}":PRINT"9 {RVS} PLAY
       {6 SPACES}{OFF}":PRINT"Ø {RVS} QUIT
       {6 SPACES}{OFF}{DOWN}"
PM 390 PRINT"{RVS}ENTER YOUR CHOICE (Ø-9)":PRINT"
       {3 SPACES}{UP}"
GX 400 GETKEYA$:IFA$<"Ø"ORA$>"9"ORA$="6"THENPRINT:
       GOSUB550:PRINT:GOTO390
MP 410 IFA$="9"THEN440
MF 420 IFA$="Ø"THENEND
EF 430 ONVAL(A$)GOSUB190,220,230,240,250,250,270,2
       90:PRINTE$:GOTO320
MQ 440 PRINTCHR$(147)"OCTAVE "MID$(OC$,2)CHR$(13)
RQ 450 PRINT"LOW{2 SPACES}PASS "LP$:PRINT"BAND PAS
       S "BP$:PRINT"HIGH PASS "HP$:PRINT
BR 460 PRINT"{RVS}CURRENT FILTER STATEMENT:":PRINT
       :PRINT"FILTER ";
CA 470 PRINTMID$(STR$(FQ),2)","MID$(STR$(LP),2)","
       MID$(STR$(BP),2)",";
EC 480 PRINTMID$(STR$(HP),2)","MID$(STR$(RE),2):PR
       INT:FILTER FQ,LP,BP,HP,RE
CX 490 PRINT"PRESS {RVS} 1 - 6 {OFF} FOR OCTAVE"CH
       R$(13)SPC(6)"{RVS} SPACE {OFF} TO EXIT"
QC 500 F$="XØ ":IFLP=1ORBP=1ORHP=1THENF$="X1 "
QD 510 A$=F$+WV$+"S":GOSUB120:TEMPO TM
MK 520 GET B$:IFB$=CHR$(32)THENGOSUB110:GOTO310
HD 530 IFB$=>"1"ANDB$<="6"THENOC$="O"+CHR$(VAL(B$)
       +48):PRINT"{HOME}"SPC(6)VAL(B$)
GG 540 A$=OC$+"CDEFGAB":GOSUB120:GOTO520
FB 550 GOSUB110:FORJ=1TO3:SOUNDJ,1ØØØ+J*5ØØ,15,Ø,Ø
       ,Ø,2,J*1ØØØ:NEXT
KQ 560 PRINT"{UP}{RVS}INAPPROPRIATE":SLEEP1:PRINT"
       {UP}{13 SPACES}{3 UP}":RETURN
HA 570 PRINTCHR$(14)CHR$(8):FORJ=54272TO54296:POKE
       J,Ø:NEXT:VOL15:D$=CHR$(19)
KS 580 FORJ=1TO15:D$=D$+CHR$(17):NEXT:FQ=1ØØØ:LP=Ø
       :BP=Ø:HP=Ø:RE=15:WV$="T7":TM=55
AD 590 FORJ=1TO35:X$=X$+CHR$(32):NEXT:E$=D$+X$+CHR
       $(13)+X$+CHR$(19)+CHR$(13)
BP 600 FORJ=ØTO9:READX$:T$(J)="{2 SPACES}"+X$:NEXT
       :OC$="O3":GOSUB110:RETURN
```

```
CH 610 DATA"PIANO{6 SPACES}","ACCORDION{2 SPACES}"
       ,"CALLIOPE{3 SPACES}","DRUM{7 SPACES}","FLU
       TE{6 SPACES}"
PS 620 DATA"GUITAR{5 SPACES}","HARPSICHORD","ORGAN
       {6 SPACES}","TRUMPET{4 SPACES}","XYLOPHONE
       {2 SPACES}"
```

Program 2-8. SOUND Editor

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
SX 10 GOSUB30:GOSUB570:GOTO320
GX 20 PRINT"{CLR}{RVS}128 SOUND EDITOR":PRINT:RETU
      RN
RD 30 FORJ=1TO3:SOUNDJ,Ø,Ø:NEXT:RETURN
RP 40 PRINTD$"CHOOSE VOICE (1-3)":INPUTA:IFA<1ORA>
      3THENGOSUB550:GOTO40
GQ 50 VC=A:RETURN
QS 60 PRINTD$"CHOOSE FREQUENCY (Ø-65535)"
MA 70 INPUTA:IFA<ØORA>65535THENGOSUB550:GOTO60
FG 80 FQ(VC)=A:RETURN
BG 90 PRINTD$"CHOOSE DURATION (600=10 SECONDS)"
SJ 100 INPUTA:IFA<ØTHENGOSUB550:GOTO90
QC 110 DU(VC)=A:RETURN
KB 120 PRINTD$"CHOOSE DIRECTION OF SOUND SWEEP"
PX 130 PRINT"Ø=UP{2 SPACES}1=DOWN{2 SPACES}2=OSCIL
       LATE":INPUTA:IFA<ØORA>2THENGOSUB550:GOTO120
ED 140 DI(VC)=A:RETURN
CF 150 PRINTD$"CHOOSE MINIMUM FREQUENCY FOR"
GF 160 PRINT"SOUND SWEEP (Ø-65535)":INPUTA:IFA<ØOR
       A>65535THENGOSUB550:GOTO150
JJ 170 IFA=>FQ(VC)THENGOSUB550:GOTO150
JG 180 MI(VC)=A:RETURN
KH 190 PRINTD$"CHOOSE STEP VALUE FOR SOUND SWEEP"
RP 200 PRINT"(LESSER OF 32767 OR"FQ(VC)-MI(VC)+1"
       {LEFT})"
SF 210 INPUTA:IFA<ØORA>32767THENGOSUB550:GOTO190
FJ 220 IFA>(FQ(VC)-MI(VC))THENGOSUB550:GOTO190
MS 230 SV(VC)=A:RETURN
HE 240 PRINTD$"CHOOSE WAVEFORM{SHIFT-SPACE}
       {5 SPACES}Ø=TRIANGLE"
JA 250 PRINT"1=SAWTOOTH{2 SPACES}2=PULSE{2 SPACES}
       3=WHITE NOISE"
FQ 260 INPUTA:IFA<ØORA>3THENGOSUB550:GOTO240
PC 270 WV(VC)=A:RETURN
CP 280 PRINTD$"CHOOSE PULSE WIDTH"
DQ 290 PRINT"(Ø-4095)":INPUTA:IFA<ØORA>4095THENGOS
       UB550:GOTO280
FC 300 PW(VC)=A:RETURN
HX 310 GOSUB20
```

```
SS 320 PRINT"1 {RVS} VOICE{6 SPACES}{OFF}"VC:PRINT
       "2 {RVS} FREQUENCY{2 SPACES}{OFF}"FQ(VC)"
       {LEFT}{4 SPACES}"
QK 330 PRINT"3 {RVS} DURATION{3 SPACES}{OFF}"DU(VC
       )"{LEFT}{4 SPACES}"
GG 340 PRINT"4 {RVS} DIRECTION{2 SPACES}{OFF}"DI(V
       C)DI$(DI(VC))
CH 350 PRINT"5 {RVS} MINIMUM{4 SPACES}{OFF}"MI(VC)
       "{LEFT}{4 SPACES}":PRINT"6 {RVS} STEP VALUE
        {OFF}"SV(VC)"{LEFT}{4 SPACES}"
JP 360 PRINT"7 {RVS} WAVEFORM{3 SPACES}{OFF}"WV(VC
       )WV$(WV(VC))
EJ 370 PRINT"8 {RVS} PULSEWIDTH {OFF}"PW(VC)"
       {LEFT}{4 SPACES}"
JH 380 PRINT"9 {RVS} HEAR SOUND {OFF}":PRINT"Ø
       {RVS} QUIT{7 SPACES}{OFF}":PRINT
PM 390 PRINT"{RVS}ENTER YOUR CHOICE (Ø-9)":PRINT"
       {3 SPACES}{UP}"
BS 400 GETKEYA$:IFA$<"Ø"ORA$>"9"THENPRINT:GOSUB55Ø
       :PRINT:GOTO39Ø
MP 410 IFA$="9"THEN44Ø
EX 420 IFA$="Ø"THENGOSUB3Ø:END
RC 430 ONVAL(A$)GOSUB4Ø,6Ø,9Ø,12Ø,15Ø,19Ø,24Ø,28Ø:
       PRINTE$:GOTO32Ø
HG 440 PRINT"{CLR}THE FOLLOWING SOUND STATEMENTS":
       PRINT"{2 SPACES}CREATE THE SOUNDS YOU HEAR.
       "
HG 450 PRINT"ZERO-DURATION SOUNDS ARE SILENT."
EM 460 FORJ=1TO3:SOUNDJ,FQ(J),DU(J),DI(J),MI(J),SV
       (J),WV(J),PW(J):NEXT
DP 470 FORJ=1TO3:PRINT:PRINT"SOUND ";
MH 480 PRINTMID$(STR$(J),2)","MID$(STR$(FQ(J)),2)"
       ,"MID$(STR$(DU(J)),2)",";
PH 490 PRINTMID$(STR$(DI(J)),2)","MID$(STR$(MI(J))
       ,2)","MID$(STR$(SV(J)),2)",";
HR 500 PRINTMID$(STR$(WV(J)),2)","MID$(STR$(PW(J))
       ,2):NEXT
GR 510 PRINT:PRINT"PRESS {RVS}RETURN{OFF} TO EXIT"
       :PRINTSPC(6)"{RVS}SPACE{OFF} TO REPEAT"
GQ 520 GETKEYA$:IFA$=CHR$(13)THENGOSUB3Ø:GOTO31Ø
RS 530 IFA$=CHR$(32)THENGOSUB3Ø:GOTO44Ø
EF 540 GOTO52Ø
GH 550 GOSUB3Ø:FORJ=1TO3:SOUNDJ,1ØØØ+J*5ØØ,15,Ø,Ø,
       Ø,2,J*1ØØØ:NEXT
KQ 560 PRINT"{UP}{RVS}INAPPROPRIATE":SLEEP1:PRINT"
       {UP}{13 SPACES}{3 UP}":RETURN
CF 570 PRINTCHR$(14):D$=CHR$(19):FORJ=54272TO54296
       :POKEJ,Ø:NEXT:FORJ=1TO15
BE 580 D$=D$+CHR$(17):NEXT:GOSUB2Ø:VOL15:FORJ=1TO3
       8:X$=X$+CHR$(32):NEXT
```

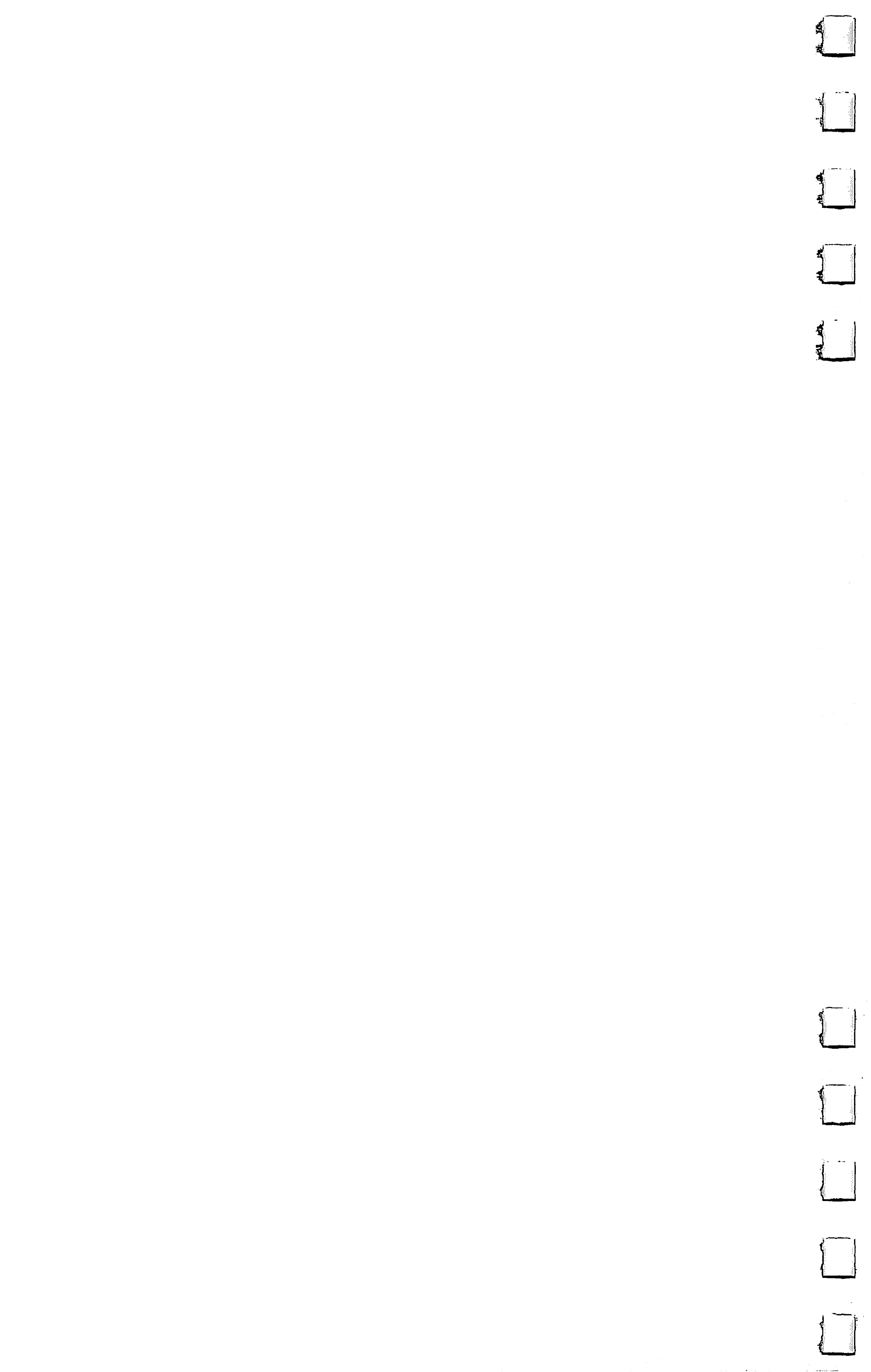
```
RD 590 VC=1:E$=D$+X$+CHR$(13)+X$+CHR$(13)+X$+CHR$(
       19)+CHR$(13)
MF 600 FORK=2000TO4000STEP220:FORJ=1TO3:SOUNDJ,K*2
       +J*20,45,2,K,K/3,2,4095-K
GG 610 NEXTJ,K:FORJ=45TO1STEP-5:SOUND1,J*1000,5,1,
       J*100,J*280,2,2300
JK 620 SOUND2,3200-J*20,5,0,0,0,2,1500:SOUND3,J*12
       00,5,1,J*120,J*300,2,3000
JX 630 NEXT:FORJ=1TO3:SOUNDJ,10000,200,1,J*2000,J*
       400,2,2300:NEXT:FORJ=1TO3
RH 640 READFQ(J),DU(J),DI(J),MI(J),SV(J),WV(J),PW(
       J):NEXT:FORJ=0TO3:READA$
MB 650 WV$(J)="---  "+A$:NEXT:FORJ=0TO2:READA$:DI$(
       J)="---  "+A$:NEXT:RETURN
KG 660 DATA10000,260,2,2000,60,2,2000,0,0,0,0,0,0,
       2000,0,0,0,0,0,0,2000
PS 670 DATA"TRIANGLE","SAWTOOTH","PULSE{3 SPACES}"
       ,"NOISE{3 SPACES}"
GA 680 DATA"UPWARD{3 SPACES}","DOWNWARD  ","OSCILLA
       TE"
```

Program 2-9. PLAY Demonstrator

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

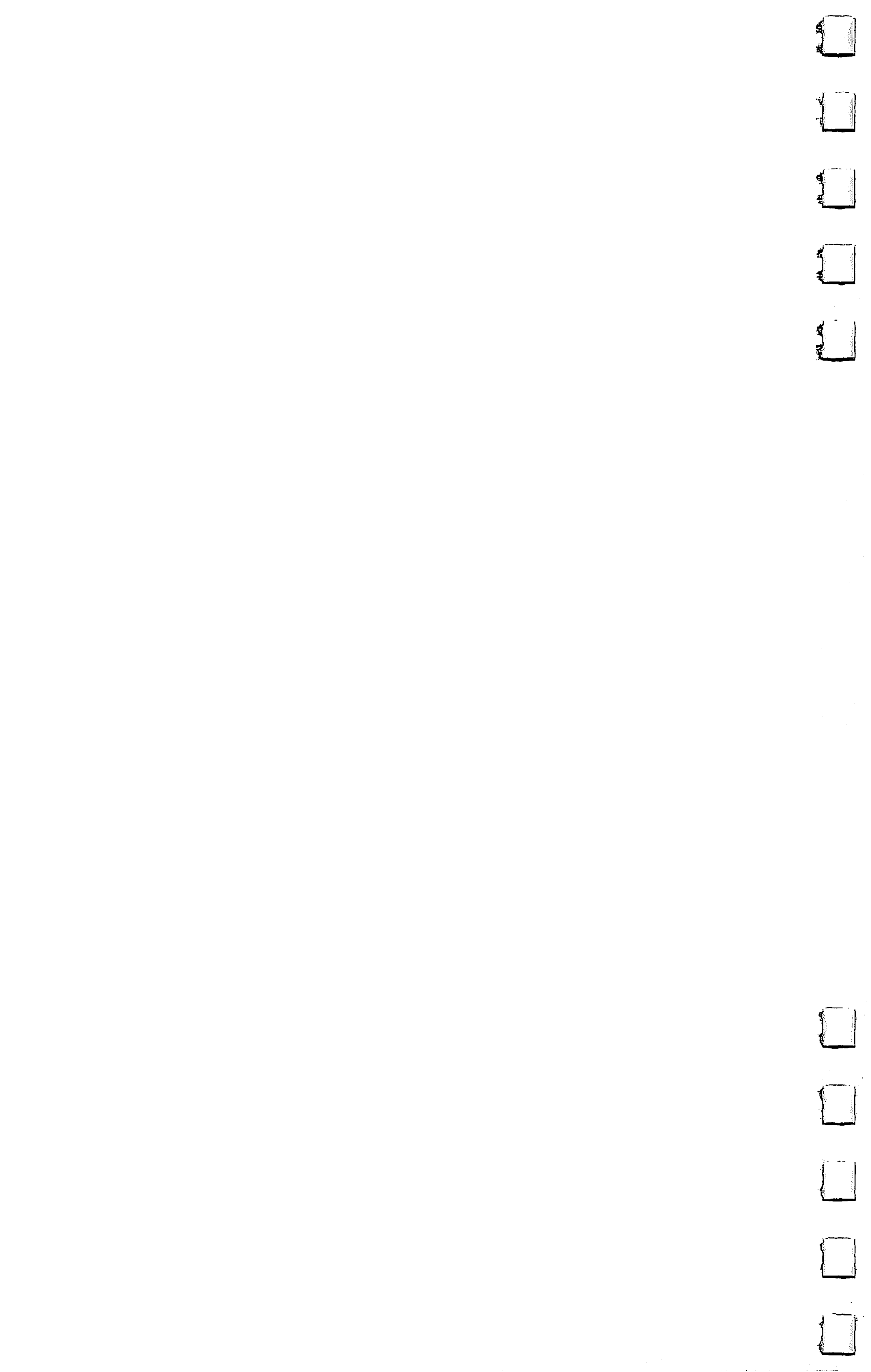
```
GA 10 GOTO30
EQ 20 PRINTA$:PLAYA$:RETURN
RR 30 PRINTCHR$(147)CHR$(14)SPC(3)CHR$(18)"128 PLA
      Y DEMONSTRATOR"CHR$(13)
KG 40 FORJ=54272TO54296:POKEJ,0:NEXT:FILTER0,0,0,0
      :FORJ=1TO3:SOUNDJ,0,0:NEXT
RF 50 READA$:IFA$<>"Z"THENGOSUB20:GOTO50
XG 60 PRINT:PRINTSPC(2)CHR$(18)"PRESS P TO PLAY AG
      AIN, Q TO QUIT"
AS 70 GETKEYG$:IFG$="P"THENRUN
KC 80 IFG$<>"Q"THEN70
ME 90 END
RQ 100 DATA U9 X0 V1 S
FH 110 DATA T7 O5 C O4 B O5 IC SO4 GRERGR
XJ 120 DATA T6 CDC O3 B O4 IC SO3 GRERGR
AH 130 DATA T7 CGDGEGDGC
BH 140 DATA O4 C O3 BAGFEDC
GE 150 DATA O5 C O4 BAGFED
MB 160 DATA T6 CGDGEGFGEGDG
MP 170 DATA CG O3 #A O4 G O3 A O4 G O3 G O4 G
BG 180 DATA O3 F R O5 FE I F S DR O4 BR O5 DR
QQ 190 DATA T2 G O6 G O5 A O6 G O5 B O6 G C O6 GDG
       FG
AH 200 DATA ERDCDGC O5 B
JB 210 DATA T4 ERDCDGC O4 B
```

```
EA 220 DATA T6 ERDCDGC O3 B
XA 230 DATA T0 ERDCDGC O2 BC
JS 240 DATA T7 O3 CDEFGABC
HD 250 DATA O4 CDEFGABC
ES 260 DATA O5 CDEFGAB
RE 270 DATA O6 CR O5 CR I O3 CR
AC 50000 DATA Z
```



Chapter 3

Games



Orbitron

Mark Tuttle and Kevin Mykytyn

You'll need to plan ahead to outmaneuver your opponent and score in "Orbitron," an interplanetary game for two players. Requires two joysticks.

Pass the ball between your three orbiting planets before your opponent grabs it. Move the ball and control to the outer orbit with perfect timing so your planet moves into position for a shot on goal. But be sure you pass in front of your goal before the other player has a chance to grab the ball. Or pass the ball between orbits just as your opponent picks up the ball, being sure that your player will be able to recapture it. Perhaps you want the other player to temporarily move the ball into position for you.

First Things First

Yes, "Orbitron" is a challenging game where two players try to outwit each other in order to pass the ball into position for a score. Playing Orbitron takes some practice. Each player must learn to control his or her three orbiting planets and pass the ball.

Player 1 is red and uses a joystick attached to port 1; player 2 is white and uses port 2. Each player has three orbiting planets: Player 1's planets orbit in a clockwise direction, while player 2's move counterclockwise.

Each player can control only one planet at any one time. The planet currently being controlled is a different color from the other two moving in the same direction. Player 1 has control over the purple planet; player 2 has control over the gray planet. To control the outer orbiting planet, press the fire button and push up—instantly the planet changes color to indicate control. Likewise, press the fire button without moving the stick to control the center planet, and pull back while pressing the button to control the inner planet.

Once You've Got Control

Once you have control of a planet, there are two things you can do: Change its speed, and pass or shoot. Move your joystick to the right to increase the speed of a planet and to the left to decrease the speed. Once the speed of a planet is set, it remains at that speed until it's changed again. The speed of each planet is shown on the bottom of the screen.

To pass the ball to another orbit, simply pull the stick down to throw the ball to an inner orbit, and push up to shoot at the goal or pass the ball to an outer orbit.

Playing the Game

The first time the game is run, there will be a minute and a half delay while the game is set up. Once the title screen appears, press either fire button to start the game.

The best way to learn to control the ball and your planets is to practice playing the game. The object of the game is to score five goals first. At the start of the game, a ball appears somewhere in the inner orbit. To score a goal, a player must move the ball from the inner orbit to the outer orbit, then shoot it into one of the two goals (on the left and right edges of the outer ring). Whenever a planet passes over the ball, it picks up the ball and carries it until it's either passed to another orbit or the other planet in the same orbit crosses over the planet carrying the ball, stealing it. You don't have to have control over the planet to steal the ball.

Remember, though, to pass the ball between orbits and to shoot, you must have control of the planet that has the ball.

To score, you must have control of your outer planet, that planet must be carrying the ball as it passes a goal, and you must shoot (push the stick up) as the planet passes the goal.

If you miss while trying to score a goal (or pass outward while on the outer ring), then the ball is transferred to the inner ring.

Typing It In

Orbitron is written in two parts. Program 3-1 is written in BASIC and should be typed in using "The Automatic Proofreader" (Appendix B). If you are using tape, you'll need to change the 8 in line 10 to a 1. Program 3-2 is the machine language section of the program and must be entered using

"MLX," the machine language editor program. MLX, and complete instructions for its use, can be found in Appendix C. Once you've typed in MLX, save a copy for use with "Meta-BASIC," found in Chapter 4, and with programs from other COMPUTE! publications.

When you're ready to enter the data from Program 3-2, load and run MLX. You'll be asked for the starting and ending address of the program. Enter:

Starting address: 7530
Ending address: 7997

Also, since Program 3-1 will automatically load the file created by MLX, you should answer the MLX prompt for a filename with ORB.OBJ when saving the data.

Tape users should save Program 3-1 at the beginning of a new tape, then remove the tape without rewinding it. Next, insert a tape with MLX on it, and load and run MLX, answering the prompts as above. When you're ready to save a copy of the data from Program 3-2, remove the tape containing MLX and insert the tape with Program 3-1 on it. Using the Save option of MLX, save the data from Program 3-2 immediately following Program 3-1.

Once you've finished entering and saving Program 3-1, and the data from Program 3-2 (using MLX), you're ready to play the game. First, load Program 3-1. Next, disk users should insert the disk with ORB.OBJ on it; if you're using tape, leave the play button down after loading Program 3-1. Run Program 3-1—it will automatically load ORB.OBJ from tape or disk.

Program 3-1. Orbitron

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
DR 10 GRAPHIC 1:GRAPHIC 0:BANK0:IFQ=0THENQ=1:LOAD"
      ORB.OBJ",8,1
XQ 20 COLOR 4,7:COLOR 0,7:PRINT"{CLR}{9 DOWN}E7}
      {2 SPACES}WHILE INITIALIZING THE SCREEN WILL
      {SHIFT-SPACE}BE"
XD 30 PRINT"{DOWN}{2 SPACES}BLANK FOR APPROXIMATEL
      Y {WHT}1.5E7} MINUTES.":SLEEP4:FAST:BANK0:GO
      SUB510
MS 40 PRINTCHR$(11)CHR$(27)"M":GRAPHIC 1,1
```

```
BG 50 CIRCLE 1,20,20,5,4:PAINT 1,20,20:SSHAPE SP$,
      10,13,33,33:FORA=2TO7:SPRSAV SP$,A:NEXT:CIRC
      LE 1,50,50,2:PAINT 1,50,50:SSHAPE SP$,40,43,
      63,63:SPRSAV SP$,1
DM 60 GRAPHIC 4,1,0:WIDTH 1:MT(1)=-4:MT(2)=127:CO=
      1:CR=35
HM 70 COLOR 0,1:COLOR 1,6:COLOR 2,8:COLOR 3,15:COL
      OR 4,1
MS 80 CIRCLE 3,3,14,2,3:SSHAPE A$(1),0,11,17,17
AD 90 CIRCLE 1,80,83,15,15:FORI=1TO3:CIRCLE CO,80,
      83,CR,CR:CO=CO+1:CR=CR+20:NEXT
AB 100 CIRCLE 2,154,83,2,6,0,180,1:CIRCLE 2,6,83,2
       ,6,180,2,1:GOSUB440
JP 110 PAINT 3,150,100,1:PAINT 2,130,100,1:PAINT 1
       ,50,100,1
PE 120 SPRITE 1,1,1,0:SPRITE 2,1,12,0:SPRITE 3,1,5
       ,0:SPRITE 4,1,2,0
HD 130 SPRITE 5,1,3,0:SPRITE 6,1,2,0:SPRITE 7,1,3,
       0:GOSUB140:GOTO160
SQ 140 MOVSPR 1,173,122:MOVSPR 2,225,124:MOVSPR 3,
       225,128:MOVSPR 4,265,124
DM 150 MOVSPR 5,265,128:MOVSPR 6,305,124:MOVSPR 7,
       305,128:RETURN
HX 160 CHAR 3,0,0,"PLAYER 1",0:CHAR 3,32,0,"PLAYER
        2",0:PRINT"{HOME}{9 DOWN}";
XD 170 GOSUB180:GOTO270
ED 180 PRINT"‹2›{4 SPACES}‹D›{RVS}‹D›‹F›{OFF}‹F›
       {RVS}‹K›‹D›‹F›{OFF}‹F›{RVS}‹K›‹D›‹F›{OFF}
       ‹F› {RVS}‹F›‹D›{OFF} ‹C›{RVS}‹F›‹D›{OFF}‹V›
       {RVS}‹K›‹D›‹F›{OFF}‹F›‹D›{RVS}‹D›‹F›{OFF}
       ‹F›{RVS}‹K›‹C›‹K›{OFF}‹K›{8 SPACES}{RVS}‹K›
       {OFF}‹K›{RVS}‹K›{OFF}";
JB 190 PRINT"‹K›{RVS}‹K›‹C›‹V›{OFF}‹V›{RVS}‹K›‹C›
       ‹V›{OFF}‹V› {RVS}‹K›{OFF}‹K›{2 SPACES}{RVS}
       ‹K›{OFF}‹K› {RVS}‹K›‹C›‹V›{OFF}‹V›{RVS}‹K›
       {OFF}‹K›{RVS}‹K›{OFF}‹K›{RVS}‹K›{2 SPACES}
       {OFF}‹K›{WHT}{8 SPACES}{RVS}‹K›{OFF}‹K›
       {RVS}‹K›{OFF}‹K›{RVS}‹K›‹D›‹C›{OFF}";
CF 200 PRINT" {RVS}‹K›{OFF}‹K›{RVS}‹K›{OFF}‹K›
       {RVS}‹K›{OFF}‹K›{2 SPACES}{RVS}‹K›{OFF}‹K›
       {SPACE}{RVS}‹K›‹D›‹C›{OFF} {RVS}‹K›{OFF}‹K›
       {RVS}‹K›{OFF}‹K›{RVS}‹K›{OFF}‹K›{RVS}‹F›
       {OFF}‹K›{9 SPACES}{RVS}‹2 I›{OFF} ‹C›‹V›‹C›
       ‹V›‹C›{RVS}‹2 I›{OFF} ";
HE 210 PRINT" {RVS}‹2 I›{OFF}{2 SPACES}‹C›‹V› ‹C›
       ‹V›‹C›‹V› {RVS}‹2 I›{OFF} ‹C›‹V›‹C›‹V›":BAN
       K15:POKE53269,0:BANK0:SLOW
HD 220 PRINTSPC(8)"{DOWN}‹7›PRESS FIREBUTTON TO PL
       AY":RETURN
```

```
MA 230 PRINT"E7}{2 SPACES}ED}{RVS}ED}EF}{OFF}EF}
       {RVS}EV}EC}{OFF} {RVS}EK}EC}{OFF}ED}{RVS}
       EK}ED}EI}{OFF}EV}{4 SPACES}ED}{RVS}ED}EF}
       {OFF}EF}{RVS}EK}{OFF}EK}{RVS}EK}{OFF}EK}
       {RVS}EK}ED}EI}{OFF}EV}{RVS}EK}ED}EF}{OFF}
       EF}{4 SPACES}{YEL}{RVS}EK}{OFF}EK}EI}EF}
       {RVS}EK}";
BK 240 PRINT"EC}EV}{OFF}EK}{RVS}EK}E2 D} EK}EC}
       {OFF}EF}{5 SPACES}{RVS}EK}{OFF}EK}{RVS}EK}
       {OFF}EK}{RVS}EK}{OFF}EK}{RVS}EK}{OFF}EK}
       {RVS}EK}EC}{OFF}EF} {RVS}EK}EC}EV}{OFF}EV}
       {4 SPACES}{GRN}{RVS}EK}{OFF}EK}{RVS}EK}
       {OFF}EK}{RVS}EK}{OFF}EK}{RVS}EK}{OFF}EK}
       {RVS}EK}{OFF}";
QE 250 PRINT"EK} {RVS} EK}{OFF}EK}{6 SPACES}{RVS}
       EK}{OFF}EK}{RVS}EK}{OFF}EK}EC}{RVS}EC}EV}
       {OFF}EV}{RVS}EK}{OFF}EK}{2 SPACES}{RVS}EK}
       ED}EC}{OFF}{6 SPACES}{RVS}E2 I}{OFF} EC}EV}
       EC}EV}EC}EV} {RVS}EI}{OFF}EC}{RVS}E2 I}";
CC 260 PRINT"{OFF}EV}{5 SPACES}{RVS}E2 I}{OFF}
       {2 SPACES}EC}EV} EC}{RVS}E2 I}{OFF}EV}EC}
       EV}EC}EV}{HOME}":RETURN
QF 270 IFJOY(1)=128ORJOY(2)=128THENGOTO320:ELSE 27
       Ø
GC 280 GOSUB430:PRINT"{DOWN}INNER{2 SPACES}E4}B
       {GRN}{RVS} {OFF}{9 SPACES}E4}B"SPC(2)"{WHT}
       INNER";
HS 290 PRINT"{2 SPACES}E4}B{RVS}{GRN} {OFF}
       {9 SPACES}E4}B"
AE 300 PRINT"E2}MIDDLE E4}B{YEL}{RVS} {OFF}
       {9 SPACES}E4}B"SPC(2)"{WHT}MIDDLE E4}B{YEL}
       {RVS} {OFF}{9 SPACES}E4}B"
RD 310 PRINT"E2}OUTER{2 SPACES}E4}BE7}{RVS} {OFF}
       {9 SPACES}E4}B"SPC(2)"{WHT}OUTER{2 SPACES}
       E4}BE7}{RVS} {OFF}{9 SPACES}E4}B{HOME}":RET
       URN
XH 320 FORA=ØTO21:GRAPHIC 4,Ø,A:FORX=1TO4Ø:NEXT:NE
       XT:GOSUB28Ø
KQ 330 GOSUB14Ø:BANK15:POKE53269,127:BANK Ø
AS 340 GRAPHIC 4,Ø,21:POKE29212,RND(1)*256:SYS3ØØØ
       Ø:GOTO36Ø
QD 350 SYS3ØØØ3
FS 360 IFPEEK(2921Ø)=ØTHENPL=2:GOTO37Ø:ELSEPL=1
XD 370 MOVSPR 1,PEEK(29213)*297+27,126:GOSUB45Ø:SC
       (PL)=SC(PL)+1:MT(PL)=MT(PL)+6
AE 380 PAINT 3,MT(PL),14,Ø:IFSC(1)=5ORSC$(2)=5THENP
       RINT"{CLR}":GOSUB43Ø:GOTO39Ø:ELSE35Ø
PE 390 SPRITE 1,Ø,1,Ø:GOSUB43Ø:GOSUB23Ø:GOSUB48Ø:S
       LEEP2
```

```
SH 400 MT(1)=-4:MT(2)=127:SC(1)=0:SC(2)=0:PRINT"
       {CLR}{7 DOWN}";
KH 410 SPRITE 1,1,1,0:MOVSPR 1,173,122:GOSUB180
JH 420 FORA=21TO0STEP-1:GRAPHIC 4,0,A:FORX=1TO40:N
       EXT:NEXT:GOSUB440:GOTO270
DA 430 PRINT"{CLR}{20 DOWN}{2}":RETURN
JB 440 N=0:FORZ=1TO2:FORI=1TO5:GSHAPE A$(1),N,11,0
       :N=N+6:NEXT:N=129:NEXT:RETURN
QS 450 FORJ=1TO15STEP5:SOUND1,J*1000,5,1,J*100,J*2
       80,2,2300
BX 460 SOUND2,3200-J*20,5,0,0,0,2,1500:SOUND3,J*12
       00,5,1,J*120,J*300,2,3000
SH 470 NEXT:RETURN
GQ 480 VOL 3:FORJ=45TO1STEP-2:SOUND1,J*1000,5,1,J*
       100,J*280,2,2300
CB 490 SOUND2,3200-J*20,5,0,0,0,2,1500:SOUND3,J*12
       00,5,1,J*120,J*300,2,3000
BF 500 NEXT:RETURN
ED 510 XC=175:YC=125:Q=3:FORR=25TO65STEP20:N=0
XF 520 FORA=0TO2*↑STEP.02:X=XC+2*R*COS(A):Y=YC+R*S
       IN(A):XH=INT(X/256)
KB 530 XL=X-XH*256:G=(Q-1)*1000+N:POKE20000+G,XL:P
       OKE23000+G,XH:POKE26000+G,Y
SS 540 N=N+1:NEXT:N=N-1:NH=INT(N/256):NL=N-NH*256:
       POKE29000+(Q-1)*2,NL
KD 550 POKE29006+(Q-1)*2,NL:POKE29001+(Q-1)*2,NH:P
       OKE29007+(Q-1)*2,NH:Q=Q-1:NEXT:RETURN
```

Program 3-2. ORB.OBJ

This data must be entered using MLX. See Appendix C.

Starting address: **7530**
Ending address: **7997**
Save using the filename: **ORB.OBJ**

```
7530:20 87 77 AD 00 FF 48 A9 11
7538:3E 8D 00 FF 20 CB 77 20 E4
7540:39 76 20 F6 77 20 26 79 DA
7548:20 39 76 20 EF 76 20 FB F7
7550:75 CE DF 79 D0 12 AD E0 48
7558:79 8D DF 79 20 17 78 20 65
7560:76 75 20 B4 75 20 6A 79 AD
7568:A5 91 C9 7F F0 FA A2 5A CE
7570:CA D0 FD 4C 48 75 A0 05 D7
7578:A2 0A A9 00 8D E4 79 AD 0D
7580:F7 79 8D D6 11 AD EF 79 7D
7588:8D D7 11 B9 F1 79 9D D8 77
7590:11 B9 F9 79 4A 2E E4 79 97
7598:B9 E9 79 9D D9 11 CA CA 57
75A0:88 10 E8 AD FF 79 8D 1D E9
```

```
75A8:72 4A 2E E4 79 AD E4 79 39
75BØ:8D E6 11 6Ø A2 ØA BD EF EC
75B8:75 85 FB BD FØ 75 85 FC 8Ø
75CØ:8A 4A A8 B9 9A 79 38 E9 49
75C8:ØF 4A 4A 4A 8D E3 79 A9 54
75DØ:ØA 38 ED E3 79 8D E3 79 ØE
75D8:AØ Ø9 A9 2Ø CC E3 79 FØ 67
75EØ:Ø2 BØ Ø2 A9 AØ 91 FB 88 9F
75E8:1Ø F2 CA CA 1Ø C8 6Ø DD EØ
75FØ:Ø7 B5 Ø7 8D Ø7 C8 Ø7 AØ 9Ø
75F8:Ø7 78 Ø7 8A 48 98 48 AD F1
76ØØ:E2 79 ØA A8 A2 Ø1 98 4A 1D
76Ø8:CD 1B 72 FØ 1F AD 1Ø 72 42
761Ø:D9 A2 79 DØ 17 AD 11 72 D2
7618:D9 A3 79 DØ ØF AD C4 79 49
762Ø:DØ ØA 98 4A 8D 1B 72 A9 17
7628:Ø1 8D C4 79 98 18 69 Ø6 27
763Ø:A8 CA 1Ø D2 68 A8 68 AA B4
7638:6Ø A2 Ø5 DE 92 79 FØ Ø3 EB
764Ø:4C A8 76 BD 9A 79 8A C9 C1
7648:Ø3 9Ø Ø3 38 E9 Ø3 A8 BD 29
765Ø:9A 79 ØA 4A 88 1Ø FC 9D EA
7658:92 79 8A ØA A8 B9 A2 79 C9
766Ø:18 79 B2 79 99 A2 79 B9 A9
7668:A3 79 79 B3 79 99 A3 79 E2
767Ø:1Ø ØC B9 48 71 99 A2 79 D4
7678:B9 49 71 99 A3 79 B9 48 1B
768Ø:71 38 F9 A2 79 8D E3 79 EØ
7688:B9 49 71 F9 A3 79 ØD E3 73
769Ø:79 BØ Ø8 A9 ØØ 99 A2 79 27
7698:99 A3 79 B9 A2 79 85 FD ØA
76AØ:B9 A3 79 85 FE 2Ø 24 77 13
76A8:CA 3Ø ØD EØ Ø2 DØ 8C 2Ø 43
76BØ:EF 76 2Ø FB 75 4C 3B 76 CØ
76B8:6Ø 2Ø 4E Ø8 52 FØ 55 2Ø 49
76CØ:4E Ø8 52 FØ 55 D8 59 CØ B1
76C8:5D A8 61 D8 59 CØ 5D A8 79
76DØ:61 9Ø 65 78 69 6Ø 6D 9Ø FE
76D8:65 78 69 6Ø 6D 18 65 FD 5E
76EØ:85 FB AD E3 79 65 FE 85 68
76E8:FC 8C E5 79 AØ ØØ 6Ø 8A 1C
76FØ:48 98 48 AD 1B 72 C9 Ø7 49
76F8:FØ ØE ØA A8 B9 A2 79 8D 86
77ØØ:1Ø 72 B9 A3 79 8D 11 72 9B
77Ø8:AD 1Ø 72 85 FD AD 11 72 B3
771Ø:85 FE AD E2 79 ØA A8 A2 4D
7718:Ø6 2Ø 24 77 2Ø 5E 77 68 DF
772Ø:A8 68 AA 6Ø B9 BA 76 8D ØC
7728:E3 79 B9 B9 76 2Ø DD 76 AØ
773Ø:B1 FB 9D F1 79 AC E5 79 8D
```

```
7738:B9 C6 76 8D E3 79 B9 C5 9B
774Ø:76 2Ø DD 76 B1 FB 9D F9 48
7748:79 AC E5 79 B9 D2 76 8D Ø7
775Ø:E3 79 B9 D1 76 2Ø DD 76 4A
7758:B1 FB 9D E9 79 6Ø AD 1B 35
776Ø:72 C9 Ø7 FØ 1Ø AD 1Ø 72 B4
7768:CD C2 79 DØ Ø8 AD 11 72 B6
777Ø:CD C3 79 FØ 11 A9 ØØ 8D 32
7778:C4 79 AD 1Ø 72 8D C2 79 A7
778Ø:AD 11 72 8D C3 79 6Ø A9 2Ø
7788:ØØ AØ ØB 8D DC 79 8D DD 9F
779Ø:79 99 A2 79 88 1Ø FA AØ A9
7798:ØB B9 76 79 99 B2 79 88 F4
77AØ:1Ø F7 AØ Ø5 B9 84 79 99 66
77A8:9A 79 A9 Ø1 99 92 79 88 1B
77BØ:1Ø F2 A9 Ø2 8D DA 79 8D 12
77B8:DB 79 8D E2 79 A9 ØA 8D E7
77CØ:DF 79 8D EØ 79 A9 Ø7 8D CB
77C8:1B 72 6Ø A2 19 A9 ØØ 9D 25
77DØ:ØØ D4 BD DC 77 9D ØØ D4 81
77D8:CA 1Ø F4 6Ø ØØ ØF ØØ ØC 1E
77EØ:81 ØF FØ ØØ ØØ ØØ ØØ ØØ 72
77E8:ØØ ØØ ØØ ØF ØØ ØC 15 ØC 2F
77FØ:ØØ ØØ ØA F3 1F 6Ø 2Ø ØA 25
77F8:78 C9 CF BØ F9 C9 6B 9Ø F9
78ØØ:F5 8D 1Ø 72 A9 ØØ 8D 11 F1
78Ø8:72 6Ø AD 1C 72 ØA ØA 38 C9
781Ø:6D 1C 72 8D 1C 72 6Ø A2 F3
7818:Ø1 BD DA 79 18 7D 46 79 A8
782Ø:8D DE 79 2Ø 3C 79 29 ØF E9
7828:C9 ØF DØ Ø5 A9 ØØ 9D DC 91
783Ø:79 AØ ØØ 2Ø 3C 79 48 29 89
7838:1Ø DØ Ø2 AØ Ø1 68 4A BØ 9E
784Ø:59 CØ Ø1 DØ Ø8 A9 ØØ 9D BF
7848:DA 79 4C Ø5 79 BD DC 79 D4
785Ø:DØ F8 AD DE 79 CD 1B 72 37
7858:DØ FØ AD 1B 72 8D 1A 72 C5
786Ø:A9 Ø7 8D 1B 72 AD E2 79 D4
7868:FØ Ø9 CE E2 79 2Ø 5A 79 96
787Ø:4C ØA 79 AD 11 72 FØ Ø7 4F
7878:AD 1Ø 72 C9 2F 9Ø 15 AD C2
788Ø:1Ø 72 C9 Ø4 9Ø Ø8 C9 99 61
7888:9Ø ØA C9 AØ BØ Ø6 68 68 5E
789Ø:68 8D ØØ FF A9 Ø2 8D E2 6C
7898:79 6Ø 4A BØ 33 CØ Ø1 DØ 22
78AØ:Ø7 A9 Ø2 9D DA 79 1Ø 5D D3
78A8:BD DC 79 DØ 58 AD DE 79 9C
78BØ:CD 1B 72 DØ 5Ø AD 1B 72 8C
78B8:8D 1A 72 A9 Ø7 8D 1B 72 F6
78CØ:AD E2 79 C9 Ø2 FØ 3E EE 4C
```

```
78C8:E2 79 2Ø 4A 79 4C ØA 79 BC
78DØ:4A BØ 13 AC DE 79 B9 9A 2B
78D8:79 C9 5F FØ 28 18 69 Ø1 69
78EØ:99 9A 79 4C Ø5 79 4A BØ 8C
78E8:13 AC DE 79 B9 9A 79 C9 F6
78FØ:ØF FØ 12 38 E9 Ø1 99 9A 8C
78F8:79 4C Ø5 79 CØ ØØ FØ ØA E3
79ØØ:A9 Ø1 9D DA 79 A9 Ø1 9D 7B
79Ø8:DC 79 CA 3Ø Ø3 4C 19 78 18
791Ø:A2 Ø1 BD 46 79 18 7D DA B2
7918:79 A8 9D Ø3 7A BD 48 79 AA
792Ø:99 28 DØ CA 1Ø EC AØ Ø6 2C
7928:CC Ø3 7A FØ ØB CC Ø4 7A AE
793Ø:FØ Ø6 B9 8A 79 99 28 DØ 5Ø
7938:88 1Ø ED 6Ø BD ØØ DC 8D 6C
794Ø:Ø2 7A AD Ø2 7A 6Ø ØØ Ø3 Ø1
7948:ØB Ø4 A9 1Ø 8D 12 D4 A9 FF
795Ø:ØA 8D ØF D4 A9 15 8D 12 A9
7958:D4 6Ø A9 1Ø 8D 12 D4 A9 ØC
796Ø:12 8D ØF D4 A9 15 8D 12 BD
7968:D4 6Ø AD 14 7A 69 19 8D ØE
797Ø:14 7A 8D Ø8 D4 6Ø FF FF 66
7978:FF FF FF FF Ø1 ØØ Ø1 ØØ 75
798Ø:Ø1 ØØ Ø1 ØØ 28 3C 5Ø 28 ØF
7988:3C 5Ø Ø1 Ø1 Ø1 Ø2 Ø2 Ø2 F3
799Ø:Ø2 Ø2 ØØ ØØ ØØ ØØ ØØ ØØ Ø5
```

Litter Patrol

Charles Brannon

BASIC 7.0 puts the power of the 128 within easy reach of the average programmer. "Litter Patrol" demonstrates how the 128's BASIC can be used to create an interesting and exciting game.

The Commodore 128 runs all 64 software, and can use virtually all 64 hardware and peripherals. This makes it easy to upgrade to the 128 and gives first-time Commodore owners instant access to the large 64 software library.

To use the expanded keyboard, full 128K memory, and RGB color 80 columns, you need to run in the true 128 mode. The 128 mode is a real upgrade of the 64, but has a familiar feel to it. The same VIC chip is used to display 40 columns, bitmap graphics, and sprites, so the screen even looks the same, except for Commodore's new power-on color choice—light green text on a dark gray screen with a light green border. You need an RGB monitor (or a monochrome monitor with an adapter cable) to use the full-color 80 column mode, which is entirely independent of the 40-column screen supported by the VIC chip.

The BASIC 7.0 is one of the most feature-packed BASICs I've seen. To learn about the BASIC, I wrote a simple *Frogger*-type game, taking advantage of the automatic sprite-movement feature supported by BASIC. It seemed that if the game were designed around the special BASIC features, I could get machine language animation and playability. I was half right. The game, "Litter Patrol," will run only in BASIC 7.0 in the 128 mode.

Playing Litter Patrol

Litter Patrol uses a joystick plugged into port 2. A joystick plugged into port 1 still interferes with the keyboard in 128 mode.

The goal of Litter Patrol is quite simple: Pick up all the bits of litter and fill all the trash cans. Your heavy-duty (but sluggish) truck can move in eight directions almost anywhere on the screen. The cars, zooming back and forth on the high-

way, are constantly throwing out bits of trash, which appear as bright dots (periods) on the road. Move the claw of your truck over the trash bit, and press the fire button. Your truck picks up the litter.

Now move the claw over any trash can (which looks like a hollow circle), and press the button. The trash drops in the can, and the lid closes. Each trash can can hold only one load of trash, so it turns solid to show you not to use it again. After you've filled all 12 trash cans, you proceed to the next level. The cars go faster, and you move more slowly—quite a handicap.

The game would be easy (and pointless) if not for the zooming cars. Dodging them provides the entire challenge for the game. If you get hit, you lose your trash bit—if you're carrying one—and one truck. The game ends when you lose all five trucks. Just to make things more interesting, you have a time limit, represented by a blue bar at the top of the screen. The bar drops by one segment every two seconds, so you have about 80 seconds to complete each level. The game ends instantly when you run out of time.

There are some safe zones for your truck where you can't be hit, medians between each roadway, and at the top and bottom of the screen. There's a secret safety zone, too, but I'll leave its discovery to you. You must move your truck halfway onto the roadway to fill a trash can, though. This makes a tough game even tougher. The hardest part of writing a game is in making it challenging but not too frustrating. Almost any game gets easier with practice, but an unfair game doesn't encourage you to try.

The Time Eaters

Litter Patrol is fun to play, but a caveat is in order. I didn't intend to program the game for its own sake, but for its educational value. Keeping in mind that the game is in BASIC, you may find it too slow. The main problem is the automatic sprite movement. The cars move by themselves once set up, but they are time eaters, stealing time during the interrupts from the mainline BASIC program. More about this below.

We'll take a walk through the program listing. Litter Patrol, Program 3-3, is too big for a line-by-line analysis, so we'll tackle it in chunks.

Lines 100–190. The GRAPHIC 0,1 command switches to the 40-column text screen and clears the screen. The COLOR

0,12 statement sets the background color to dark gray (even though this is the default color), and COLOR 4,6 sets the border color to green. Note that the colors are numbered 1–16, not 0–15 as in POKEs. We GOSUB 760 to fill sprite shape strings from the DATA statements.

The roadways will be the background color showing through other areas printed with reverse spaces. This lets us put yellow and white lines on the road. We'll print green reverse spaces to represent grass, delineating the roadways. To print the median lines and grass, we create 40 character strings within the FOR-NEXT loop. It may be easier for the programmer just to define the literal strings as 40 characters within quotation marks (like SP$="{40 SPACES}"), but it's easier to type in the program if we use a FOR-NEXT loop. Everything is done with CHR$ codes. Instead of printing color codes, we use the COLOR command to change the text color. However, you'll occasionally see a {SPACE}. Just type one space instead of the word in braces.

Line 140 turns off all sprites that may have been active from a previous run of the game. We then print the roadways with green bars above the road, white or yellow median bars for the middle of the road, and blank lines for the road itself. The time line is printed in blue at the top of the screen.

Lines 200–250. Line 200 is trying to print a 40-column reverse string at the bottom of the screen. You can't normally do this without scrolling, but it's possible if you print 39 characters, cursor left, use the INST/DEL key to insert the thirty-ninth character into the fortieth position, then print another character to fill the gap created by INST/DEL.

The title of the game is printed with the CHAR command. CHAR is a usable substitute for PRINT AT. It lets you print any string at any x,y position on the screen, and in normal or reverse field. Combine it with COLOR to change the text color. The subroutines at 720 and 730 are used to display the score and number of trucks ("lives") remaining. The FOR-NEXT loop in lines 220–230 draws all the trash cans, at rows 2, 9, 16, 23, and columns 8, 20, and 32.

We then build a music string. It's a cutesy, happy melody, but all that's important here is to notice the PLAY syntax. The letters C-D-E-F-G-A-B stand for notes. The V1 sets the voice to voice 1, O2 sets the octave to 2 (that's an *O*, not a zero), and T0 selects a piano-like instrument setting. The letter *I* sets

the note duration to eighth notes; *Q* is used for quarter notes, with a period for a dotted quarter. The sharp (#) precedes the note it modifies. And *R* is used as a rest. We'll play the string in line 360.

Lines 260–320. The automatic car sprites are set up. SPRCOLOR sets the sprite multicolor registers to white and black. All sprites share these colors. White is used for the windshield (or the claw on the truck), and black for the tires. I used the built-in sprite editor to design the sprites, then made DATA statements for them. The DATA statements are read into strings, then each string is assigned to a sprite with the SPRSAV command. Sprite strings are 67 characters long, not 64 as you might expect.

For the six sprites (FOR I=2 TO 7), we read (line 270) the sprite x and y positions from a DATA statement at line 1110. Notice that you can now RESTORE to any line number. The SPRITE command turns on the sprite, sets its color, and specifies multicolor mode. It can also be used to select sprite/foreground priority, and x,y expansion. Nonexpanded sprites offer the greatest detail.

The MOVSPR command can move a sprite to any position, up or down by any amount, or automatically at any angle and at 16 speeds. We use the automatic syntax (the two arguments are separated by a # sign instead of a comma). The angle is either 90 (right) or 270 (left). Angle 0 is pointing straight up in the sprite angular system. Whether a sprite goes left or right depends on its sprite number. If (SN=2) is true (−1), then 180 is added to 90, giving us 270. Otherwise, the angle is 90.

The speed, which can range from 0 to 15, varies from up to five speeds from the base speed, DF. This sets the difficulty level. A higher DF gives generally faster cars. We save the angles and speeds in arrays so that we can later pause the game (all speeds go to zero) and restart it from the arrays.

The automatic sprite movement is amazing. Even if you stop the program, the sprites continue. You can LIST your program, and the sprites still whiz by. However, you'd notice a suspicious slowness to the listing. When you use automatic sprites, everything else slows down drastically. The more sprites are moving, and the faster they go, the less time is available for the main program. This made the truck-moving

part of Litter Patrol quite sluggish and explains why the truck moves more slowly as the cars go faster.

While automatic sprites give you smooth, fast motion, this motion is not under your control. Speed is the reason you would use the automatic sprites in the first place, but the time saved by the automation is stolen from your main program. You can achieve a workable compromise if you plan your game around the limitations.

Lines 340–360. We synchronize the truck's position with the character screen so that the claw will cover the dots that represent trash bits. The truck always moves eight notches at a time, as if it were a character. Therefore, it's always synchronized with the character grid.

Line 350 turns on the collision interrupts. Any time a sprite hits a sprite, the program goes to line 580. Since all the sprites are in separate lanes, this can happen only when the truck is smashed. When we RETURN from the subroutine at line 580, the program picks up where it left off when the collision occurred.

We play the tune in line 360 only at the beginning of the first level (IF DF=1).

Lines 370–470. We enter the main loop here. While the car sprites move automatically, we must move the truck ourselves. First, if two seconds have passed (TI−T>120), we erase a character from the time line. If the time line hits zero, we go to the "game is over" routine at line 640. In 380, we check for a keystroke. If a key is pressed, we halt all sprites and wait for a new keystroke with GETKEY, then turn all the sprites back on.

In line 390, we check for the highly probable: Is the value of RND(1), which randomly varies between 0 and 1, less than 0.95? About 95 times out of 100, it will be, skipping lines 400 and 410. Five percent of the time, though, RND(1) will be greater than or equal to 0.95, so we pick a sprite number, read its x,y position, translate the sprite coordinates to character coordinates, and draw a white period to represent an empty cola can (or whatever litterbugs throw out car windows). They all look like little dots, though, from your aerial perspective. The random statement controls the timing of litter dropping. Without it, there would be a stream of trashy bits flowing from all cars.

Lines 420–460. These lines move the truck. The JOYstick command returns a number from 0 to 8, and is greater than 128 if the fire button is pressed. We use the JOY value as an index into the DX and DY arrays. These arrays contain the values −8, 0, or 8 for each position. For example, the southwest position of the joystick is down eight (+8) and left eight (−8). Remember that we're moving eight spaces at a time. We add this displacement to the current x and y positions of the sprite, then relocate the sprite to the new position. We subtract the displacement if that would put the sprite off the screen.

Lines 480–570. This is the fire button routine, called by line 430 if it's pressed. It first figures out the position of the character underneath the truck claw, then PEEKs screen memory to see what the character is. If it's a period (a trash bit), and if the truck is not carrying a trash bit, we POKE directly into the sprite shape to put a dot in the claw, then POKE a space into the position where the period was. So even in BASIC 7.0, you sometimes need to use PEEK and POKE. One point is added to the player's score, which is redisplayed using the subroutine at 720.

If the character is an empty trash can (hollow ball), and if the truck is carrying a piece of trash, we change that hollow ball to a solid ball, increment the filled trash can counter, and award ten points. If all 12 trash cans are full, we award a 1000 point bonus and increment the difficulty level, without letting the difficulty level exceed 3. The game is restarted at line 140.

Notice the use of BEGIN and BEND. BEGIN starts a block of code that is executed only if a preceding IF was true. BEND ends the block. So BEGIN–BEND lets you extend the statement after a THEN into several lines. I placed a colon on these extended lines to remind myself that they are part of a BEGIN–BEND block.,

Lines 580–700. This is the collision routine, called automatically whenever the truck is hit. The function BUMP(1) reads the sprite-to-sprite collision register. The collision routine should be called only when the sprites collide, but I found it was entered twice for every time the truck was hit. The check in line 580 prevents false collisions. I still don't know why this is necessary.

For the collision, we print a silly message, make a high-pitched sound effect, move the truck back to the bottom of the screen, remove any trash bit the truck may be carrying, reset the collision with A=BUMP(1), then decrease the number of trucks. If there are still trucks remaining, we continue with the game by RETURNing from the sprite interrupt.

For the "game over" routine, we play another tune, print the GAME OVER message, and wait for the fire button to be pressed while we redraw GAME OVER in different colors. Before we check for the button press, we first wait for the player to let go of the button in case the player was picking up or dropping a trash bit. Otherwise, the game would instantly restart.

Lines 720–1110. These are simple subroutines. Line 720 updates the score; line 730 updates the number of remaining trucks; 740 stops all sprites; 750 restarts them; and 760–780 read in the joystick displacements and sprite shapes. The rest of the program is made up of DATA statements for the cars and the truck.

The descriptions above can give you an idea of the detail required to program even a simple game. This is not meant to discourage, but to challenge.

Program 3-3. Litter Patrol

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
AE 100 GRAPHIC 0,1:COLOR 0,12:COLOR 4,6:GOSUB760:D
       F=1:LV=5:R=RND(-TI):R=RND(0)
KR 110 FORI=1TO40:SP$=SP$+" ":LN$=LN$+"-"
HX 120 BL$=BL$+CHR$(210):UL$=UL$+CHR$(183):NEXT
CE 130 BL$=CHR$(146)+BL$:UL$=CHR$(146)+UL$:SP$=CHR
       $(18)+SP$
AX 140 FORI=1TO8:SPRITE I,0:MOVSPR I,0,0:NEXT
MA 150 PRINTCHR$(147);:COLOR 5,6:PRINTSP$
HJ 160 TL=38:COLOR 5,7:PRINTLEFT$(SP$,40);:COLOR 5
       ,6:PRINT" "SP$
JE 170 PRINT:PRINT:COLOR 5,16:PRINT LN$:PRINT:PRIN
       T
RB 180 COLOR 5,6:PRINT SP$SP$:PRINT:PRINT:COLOR 5,
       16:PRINTLN$:PRINT:PRINT
JS 190 COLOR 5,6:PRINT SP$SP$:PRINT:PRINT:COLOR 5,
       8:PRINTBL$;UL$:PRINT:PRINT
AF 200 COLOR 5,6:PRINTSP$LEFT$(SP$,40)CHR$(157)CHR
       $(148)CHR$(32);
CP 210 COLOR 5,14:CHAR 1,12,0," LITTER PATROL ",1:
       GOSUB720 :GOSUB730
```

```
FM 220 COLOR 5,6:FOR Y=2 TO 27 STEP 7
ED 230 FOR X=8 TO 32 STEP 12:CHAR 1,X,Y,CHR$(215),
       1:NEXT:NEXT
SC 240 M$="V1 O2 T0 IEGGEGGEGG .Q #E I DFFDFFDFF .
       QE"
PQ 250 M$=M$+"I EGGEGGEGG .Q A I AAFGGEFFD .Q CRRR
       "
MQ 260 SPRCOLOR 2,1
BX 270 RESTORE 1110:FORI=2TO7:READ SY,SN:SPRSAV SS
       $(SN),I
CQ 280 MOVSPR I,0,56+SY*8
FM 290 SPRITE I,1,I+1,0,0,0,1
XK 300 ANG%(I)=90-180*(SN=2):SPD%(I)=5*RND(1)+DF
MX 310 MOVSPR I,ANG%(I)#SPD%(I)
RF 320 NEXT
QJ 330 XP=102:YP=237:MOVSPR 1,XP,YP
DC 340 SPRITE 1,1,11,0,0,0,1:SPRSAV SS$(0),1
SS 350 COLLISION 1,580
GK 360 IF DF=1 THEN PLAY M$:SOUND 1,0,0
KH 370 IFTI-T>120THENCOLOR 5,6:CHAR 1,TL,1,CHR$(32
       ),1:T=TI:TL=TL-1:IFTL<0THEN640
KH 380 GET A$:IF A$<>"" THEN GOSUB740:GETKEY A$:GO
       SUB750
KS 390 IF RND(1)<.95 THEN420
RG 400 S%=2+6*RND(1):X=RSPPOS(S%,0):Y=RSPPOS(S%,1)
DP 410 IF X>31 AND X<336 THEN COLOR 5,2:CHAR 1,(X-
       24)/8,(Y-50)/8+1,"."
SC 420 J=JOY(2):IF J=0 THEN370
EE 430 IF J AND 128 THEN 480
PC 440 XP=XP+DX(J):IF XP<24 OR XP>343 THEN XP=XP-D
       X(J)
FE 450 YP=YP+DY(J):IF YP<61 OR YP>237 THEN YP=YP-D
       Y(J)
PC 460 MOVSPR 1,XP,YP
JD 470 GOTO 370
CR 480 X%=(XP-24)/8+1:Y%=(YP-50)/8:SP=1024+X%+40*Y
       %:C=PEEK(SP)
BC 490 IF C=46 AND HT=0 THEN BEGIN:POKE 3584,65:PO
       KESP,32:HT=1
JK 500 :SOUND 1,700,20,0,600,10,3:SC=SC+1:GOSUB720
       :BEND
FS 510 IF HT AND C=215 THEN BEGIN:POKE SP,209:POKE
        3584,64:SOUND 1,5000,5,,,,3
DA 520 :HT=0:F=F+1:SC=SC+10:GOSUB720:FL=FL+1:IF FL
       <12 THEN 370
KJ 530 :GOSUB740:FORI=0TO63
MM 540 :COLOR 5,(IAND15)+1:CHAR 1,4,12,"BONUS 1000
        POINTS FOR COMPLETION",1
JX 550 ::NEXT:SC=SC+1000:DF=DF-(DF<3):FL=0:GOTO140
PH 560 BEND
```

```
DK 570 GOTO370
XG 580 IF BUMP(1)=0 THEN RETURN
PE 590 COLOR 5,9:CHAR 1,12,0,"OH! YOWEE OUCH!",1
HJ 600 FORI=1TO11:SPRITE1,1,I:SOUND 1,2000+RND(1)*
       1000,1,,,,3:NEXT
RB 610 COLOR 5,14:CHAR 1,12,0," LITTER PATROL ",1
AQ 620 XP=102:YP=237:MOVSPR 1,XP,YP:POKE 3584,64:H
       T=0:A=BUMP(1)
SX 630 LV=LV-1:GOSUB730:IF LV THEN RETURN
QA 640 COLLISION 1:PLAY "T0 O2 I C C E E G R B R A
        A F D Q C R R":SOUND 1,0,0
GH 650 COLOR 5,16:CHAR 1,7,12,"GAME OVER -- PRESS
       {SPACE}TRIGGER",1:C=0
HS 660 IF JOY(1)=128 THEN660
BB 670 IF JOY(2)=128 THEN690
EH 680 COLOR 5,C+1:CHAR 1,7,12,"GAME OVER",1:C=(C+
       1)AND15:GOTO670
DJ 690 FORI=1TO8:SPRITE I,0:MOVSPR I,0,0:NEXT
AM 700 RUN
FS 720 COLOR 5,15:CHAR 1,0,0,"SCORE:"+MID$(STR$(SC
       ),2),1:RETURN
AK 730 COLOR 5,4:CHAR 1,30,0,"TRUCKS:"+STR$(LV),1:
       RETURN
BH 740 FORQQ=2 TO7:MOVSPR QQ,90#0:NEXT:RETURN
MK 750 FOR I=2TO7:MOVSPR I,ANG%(I)#SPD%(I):NEXT:RE
       TURN
DF 760 FOR I=0TO8:READ DX(I),DY(I):NEXT
CF 770 FOR I=0 TO 2:FOR J=1 TO 67:READ A$:SS$(I)=S
       S$(I)+CHR$(DEC(A$)):NEXT:NEXT
RG 780 RETURN
XG 790 DATA 0,0,0,-8,8,-8,8,0,8,8,0,8,-8,8,-8,0,-8
       ,-8
CM 810 DATA 40,10,00,40,10,00,40,10,00,15,40,00,05
       ,00,00,05
JM 820 DATA 00,00,2A,80,00,EA,B0,00,EA,B0,00,2A,80
       ,00,2A,80
KR 830 DATA 00,EA,B0,00,EA,B0,00,00,00,00,00,00,00
       ,00,00,00
BD 840 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00
       ,00,00,17
QM 850 DATA 00,14,00
QB 900 REM CAR FACING LEFT
RM 910 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00
       ,00,00,00
FF 920 DATA 00,00,0F,00,3C,AB,96,A9,2B,AA,6A,3A,7D
       ,6A,2A,7D
BM 930 DATA 6A,3A,7D,6A,2B,AA,6A,AB,96,A9,0F,00,3C
       ,00, 00,00
MK 940 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00
       ,00,00,17
```

```
AA 950 DATA ØØ,14,ØØ
GQ 1ØØØ REM CAR FACING RIGHT
KX 1Ø1Ø DATA ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,Ø
        Ø,ØØ,ØØ
BK 1Ø2Ø DATA ØØ,ØØ,ØØ,3C,ØØ,FØ,6A,96,BA,69,AA,E8,A
        9,7D,AC
XJ 1Ø3Ø DATA A9,7D,A8,A9,7D,AC,69,AA,E8,6A,96,BA,3
        C,ØØ,FØ
QR 1Ø4Ø DATA ØØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,
        ØØ,ØØ,ØØ
SP 1Ø5Ø DATA ØØ,ØØ,ØØ,17,ØØ,14,ØØ
GC 11ØØ REM POSITION AND DIRECTION OF EACH CAR
PS 111Ø DATA 2,1,5,1,9,2,12,2,16,1,2Ø,2
```

Word Search

Michael B. Williams
128 Version by Patrick Parrish

"Word Search" is a computerized puzzle-maker that can provide hours of challenging fun. A printer is required.

You're probably familiar with word search puzzles: Certain words are hidden in a rectangle of nonsense letters, and it's your job to hunt them down. "Word Search" lets you create such puzzles on your computer's printer with words of your own choice. Since you design the puzzle, you can make it as easy or as difficult as you want, using up to 100 different words. Topical puzzles make the game even more interesting. For example, you might include only computer words, the names of foreign cities, or stumpers like *uxorious* and *bougainvillaea.* Parents and teachers can make puzzles for children using weekly vocabulary lists.

Save a copy of Word Search and refer to the notes below before running the program.

Word Search begins by asking you for the number of words to be hidden. When you've answered that question, the computer asks you to choose the number of rows and columns for the puzzle grid. Since the grid must be big enough to hide all the words, the computer tells you when you've made the grid too small and lets you try again.

Next, Word Search lets you enter the words one by one. There's no particular limit on word length, but keep in mind that the words must fit inside the grid. (For example, you can't fit a 12-letter word in a 6 × 6 grid.) Since longer words are harder to fit into the grid, the computer sorts the words by length (from longest to shortest) so it can place the longest words first. When many words are involved, this can take a few minutes, so be patient.

Once the words are sorted, you're allowed to name the puzzle. You also have the option of printing the solution to the puzzle (parents and teachers might want to separate the solution from the puzzle until the puzzle has been tried). After printing one puzzle, you can create another, using the same word list (the words will be rearranged) or entirely new

words. Word Search is designed to permit a maximum of 100 words in a 99 × 99 grid. However, puzzles of that size can take a long time to create—over an hour in some cases. In addition, many printers can't print more than 80 columns unless you first send the printer a special escape code for condensed type (see your printer manual).

Program 3-4. Word Search

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
QA 95 MC=99
DD 100 DIM FF$(100),S$(99),W$(100),CC(100),RR(100)
       ,L(100),E$(2,2)
EE 110 FOR I=-1 TO 1
BH 120 FOR J=-1 TO 1
SX 130 READ E$(I+1,J+1)
JX 140 NEXT J
QX 150 NEXT I
AC 160 DATA "NW"," N","NE"," W","{2 SPACES}"," E",
       "SW"," S","SE"
BM 170 FOR I=1 TO MC
RA 180 G$=G$+" "
RC 190 NEXT I
DH 200 FOR I=1 TO 8
BK 210 READ D(1,I),D(2,I)
QD 220 NEXT I
SA 230 DATA -1,-1,-1,0,-1,1,0,-1
HA 240 DATA 0,1,1,-1,1,0,1,1
QC 250 GOTO 1220
PD 260 REM SHELL SORT
KS 270 PRINT "SORTING..."
HA 280 X=1
AX 290 X=2*X
QS 300 IF X<=W0 THEN 290
JP 310 X=INT(X/2)
HJ 320 IF X<>0 THEN 340
DM 330 RETURN
QQ 340 FOR Y=1 TO W0-X
RD 350 Z=Y
XR 360 A=Z+X
GB 370 IF L(Z)>=L(A) THEN 460
BH 380 X$=W$(Z)
QS 390 W$(Z)=W$(A)
PS 400 W$(A)=X$
GB 410 B=L(Z)
JQ 420 L(Z)=L(A)
EM 430 L(A)=B
XK 440 Z=Z-X
BG 450 IF Z>0 THEN 360
```

```
RP 460 NEXT Y
SA 470 GOTO 310
QC 480 REM HIDE WORDS
DX 490 FOR X=1 TO W0
JE 500 FOR Y=1 TO 50
HG 510 R1=INT(RND(1)*R0)
JC 520 C1=INT(RND(1)*C0)
KD 530 D1=INT(RND(1)*8)+1
GE 540 O1=D1
AH 550 DX=D(1,D1)
MG 560 DY=D(2,D1)
GS 570 IF R1+DX*L(X)<1 OR R1+DX*L(X)>R0 OR C1+DY*L
       (X)<1 THEN 590
ME 580 IF C1+DY*L(X)<=C0 THEN 630
PK 590 D1=D1*(D1<8)*(1=1)+1
XH 600 IF D1<>O1 THEN 550
CD 610 NEXT Y
BM 620 GOTO 800
FS 630 FOR Z=1 TO L(X)
EH 640 IF MID$(W$(X),Z,1)<"A" OR MID$(W$(X),Z,1)>"
       Z" THEN 680
RA 650 R1=R1+DX
CJ 660 C1=C1+DY
CM 670 IF MID$(S$(R1),C1,1)<>" " AND MID$(S$(R1),C
       1,1)<>MID$(W$(X),Z,1) THEN 590
GK 680 NEXT Z
QG 690 FOR Z=L(X) TO 1 STEP -1
AE 700 IF MID$(W$(X),Z,1)<"A" OR MID$(W$(X),Z,1)>"
       Z" THEN 770
FD 710 S$(R1)=MID$(S$(R1),1,C1-1)+MID$(W$(X),Z,1)+
       MID$(S$(R1),C1+1)
DS 720 RR(X)=R1
BX 730 CC(X)=C1
XC 740 FF$(X)=E$(DX+1,DY+1)
GJ 750 R1=R1-DX
EB 760 C1=C1-DY
KX 770 NEXT Z
RX 780 NEXT X
JM 790 GOTO 890
QK 800 GOSUB 1720
GB 810 PRINT "SORRY, BUT I CAN'T FIT WORD NUMBER "
       ;STR$(X);" , ";W$(X);" , ";
DC 820 PRINT "INTO THE GRID. SHOULD I SKIP IT, STA
       RT OVER, OR TRY AGAIN"
PH 830 INPUT X$
BA 840 IF MID$(X$,1,2)="ST" THEN 1660
SP 850 IF MID$(X$,1,2)="TR" THEN 500
JH 860 IF MID$(X$,1,2)<>"SK" THEN 830
AH 870 W$(X)="/"
BB 880 GOTO 780
```

```
RF 890 FOR X=1 TO R0
GK 900 FOR Y=1 TO C0
KG 910 IF MID$(S$(X),Y,1)<>" " THEN 930
CC 920 S$(X)=MID$(S$(X),1,Y-1)+CHR$(INT(26*RND(1)+
       65))+MID$(S$(X),Y+1)
JJ 930 NEXT Y
EJ 940 NEXT X
XA 950 REM DONE
KE 960 PRINT
JF 970 PRINT "I AM FINISHED. WHAT DO YOU WANT TO C
       ALL THE WORD SEARCH"
EX 980 INPUT T$
FK 990 SL=0
JP 1000 PRINT
XF 1010 PRINT "DO YOU WANT TO PRINT THE SOLUTION (
        Y/N)"
RG 1020 GOSUB 1180
BX 1030 IF A$="N" THEN 1050
PK 1040 SL=1
DC 1050 GOSUB 2000
PP 1060 GOSUB 1720
ME 1070 F=0
PG 1080 PRINT "DO YOU WANT ANOTHER GRID (Y/N)"
GR 1090 GOSUB 1180
HG 1100 IF A$="Y" THEN 1120
SE 1110 END
KD 1120 PRINT
FM 1130 PRINT "DO YOU WANT TO USE THE SAME WORDS (
        Y/N)"
RS 1140 GOSUB 1180
PM 1150 IF A$="N" THEN 1280
MM 1160 F=1
JM 1170 GOTO 1340
XM 1180 INPUT A$
JJ 1190 IF A$<>"Y" AND A$<>"N" THEN 1180
PP 1200 RETURN
HC 1210 REM INITIALIZATION
AC 1220 GOSUB 1720
GF 1230 LL=6
GH 1240 GOSUB 1740
HP 1250 PRINT "{8 SPACES}WORD SEARCH"
XH 1260 LL=4
KJ 1270 GOSUB 1740
EA 1280 FOR I=1 TO W0
FQ 1290 W$(I)=""
DM 1300 L(I)=0
ER 1310 NEXT I
RP 1320 PRINT TAB(5);"HOW MANY WORDS WOULD YOU":PR
        INT TAB(5);"LIKE IN YOUR WORD SEARCH"
CP 1330 INPUT W0
```

```
QB 1340 PRINT
JH 1350 PRINT "HOW MANY ROWS AND COLUMNS IN THE GR
        ID"
SM 1360 INPUT RØ,CØ
HC 1370 PRINT
AF 1380 PRINT
GH 1390 IF RØ*CØ>=1Ø*WØ THEN 144Ø
ED 1400 PRINT "I DON'T THINK I COULD DO THIS."
RF 1410 FOR I=1 TO 1ØØØ
MC 1420 NEXT I
PM 1430 GOTO 134Ø
QM 1440 PRINT "I THINK I CAN DO THIS."
SS 1450 IF CØ<=MC THEN 147Ø
KP 1460 PRINT "(BUT IT WON'T FIT ON THE PAPER.)"
JH 1470 IF F=1 THEN 166Ø
SF 1480 LL=3
CG 1490 GOSUB 174Ø
PQ 1500 PRINT "ENTER THE ";STR$(WØ);" WORDS.":PRIN
        T"TO CORRECT A MISTAKE, ENTER X"
BM 1510 PRINT
PC 1520 FOR I=1 TO WØ
JS 1530 PRINT "WORD NUMBER ";I;":"
QB 1540 INPUT X$
MS 1550 IF LEN(X$)<=RØ AND LEN(X$)<=CØ AND X$<>"X"
         THEN 161Ø
FM 1560 IF X$<>"X" THEN 159Ø
EQ 1570 I=I-(I>1)*(1=1)
EH 1580 GOTO 153Ø
DR 1590 PRINT "OOPS...THE WORD IS TOO LONG."
CH 1600 GOTO 153Ø
HG 1610 W$(I)=X$
XB 1620 L(I)=LEN(X$)
CA 1630 NEXT I
HX 1640 GOSUB 172Ø
CG 1650 GOSUB 27Ø
KE 1660 PRINT
MD 1670 PRINT "OKAY, I WILL GO TO WORK (WISH ME LU
        CK)."
SF 1680 FOR I=1 TO RØ
XX 1690 S$(I)=LEFT$(G$,CØ)
DE 1700 NEXT I
KM 1710 GOTO 49Ø
PS 1720 PRINT CHR$(147)
CQ 1730 RETURN
PQ 1740 FOR I=1 TO LL
AM 1750 PRINT
RJ 1760 NEXT I
XB 1770 RETURN
DA 1999 REM PRINTER ROUTINE
MC 2000 OPEN3,4:PRINT#3,T$:PRINT#3
```

```
QD 2010 PRINT#3,"{4 SPACES}";:FORI=1TOC0:IFI/10<>I
        NT(I/10)THENPRINT#3," ";:GOTO2030
EC 2020 PRINT#3,MID$(STR$(I),2,1);
MM 2030 NEXTI:PRINT#3
SX 2040 PRINT#3,"{4 SPACES}";:FORI=1TOC0:PRINT#3,R
        IGHT$(STR$(I),1);:NEXTI:PRINT#3
KP 2050 FORX=1TOR0:IFX<10THENPRINT#3," ";
QX 2060 PRINT#3,STR$(X)" ";
SH 2070 FORY=1TOC0:PRINT#3,MID$(S$(X),Y,1);
XE 2080 NEXTY:PRINT#3:NEXTX:PRINT#3:PRINT#3:PRINT#
        3,"WORD LIST:"
CQ 2090 FORX=1TOW0:IFW$(X)="/"THEN2110
BM 2100 PRINT#3,W$(X)
PQ 2110 NEXTX:FORI=1TO5:PRINT#3:NEXTI:IFSL=0THEN21
        80
ER 2120 PRINT#3,"SOLUTION LIST:":PRINT#3,"WORD
        {21 SPACES}ROW{3 SPACES}COLUMN";
XE 2130 PRINT#3,"{3 SPACES}DIR"
JM 2140 FORX=1TOW0:IFW$(X)="/"THEN2170
MM 2150 PRINT#3,W$(X);LEFT$(G$,25-LEN(W$(X)));RR(X
        );LEFT$(G$,8-LEN(STR$(RR(X))));
BP 2160 PRINT#3,CC(X);LEFT$(G$,6-LEN(STR$(CC(X))))
        ;FF$(X)
CM 2170 NEXTX
XE 2180 CLOSE3:RETURN
```

Switchbox

Todd Heimarck

It looks easy, but takes time to master. "Switchbox" is a challenging strategy game that's fun to watch and play.

Playing "Switchbox" is like putting dominos in place for a chain reaction—either you're setting them in position or you're knocking them over. Winning requires skill and a sense of when to go for points and when to lay back and wait for a better board. The goal is simple: You try to score more points than your opponent by dropping balls into a boxful of two-way switches. Each switch has a trigger and a platform. If the ball lands on an empty platform, it stops dead. But if it hits a trigger, it reverses the switch and continues. In many cases dropping a single ball creates a cascading effect—one ball sets another in motion, which sets others in motion, and so on, all the way down.

Type in Program 3-5 using "The Automatic Proofreader" and save a copy before you run it.

A Box of Switches

Switchbox is a tale of twos: Each switch has two parts, two positions, two states, two paths in, and two paths out. The two parts are the platform and the trigger. A switch can lean to the left (platform left, trigger right) or to the right (platform right, trigger left).

Figure 3-1. Trigger States

The trigger is weak and always allows balls to pass. But the platform is strong enough to hold a single ball. So the platform either holds a ball—it's full—or it does not and is empty. When a ball sits on a platform, the switch is said to be loaded, or full.

Figure 3-2. Loaded Trigger

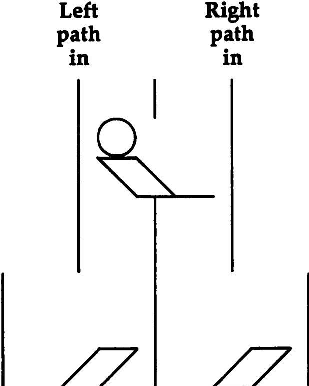


Figure 3-2 shows a full switch over two empty switches. The platform holds a ball and leans to the left. The trigger extends to the right. Note that the switch on top has two pathways leading in, the left path and the right, and that the right path leading out is the left path into one of the switches below. The left path of the top switch leads into the right path of the other, the switch below and to the left. If you drop a ball down the right-hand path, it hits the trigger and flips that switch to the right. Then it continues down, hits the left-hand trigger below, and flips that switch as well.

In the meantime, the ball on the platform is set in motion (when the switch is flipped) and then hits the trigger. The top switch is reset to point to the left. The second ball then drops a level to the platform below, where it stops. The playing field is composed of five levels, with four switches in the first level and eight in the bottom level. At the beginning of the game, there are no balls on the field—all platforms are empty—and the position of each switch is chosen randomly.

Moving Down the Path

Players alternate dropping balls into one of eight entry points. These balls (and others) may or may not make it all the way through the switchbox to one of the 16 exit paths. Balls fall straight down (with one exception), so a ball's movement is always predictable. When a ball hits an empty switch, one of two things can happen. If it lands on the empty platform, it stops dead in its tracks. But if it lands on a trigger, it falls through to the next level below.

Moving balls always make it through loaded switches. Triggers allow balls to continue and move the switch to the other position. If it's loaded, the dead ball on the platform is put into motion and it hits the trigger that just moved over. This makes the switch go back to its original position, but with an empty platform. So when a ball hits the trigger of a loaded switch, its motion continues unabated. The switch moves, the ball on the platform begins to fall, and it hits the newly placed trigger. The newly emptied switch moves back again, and the two balls drop to the next level.

There's one more possibility: a ball dropping onto a platform that already holds a ball. A platform can't hold more than one ball, so when this happens, one of the balls slides over to the trigger. So the ball does not move straight down—it slides over to the next pathway. This is the exception to the rule that balls drop in a straight line. Of course, when the ball hits the trigger, the switch changes position, causing the other ball to drop and hit the trigger.

The Chain Reaction

At the game's start, all platforms are empty, so four of eight entry paths are blocked. Remember that your turn ends when a ball hits an empty platform and stops. As the switches fill up, the chances increase that a ball will descend through several levels. The goal is to score points by getting balls to pass all the way through the maze of the switchbox. The best way to collect a lot of points is to cause a chain reaction.

A ball that hits a loaded switch from either side continues on its way. And the previously inert ball on the platform starts moving. One enters, two exit. If both of those balls encounter full platforms, four drop from the switches. The pathways are staggered, so the effects can spread outward, with more and more balls cascading toward the bottom.

Rather than taking an easy point or two, it's often worthwhile to build up layers of loaded switches. Watch out for leaving yourself vulnerable, though. Because players take turns, you'll want to leave positions where your opponent's move gives you a chance to create a chain reaction. The best strategy is to play defensively. Look ahead a move or two, and watch for an opening that allows you to score several points at once.

Four Quarters

A game of Switchbox always lasts four rounds. In the first (equality), each exit counts for two points. Your goal is to score ten points. The second quarter has more points available as well as a higher goal. If you look at the exits, you'll see that the farther away from the middle, the higher the point value. The numbers increase in a Fibonacci sequence: 1, 2, 3, 5, 8, and so on. Each number is the sum of the previous two (1+2 is 3, 2+3 is 5, 3+5 is 8, and so on). The target score in round 2 is 40.

In round 3 the numbers are a bit lower. They increase arithmetically (1, 2, 3, 4, up to 8 in the corners). A goal of 20 points brings you to round 4, where you can score big. Here the numbers are squares: 1, 4, 9, 16, 25, all the way to 64 at the edges. In rounds 2–4, it's sometimes prudent to leave a middle path open for your opponent to score a few points in order to gather a high score on the big numbers to the left and right.

Each round lasts until one player has reached the goal. At that point the other player has one last turn before the round ends. It's possible to win the round on this last-chance play; watch out for barely topping the goal and leaving a chain reaction open for the other player. An arrow points to the scoreboard of the player whose turn it is. On the other side of the screen, you'll see a number where the arrow should be. That's the goal for the current round.

Bonus points are awarded at the conclusion of each round. Four numbers appear below the scorecards. The first is simply the total so far. The second is the total plus a bonus of the goal for the round if the player's points are equal to or greater than the goal. For example, if the goal is 20 and you get 18, there's no bonus. If you score 22, the bonus is the goal

for that round (20), and you'd have 42 points. The third number under the scoreboard is the difference between scores for the rounds. If you win by two points, two is added to your score (and two is subtracted from the other player's). The final number is the grand total of the first three scores and bonuses. Rounds 1 and 3 are fairly low-scoring with low goals. You may want to seed the field with extra balls during these quarters so that you can collect more points in the second and fourth quarters.

Variations

Although the goal of the game is to score the most points, there's no reason you couldn't agree to play for low score. In a "lowball" game, you would try to avoid scoring points. You wouldn't necessarily play backward; you would have to adjust the strategy of where to place the balls. Fill up the board as much as possible and leave your opponent in a situation where he or she is forced to score points.

The DATA statements at the beginning of the program determine the goal for each round and the point values for the exit paths. You can prolong the game by doubling the goals; this also dilutes the value of a big score at the beginning of a round, preventing one player from winning on the first or second turn. An interesting variation is to assign negative values to some slots. If some paths score negative points, you are forced to think harder about where the balls will drop.

In addition to the numbered keys (1–8), the plus (+) and minus (−) keys are active. Pressing the plus key drops a ball at random down one of the eight entry paths. Pressing minus allows you to pass your turn to your opponent.

Once you've mastered the regular game, you can add some new rules. Each player gets three passes per half, similar to the three timeouts in a football game. If you don't like the looks of the board, press the minus key to use one of your passes. After one player has skipped a turn, the other player must play (this prevents the possibility of six passes in a row). It's also a good idea to make a rule that a player can't pass on two consecutive turns. You can also give each player two random moves to be played for the opponent. In other words, after making a move, you could inform your opponent that

you're going to give him or her one of your random moves and you would press the plus key.

Here's one more change you could make: Instead of alternating turns, allow a player to continue after scoring. When a player drops a ball and scores some points, the other player would have to pass (by pressing the minus key). If the first player scores again, the opponent passes again, and so on, until no more points are scored.

Playing Solitaire

To drop a ball, press a number key (1–8). The numeric keypad is convenient for choosing a move. By using the pass and random-turn options, you can play against the computer. Here are the rules for solitaire play:

1. The computer always scores first. At the beginning of every round, the computer plays randomly until at least one point is acquired. Press the plus key for the computer's turn. You must continue passing (skip your turn with the minus key) until the computer puts points on the board.
2. After the first score by the computer, you can begin to play. When the computer has a turn, press the plus key for a random move.
3. Whenever you make points, you must pass again until the computer scores. When the computer gets more points, you can begin to play again. This rule means you should hold back on the easy scores of a few points; wait until there's an avalanche available.
4. If you're the first to reach the goal, the computer gets a last chance. Don't make this move randomly; figure out the best opportunity for scoring and play that move for the last-chance turn.

In the interest of keeping these programs to a manageable length, no attempt has been made to provide an "intelligent" computer opponent. Once you become familiar with the game, you might find it an interesting project to try adding some routines that give the computer a rational basis for picking one move over another.

Program 3-5. Switchbox

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
FP 1Ø  DIMSW(4,7,1),SP$(1),LB(32,4),AR$(1),PT(4,16)
       ,SC(1,8)
DE 12  SP$(Ø)="{OFF}E*3{RVS}E*3{OFF}E@3":SP$(1)="
       E@3{RVS}£{OFF}£":AR$(Ø)="<I {DOWN}{2 LEFT}
       JEW3":AR$(1)="EQ3K{UP}{2 LEFT} U>":QR=1:PRIN
       TCHR$(27);"M"
EC 14  COLORØ,16:COLOR4,7:COLOR5,7:TX=RND(-TI/137)
QS 2Ø  FORJ=1TO4:READPT(J,Ø):REM NAME AND GOAL
XC 22  FORK=1TO8:READL:PT(J,K+8)=L:PT(J,9-K)=L:NEXT
       K,J:REM POINTS
RP 24  DATA 1Ø:REM ROUND 1 (EQUAL)
PE 25  DATA 2,2,2,2,2,2,2,2
PF 26  DATA 4Ø:REM ROUND 2 (FIBONACCI)
HP 27  DATA 1,2,3,5,8,13,21,34
KJ 28  DATA 2Ø:REM ROUND 3 (ARITHMETIC)
BG 29  DATA 2,3,4,5,6,7,8,9
EB 3Ø  DATA 8Ø:REM ROUND 4 (SQUARES)
SF 31  DATA 1,4,9,16,25,36,49,64
PB 4Ø  SCNCLR:INPUT"PLAYER 1";P1$:INPUT"PLAYER 2";P
       2$:P1$=LEFT$(P1$,5):P2$=LEFT$(P2$,5):PRINTP1
       $;" VS ";P2$
JD 42  PRINT"IS THIS CORRECT?":GETKEYA$:IFASC(A$)<>
       89THEN4Ø
PB 5Ø  GOSUB5ØØ:GOSUB7ØØ:REM SETUP
HD 6Ø  FORRR=1TO4:TX=1Ø72+4Ø*RR:POKETX,9Ø:POKETX+22
       ,9Ø
XG 62  GOSUB62Ø:REM PUT SCORES AT BOTTOM
SF 65  QR=1-QR:COLOR5,7:TY=QR*2Ø:TX=28-TY:WINDOWTX,
       Ø,TX+2,1,1:PRINTRIGHT$(STR$(PT(RR,Ø)),3):PRI
       NT"{2 HOME}":TX=8+TY:CHAR1,TX,QR,AR$(QR)
SK 7Ø  GOSUB9ØØ:IFSC(1-QR,RR)=>PT(RR,Ø)THEN3ØØ:REM
       {SPACE}END OF ROUND
AK 8Ø  GOTO65
EX 3ØØ  FORJ=ØTO1:FORK=5TO8:SC(J,K)=Ø:NEXTK,J
QP 31Ø  FORJ=ØTO1:FORK=1TO4:GL=PT(K,Ø):AC=SC(J,K):S
        C(J,5)=SC(J,5)+AC:SC(J,6)=SC(J,6)-(AC=>GL)*
        GL:SC(J,7)=SC(J,7)+(SC(J,K)-SC(1-J,K)):NEXT
        K,J
QB 32Ø  FORJ=ØTO1:FORK=6TO7:SC(J,K)=SC(J,K)+SC(J,5)
        :NEXTK,J
SC 33Ø  FORJ=ØTO1:FORK=5TO7:SC(J,8)=SC(J,8)+SC(J,K)
        :NEXTK,J
ME 34Ø  COLOR5,12:FORJ=ØTO1:FORK=5TO8:Y$=STR$(SC(J,
        K)):L=LEN(Y$):TX=6+J*31-L:TY=3+K:CHAR1,TX,T
        Y,Y$:NEXTK,J
CX 4ØØ  NEXTRR:REM END OF MAIN LOOP 6Ø-499
EB 499  GETKEYA$:RUN
```

```
BA 500 SCNCLR:PRINTSPC(11);"⁅A⁆{RVS}⁅O⁆{OFF}";:FOR
       J=1TO7:PRINT"{OFF}⁅R⁆{RVS}⁅O⁆";:NEXT:PRINT"
       {OFF}⁅S⁆":LL=7
QB 510 FORJ=ØTO4:TX=9-2*J:TY=1+J*4:BX=TX+2Ø+J*4:BY
       =TY+4:WINDOWTX,TY,BX,BY:R$="- "
CD 520 FORK=1TO2:PRINT"{2 SPACES}{RVS} {OFF} ";:GO
       SUB6ØØ:PRINT"{RVS} ":NEXT
BQ 530 PRINT" {RVS}£ {OFF} ";:GOSUB6ØØ:PRINT"
       {RVS} ⁅*⁆"
PM 540 LL=LL+2:PRINT"{RVS}£{OFF}£";:GOSUB6ØØ:PRI
       NT"{LEFT}⁅*⁆{RVS}⁅*⁆{OFF}";:NEXTJ
JP 550 WINDOW1,21,38,23:PRINT"⁅R⁆ ";:GOSUB6ØØ:PRIN
       T"⁅R⁆"
BF 560 R$="{RVS}⁅U⁆{OFF}⁅E⁆":LL=LL+1:PRINT"⁅Z⁆";:G
       OSUB6ØØ:PRINT"{LEFT}⁅X⁆":WINDOWØ,Ø,39,24
QS 599 RETURN
KX 6ØØ FORL=1TOLL:PRINTR$;:NEXT:RETURN
MA 62Ø COLOR5,12:FORJ=1TO16:K=PT(RR,J):JJ=2+J*2
RK 63Ø IFK>9THENL=INT(K/1Ø):L$=MID$(STR$(L),2,1):E
       LSEL$=CHR$(32)
MC 64Ø CHAR1,JJ,23,L$:CHAR1,JJ,24,RIGHT$(STR$(K),1
       ):NEXTJ:RETURN
SX 7ØØ FORJ=ØTO4:SY=4+J*4:FORK=ØTOJ+3:SX=12-J*2+K*
       4:CHAR1,SX+1,SY-1," "
MX 71Ø WP=INT(RND(1)*2)
HA 72Ø SW(J,K,Ø)=WP:SW(J,K,1)=Ø:GOSUB8ØØ
RM 73Ø NEXTK,J
SK 74Ø FORJ=1TO8:POKE1Ø74+J*2,48+J:NEXT
XJ 75Ø FORJ=ØTO1:BX=J*31:WINDOWBX,Ø,BX+7,7
BQ 76Ø PRINT"{OFF}{BLK}⁅D⁆{RVS}{PUR}{7 SPACES}
       {BLK}⁅K⁆{PUR}⁅D⁆⁅5 I⁆⁅F⁆";
EQ 77Ø FORK=1TO4:PRINT"{RVS}{BLK}⁅K⁆{OFF}{PUR}⁅K⁆
       {5 SPACES}{RVS}⁅K⁆";:NEXT
KQ 775 PRINT"{RVS}{BLK}⁅K⁆{PUR}⁅C⁆{OFF}⁅5 I⁆{RVS}
       ⁅V⁆{OFF}{BLK}⁅C⁆{RVS}⁅6 I⁆{OFF}⁅V⁆";
HK 78Ø NEXT:PRINT"{2 HOME}":COLOR5,5
RE 79Ø CHAR1,3+(LEN(P1$)=5),Ø,P1$,1
QJ 791 CHAR1,34+(LEN(P2$)=5),Ø,P2$,1
RP 799 RETURN
BA 8ØØ COLOR5,2:CHAR1,SX,SY,SP$(WP):RETURN
JJ 9ØØ FORJ=ØTO32:LB(J,Ø)=Ø:NEXT:NB=1:POKE2Ø8,Ø
RC 91Ø GETKEYA$:IFA$="-"THENRETURN:ELSEIFA$="+"THE
       NA$=STR$(INT(RND(1)*8+1))
FX 915 A=VAL(A$):IF(A<1)OR(A>8)THEN91Ø
FK 92Ø LB(Ø,Ø)=1:FORJ=1TO3:LB(Ø,J)=Ø:NEXT:LB(Ø,4)=
       1Ø+A*2
SF 1ØØØ DO:EX=1
KR 1Ø1Ø FORJ=ØTO32:IFLB(J,Ø)THENEX=Ø:GOSUB11ØØ
GP 1Ø2Ø NEXT:IFEXTHENEXIT
EF 1Ø3Ø LOOP:RETURN
```

```
KJ 1100 DY=LB(J,Ø):DX=LB(J,1):LY=LB(J,2):NY=LB(J,3
        ):NX=LB(J,4):SM=1Ø64+NX+LY*16Ø+NY*4Ø:IF(LY
        +NY)THENPOKESM,32
GJ 111Ø LB(J,3)=(NY+1)AND3:ONNY+1GOTO12ØØ,13ØØ,14Ø
        Ø,15ØØ
EE 12ØØ IFLY>4THENLB(J,Ø)=Ø:GOTO17ØØ:REM SCORING R
        OUTINE
QE 122Ø POKESM+4Ø,81:ONINT(RND(1)*3+1)GOTO18ØØ,181
        Ø,182Ø
QS 13ØØ VX=Ø:GOSUB16ØØ:IF SW(WY,WX,1)AND(SW(WY,WX,
        Ø)=SD)THEN VX=1-2*SD:LB(J,1)=VX:LB(J,3)=NY
        +1:LB(J,4)=NX+VX:POKESM+4Ø+VX,81:GOTO184Ø
EG 131Ø IF SW(WY,WX,Ø)=SDTHENLB(J,Ø)=Ø:SW(WY,WX,1)
        =1:POKE SM+4Ø,81:GOTO183Ø
HC 132Ø LB(J,3)=NY+1:POKESM+4Ø,81:ONINT(RND(1)*3+1
        )GOTO18ØØ,181Ø,182Ø
QD 14ØØ LB(J,1)=Ø:LB(J,4)=NX+DX:POKESM+4Ø+DX,81:GO
        TO185Ø
FD 15ØØ LB(J,2)=LY+1:POKESM+4Ø,81:GOSUB16ØØ:SW(WY,
        WX,Ø)=1-SW(WY,WX,Ø)
DA 151Ø IF SW(WY,WX,1)THENLB(NB,Ø)=1:LB(NB,1)=Ø:LB
        (NB,2)=LY:LB(NB,3)=Ø:LB(NB,4)=NX+2-SD*4:NB
        =NB+1:SW(WY,WX,1)=Ø:POKESM-4Ø+2-SD*4,32:GO
        SUB186Ø
PA 152Ø SX=12-WY*2+WX*4:SY=4+WY*4:WP=SW(WY,WX,Ø):G
        OSUB8ØØ:GOTO184Ø
FH 16ØØ WY=LY:JX=(NX/2)+LY-6:WX=INT(JX/2):SD=JXAND
        1:RETURN
KX 17ØØ SF=PT(RR,NX/2-1)
RA 171Ø SG=SC(QR,RR)+SF:COLOR5,12
GG 172Ø TX=5+31*QR+(SG>9)+(SG>99)+(SG>999)
QS 173Ø TY=1+RR:A$=MID$(STR$(SG),2)
JJ 174Ø CHAR1,TX,TY,A$:SC(QR,RR)=SG:GOTO187Ø
MJ 18ØØ SOUND1,45ØØ,8:RETURN
CP 181Ø SOUND1,9ØØØ,8:RETURN
FC 182Ø SOUND1,675Ø,8:RETURN
AH 183Ø SOUND2,75ØØ,8,1,625Ø,125,1,1Ø24:RETURN
QD 184Ø SOUND2,6ØØØ,12,2,42ØØ,15Ø,3:RETURN
EH 185Ø SOUND2,3ØØØØ,12,2,1ØØØØ,5ØØØ,3:RETURN
BX 186Ø SOUND3,15ØØ,24,Ø,145Ø,25,3:RETURN
RQ 187Ø SOUND1,12ØØØ,24:SOUND2,75ØØ,12,Ø,73ØØ,25:S
        OUND3,9ØØØ,18:RETURN
```

Lexitron

Ron Wilson

Like a bowl of alphabet soup, the "Lexitron" screen appears to be just a jumbled mass of letters. Can you find the ten hidden words before time runs out? A joystick is required.

If you enjoy the hidden word games often found in newspapers and magazines, you'll like "Lexitron." But unlike the ones done with pencil and paper, Lexitron adds a few twists. There's a time limit, and you can select one of three difficulty levels.

The game is written entirely in BASIC. After typing it in, be sure to save a copy. Be especially careful when typing in the DATA statements in lines 1200 and 1210. These lines hold the word pool from which Lexitron selects.

Up, Down, Left, and Right

To play Lexitron, load it and type RUN. Be sure to have a joystick plugged into port 2. First, you'll be asked to select one of three skill levels. Level 1 is the easiest, with all the hidden words spelled left to right or top to bottom. Level 2 is more difficult. Besides forward spellings, words are also formed in their reversed spelling order (from right to left or bottom to top). Level 3 is the most difficult, with words spelled diagonally, and both forward and reversed diagonal words being formed. You might want to stay clear of level 3 until you've played a few times.

After you've selected a skill level by moving your joystick to the appropriate number, press the fire button. The screen will clear for a few seconds while the game words are being selected and hidden. But don't leave your seat—the timer starts as soon as the game appears on the screen.

Using your joystick, move the cursor to the word you've found, and press the fire button on each letter until you complete the word. Each time a correct letter is registered, the time level, which moves from top to bottom, is pushed back toward the top. Avoid guessing letters by trial and error. Wrong entries only reduce the amount of time.

All valid game words are at least six letters long. This rule is in force so that accidental (and sometimes humorous) letter combinations do not cost you time and effort. You'll often see words like MAN, CAR, SEE, or TRY, but Lexitron does not recognize them. Also be aware of letter additions. For example, Lexitron may choose and hide the word AMERICA, but by chance the letter following could be an *N*, thus AMERICAN. Lexitron may not recognize the extra *N*.

Easy Modifications

The Lexitron vocabulary words are coded so that players cannot list the program and get an illegal sneak preview. If you wish to add your own words to the program, the code is simple. Each letter represents the letter which alphabetically follows. For example, the letter *A* is coded as *B*. ABACUS would be coded as BCBDVT. If you decide to add your own words, start with a new line—1220—and remember that all words must be at least six letters long. Be sure the last word in the list is FOE (the word END in code). This signals to the program that it's reached the end of the word list. You might want to avoid using words with the letters *X*, *Z*, or the *Q-U* combination. A sharp player can spot words with those letters in seconds.

If you find that Lexitron is too easy or too difficult, you can change the value .009 in line 330. This controls the timer. Raising and lowering this value will change the allotted time, thus the difficulty of the game. A value less than .009 (such as .007) makes the game easier, and, conversely, increasing the value makes it more difficult.

There are a few strategies to consider when playing Lexitron. For instance, in some cases it's not to your advantage to enter a word as soon as you find it. If you have trouble finding some of the hidden words, Lexitron sometimes provides a clue by flashing a word at the bottom of the screen.

Program 3-6. Lexitron

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
EM 10 BANK 15
HB 100 PRINT"{CLR}{3 DOWN}"SPC(11)"L E X I T R O N
       {3 DOWN}":DIMAR$(20,15),AR(20,15),LN(200)
FQ 110 P$="{HOME}{23 DOWN}{15 SPACES}":FORA=0TO10:
       READQ(A):NEXT
```

```
CB 120 FORA=ØTO8:READD(A):NEXT:DATA Ø,1,5,Ø,7,8,6,
       Ø,3,2,4,Ø,-4Ø,-39,1,41,4Ø,39,-1
JA 13Ø DIML(2ØØ):DATA -41:PRINTSPC(3)"PLEASE WAIT
       {SPACE}WHILE LOADING WORDS"
MD 14Ø S=54272:FORA=STOS+23:POKEA,Ø:NEXT:POKEA,15:
       POKES+5,28:POKE5328Ø,6
KG 15Ø SC=1274:CD=54272:PT=5632Ø:BC=53281:W=15:BL=
       6:SB=53265:CS=15
PB 16Ø Z=646:A1=15:A2=14:A3=1:A4=7:A5=6:A6=1
EK 17Ø GOSUB116Ø:GOSUB112Ø
SD 18Ø GOSUB1Ø9Ø
KE 19Ø GW=Ø:FORA=ØTO19:FORB=ØTO14:AR(A,B)=Ø:NEXTB,
       A:FORA=ØTONW:L(A)=Ø:NEXT:PS=SC
ME 2ØØ LL=7:POKEBC,W:POKEZ,A5:PRINT"{CLR}{3 DOWN}
       {5 SPACES}USE JOYSTICK TO CHOOSE A LEVEL"
CS 21Ø POKEZ,A6:PRINTSPC(1Ø)"{2 DOWN}{5 SPACES}
       {RVS}1";:POKEZ,A5:PRINT"{2 SPACES}2
       {2 SPACES}3":MS=SC+5
HQ 22Ø PRINT"{4 DOWN}{3 SPACES}{RVS}1{OFF}
       {2 SPACES}ALL WORDS SPELLED FORWARDS
       {14 SPACES}NO DIAGONALS"
SM 23Ø PRINT"{3 SPACES}{2 DOWN}{RVS}2{OFF}
       {2 SPACES}FORWARD & REVERSED SPELLINGS
       {12 SPACES}NO DIAGONALS"
CM 24Ø PRINT"{3 SPACES}{2 DOWN}{RVS}3{OFF}
       {2 SPACES}FORWARD & REVERSED SPELLINGS
       {12 SPACES}DIAGONALS INCLUDED"
PG 25Ø GOSUB41Ø:IFJ=ØTHEN25Ø
RS 26Ø POKEMS+CD,A5:MS=MS-3*(J=3)*-(MS<SC+11)+3*(J
       =7)*-(MS>SC+5):POKEMS+CD,A6
QS 27Ø IFJ<128THEN25Ø
QJ 28Ø PRINT"{CLR}{BLK}{8 DOWN}"SPC(14)"PLEASE WAI
       T":FORTD=1TO1ØØØ:NEXT
RM 29Ø LV=(MS-(SC+5))/3+1:POKESB,PEEK(SB)AND239:PO
       KEBC,BL:GOSUB97Ø
XK 3ØØ GOSUB85Ø:QF=1:T=1:GOSUB48Ø:POKESB,PEEK(SB)O
       R16:TI$="ØØØØØØ":GOTO39Ø
BS 31Ø REM JOYSTICK ROUTINE
EP 32Ø IFRND(1)<.ØØ3THENGOSUB67Ø
FP 33Ø IFRND(1)<.ØØ9THENGOSUB54Ø
BA 34Ø GOSUB41Ø:IFJ=ØTHEN32Ø
AM 35Ø IFJ>127THENGOSUB43Ø:GOTO32Ø
RR 36Ø POKEPS,PEEK(PS)AND127:TP=PS
QE 37Ø IF(PEEK(PS+CD)ANDCS)=A3THENPOKEPS+CD,A2
CP 38Ø PS=PS+D(JAND15):IFPEEK(PS)<1ORPEEK(PS)>26TH
       ENPS=TP
MG 39Ø POKEPS,PEEK(PS)OR128:IF(PEEK(PS+CD)ANDCS)<>
       A4THENPOKEPS+CD,A3
JP 4ØØ GOTO32Ø
```

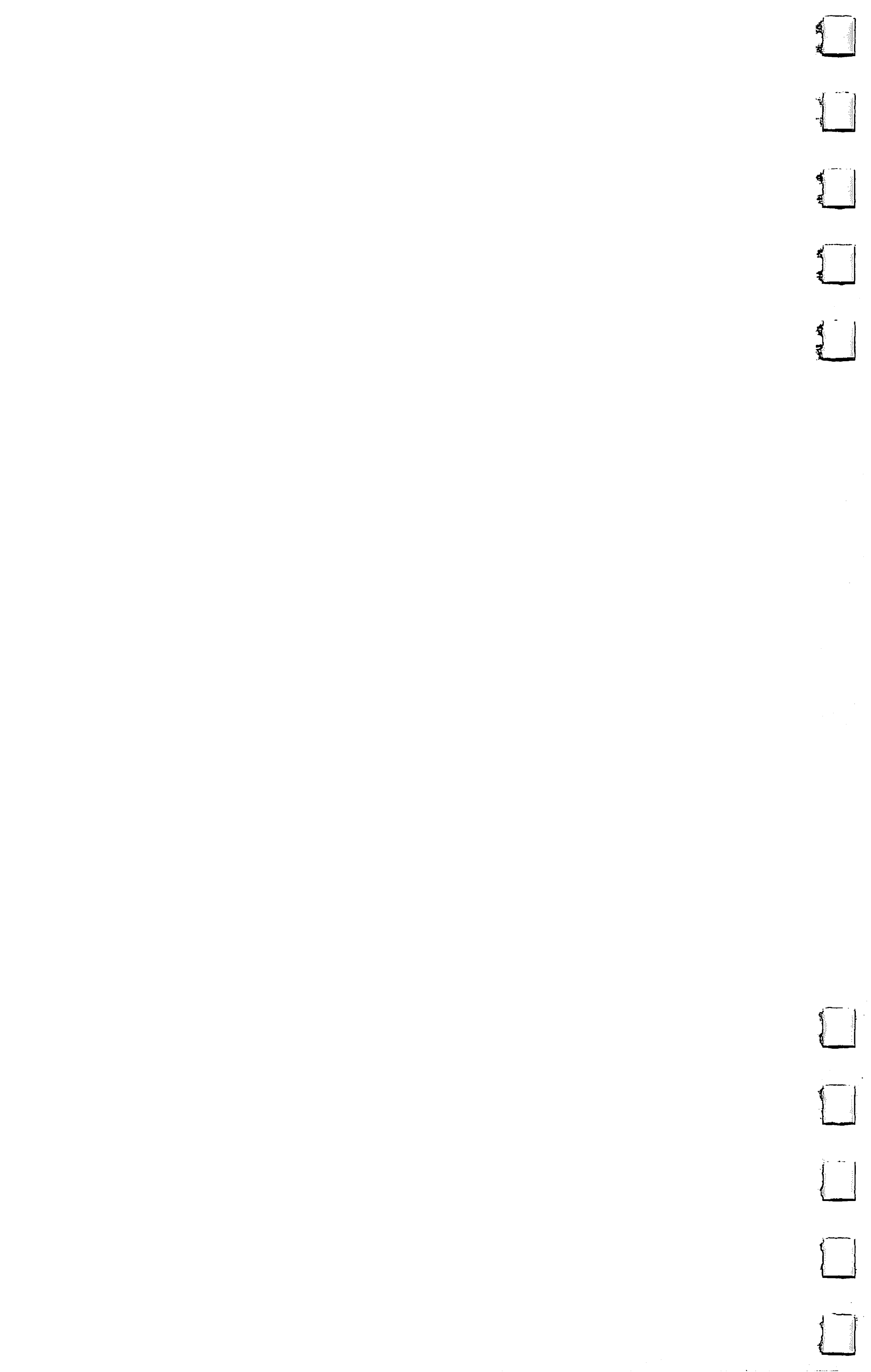
```
EC 410 J=Q(15-(PEEK(PT)AND15))-128*((PEEK(PT)AND16
       )=Ø):RETURN
BH 420 REM PRESS FIREBUTTON
EH 430 IF(PEEK(PS+CD)ANDCS)=A4THENRETURN
QX 440 Y=INT((PS-SC)/4Ø):X=PS-SC-Y*4Ø
KJ 450 IFAR(X,Y)=ØTHENPRINTP$"TRY ANOTHER":GOSUB54
       Ø:RETURN
QG 460 PRINTP$"GOOD ANSWER"
BP 470 POKEPS+CD,A4:T=AR(X,Y)-1:L(T)=L(T)+1
GD 480 POKES+4,16:POKES+4,17:POKES+1,1Ø+1Ø*L(T)
FP 490 FORA=SC+554TO(SC+554)-4Ø*LLSTEP-4Ø:POKEA+4Ø
       ,93:POKEA+71,93
KM 500 POKEA,16Ø:POKEA+31,16Ø:FORTD=1TO2Ø:NEXT:NEX
       T:IFQF=1THENQF=Ø:RETURN
GF 510 GOSUB59Ø:LL=LL-(LL<15):AR(X,Y)=Ø:IFL(T)=LN(
       T)THENGOSUB61Ø
EG 520 RETURN
JH 530 REM ENERGY DRAIN
CJ 540 POKES+4,32:POKES+4,33:POKES+1,3Ø:FORTD=1TO2
       ØØ:NEXT:POKES+1,15
SD 550 FORA=SC-6TO(SC-6)+4Ø*(15-LL)STEP4Ø:POKEA-4Ø
       ,93:POKEA-9,93:POKEA,16Ø
AP 560 POKEA+31,16Ø:FORTD=1TO2Ø:NEXT:NEXT:GOSUB59Ø
       :LL=LL-1:IFLL=-1THEN7ØØ
JM 570 RETURN
RJ 580 REM CLEAR MESSAGE
XQ 590 PRINTP$"{19 SPACES}":RETURN
GA 600 REM GOT A WORD
AK 610 GW=GW+1:IFGW=1ØTHEN77Ø
JK 620 FORTD=1TO3ØØ:NEXT:PRINTP$"WELL DONE!"
AB 630 POKES+4,32
SD 640 POKES+4,35:FORA=6TO2Ø:POKES+1,A:FORTD=3ØTO4
       Ø:POKES+15,TD:NEXT:NEXT
RK 650 GOSUB59Ø:RETURN
GJ 660 REM GIVE A CLUE
SX 670 A=RND(1)*1Ø:IFL(A)=ØTHENPRINTP$"CLUE: "W$(A+
       M):FORT=1TO8ØØ:NEXT
BS 680 GOSUB59Ø:RETURN
SC 690 REM END OF GAME
KQ 700 PRINT"{HOME}{4 DOWN}{BLK}":FORA=ØTO14:PRINT
       :PRINTSPC(1Ø);:FORB=ØTO19
RB 710 IFAR(B,A)<>ØTHENPRINTAR$(B,A);:GOTO73Ø
JQ 720 PRINT"{RIGHT}";
HG 730 NEXT:NEXT:PRINT:POKEZ,A4:PRINT "{2 DOWN}
       {6 SPACES}PRESS FIREBUTTON TO CONTINUE"
RX 740 GOSUB41Ø:IFJ<128THEN74Ø
XJ 750 GOSUB41Ø:IFJ>127THEN75Ø
JS 760 POKEZ,A5:PRINT"{CLR}{4 DOWN}{6 SPACES}SORRY
       , YOU DIDN'T MAKE IT{4 DOWN}":GOTO79Ø
```

```
GE 770 POKEZ,A5:PRINT"{CLR}{4 DOWN}{6 SPACES}CONGR
       ATULATIONS!{2 SPACES}YOU WON"
QD 780 PRINT"{3 DOWN}{9 SPACES}YOUR TIME WAS "MID$
       (TI$,3,2)":"RIGHT$(TI$,2)
XR 790 FORI=1TO1000:NEXT:POKEBC,W:PRINT"{5 DOWN}
       {4 SPACES}PRESS FIREBUTTON TO PLAY AGAIN"
PA 800 PRINT"{2 DOWN}{10 SPACES}PRESS DOWN TO QUIT
       "
RG 810 GOSUB410:IFJ>127THENPRINTSPC(13)"{2 DOWN}
       {RVS}PLEASE WAIT":GOSUB1170:GOTO180
QE 820 IFJTHENPRINT"{CLR}":END
RK 830 GOTO810
MR 840 REM PRINT ARRAY
KG 850 PRINT"{CLR}{4 SPACES}{CYN}{RVS}{K}{OFF}{K}
       {2 SPACES}{RVS}{K}{D}{I}{OFF}{V}{RVS}{K}
       {OFF}{K}{RVS}{K}{OFF}{K} {RVS}{F}{D}{OFF}
       {C}{RVS}{F}{D}{OFF}{V}{RVS}{K}{D}{F}{OFF}
       {F}{D}{RVS}{D}{F}{OFF}{F}{RVS}{K}{C}{K}
       {OFF}{K}"SPC(8);
CP 860 PRINT"{RVS}{K}{OFF}{K}{2 SPACES}{RVS}{K}{C}
       {OFF}{F}{2 SPACES}{RVS}{F}{D}{OFF}
       {2 SPACES}{RVS}{K}{OFF}{K}{2 SPACES}{RVS}
       {K}{OFF}{K} {RVS}{K}{C}{V}{OFF}{V}{RVS}{K}
       {OFF}{K}{RVS}{K}{OFF}{K}{RVS}{K}{2 SPACES}
       {OFF}{K}"SPC(8)"{7}{RVS}{K}{OFF}{K}";
FG 870 PRINT"{2 SPACES}{RVS}{K}{OFF}{K}{2 SPACES}
       {D}{RVS}{D}{F}{OFF}{F} {RVS}{K}{OFF}{K}
       {2 SPACES}{RVS}{K}{OFF}{K} {RVS}{K}{D}{C}
       {OFF} {RVS}{K}{OFF}{K}{RVS}{K}{OFF}{K}{RVS}
       {K}{OFF}{K}{RVS}{F}{OFF}{K}"SPC(8)"{C}{RVS}
       {2 I}{OFF}{V}{C}";
AG 880 PRINT"{RVS}{2 I}{OFF}{V}{C}{V}{C}{V} {RVS}
       {2 I}{OFF}{2 SPACES}{C}{V} {C}{V}{C}{V}
       {RVS}{2 I}{OFF} {C}{V}{C}{V}{DOWN}"
MX 890 POKEZ,A1:PRINTSPC(4)"B{4 SPACES}{RVS}{A}***
       *****************{S}{OFF}{4 SPACES}B"
PJ 900 PRINTSPC(4)"B{4 SPACES}{RVS}B{OFF}";:FORA=0
       TO14:FORB=0TO19
FH 910 POKEZ,A2:IFAR$(B,A)=" "THENPRINTCHR$(65+RND
       (1)*26);:GOTO930
PC 920 PRINTAR$(B,A);
KG 930 NEXT:POKEZ,A1:PRINT"{RVS}B{OFF}{4 SPACES}B"
       :PRINTSPC(4);"B{4 SPACES}{RVS}B{OFF}";:NEXT
HB 940 PRINT:PRINTSPC(4)"{UP}B{4 SPACES}{RVS}{Z}**
       ******************{X}{OFF}{4 SPACES}B"
CD 950 RETURN
KK 960 REM PUT WORDS IN THE ARRAY
MB 970 B=INT(RND(1)*(NW-10))
GH 980 M=B:FORA=0TO9:W$=W$(B+A)
```

```
HR 990  DR=RND(1)*2↑LV:DY=DY(DR):DX=DX(DR):L=LEN(W$
        )
FF 1000  SX=INT(RND(1)*19+1):SY=INT(RND(1)*14+1):RX
         =SX:RY=SY
GD 1010  NX=SX+(L-1)*DX:NY=SY+(L-1)*DY:IFNX<0ORNY<0
         ORNX>19ORNY>14THEN990
KB 1020  FL=0:FORL=1TOLEN(W$):IFAR$(SX,SY)<>" "THEN
         FL=1:L=LEN(W$)
XJ 1030  SX=SX+DX:SY=SY+DY
SQ 1040  NEXT:IFFLTHENA=A-1:NEXT
MX 1050  FORL=1TOLEN(W$):AR$(RX,RY)=MID$(W$,L,1)
JD 1060  AR(RX,RY)=A+1:RX=RX+DX:RY=RY+DY:NEXT:LN(A)
         =LEN(W$):NEXT
MC 1070  RETURN
CH 1080  REM SHUFFLE WORDS
HE 1090  FORA=1TONW/2:B=RND(1)*NW:C=RND(1)*NW:T$=W$
         (B):W$(B)=W$(C):W$(C)=T$
JX 1100  NEXT:RETURN
HH 1110  REM READ IN WORDS
DM 1120  DIMW$(200):NW=-1
MA 1130  NW=NW+1:READW$:FORA=1TOLEN(W$):W$(NW)=W$(N
         W)+CHR$(ASC(MID$(W$,A,1))-1)
FP 1140  NEXT:IFW$(NW)<>"END"THEN1130
SK 1150  RETURN
BJ 1160  FORA=0TO7:READDX(A),DY(A):NEXT
BF 1170  FORA=0TO19:FORB=0TO14:AR$(A,B)=" ":NEXTB,A
FM 1180  RETURN
AX 1190  DATA 0,1,1,0,0,-1,-1,0,-1,1,1,1,1,-1,-1,-1
CX 1200  DATA DBSOJWBM,DJSDVT,DBOBEB,VOJUFE,UFMFQIP
         OF,NPOLFZ,DPNQVUFS
RB 1210  DATA KPZTUJDL,NPOTUFS,TUBQMFS,NBHJDBM,TIVG
         GMF,FOE
```

Chapter 4

Utilities



MetaBASIC: Programmer's Problem Solver

Kevin Mykytyn

"MetaBASIC," originally written for the Commodore 64, has been rewritten to work with the 128 running in 128 mode. It adds 11 new debugging and testing commands to BASIC 7.0.

You've bought your first car and it runs well. But when you take it out on the highway, you're dismayed to find that it won't go faster than 45 miles per hour. What do you do?

Take it to your favorite mechanic who might give you three options: Remove the engine and replace it with a brand-new one. Or add some fancy turbocharging, fuel-injected doohickeys to the engine you already have. Or, without adding anything, you could tune it up, using a special machine that measures the engine's performance.

A BASIC Tune-Up

You can add new programming commands to your 128 in three similar ways. The first is to toss out BASIC and create a whole new language (a more powerful engine) based on your ideas of what a programming language should do.

The second method, a language extension, keeps BASIC, but adds some new programming commands. You keep the BASIC engine, but add some additional parts which make it work faster or more efficiently.

The third way is like a tune-up which doesn't change the engine. You add direct mode commands for debugging. This is not a new language or even an extension of BASIC; it's more properly called a *development system* or *writing/debugging tool.* The new commands you add cannot be used inside a program; they work only in immediate mode.

New languages and extensions have several advantages. But they also have a major drawback: You have to load the language or extension *before* you load the main program, or the program just won't work.

The nice thing about a development system like "MetaBASIC" is that it's there when you need it, during the time you're writing and tuning up a program. But once you've finished the program, you don't need MetaBASIC to run it—you can disconnect the tune-up machine.

Typing It In

MetaBASIC is written entirely in machine language, and "MLX," the Machine Language Editor, is required to type it in. MLX can be found in Appendix C. Be sure to read the specific directions for using MLX before typing in MetaBASIC. Save a couple of copies of MLX; you'll need it to type in "Orbitron" and other programs which appear in other COMPUTE! publications.

Load and run MLX. Give it the following information:

Starting Address: 1300
Ending Address: 18BF

Next, following the MLX instructions, enter MetaBASIC and save it.

To use MetaBASIC, follow these steps:

1. BLOAD "MetaBASIC" (for disk) or
   LOAD"MetaBASIC",1,1 (for tape)
2. SYS 4864

After the SYS, it may seem that nothing is happening. But MetaBASIC is running in the background, and you now have 11 new commands to help you write and debug programs.

Using MetaBASIC

MetaBASIC uses English mnemonics, so you don't have to memorize a lot of SYS numbers. Once MetaBASIC is active, you'll have these 11 additional commands: AID, CHANGE, DEFAULT, DLIST, FIND, MERGE, QUIT, READ, RESAVE, START, and UNNEW.

The commands work only in direct mode; you cannot add them to programs. Also, you're limited to one command per line (although you can still use multistatement lines inside your programs). Unlike ordinary BASIC commands, there are no abbreviations. You must type out the entire MetaBASIC command. If you wish to stop the execution of a command, press the RUN/STOP key (*not* RUN/STOP–RESTORE). If it

seems to be working incorrectly, make sure the syntax is correct.

Machine language programmers should remember that MetaBASIC occupies memory locations $1300–$18BF (4864–6335) and uses zero page locations $FB–$FE (251–254) and $AC–$AF (172–175).

MetaBASIC Commands

Here's an alphabetical list of the new commands and how to use them, with examples. MetaBASIC commands and strings appear in boldface and numbers appear in italics. Anything enclosed in parentheses is optional.

If something is described as a disk command, it won't work unless you have a disk drive. However, some of the ML programming aids can be useful in BASIC and vice versa.

AID

Syntax: **AID**

Lists all available MetaBASIC commands.

CHANGE

Syntax: **CHANGE @*old string*@*new string*@** (*,startnum, endnum*)
CHANGE @*old string*@*new string*@ (*,startnum*)
CHANGE @*old string*@*new string*@ (*,, endnum*)
CHANGE /*old string*/*new string*/ (*,startnum, endnum*)
CHANGE /*old string*/*new string*/ (*,startnum*)
CHANGE /*old string*/*new string*/ (*,, endnum*)

See also FIND.

CHANGE searches through the program in memory, changing every occurrence of the old string to the new one. The strings can be up to 30 characters long and must be bracketed by the commercial at sign (@) or the slash (/). All lines in which changes are made are listed to the screen. The format with the commercial at sign is the tokenized form and should be used to change BASIC commands and variable names. The ASCII form (the slash format) is useful when you want to change a

word in a string without changing keywords. For example,

CHANGE /print/write/

will change all occurrences of the word *print* within quotation marks without changing any PRINT statements.

Use the slash format to change anything inside quotation marks or after a REM statement; use the at sign format to change anything not inside quotation marks or after a REM statement. Remember that mathematical operators within programs such as +, −, *, /, >, <, and = are stored as tokens, not characters, so you must use the @ format when searching for one of these.

If you omit the line numbers, CHANGE affects the whole program. If you want to change only one section, add the starting and ending line numbers, marked off by commas.

Example: **CHANGE @X@QQ@,,200** changes the variable X to QQ in all lines up to and including 200. To change the name Charles to John throughout the program, **CHANGE /CHARLES/JOHN/**.

DEFAULT

Syntax: **DEFAULT** ***border, background, text***

See also QUIT.

When you hit RUN/STOP–RESTORE, the screen reverts to the default colors green and black. DEFAULT lets you change these values to whatever you prefer.

If your 128 is hooked up to a black-and-white TV, change the character/background color to a more readable combination.

To disable DEFAULT (and go back to normal colors), use the QUIT command.

Example: **DEFAULT1,1,0** changes border and background to white, and characters to black. If you press RUN/STOP–RESTORE, you'll see white characters on a black background.

The border and background color changes affect only the 40-column screen; the text color change affects both the 40- and 80-column displays.

DLIST

Syntax: **DLIST** *"filename"*

See also READ.

This command lists a BASIC program from disk to the screen without affecting what's currently in memory. The program name must be enclosed in quotation marks. DLIST enables you to look at a program before using MERGE or SCRATCH.

Example: **DLIST "BASICPROGRAM"** reads the file from disk and lists it to the screen.

FIND

Syntax: **FIND @***string***@** (*,startnum, endnum*)
FIND @*string***@** (*,startnum*)
FIND @*string***@** (*,, endnum*)
FIND */string/* (*,startnum, endnum*)
FIND */string/* (*,startnum*)
FIND */string/* (*,, endnum*)

See also CHANGE.

This allows you to find any word, variable, or other string within a program. Each line containing the search string is listed to the screen. If you wish to search just one section of the program, add the starting and ending line numbers, separated by commas.

If you're trying to find BASIC keywords (like PRINT or REM), use the first format. It also works for variables and numbers. The second format should be used when you're looking for strings or items inside quotation marks.

Example: **FIND @A=@** searches for lines where variable A is defined.

MERGE

Syntax: **MERGE** *"program name"*

MERGE reads a program from disk, lists each line to the screen, and adds the line to the program in memory. If the programs have common line numbers, the program on disk takes precedence. Say they both contain a line 250. The line 250 from the disk program will replace line 250 in memory.

Before using this command, you may want to use DLIST to make sure you're merging the right program. And if there are conflicting line numbers, you can use RENUMBER to renumber one of the two programs. If you want to merge just part of one program, use DELETE to eliminate the unwanted lines.

QUIT

Syntax: **QUIT**

This resets all vectors and disables all MetaBASIC commands. MetaBASIC is still protected from BASIC. Reenter the program with SYS 4864.

READ

Syntax: **READ** ***"filename"***

See also DLIST.

READ allows you to examine sequential disk files. The information in the file is displayed to the screen, without altering whatever program is in memory.

In the rare case that you want to use the BASIC READ statement from direct mode (to see if all DATA statements have been read), you can precede it with a colon to distinguish it from MetaBASIC's READ command.

RESAVE

Syntax: **RESAVE** ***"filename"***

The disk command save-with-replace (SAVE "@0:*filename*") first saves the program and then scratches the older version, so there must always be enough free space on the disk for the new version of the program. Thus, the command can cause problems if you don't have enough available disk space for the new version. Save-with-replace also has other problems; see the article "Save-with-Replace: Debugged at Last" earlier in this book.

RESAVE reverses the order—first it scratches the old version of your program from disk, and then it does a regular SAVE, solving both of the above problems.

START

Syntax: **START** *"filename"*

If you forget where a machine language program begins, put the disk in the drive and use this command. This can help when you have forgotten the SYS that starts a program.

Example: **START "METABASIC"** should display 4864 on the screen.

UNNEW

Syntax: **UNNEW**

You may never need this command, but it's nice to have it available. If you accidentally type NEW and you want to retrieve the program, use UNNEW to get it back.

Program 4-1. MetaBASIC

This data must be entered using MLX. See Appendix C.

Starting address: 1300
Ending address: 18BF

```
1300:4C 15 13 4C E5 5E 4C 32 2C
1308:8E 4C E8 4D 4C A0 50 4C 4C
1310:E5 50 4C 64 50 A2 34 A0 23
1318:13 D0 0E A9 03 8D 00 0A B0
1320:A9 40 8D 01 0A A2 0D A0 82
1328:43 8E 04 03 8C 05 03 A2 65
1330:80 6C 00 03 AD 00 FF 8D DC
1338:C1 18 A9 00 8D 00 FF A9 90
1340:FB 8D 28 03 A9 17 8D 29 EA
1348:03 A9 0F A8 A2 08 20 BA F6
1350:FF 20 EA 17 A9 00 20 BD 98
1358:FF 20 C0 FF A2 00 8E C2 93
1360:18 A0 FF C8 B9 00 02 F0 0A
1368:3F C9 20 F0 F6 DD B7 13 65
1370:D0 28 C8 E8 BD B7 13 29 CC
1378:7F D9 00 02 D0 1C BD B7 1F
1380:13 10 EF AD C2 18 0A AA 42
1388:8C C3 18 84 3D A9 02 85 4B
1390:3E BD EF 13 48 BD EE 13 9E
1398:48 60 E8 BD B7 13 F0 08 E7
13A0:10 F8 E8 EE C2 18 10 B9 69
13A8:18 A9 0F 20 C3 FF AD C1 64
13B0:18 8D 00 FF 4C 0D 43 41 A4
13B8:49 C4 43 48 41 4E 47 C5 39
13C0:44 45 46 41 55 4C D4 44 01
13C8:4C 49 53 D4 46 49 4E C4 D7
```

```
13D0:4D 45 52 47 C5 51 55 49 15
13D8:D4 52 45 41 C4 52 45 53 08
13E0:41 56 C5 53 54 41 52 D4 4C
13E8:55 4E 4E 45 D7 00 03 14 44
13F0:53 14 1D 14 7F 16 50 14 B3
13F8:7C 16 1A 13 34 17 4F 17 0B
1400:C8 17 EF 17 A0 FF C8 B9 12
1408:B7 13 F0 0F 08 29 7F 20 E3
1410:D2 FF 28 10 F1 20 30 17 2F
1418:4C 06 14 4C 01 18 20 A8 80
1420:18 8E B7 18 20 A8 18 8E D2
1428:B6 18 20 A8 18 8E B8 18 C4
1430:A9 3D 8D 00 0A A9 14 8D DA
1438:01 0A 4C 01 18 AD B6 18 F9
1440:8D 21 D0 AD B7 18 8D 20 C5
1448:D0 AD B8 18 85 F1 4C 03 6C
1450:40 A9 00 2C A9 01 8D C5 F7
1458:18 20 3A 16 A2 FF 20 5C EE
1460:16 AD C5 18 F0 03 20 5F 6C
1468:16 A9 FF 85 47 85 48 A5 E4
1470:2D 85 FC A5 2E 85 FD 20 2E
1478:A1 18 90 17 20 12 13 A5 10
1480:61 85 FC A5 62 85 FD 20 F9
1488:86 03 20 A4 18 90 04 86 94
1490:47 85 48 A0 00 B1 FC 8D 1F
1498:C2 18 C8 B1 FC 0D C2 18 16
14A0:D0 06 20 03 13 4C 01 18 CA
14A8:C8 B1 FC 85 16 C8 B1 FC CD
14B0:85 17 A5 47 38 E5 16 A5 B5
14B8:48 E5 17 B0 02 90 E3 A2 29
14C0:00 C8 B1 FC F0 30 DD C7 EC
14C8:18 D0 F4 8C C6 18 C8 E8 A9
14D0:BD C7 18 F0 09 D1 FC F0 56
14D8:F5 AC C6 18 D0 E1 8C C2 6B
14E0:18 A5 FC 85 61 A5 FD 85 99
14E8:62 20 0F 13 AD C5 18 D0 E2
14F0:15 AC C6 18 D0 C9 C8 98 01
14F8:18 65 FC 85 FC A5 FD 69 62
1500:00 85 FD 4C 93 14 AD C2 1B
1508:18 18 65 FC 8D B9 18 A5 E9
1510:FD 69 00 8D BA 18 AD C6 C4
1518:18 18 65 FC 8D BD 18 A5 0A
1520:FD 69 00 8D BE 18 AD BD EB
1528:18 18 6D C0 18 8D BD 18 A8
1530:AD BE 18 69 00 8D BE 18 46
1538:20 8E 15 AD BD 18 38 ED 40
1540:B9 18 8D BD 18 AD BE 18 E7
1548:ED BA 18 8D BE 18 AD 10 B5
1550:12 18 6D BD 18 8D 10 12 3C
1558:AD 11 12 6D BE 18 8D 11 39
```

```
1560:12 A0 00 B1 FC 18 6D BD B7
1568:18 91 FC C8 B1 FC 6D BE 4A
1570:18 91 FC A2 FF E8 BD C7 BB
1578:18 D0 FA AC C6 18 88 E8 9D
1580:C8 BD C7 18 F0 04 91 FC B0
1588:D0 F5 88 4C BF 14 AD 10 28
1590:12 38 ED B9 18 8D BB 18 B1
1598:AD 11 12 ED BA 18 8D BC 0D
15A0:18 AD B9 18 38 ED BD 18 08
15A8:8D C2 18 AD BA 18 ED BE F8
15B0:18 0D C2 18 D0 01 60 B0 FF
15B8:3E AE BC 18 18 8A 6D BA 47
15C0:18 8D E4 15 AD B9 18 8D 5A
15C8:E3 15 18 8A 6D BE 18 8D F9
15D0:E7 15 AD BD 18 8D E6 15 9F
15D8:E8 AC BB 18 D0 04 F0 0D 21
15E0:A0 FF B9 FF FF 99 FF FF F8
15E8:88 C0 FF D0 F5 CE E4 15 5E
15F0:CE E7 15 CA D0 EA 60 AD 6C
15F8:B9 18 8D 1C 16 AD BA 18 6E
1600:8D 1D 16 AD BD 18 8D 1F 60
1608:16 AD BE 18 8D 20 16 AE CB
1610:BC 18 F0 20 A9 00 8D BF E8
1618:18 A0 00 B9 FF FF 99 FF 47
1620:FF C8 CC BF 18 D0 F4 EE F0
1628:1D 16 EE 20 16 E0 00 F0 6D
1630:08 CA D0 E0 AD BB 18 D0 98
1638:DD 60 EE C3 18 AC C3 18 98
1640:B9 00 02 D0 03 4C 2C 18 50
1648:C9 20 F0 EE C9 2F F0 09 64
1650:C9 40 D0 F1 48 20 0D 43 CA
1658:68 85 FC 60 AC C3 18 A9 0E
1660:FF 8D C0 18 C8 E8 B9 00 E6
1668:02 F0 DA 9D C7 18 EE C0 44
1670:18 C5 FC D0 EF 84 3D A9 7C
1678:00 9D C7 18 60 A9 00 2C 5C
1680:A9 01 8D C5 18 A0 00 20 33
1688:5F 18 20 E4 FF 20 E4 FF 07
1690:20 E4 FF 8D C2 18 20 E4 7A
1698:FF 0D C2 18 D0 03 4C 01 0E
16A0:18 A0 02 A9 02 8D 01 02 26
16A8:20 E4 FF 99 14 00 99 00 8B
16B0:02 C8 C0 04 D0 F2 88 C8 94
16B8:20 E4 FF 99 00 02 C9 00 63
16C0:D0 F5 8C 00 02 A2 02 C8 CB
16C8:99 00 02 CA D0 F9 A9 00 70
16D0:85 61 A9 02 85 62 38 20 B3
16D8:0F 13 AD C5 18 D0 03 4C B9
16E0:E5 16 4C 90 16 A9 04 85 FC
```

```
16E8:3D A9 02 85 3E A2 FE A0 D1
16F0:16 20 0C 17 AD 00 02 38 CC
16F8:E9 03 A8 4C 09 13 EE 20 47
1700:D0 20 1F 17 A9 00 8D 00 5C
1708:FF 4C 90 16 AD 02 03 8D C5
1710:C3 18 AD 03 03 8D C4 18 FB
1718:8E 02 03 8C 03 03 60 AD C9
1720:C3 18 8D 02 03 AD C4 18 78
1728:8D 03 03 60 A9 20 D0 02 B5
1730:A9 0D 4C D2 FF A0 02 20 D3
1738:5F 18 20 E4 FF 8D C3 18 44
1740:A5 90 D0 09 AD C3 18 20 DC
1748:D2 FF 4C 3A 17 4C 01 18 11
1750:A2 0F 20 C9 FF 20 3A 18 41
1758:AC C3 18 8A 38 6D C3 18 90
1760:AA A9 00 9D 00 02 A2 02 77
1768:BD B0 17 99 00 02 88 CA 02
1770:10 F6 AE C3 18 CA CA A9 A1
1778:02 20 B3 17 20 CC FF AD 79
1780:C4 18 AE C3 18 E8 A0 02 D0
1788:20 BD FF 20 EA 17 A0 02 2F
1790:20 E3 17 A2 01 B5 2D 95 A3
1798:FC CA 10 F9 A9 FC AE 10 48
17A0:12 AC 11 12 20 D8 FF 20 CA
17A8:7E 18 20 30 17 4C 01 18 27
17B0:53 30 3A 85 FD 98 48 86 9D
17B8:FC A0 00 B1 FC D0 03 68 42
17C0:A8 60 20 D2 FF C8 4C BB 04
17C8:17 A0 02 20 5F 18 20 E4 6D
17D0:FF 8D C3 18 20 E4 FF AE 9F
17D8:C3 18 20 06 13 20 30 17 E3
17E0:4C 01 18 A9 02 A2 08 4C 0A
17E8:BA FF A9 00 AA 4C 68 FF 01
17F0:A0 01 98 91 2D 20 03 13 DE
17F8:4C 01 18 20 6E F6 F0 01 C4
1800:60 20 21 18 18 A9 02 20 99
1808:C3 FF 18 A9 0F 20 C3 FF 38
1810:20 CC FF 20 30 17 AD C1 80
1818:18 8D 00 FF A2 80 6C 00 A7
1820:03 A9 6E 8D 28 03 A9 F6 7A
1828:8D 29 03 60 18 A9 0F 20 75
1830:C3 FF 20 21 18 A2 0B 6C 26
1838:00 03 AC C3 18 C8 B9 00 52
1840:02 F0 E9 C9 22 D0 F6 8C 56
1848:C3 18 A2 FF C8 E8 B9 00 12
1850:02 F0 04 C9 22 D0 F5 CA E5
1858:30 D2 E8 8E C4 18 60 20 C2
1860:E3 17 20 EA 17 20 3A 18 C0
1868:8A AE C3 18 E8 A0 02 20 71
```

```
1870:BD FF 20 C0 FF 20 7E 18 25
1878:A2 02 20 C6 FF 60 A2 0F C0
1880:20 C6 FF 20 E4 FF C9 30 5F
1888:D0 0D 20 E4 FF C9 0D D0 C8
1890:F9 4C CC FF 20 E4 FF 20 1F
1898:D2 FF C9 0D D0 F6 4C 01 38
18A0:18 20 80 03 C9 2C D0 0C D1
18A8:20 80 03 20 0C 13 A6 16 7B
18B0:A5 17 38 60 18 60 00 00 C8
18B8:00 00 00 00 00 00 00 00 E8
```

REM Highlighter

Don A. Ellis

If you headline routines in your programs with REMs, here's a short and clever utility that helps you find important sections of code more quickly. It highlights REMs on your screen and your printer.

Trying to find the REMark statements in a crowded program listing as it scrolls by is difficult, particularly after a few late-night hours in front of the screen. Like many other programmers, I use asterisks (***), but that's only marginally effective. Blank REM lines inserted to set off the comments and identify program routines work better, but use up both screen space and memory.

A Better Solution

"REM Highlighter" automatically *tweaks* another program, at no cost to memory, so that REM statements on separate lines will be displayed in reverse, standing out sharply. The adjusted program may be saved normally, and this version will retain its reversed comments when reloaded.

First type in and save Program 4-2, REM Highlighter. Be sure to save a copy of the program before you run it since it erases itself from memory when it loads the program to be highlighted.

Load and run Highlighter, and enter the name of the program you wish to tweak. If you're using disk, that's all there is to it. If you're using tape, the process is a little less automatic, but still simple (see details below).

Be sure to enter the program exactly as in the listing; it depends on precise screen layout to function, so the spacing is tight. Common keyword abbreviations must be used; when you see an underlined character, it means to enter it with the SHIFT key held down.

The disk version uses the dynamic keyboard technique. It POKEs keystrokes into the keyboard buffer so that when the program ends, the computer is fooled into thinking that certain keys have been pressed. REM Highlighter first loads the

program to be modified, so the Highlighter itself is overwritten (and lost). But several lines of BASIC (63994–63999) have been left on the screen. The 13's in the keyboard buffer are carriage returns, so the computer prints RETURN over lines, adding them to the program just loaded. The final line tells the program to GOTO 63995, and the program obliges by jumping to the Highlighter routine. When it's finished, blank lines numbered 63994–63999 are printed on the screen. The dynamic keyboard is again used to press RETURN over the lines, erasing them from memory. You're left with the program with reversed REMs. You can now save back to disk.

Using the Program with Tape

A special procedure is required for using Highlighter with tape. First type in Program 4-2.

It's necessary to append Highlighter to the program you wish to tweak. To accomplish this:

1. Load the program you wish to be highlighted.
2. Clear the screen; in direct mode, enter the following line:

```
PRINT45;PEEK(45),46;PEEK(46):A=PEEK(4624)
  +PEEK(4625)*256-2:C=INT(A/256):
  B=a-C*256:POKE45,B:POKE46,C
```

3. Load REM Highlighter.
4. Using the values displayed (by step 2), POKE 45 and 46 with their original values again.
5. Type RUN 63993.

Program 4-2. REM Highlighter

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
GE 10 N=208:P=842:COLOR0,7:BANK0
JD 12 PRINT"{CLR}"CHR$(14)CHR$(8):PRINT"{7}REM
      {RVS}HIGHLIGHTER{DOWN}
JS 14 INPUT"{2 DOWN}PROGRAM NAME";N$:IFN$=""THENPR
      INT"{5 UP}":GOTO14
DP 16 Q$=CHR$(34):PRINT"{CLR}{22 DOWN}LOADING
      {DOWN}":PRINTN$"{HOME}{BLU}LOAD"Q$N$Q$",8
BH 18 PRINT"{4 DOWN}63994S=PEE(S)+PEE(S+1)*256
SJ 20 PRINT"63995IFPEE(S+4)=143TH?"Q$"{HOME}{CYN}"
      Q$"PEE(S+2)+PEE(S+3)*256:T=S+4:GOS63997
HC 22 PRINT"63996ON-(S<>.)GO63994:?"Q$"{CLR}{BLU}"
      Q$";:GO63999
```

```
QK 24 PRINT"63997T=T+1:IFPEE(T)=.THRET
PK 26 PRINT"63998ON-(PEE(T)=32)GO63997:POKT+(T>S+5
      ),18:RET
BA 28 PRINT"63999POKP,19:FOI=1TO6:POKP+I,13:?63993
      +I:NE:POKP+I,154:POKN,8
KC 30 PRINT"?"Q$"{6 DOWN}{RVS}{WHT}HIGHLIGHTING"Q$
      ":N=208:P=842:S=45:GO63994
BM 32 POKEP,19:FORI=1TO8:POKEP+I,13:NEXT:POKEN,9
```

Program 4-3. REM Highlighter, Tape Modifications

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
PK 63993 BANK0:PRINT"{CLR}":N=208:P=842:S=PEEK(45)
         +PEEK(46)*256:GOTO63995
AF 63994 S=PEEK(S)+PEEK(S+1)*256
FR 63995 IFPEEK(S+4)=143THENPRINT"{HOME}"PEEK(S+2)
         +PEEK(S+3)*256:T=S+4:GOSUB63997
HM 63996 ON-(S<>.)GOTO63994:GOTO63999
GB 63997 T=T+1:IFPEEK(T)=.THENRETURN
DE 63998 ON-(PEEK(T)=32)GOTO63997:POKET+1*(T>S+5),
         18:RETURN
RH 63999 PRINT"{CLR}";:POKEP,19:FORI=1TO8:POKEP+I,
         13:PRINT63991+I:NEXT:POKEN,9
```

Blick

Plummer Hensley

Use a blick *to spice up your programs by adding a blink and a click to the PRINT command. Anytime you type a character to the screen, you'll see an underline cursor accompanied by a brief sound.*

If you don't think sound is important, try playing your favorite action game with the volume turned all the way down. It's just not as much fun without the explosions, zaps, and other noises.

Sounds help to liven up games, so why not make PRINT statements a little more interesting? This program gives you a blink and a click (a *blick)* every time a character is printed.

Typing It In

Enter Program 4-4 and save it to tape or disk before proceeding. Saving is important because the last command in line 120 is a NEW, which erases the program currently in memory.

"Blick" is written in machine language (ML), but you don't need to know ML to use it. It is presented in the form of a BASIC loader that reads DATA statements and POKEs the routine into memory. After running it, you should see the message BLICK ENABLED.

Once Blick is in memory, try printing a message, PRINT "THIS IS BLICK", for example. Or load a program and list it. See the table below for ideas on customizing the program.

If you should accidentally disable Blick by pressing RUN/STOP–RESTORE or RUN/STOP–RESET, enter SYS 3072 to reenable Blick. To turn off Blick, enter the following all on one line:

POKE 806,121:POKE 807,239

To change the cursor character, substitute the appropriate ASCII value for *x* in the following POKE:

POKE 3128,*x*

Finally, the blinking speed can be modified by substituting any number from 0 to 255 for *y* (anything above 234 will speed it up) in this statement:

POKE 3133,*y*

How It Works

Blick is a "wedge" that temporarily diverts the PRINT command into a routine that prints an underline character, makes a sound, and erases the underline. When it's finished, it goes on to the main PRINT command.

PRINT is a common, easy-to-use command in BASIC. But at the machine language level, PRINT is more complex; it has to do a lot of work. First, the computer looks ahead to see whether it will be printing a variable, a number, a string, or maybe even a long calculation. Once that's straightened out and BASIC knows the sequence of characters to be printed, it goes through the Kernal routine for printing characters (always at location $FFD2 on the 128). The Kernal routine looks at locations 806–807 to find the actual ROM routine for printing a character.

This pointer was deliberately designed to be the weak link in the process. If we change the address there, anytime the computer wants to print a character, it runs into a detour we have set up. This detour handles the blink and the click before jumping back to the main PRINT routine.

Program 4-4. Blick

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
BA 100 FORI=3072TO3158:READA:POKEI,A:CK=CK+A:NEXT
XD 110 IFCK<>10998THENPRINT"ERROR IN DATA STATEMEN
       TS.":STOP
FE 120 SYS3072:PRINT"BLICK ENABLED":NEW
SF 130 DATA 169,15,141,24,212,141,19,212,169,120
SM 140 DATA 141,15,212,169,1,141,14,212,169,0
RH 150 DATA 141,20,212,162,34,160,12,142,38,3
QG 160 DATA 140,39,3,96,72,169,0,141,0,255
SD 170 DATA 104,32,121,239,133,167,134,168,132,169
FG 180 DATA 169,33,141,18,212,169,175,32,121,239
PS 190 DATA 162,234,160,0,200,208,253,232,208,250
SS 200 DATA 169,32,141,18,212,169,20,32,121,239
EF 210 DATA 165,167,166,168,164,169,96
```

Word Counter

Thomas K. Tucker

If you ever need a quick word count of a document, this program is for you. It works with text files—program or sequential—created by almost any word processor. A disk drive is required.

Teachers are fond of giving assignments in terms of words: a "3,000-word term paper" or a "500-word essay," for example. I recently wrote such a paper using the word processor *Speed-Script,* but when I finished writing, I had no idea of the number of words. It seemed to me it would be a fairly easy task to write a program to count the words in a file, but first I had to determine what constituted a word.

Spaces separate words from neighboring words, so the number of spaces in a document should equal the number of words. The only snag would be multiple spaces in the file. I didn't want to count *all* the spaces, just the ones immediately preceded by a character that was not a space.

The BASIC program I came up with looked something like this:

```
10 Z=0:A$=" ":B$=" "
20 INPUT"FILENAME";F$
30 OPEN 1,8,0,F$+",P,R"
40 GET#1,A$
50 IF 64 AND ST THEN 90
60 IF A$=CHR$(32) THEN IF B$<>
   CHR$(32) THEN Z=Z+1
70 B$ = A$
80 GOTO 40
90 CLOSE 1
100 PRINT"NUMBER OF WORDS IN
    FILE : ";Z+2
110 END
```

Line 50 checks for the end of the file. Line 60 rules out counting consecutive spaces as more than one word. By experiment, I found that by adding 2 to the counter (Z), a more accurate count is shown. Since printer format codes and carriage returns are counted as words, a 100 percent accurate count is not possible. But it's rarely important that the final

number of words is exact. (Is anyone penalized for being six words short in a 2,000-word paper?)

Speeding It Up

The BASIC program above took over four minutes to count about 2,500 words. Much too slow.

Writing the loop part of the program (lines 40 to 80) in machine language (ML) seemed to be the answer. Since it's a short routine, it fits nicely into the RS-232 input buffer at $0C00 (3072). The ML data is POKEd into the input buffer by using DATA statements.

Later I added the directory routine and the option of counting sequential as well as program files. This program should read files written on any word processor—but remember, the more printer code strings used in the file, the less accurate the word count. In any case the program is pretty fast, taking about 40 seconds to count a 2,500-word, 60-block file.

How to Use It

"Word Counter" is easy to use. Type in and save Program 4-5. The program as listed is for the 128 mode, but it will count words created by a word processor in 64 or 128 mode.

When you've finished writing and saving your document, load Word Counter and type RUN. The first prompt is PRESS D FOR DIRECTORY. Insert the disk containing the text file and press D. You're then asked to type in the filename and type P (Program) or S (Sequential) for file type. Word Counter reads through the file, and seconds later displays the number of words. You're then asked if you'd like to count the words in another file.

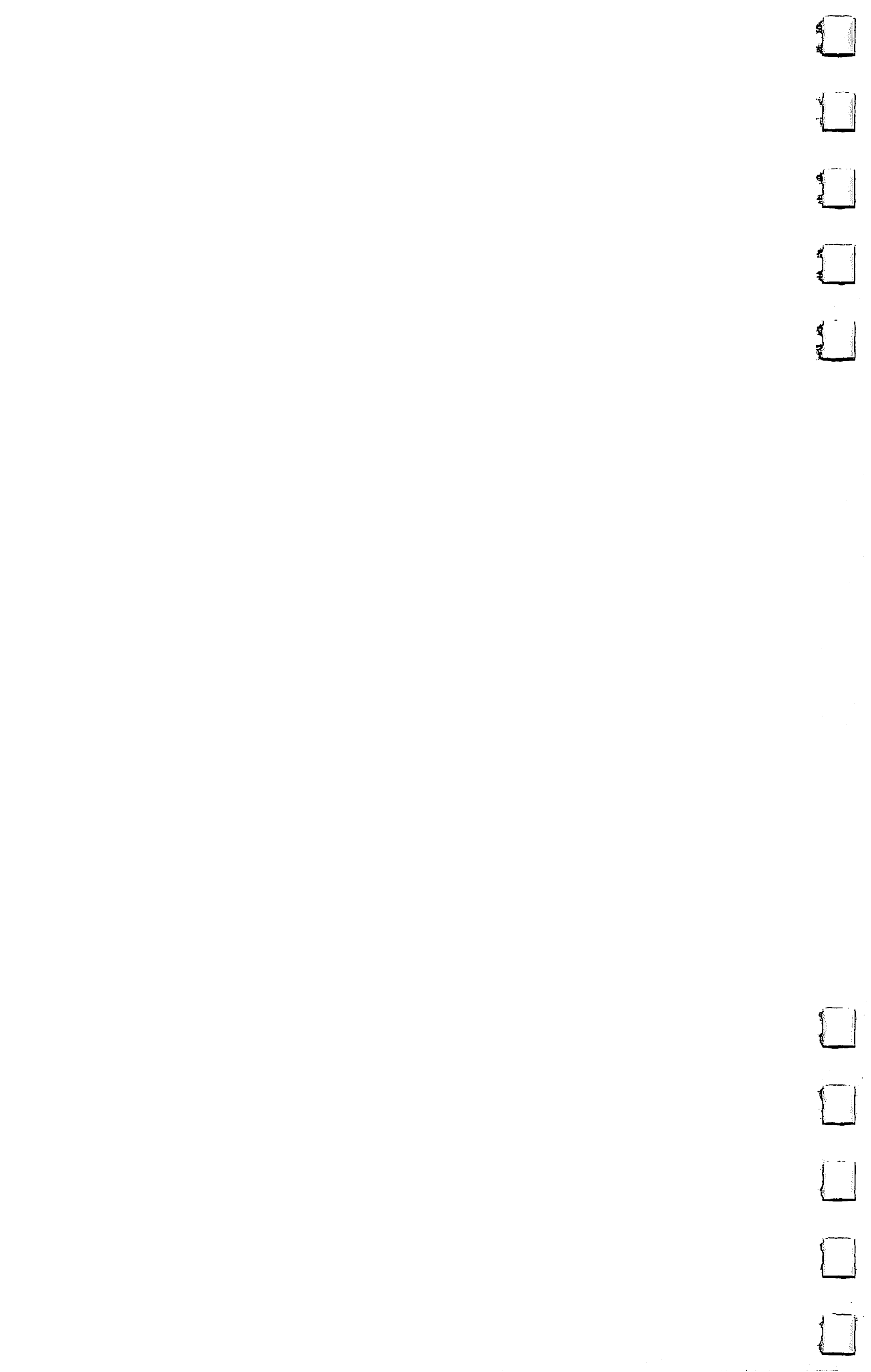
Editor's Note: We tested Word Counter with text files created by SpeedScript, Paperback Writer *(128 and 64 versions), and* Word Writer 128. *The program gave a reasonably accurate count with these files (program or sequential), which were of varying length.*

Program 4-5. Word Counter

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

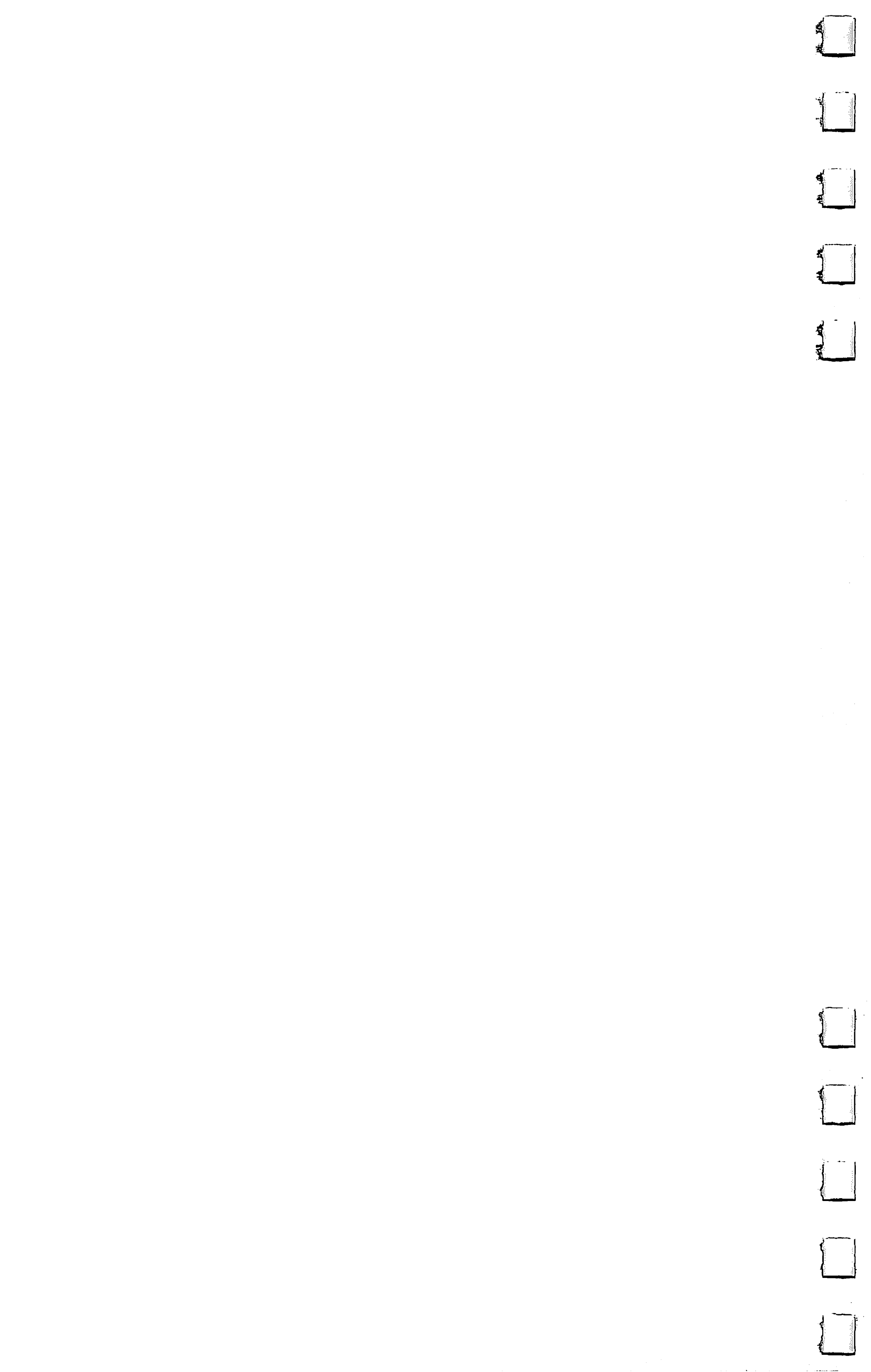
```
PB 10 PRINT"{CLR}"CHR$(142):BS=3072:COLOR0,7:COLOR
      4,7
CP 20 I=I+1:READA:IFA<0THEN50
```

```
QQ 30 POKEBS+1+I,A
AC 40 GOTO20
RA 50 Z=0:D$=""
EF 60 PRINT"{CLR}{2 DOWN}PRESS D FOR DIRECTORY"
KG 70 GETA$:IFA$=""THEN70
FX 80 IFA$<>"D"THEN160
XG 90 OPEN1,8,0,"$0"
ES 100 PRINT:FORA=1TO32:GET#1,C$:NEXT
KR 110 GET#1,B$:IFST<>0THENCLOSE1:SYS65484:GOTO160
BC 120 IFB$<>CHR$(34)THEN110
PC 130 GET#1,B$:IFB$<>CHR$(34)THEND$=D$+B$:GOTO130
MB 140 GET#1,B$:IFB$=CHR$(32)THEN140
AE 150 PRINT" ";B$;"{3 SPACES}";D$:D$="":GOTO110
HD 160 INPUT"{2 DOWN}FILE NAME";F$
AJ 170 IFF$=""THENPRINT"{4 UP}":GOTO160
FE 180 PRINT"{CLR}{2 DOWN}FILE TYPE?"
EM 190 PRINT"{2 DOWN}{RVS}P{OFF}ROGRAM"
GM 200 PRINT"{DOWN}{RVS}S{OFF}EQUENTIAL"
HR 210 GETG$:IFG$<>"P"ANDG$<>"S"THEN210
KX 220 PRINT"{2 DOWN}COUNTING"
BR 230 IFG$="S"THEN250
JK 240 OPEN1,8,0,F$+",P,R":GOTO260
MP 250 OPEN1,8,0,F$+",S,R"
FA 260 SYSBS+2
AK 270 Z=PEEK(BS)+256*PEEK(BS+1)+2
RC 280 PRINT"{CLR}{DOWN}NUMBER OF WORDS:"Z:CLOSE1
DF 290 OPEN15,8,15,"I0":CLOSE15
GP 300 PRINT"{DOWN}ANOTHER FILE?{2 SPACES}(Y/N)
AQ 310 GETA$:IFA$="Y"THEN50
AB 320 IFA$="N"THENEND
CG 330 GOTO310
HH 340 DATA 169,0,141,0,12,141,1,12,141,66,12,141,
       67,12,162,1,32
BK 350 DATA 198,255,32,183,255,41,64,208,34,32,207
       ,255,141,66,12,201,32,208,15
QJ 360 DATA 32,207,255,201,32,240,8,238,0,12,208,3
       ,238,1,12,173,66,12,141
BQ 370 DATA 67,12,76,21,12,32,231,255,96,3,4,-1
```



Chapter 5

Peripherals



Storage and Display

The two most important peripherals you'll use with your 128 are a disk drive and a monitor. Here's some helpful information on the 1571 drive and the video display options that are available. An excerpt from Chapter 4 of COMPUTE!'s 128 Programmer's Guide.

As you probably know, the 1571 drive has two separate operating modes. It can be either a fast, double-sided drive (1571 mode) or a slower, single-sided drive (1541 mode). In ordinary circumstances, DOS automatically uses 1571 mode when the computer is in 128 mode, and 1541 mode when you switch the computer to 64 mode with GO 64. However, it's possible to switch the drive from one mode to another under program control. The following program switches from 1571 mode to 1541 mode:

```
10 OPEN 15,8,15
20 PRINT#15,"U0>M0"
30 CLOSE 15
```

After you run the program, the drive will read and write to the top side of the disk alone, just like a 1541. To return to 1571 mode, replace line 20 as shown here and rerun the program:

```
20 PRINT#15,"U0>M1"
```

You can also switch from one read/write head to another in a program. Switch your drive to 1541 mode, then format a new disk using the disk name SIDE ZERO. Now replace line 20 as shown here and rerun the program:

```
20 PRINT#15,"U0>H1"
```

Now format the disk again using the disk name SIDE ONE, then read the disk directory.

Replace line 20 with this line and rerun the program:

```
20 PRINT#15,"U0>H0"
```

At this point, the disk directory should show SIDE ZERO. You now have a disk formatted on both sides in 1541 mode. Either side may be accessed by changing disk heads as shown above. If you have old single-sided disks around, this is one

way to extend their usefulness. Note, however, that the second side cannot be read in a 1541 drive. It is only readable by a 1571 drive, which has two read/write heads rather than one.

Why the 1571 Is Faster

A 1541 disk drive can read a disk pretty quickly. That is, it can copy data from the surface of the disk to its internal memory fast enough. But it transfers data to the computer at a much slower rate. The problem lies in the 1541's communication protocol, which Commodore has fittingly dubbed "slow mode." There are now an abundance of Commodore 64 programs which increase the speed of the 1541 drive. Such utilities reprogram both the drive and the computer to accelerate the data transfer rate at the risk of less reliable communications. The 1571 offers much faster transfer modes, and we'll show you how to access them under program control. But you first need to understand some simple facts about serial data transfer.

Most Commodore peripherals, including 1541 and 1571 drives, attach to the computer through a serial communication bus. The term *bus* is a jargon name for "group of wires," and *serial* means that only one bit (logical 1 or 0) of data can move along the bus at a time. To send a byte of data over the serial bus, a device first has to break the data down into its eight component bits. This sounds slower than it really is. Some serial communication systems—the Ethernet local-area network, for instance—are very fast. But there's one reason why Commodore's serial bus is so slow. Commodore computers built before the 128 don't use any special-purpose hardware for serial bus communications. Instead, the computer's microprocessor executes a program to convert a byte to bits and send both the data and associated control signals (like "Here's another bit" and "I'm done, now it's your turn to talk") down the bus. Since the microprocessor isn't specifically designed for such operations, it can't do them as fast as a special-purpose serial device.

The 128 and 1571 use a new system called *fast serial mode* to relieve the microprocessor of most serial communication chores. In the original serial protocol, one wire in the serial bus cable was named—Service Request (SRQ)—but never used. Now this wire has been put to work carrying a high-

speed clock signal to accompany data sent at a faster rate. Since the clock signal and the data move faster than the microprocessor can follow, the Complex Interface Adapter chip at location $DC00 (CIA #1) assumes more responsibility for communications. The CIA's Serial Data Register (SDR) at location $DC0C was unused on the 64. But in the 128, it reads and writes to the serial bus's data line. Thus, in the 128's fast serial mode, the most speed-critical tasks are done by hardware that was unused (or underused) on the 64.

Why Learn About Fast Mode?

Some people—Commodore's programmers, for instance—have to know the details of serial communication backward and forward, but you can ignore all the bits and bytes and three-letter words and still benefit from fast serial mode. Every serial input/output routine in the 128's operating system has been written to use fast mode whenever it can. Whenever a serial device is active, the 128 checks to see if it can handle fast mode. Whether you access the device from BASIC or machine language, fast mode will be used if you have a 1571 connected to your 128.

For some disk operations, the 1571 is even faster because of new *burst mode* disk access commands. In conventional disk access, each request for data from the drive returns only one byte. When you're loading a large file, the computer spends most of its time saying "next byte, please" over and over again. Burst commands, on the other hand, tell the disk drive to pass many *blocks* (256-byte packages) of data without any further instruction. These commands can operate on as many as 256 blocks of disk data at a time. There is also a new fast load command that reads a complete disk file in one operation. As CP/M users will be glad to learn, burst mode is also available with any command that relates to MFM-formatted (non-Commodore CP/M) disks.

Since the computer's LOAD routine knows about fast load, ordinary BASIC commands like LOAD, DLOAD, and BLOAD (as well as ML routines which call Kernal LOAD) use burst mode if it's available. Unfortunately, since there is no corresponding fast save command, all save operations transfer data at the normal byte-by-byte rate.

Video Displays

Like virtually every other microcomputer, the Commodore 128 communicates with you, the programmer, chiefly through a *monitor* or some type of display screen. Several options are available to accommodate a wide variety of needs.

The least expensive alternative is to connect the computer to an ordinary TV, using the cable and switch box supplied by the manufacturer. The RF (Radio Frequency) modulated output of the computer contains both audio and video signals, which the TV receives through its antenna input like an ordinary broadcast signal. Though it has the advantage of low cost, a TV hookup rarely provides as clear a display as using a dedicated monitor. The RF cable acts as an antenna of sorts, picking up stray signals from the general vicinity. In the past few years, an increasing number of combination TV/monitor devices have appeared on the market. These are essentially televisions with extra connections for direct input from a home computer or video-recording device. In monitor mode, the device bypasses the TV circuits that receive broadcast signals, usually providing a display equal to that of a dedicated monitor.

If you have owned or used another Commodore computer before purchasing your 128, chances are good that you're familiar with the most popular type of dedicated monitor—the *composite* monitor. This type includes the Commodore 1701 and 1702 monitors (and many similar non-Commodore devices), which display an excellent image in 40-column mode. These monitors are connected to the computer through the eight-pin video connector at the computer's rear. Commodore composite monitors usually produce the best picture through the rear connectors, which split the color portion of the signal into separate chroma (color) and luma (brightness) signals, rather than those in the front of the device.

A monochrome monitor, though it provides no color (and often no sound), offers another inexpensive alternative. For a 40-column display, simply connect the luma output plug of the video output cable to the monitor's input (luma is essentially the video signal stripped of its color information). A diagram of the pins in the video output appears in your *128 System Guide*. If you're not sure which plug is luma, go ahead and experiment. You can't harm either device by momentarily connecting the wrong plug to the monitor. Monochrome displays are usually very sharp—more distinct than the best dis-

play produced by a composite monitor.

The third major type of monitor is the RGBI (Red/Green/Blue/Intensity), in which a separate signal is provided for each of the three primary video colors—red, green, and blue. The 128's 80-column display is in RGBI format. Since the nine-pin RGBI connector at the rear of the 128 is much like that on an IBM PC or PCjr, you should be able to use any color monitor compatible with those machines. Besides providing an extremely clear image, an RGBI monitor gives you 80 columns of characters on the screen—ideal for word processing, spreadsheets, and so on. The Commodore 1902 monitor, designed expressly for the 128, is a dual monitor: You can switch it from 40-column composite mode to 80-column RGBI mode simply by pressing a switch.

Though it requires making your own connector cable, you can get an acceptable 80-column monochrome display on a Commodore 1701 or 1702 monitor. The first step is to purchase a standard male nine-pin D connector (Radio Shack part #276-1537 is acceptable), a length of shielded coaxial cable, and an ordinary RCA phono plug. As shown on page 352 of the *Commodore 128 System Guide,* pin 1 of the RGBI connector is ground, and pin 7 is monochrome output. (Note that the diagram on page 352 shows the pins as if you are *inside the computer looking out.* The pins of the plug you buy should be numbered; just look for pins 1 and 7.) To make an 80-column cable, you need only connect pin 7 of the D connector to the signal (inner) portion of the RCA plug and connect pin 1 to the ground (outer) portion of the plug (via the coaxial cable, of course). If you don't know how to do this yourself, any friend with a soldering iron and some electronics experience should be able to do it for you.

A homebrew cable of this type produces an excellent 80-column image on any monochrome monitor that accepts composite output, and an acceptable display on a Commodore 1701 or 1702 monitor. To use it with a Commodore composite monitor, turn the contrast down and plug the RCA connector end of the cable into the VIDEO connector on the front or the LUMA connector on the back. The ordinary light-on-dark display will probably not be very readable: Press ESC-R to switch to dark characters on a light background. Though it's not quite RGBI quality, the image is definitely usable and gives you access to 80 columns at a cost of only a few dollars.

Disk Commands

Todd Heimarck

Whether you have a 1541 disk drive or a new 1571, there are a number of powerful disk commands available to you; also included here are some useful hints and shortcuts.

BASIC 7.0 is a vast improvement over previous Commodore BASICs. The computer has its share of flashy new commands, the ones that give you POKEless sprites, easy-to-program music and sound effects, and hi-res graphics. The glamour of these powerful keywords can easily bewitch a new 128 owner.

Disk commands, on the other hand, are just disk commands. They're mundane. But if you learn about the new ways of loading, saving, and handling files, you'll save a lot of time, time that could be spent programming—or playing with sprites, music, and hi-res graphics.

We'll concentrate on using the 128 disk commands, most of which work equally well on the 1541 disk drive or the new 1571. But we'll also touch briefly on a few of the new 1571 DOS commands.

A Dozen Ways to Load

If you want to load a BASIC program, you have four choices:

1. LOAD *"filename"*,8
2. DLOAD *"filename"*
3. RUN *"filename"*
4. Press SHIFT–RUN/STOP

For machine language or binary files:

5. LOAD *"filename"*,8,1
6. BLOAD *"filename"*
7. BLOAD *"filename"*, B*bank*, P*address*
8. BOOT *"filename"*

From within the machine language monitor:

9. L *"filename"*,8
10. L *"filename"*,8,*address*

Finally, there are two ways to start up autoboot programs:

11. BOOT
12. Turn on or reset the computer with an autoboot disk in the drive.

Loading BASIC Programs

As in BASIC 2.0, the LOAD command defaults to tape, so you must include the device number when loading from disk. But LOAD should never be necessary when DLOAD and RUN are available.

DLOAD is a new command; the D stands for "Disk," and it defaults to drive 0, device 8. If you own a dual drive, you can add a comma and either D0 or D1 to pick a drive for loading. Unfortunately, 128 owners may never see the 1572 dual drive; as of this writing, Commodore has apparently decided not to manufacture it. You can still add single drives to your system, though. To access a second or third drive, follow DLOAD with a comma and U9, U10, and so on. The current device number of the 1571 can be selected by flipping switches on the back. To change to device 9, for example, make sure it's turned off and flip down the switch nearest to the cords. This is much simpler than what's required to modify a 1541, opening it up and cutting a solder trace.

The next command on the list, RUN, has been modified. By itself, it still runs a program, but if you add a program name, the program loads and runs. As with DLOAD and most other disk commands, you can specify a drive number with D or a device number with U after the program name.

In 64 mode, pressing SHIFT–RUN/STOP still loads and runs the first program from tape. But in 128 mode, this combination loads and runs the first program on disk.

BLOADing Binary Files

A binary file is most often a machine language program, although there are several other possibilities: sprite shapes, redefined characters, function key definitions, hi-res pictures, to name just a few. With binary files it's usually important that they load into a specific area of memory.

If you're familiar with the VIC or 64, you'll recognize LOAD *"filename"*,8,1. It loads a file back into the part of memory from which it was saved.

BLOAD does the same thing, but you don't have to include the 8 and the 1. BLOAD can also send a file to a different section of memory if you append a B (for Bank number) and a P (Position). With an unexpanded 128, the only two choices for the bank are 0 and 1. BASIC programs are stored in bank 0, variables in bank 1. The position can be any memory location in the range 0–65535.

BOOT *"filename"* loads a machine language program and executes a SYS to the starting address. It's the machine language equivalent of RUN *"filename"* for BASIC programs.

You can also load from within the machine language monitor with the L command. After the filename, you must include a comma and an 8 (for device 8, the disk drive). If you wish to relocate the program to a different section of memory, you can include the new address as well.

Autoboot Sectors

When you first turn on a 128, it checks to see whether a disk drive is attached and turned on. If so, it tries to read track 1, sector 0 into memory (the 256 bytes of the boot sector are read into locations $0B00–$0BFF). If the letters *CBM* are found at the beginning of that disk sector, the autoboot sequence begins.

You can see how this works by following this power-on sequence:

1. Turn on your TV/monitor and disk drive, but not the 128.
2. Insert the CP/M disk that came with the 128 into the 1541 or 1571.
3. Turn on the 128.

The CP/M disk has an autoboot sector; it's designed to load and run CP/M automatically. An alternative to resetting the computer is to enter BOOT without a filename.

Autoboot sectors aren't limited to CP/M. It's possible to create disks that automatically load and run a BASIC or an ML program. To create such a disk, load and run the AUTOBOOT MAKER program on the disk that comes with the 1571.

The first three bytes of track 1, sector 0 (the characters *C*, *B*, and *M*) are followed by the low byte of the load address, the high byte, the bank number for the load, and the number of sequential disk sectors to be loaded. These four bytes aren't important when you're autobooting BASIC programs, so they

should usually be zeros. Starting at the eighth byte, you put the disk name (for the BOOTING message), and end with a zero. Next is the name of the program you wanted to load, again terminated by a zero. Finally, there's machine language which will be called after the load.

Chained Programs

Commodore computers have always had problems with chaining, the process of loading and running one program from within another. The difficulties stemmed from the way variables were stored in memory in previous Commodores: The beginning of variable storage immediately followed the end of the BASIC program.

Chaining is a snap on the 128. Since the program is kept separate from variables, you don't need to worry about program length. To load and run another program, just follow these rules:

1. If you want to keep the variables from the first program, use DLOAD. The second program loads and runs. All variable values are retained.
2. If you want to clear the variables, use RUN *"filename"*, where *filename* is the name of the second program.
3. To load a binary file, use either BLOAD *"filename"* or BOOT *"filename"*.

A Shortcut

There's a quick and convenient way to DLOAD or RUN a program if you save it a certain way. Include this line at the beginning of the program you're working on:

```
1 REM      DSAVE "01PROGRAM-
  NAME {SHIFT-SPACE}:
```

The {SHIFT–SPACE} means hold down SHIFT and press the space bar. Play with the spacing of the line so that pressing TAB once puts the cursor in front of DSAVE and pressing it twice lands the cursor on the 1 in front of the program name. When you want to do a safety save of an incomplete program, LIST 1 and TAB twice. Change version number 01 to 02, and press RETURN. Now cursor up to the beginning of the line and TAB once. Tap the ESC key (next to TAB) and then press P. This erases everything from the cursor to the be-

ginning of the line (ESC-Q erases everything to the end of a line, and you can remember these two ESC commands if you mind your p's and q's). Press RETURN, and your program is saved to disk with the new version number.

Later, when you come back to work on the program, press F3 to see the directory (if it goes by too fast, the Commodore key slows it down; the NO SCROLL key temporarily pauses it). When you see the latest version, press RUN/STOP. Cursor up to the program name and type DLOAD or RUN. Better yet, press F2 (DLOAD) or F6 (RUN). The SHIFT–SPACE in line 1 puts a quotation mark between the program name and the colon. Without the colon, DLOAD or RUN would interpret PRG as part of the command.

Another advantage to including the DSAVE on line 1 is that when you send a program listing to your printer, the version number is right there at the top of the page.

Saving

Here are a few ways to save programs:

1. SAVE *"filename"*,8
2. DSAVE *"filename"*
3. BSAVE *"filename"*, B*bank*, P*start* TO P*end*
4. From the ML monitor: S *"filename"*,8, *start*, *end*+1

The first two, SAVE and DSAVE, are just ordinary ways to save ordinary BASIC programs. BSAVE and the monitor save are a little more interesting. They save a section of memory as a binary file. Note that when you're in the monitor, you have to add 1 to the ending address of the memory being saved.

You might think these two methods would be most useful for saving ML programs. They are good for that, of course, but there are also several areas of memory you may want to BSAVE for use in a BASIC program:

$0E00–$0FFF	Sprite definitions
$1000–$10FF	Ten function key definitions
$1C00–$3FFF	Hi-res screen

The addresses are listed in hexadecimal. To convert to decimal, use the DEC function, PRINT DEC("0E00"), for example.

If you create several sprites with SPRDEF for a game, you can BSAVE the sprite area to disk. In the game, you would

then BLOAD them back into memory. This works a lot faster than POKEing them into memory or reading a sequential file, especially if you're using a 1571.

In case you're wondering about the reference above to ten function keys, yes, there are ten redefinable keys. There are the eight you can define with the KEY command (labeled F1–F8), but also SHIFT-RUN/STOP and HELP. If you go into the monitor and do a memory display of 1000–10FF, you can see the ten key definitions. The first ten bytes in this area list the length of each function key. The rest are the actual characters that print when you press one of them. The number 13 is ASCII for a carriage return, the equivalent of pressing the RETURN key. After redefining the keys, you can BSAVE their new values. To retrieve the previous key definitions, use BLOAD.

Handling Sequential Files

DOPEN and DCLOSE are new ways of establishing and breaking connections with a sequential file. There's not much to say about them; if you already know how to OPEN and CLOSE sequential files, you'll catch on quickly. The difference in syntax is illustrated below:

OPEN 3,8,4,"*filename*,S,W"
DOPEN#3,"*filename*",W

Note that DOPEN doesn't need as much information as OPEN. OPEN is a general-purpose command; it can set up a logical file to a disk file, a tape file, a printer, a modem, and so on. DOPEN, on the other hand, is for disk files only. So OPEN needs the device number and disk channel (,8,4), but DOPEN doesn't. The *S* after the filename indicates that a sequential file will be opened. Since DOPEN defaults to sequential files, it too is unnecessary. Also, note that the *W* for Write is outside the quotation marks in the second example.

APPEND is a variation on DOPEN. It opens an already existing disk file for a write operation. Any information written to the sequential file is added to the end. Data at the beginning of the file is safe and unchanged.

There aren't any new ways of reading or writing files. You still PRINT# to send data and either INPUT# or GET# to read a file.

Relative Files Are Much Easier

Being able to randomly access records in a file can sometimes greatly speed up a program. With sequential files, you may sometimes have to read through 50 records just to get to the fifty-first. A relative file allows you to obtain the information you need almost immediately.

In BASIC 2.0, creating and maintaining a relative file requires sending a number of CHR$ codes. If you write programs for relative files in 64 mode, you'll have to learn the complexities of relative files. But not on the 128. In just a few lines, you can open and write to a relative file. Let's say you wanted 100 records with 20 characters in each record. Your program to set up a file would look something like this:

```
10 DOPEN#3,"XYZFILE",L20
20 RECORD#3,100
30 PRINT#3,"LAST RECORD"
40 RECORD#3,100
50 CLOSE3
```

That's all there is to it. When DOPEN is followed by an *L* and a number, it opens a relative file. The length of each record is set by L20. Records can be from 1 to 254 bytes long. Because the record length is stored in the directory, you need to use the L parameter only when the file is first created.

RECORD# positions the pointer to the desired record (up to 65535 can be accessed, depending on the record length). You must include the logical file number and the record number. A third number can be added if you want to start reading or writing partway into the record. If this number is omitted, you'll begin at the first byte of the record.

In line 30, we PRINT# to record number 100. Printing to a previously nonexistent record forces the disk drive to create that record and all previous records. Line 40 positions the pointer again to avoid a rare bug that sometimes corrupts files, and then file 3 is closed.

Once the file is created, you can easily access records with DOPEN and RECORD#. You PRINT# to write and either GET# or INPUT# to read records.

Utilities

There are more new commands that help when you're programming. The F3 key is defined to print DIRECTORY. So,

with the press of a single key, you can see what's on a disk.

Two very useful reserved variables are DS (short for Disk Status) and DS$. The first returns the disk error number; the second prints out the error message. If the red light on a 1541 starts flashing (the green light on a 1571), just enter PRINT DS$ and you can see what went wrong. Consult your disk drive manual for a complete list of error messages.

Within a program, DS is usually more helpful than DS$. After a disk operation, add a line **IF DS>19 THEN 500**, where line 500 is the beginning of an error-handling routine.

The variable DS will normally hold a zero if no errors occurred. But if DS is equal to 20 or more, something has gone wrong. There are a few exceptions, though: Error 01 is not an error; it's triggered after a SCRATCH operation. The error message will be FILES SCRATCHED, followed by a comma and the number of files that were scratched. Error 50 (RECORD NOT PRESENT) is no matter for concern if you've just created or expanded a relative file. If you write to a previously nonexistent record in a relative file, it's added to the disk. The record was not present before the operation and thus causes the error 50. Finally, when you first turn on or reset a disk drive, you'll receive an error 73, which is simply an announcement of which version of DOS is inside the drive.

Several other new commands make file management easier. RENAME and SCRATCH are fairly straightforward. SCRATCH is followed by a filename inside quotation marks. Pattern matching, using wildcards like question marks or asterisks, is available for those times when you want to scratch several files with similar names. To change the name of a file, RENAME *"oldname"* TO *"newname"*. This syntax is certainly easier to remember than OPEN 15,8,15, "R0:*newname=oldname*", the required syntax on the VIC or 64.

COLLECT validates the disk. It's used mostly for cleaning up the block allocation map (BAM) to get rid of improperly closed files. These were formerly called "poison files," but the 1571 disk drive manual refers to them as "splat files." They're marked by an asterisk in the directory, *PRG, for example.

DCLEAR initializes the disk; it's the same as sending "I0" to channel 15.

CONCAT combines the contents of two sequential files. You can use it on program files, but the result won't be a

merged program because the two zeros that mark the end of a BASIC program get in the way.

Two disk commands designed primarily for dual drives are COPY and BACKUP. The first copies a file from one drive to another. But you must use a dual drive—COPY won't work with two single drives. It can also make a copy to the original disk (if you want to rearrange a directory, for example). BACKUP copies a whole disk. It too requires a dual drive.

A Few Quirks

There are a few annoying features of the 128—not bugs, just bothersome quirks.

The most serious of these is that SHIFT–RUN/STOP loads and runs the first file on disk. A nice feature if that's what you want, but sooner or later, while programming, you'll accidentally press SHIFT–LOCK and RUN/STOP or the Commodore key and RUN/STOP. When the disk drive starts spinning, you have only a few seconds to unlock the SHIFT–LOCK key and press RUN/STOP to prevent the first program from loading. If you fail to stop it, the program loads and runs and you've lost any part of your other program that was not saved. To avoid this situation, you may want to put a sequential file at the beginning of a disk. If you accidentally type SHIFT–RUN/STOP, the computer will try to load the sequential file, but it won't work. The program you're working on will be safe if this precaution is taken.

You can also accidentally save a program. The default values for F7 and F3 are LIST and DIRECTORY. Very helpful when you want to take a look at what's on a disk or what's in a program. But in between these two keys is F5, which is defined as DSAVE. If you reach up to list a program and accidentally press F5 and F7, the computer will print DSAVE"LIST and begin saving your program under that name.

VERIFY and DVERIFY don't always work as you would expect. Each line of a BASIC program contains a memory pointer to the beginning of the next line. When you allocate a graphics area, the BASIC program is moved up by 9K, and all the line links change. Line links that don't match will lead to a false VERIFY ERROR. You can test this by entering a one-line program and saving it to disk. Now type **GRAPHIC1: GRAPHIC0** to allocate a graphics area. List the program and

use DVERIFY to check your save. You should see an error message.

Something to remember when you're using disk commands is that variables must be enclosed in parentheses. The following two examples show the right and wrong ways to use variables:

Wrong **RENAME H$ TO "FINALFINAL"**
Correct **RENAME (H$) TO "MOSTRECENT"**

How Fast Is the 1571?

If you already own a 1541 drive, you can use it with a 128. You don't need to buy a 1571, unless speed is important. Here's how the two drives compare:

	1541	**1571**
9K LOAD (hi-res screen)	27 seconds	4 seconds
Disk format	89 seconds (one side)	43 seconds (two sides)
Quick format (no ID)	2.5 seconds (one side)	3.4 seconds (two sides)

Going into 80 columns and using the FAST command to double the speed of the microprocessor saves only a few tenths of a second on disk operations. So the speed of the computer is not a factor. The bottleneck is the speed at which the data travels through the serial cable.

Note that formatting, which is handled completely within the disk drive, is twice as fast for twice the disk capacity. This suggests that writing operations are quicker on the 1571.

Even when you send the command to make the 1571 act like a 1541, it's faster. A 1541 takes nearly a minute and a half to format a disk. The 1571 in 1541 mode takes only one minute and 12 seconds.

The "act like a 1541" command is **OPEN 15,8,15: PRINT#15, "U0>M0"**. To reset to 1571 mode, **PRINT#15, "U0>M1"** (these commands can be used in 64 mode as well).

While the 1571 is emulating the 1541, you can choose which read/write head is used with **PRINT#15, "U0>H0"** or **PRINT#15, "U0>H1"**. By switching heads, you can format both sides of a disk as if they were separate disks.

Another plus for the 1571 is its ability to read a variety of CP/M formats. If you plan to do much with CP/M, the 1571 provides more flexibility. Even if you don't, it's faster and can handle more data than the 1541.

Cataloger

Kevin Mykytyn

Organize your disk library by making it easy to find any program on any disk. Included are options to print out a master directory and alphabetically sort all filenames.

After owning a computer and disk drive for a while, it doesn't take long before you find yourself inundated by programs and disks. No matter how well your disks are organized, you may still find yourself loading several directories searching for that one program. "Cataloger" offers a practical solution. It's a straightforward, menu-based program that creates a master directory for all your disk-based programs. Several extra features make it especially useful.

Managing Hundreds of Files

Cataloger is written entirely in BASIC. As listed, it can handle up to 2000 filenames in 128 mode. In 64 mode, the program can handle only 800 filenames—if you want to use the program in 64 mode, you'll need to change the value of NR in line 10 from 2000 to 800.

After you've typed in the program using "The Automatic Proofreader," save a copy. To use it, type RUN. A menu of nine choices is displayed:

1. CATALOG A SET OF DISKS
2. SEE ALL FILE NAMES
3. PRINT ALL FILE NAMES
4. SORT NAMES ALPHABETICALLY
5. CREATE A SEQUENTIAL FILE
6. CREATE A PROGRAM FILE
7. LOAD AN OLD FILE
8. DELETE A DISK
9. START NEW FILE

If you're using the program for the first time, choose option 1, Catalog a Set of Disks. You'll then be asked for a disk name up to 16 characters long. This should be the name on the label of the disk you wish to catalog. Put the disk in the drive and press RETURN. The directory of the disk is dis-

played on the screen and the filenames are entered into the master directory. Next, you're asked whether you want to continue or quit. If you have more disks to catalog, press any key. Otherwise, type Q to go back to the main menu.

After entering all the disks you wish to catalog, you can view the master directory. Option 2, See All File Names, displays the master directory on the screen. You can also print it out on any Commodore printer by using option 3. Make sure the printer is turned on before you choose this option.

The filenames are stored in a format slightly different from a standard disk directory. The filename is followed by the number of blocks used, then the disk name.

Once you've viewed the directory, use option 5, Create a Sequential File, to save it. Choose this option if you wish to make changes later to the master directory.

Option 6 also saves your master directory to disk, but instead of creating a sequential file, it creates a program beginning at line 100. Whenever you wish to view the master directory, you can load the program created.

Adding, Deleting, and Sorting

At some point, you'll want to add or delete disks from your master directory. Options 7 and 8 are used for this purpose. Option 7 is used to load any directory stored in sequential file format (with option 5). After loading a file, you can choose option 1, and all new filenames will be appended to the old directory. Make sure you save your changes to disk when you're through.

To delete a disk from the master directory, choose option 8. In order for this option to work correctly, the directory must be *unsorted*. If you wish to have a sorted master directory on disk, you should first save it unsorted.

The delete option removes all filenames from the disk name specified. Therefore, it's important that all disks have a unique name. If you've made changes to a disk and wish to enter the changes in the master directory, use option 8 to delete that disk; then use option 1 to enter the newer disk version.

You'll find it easier to locate a specific file if the master directory has been sorted. Once the file has been saved in sequential file format, use option 4 to sort the names.

To delete the directory in memory and start a new directory, use option 9.

Program 5-1. Cataloger

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
GF 10 NR=2000:DIMB$(NR):IFPEEK(794)<>74THEN30
FR 20 POKE53281,15:POKE53280,15:KB=198:RO=214:GOTO
      60
EJ 30 IFPEEK(794)<>189THEN50
KH 40 COLOR 0,16:COLOR 4,16:KB=208:RO=235:PRINTCHR
      $(27)"M":GOTO60
BA 50 RO=205:KB=239:POKE65301,241:POKE65305,241:PO
      KE2025,255
RE 60 S$="{20 SPACES}":C=0:OPEN15,8,15:D$="
      {9 DOWN}"
GJ 70 PRINT"{CLR}{RVS}{RED}{6 SPACES}CHOOSE ONE OF
       THE FOLLOWING{7 SPACES}{BLU}{OFF}"
AG 80 PRINTSPC(5)"{2 DOWN}1. CATALOG A SET OF DISK
      S"
BK 90 PRINTSPC(5)"{DOWN}2. SEE ALL FILE NAMES"
JJ 100 PRINTSPC(5)"{DOWN}3. PRINT ALL FILE NAMES"
SP 110 PRINTSPC(5)"{DOWN}4. SORT NAMES ALPHABETICA
       LLY"
KD 120 PRINTSPC(5)"{DOWN}5. CREATE A SEQUENTIAL FI
       LE"
AR 130 PRINTSPC(5)"{DOWN}6. CREATE A PROGRAM FILE"
SX 140 PRINTSPC(5)"{DOWN}7. LOAD AN OLD FILE"
SM 150 PRINTSPC(5)"{DOWN}8. DELETE A DISK"
QH 160 PRINTSPC(5)"{DOWN}9. START NEW FILE"
MP 170 PRINTSPC(12)"{RED}{DOWN}{RVS} PRESS Q TO QU
       IT{BLU}{OFF}"
DD 180 POKEKB,0:WAITKB,1:GETA$:IFA$="Q"THENCLOSE15
       :PRINT"{CLR}":END
SP 190 IFA$<"1"ORA$>"9"THEN180
FS 200 ONVAL(A$)GOSUB210,510,520,640,730,740,850,9
       40,1040:GOTO70
JK 210 F$="DISK":GOSUB1050:IFDN$="Q"THENRETURN
QR 220 PRINT"{CLR}{RVS}{RED}"SPC(20-LEN(DN$)/2)DN$
       "{2 DOWN}{BLU}"
CP 230 CLOSE2:OPEN2,8,0,"$":GOSUB390:IFA$="Q"THENR
       ETURN
KG 240 X=4:NF=0:IFEXTHEN220
QK 250 B$(C)="":FL=-1:GOSUB310:X=2:GET#2,LN$
BJ 260 GET#2,HN$:NM=ASC(LN$+CHR$(0))+256*ASC(HN$+C
       HR$(0)):PRINTSPC(5)NM;
PF 270 GET#2,A$:IFA$=""THEN320
XD 280 IFA$=CHR$(34)THENFL=-FL
JJ 290 IFFL=1THENB$(C)=B$(C)+A$
RC 300 PRINTA$;:GOTO270
```

```
ER 310 FORA=1TOX:GET#2,A$:NEXT:RETURN
EQ 320 IFB$(C)=""THENCLOSE2:GOSUB480:IFA$<>"Q"THEN
       210
AC 330 IFB$(C)=""THENRETURN
XQ 340 PRINT:NM$=STR$(NM)
FA 350 B$(C)=RIGHT$(B$(C),LEN(B$(C))-1)
SC 360 B$(C)=B$(C)+LEFT$(S$,19-LEN(B$(C))-LEN(NM$)
       )+NM$+"{2 SPACES}"+DN$
AR 370 IFNF=1THENC=C+1:IFC=NR THEN1080
BS 380 NF=1:GOTO250
DQ 390 RF=0:EX=0:A$="":INPUT#15,EN,M$,T,S
RC 400 IFEN<20THENRETURN
KQ 410 PRINT"{HOME}"D$:FORA=1TO5:PRINT"{35 SPACES}
       ":NEXT
HB 420 EX=1:PRINT:PRINT"{HOME}{BLU}"D$EN;M$;T;S:IF
       EN<>63THEN470
RD 430 RF=1:PRINT"{DOWN} DO YOU WANT TO REPLACE TH
       E FILE? (Y/N)"
AB 440 GETA$:IFA$="N"THENRETURN
CM 450 IFA$<>"Y"THEN440
JA 460 EX=0:PRINT#15,"S"+DN$:RETURN
EK 470 CLOSE2
RS 480 PRINT:PRINT"{DOWN}{BLU}{2 SPACES}PRESS ANY
       {SPACE}KEY TO CONTINUE {RVS}Q TO QUIT{OFF}"
       :POKEKB,0:WAITKB,1
PR 490 GETA$:RETURN
GP 500 PRINT:PRINT"{DOWN}{BLU}{6 SPACES}PRESS ANY
       {SPACE}KEY TO CONTINUE":POKEKB,0:WAITKB,1:R
       ETURN
BG 510 DV=3:SA=0:SP=1:DN$="":NM=0:GOTO550
ER 520 PRINT"{CLR}"D$"PRESS [M] TO RETURN TO THE M
       ENU ANY{5 SPACES}OTHER KEY TO PRINT"
BF 530 POKEKB,0:WAITKB,1:GETA$:IFA$="M"THEN70
JP 540 DV=4:SP=20:SA=0:DN$="":NM=0
MB 550 IFC<=0THENPRINT"{CLR}"D$SPC(10)"{BLU}NO FIL
       ES IN MEMORY":GOTO500
AX 560 IFNMTHENGOSUB1050:DN$="0:"+DN$+",S,W"
QR 570 CLOSE2:OPEN2,DV,SA,DN$:GOSUB390:IFA$="Q"THE
       NRETURN
MJ 580 IFRFANDEX=0THEN570
HX 590 IFRFTHEN560
QD 600 IFEXTHEN570
SP 610 PRINT"{CLR}{DOWN}":FORA=0TOC:PRINT#2,SPC(SP
       )B$(A)
FJ 620 IFPEEK(RO)=21THENGOSUB500:PRINT"{CLR}
       {3 DOWN}"
SS 630 NEXT:CLOSE2:GOSUB500:RETURN
GX 640 IFC<=0THENPRINT"{CLR}"D$SPC(10)"{BLU}NO FIL
       ES IN MEMORY":GOTO500
```

```
MG 650 D=C-1:M=D:PRINT"{CLR}"D$SPC(15)"SORTING....
       "
AB 660 M=INT(M/2):IFM=0THENRETURN
EJ 670 J=0:K=D-M
JS 680 I=J
RK 690 L=I+M
MR 700 IFB$(I)>B$(L)THENT$=B$(I):B$(I)=B$(L):B$(L)
       =T$:I=I-M:IFI>0THEN690
CQ 710 J=J+1:IFJ>KTHEN660
PF 720 GOTO680
PS 730 DV=8:SP=5:SA=2:F$="FILE":NM=1:DN$=DN$+",S,W
       ":GOTO550
SX 740 GOSUB1050:DN$="0:"+DN$:IFDN$="0:Q"THENRETUR
       N
BS 750 CLOSE2:OPEN2,8,2,DN$+",P,W":GOSUB390:IFA$="
       Q"THENRETURN
KF 760 IFRFANDEX=0THEN750
GJ 770 IFRFTHEN740
KX 780 IFEXTHEN750
MK 790 PRINT#2,CHR$(1)CHR$(8);:FORA=0TOC-1
FG 800 PRINT#2,CHR$(4)CHR$(4);:LN=100+A
BE 810 HB=INT(LN/256):LB=LN-HB*256:PRINT#2,CHR$(LB
       )CHR$(HB);
AJ 820 PRINT#2,CHR$(34)B$(A)CHR$(34)CHR$(0);
QG 830 NEXTA:PRINT#2,CHR$(0)CHR$(0);
RJ 840 CLOSE2:RETURN
QH 850 F$="FILE":GOSUB1050:IFDN$="Q"THENRETURN
EC 860 OPEN2,8,2,"0:"+DN$+",S,R":GOSUB390:IFEXTHEN
       850
XE 870 FL=-1:B$(C)="":FORA=1TO5:GET#2,A$:NEXT
SP 880 GET#2,A$:IFA$=CHR$(13)THEN910
BG 890 B$(C)=B$(C)+A$
QB 900 GOTO880
GP 910 IFSTTHENCLOSE2:RETURN
GE 920 C=C+1:IFC=NR THEN1080
KD 930 GOTO870
QM 940 F$="DISK":GOSUB1050:IFDN$="Q"THENRETURN
DP 950 PRINT"{CLR}"D$SPC(14)"DELETING":EN=0
KD 960 FL=0:FORA=0TOC-1
XH 970 IFMID$(B$(A),22,16)=DN$ANDFL=0THENFL=1:SN=A
RE 980 IFMID$(B$(A),22,16)<>DN$ANDFL=1THENEN=A:A=C
BP 990 NEXT:IFFL=0THENPRINT"{DOWN}"SPC(12)"FILE NO
       T FOUND":GOTO1030
PE 1000 IFEN=0THENEN=C+1:GOTO1020
GS 1010 FORA=SNTOC-(EN-SN):B$(A)=B$(EN+A-SN):NEXT
QG 1020 C=C-(EN-SN):PRINT"{DOWN}"SPC(14)"DELETED"
MH 1030 GOSUB500:C=C-(C<0):RETURN
FF 1040 C=0:RETURN
BK 1050 PRINT"{CLR}{4 DOWN} PUT DISK IN DRIVE AND
        {SPACE}ENTER "F$" NAME"
```

```
XG 1060 PRINT" (UP TO 16 CHARACTERS) {RVS}Q TO QUI
        T{OFF}"
RH 1070 POKE19,1:INPUT"{DOWN}{RED} ";DN$:POKE19,0:
        DN$=LEFT$(DN$,16):RETURN
HG 1080 PRINT"{CLR}"D$"{RED}THE MASTER DIRECTORY I
        S FULL.{2 SPACES}SAVE THE PRESENT FILE AND
         ";
QR 1090 PRINT"START A NEW DIRECTORY.{BLU}":GOSUB50
        0:RETURN
```

Autoboot

Steve Stiglich

These four programs allow you to create boot disks that automatically load and run a program when your 128 is turned on. You don't even have to type LOAD—it's all done for you. The "autorun" disks can boot programs for either 128 mode or 64 mode from the 1541 or 1571 drive.

For many Commodore users, the idea of a *boot disk* is brand-new. But if you've done much work with an Apple, IBM, or Atari, you know how convenient it is to insert a disk, turn on the system, and see a program automatically load and run. When a Commodore 128 is turned on (or when it's reset with SYS 65341), it checks for the presence of a disk drive. If the drive is turned on and contains a disk, the 128 checks track 1, sector 0, for an *autoboot sector*. Upon finding one, it follows the instructions there for loading and running a program.

The programs presented here allow you to write an autoboot sector to any disk. In addition, you can create a disk that makes the 128 go into 64 mode and automatically load and run your favorite program for the 64. You could make a *SpeedScript* boot disk, for example. All you'd have to do is turn on the drive, insert the disk, and power on the 128.

All four programs are written in BASIC, although some contain short machine language routines inside DATA statements. Program 5-2 writes an autoboot sector to a disk. The boot sector attempts to load and run a 128 program called "HI", so you should have a file by that name on the same disk. Program 5-3 is a menu program that gives you five choices. If you want this to be your boot program, save it to disk under the name "HI". Program 5-4 creates a machine language autorun 64 program file that causes another 64 program to load and run automatically after going into 64 mode. Program 5-5 uses the dynamic keyboard technique to load and run a BASIC or machine language program for the 64.

Special Typing Instructions

Programs 5-2, 5-3, and 5-4 should be typed in and saved from 128 mode. *You must not have a graphics area allocated when these programs are saved.* If you've been working with hi-res graphics, enter GRAPHIC CLR before saving the programs. The name given to Program 5-3 should be "HI" if you want the menu to come up automatically when you turn on the system (if you plan to boot any other 128 program, you can skip Program 5-3). It's of no importance which names you use for Program 5-2 or Program 5-4.

Program 5-5 is written for 64 mode, so you should enter GO 64 before beginning to type it in. *This program must be saved from 64 mode.* Be very careful with lines 10 and 20; type them exactly as listed. Save this program (from 64 mode) under the name "64LOADER". It should be on the same disk as Programs 5-3 and 5-4.

Creating an Autoboot Sector for 128 Programs

Program 5-2, "Create 128 Autoboot Sector," is fairly straightforward. It writes an autoboot sector to track 1, sector 0, of a disk. Load Program 5-2, insert a disk into your 1541 or 1571, and type RUN.

It first prompts you for a screen color. Enter a number in the range 0–15. If you want the screen to be white, for example, enter 1. Next, you're asked for a border color. The default values (if you don't answer the questions) are a blue screen with a light gray border. The screen and border will take on these colors when the disk is booted.

When you've set the screen and border colors, Program 5-2 writes the autoboot sector to disk. If you reset your 128 by turning it off and then on, it will read the sector and try to load a file called "HI". The HI file can be any valid program saved from 128 mode. It will load and automatically run. If you should want to load and run a machine language program, use these two lines, substituting the filename and SYS address of your ML (machine language) program:

10 BLOAD *"filename"*
20 SYS *xxxxx*

DSAVE this short program as "HI", and when it boots, it will load the ML and start it running.

Using the Menu from Program 5-3

If you load and run Program 5-3, you'll see a menu with five options:

64 MODE—BASIC
64 MODE—RUN FILE
BOOT CP/M DISK
128 MODE—BASIC
128 MODE—RUN FILE

Use the cursor keys to select one of the options, then press RETURN. If you should choose 128 MODE—BASIC, the program NEWs itself and exits to BASIC; 128 MODE—RUN FILE prompts you for the program name and then loads and runs that file.

If you opt for CP/M, it asks you to insert a CP/M disk. After doing so, press RETURN, and CP/M will boot.

Choose 64 MODE—BASIC to go directly to 64 mode without loading a program. The second choice, 64 MODE—RUN FILE, leads into several questions. First, you're asked DOES FILE CONTAIN AUTORUN CODE (Y/N)? If you've used Program 5-4 to create an autorun file (see below), you should answer yes and provide a filename. Program 5-3 then BLOADs the autorun file and executes the GO 64 command.

If you answer no to the prompt, then there must be a copy of the "AUTORUN.C64" file on the disk (see the description of Program 5-4 below for instructions on how to create this file). The program loads AUTORUN.C64, and inserts the name of your program. Next, you're asked IS THIS A SELF-STARTING FILE (Y/N)? If the 64 program to be loaded is in BASIC, answer yes. If not, you'll have to provide a SYS address. Finally, the 128 goes into 64 mode and runs the program you've requested.

A Memory-Based Cartridge for the 64?

We've already seen that the 128 checks for an autoboot disk sector when power is first turned on. But how does this program go into 64 mode and, in the process, cause the 64 to load and run a program? The answer can be found in the 64's own power-up routine. When you turn on a 64 (or GO 64 on the 128), the computer doesn't access the disk drive, but it *does* look for a cartridge. If it finds a certain sequence of letters and numbers at location 32768 (hexadecimal $8000), it does

not enable BASIC, but instead surrenders control to the program in the cartridge.

All three modes of the 128 (128, 64, and CP/M) share certain areas of memory. So, if we write a cartridge emulator, it can be BLOADed into location 32768 (in 128 mode). With that special program in place, the GO 64 command makes the computer turn into a 64 and begin the reset routine. As 64 mode is initializing itself, it finds something that looks like a cartridge at 32768. The memory-based cartridge tells it to load and run a program from disk.

In order to get this to work, we have to create the cartridge emulator with Programs 5-4 and 5-5.

First, enter Program 5-5 (remember, *this must be typed and saved from 64 mode*). Let's say you call it "64LOADER". Now turn the 128 off and back on, and enter Program 5-4 (from 128 mode). Now we can begin.

Run Program 5-4. Answer the first prompt with "64LOADER" (which must be on the disk currently in the drive). The disk drive light will turn on while the program is read into memory. When it's finished, it will ask for a filename. Answer "AUTORUN.C64". It will create the generic cartridge emulator used by Program 5-3. If this emulator is on a disk, Program 5-3 will be able to put the 128 into 64 mode and run any BASIC or ML program.

You can also generate a cartridge emulator for running a specific program. Let's say you want to create an autorun file for a game called SPACEGAME (saved as a BASIC program from 64 mode). Load and run Program 5-4. When it asks for the program name, answer "SPACEGAME". After reading the file, it will ask for a filename. Call it something like "SPACE.BOOT".

With a specific boot program in place, you can go into 64 mode and run it with the following sequence of events:

1. Insert the disk in the drive and turn on your 128.
2. The boot sector (which was created by Program 5-2) is loaded. It, in turn, loads the "HI" program (Program 5-3, saved under the name HI) from the disk.
3. Program 5-3 presents you with the menu of choices. Cursor down to 64 MODE—RUN FILE and press RETURN.
4. Answer yes to DOES FILE CONTAIN AUTORUN CODE? and enter SPACE.BOOT as the name of the program. The computer then goes into 64 mode. The boot program, acting

like a cartridge, then loads and runs the 64 program SPACEGAME.

Bypassing the Menu

There's one final option. Let's say you want to set up a boot disk that goes into 64 mode and loads (and runs) *SpeedScript*. You don't need Program 5-3, because you don't want the menu to appear—you just want *SpeedScript* to load and run. Follow these steps:

1. Load *SpeedScript* (from 64 mode) and save it to a freshly formatted disk.
2. Load Program 5-5 (again from 64 mode) and change lines 10–20. The first line should say **A$="SPEEDSCRIPT"** and line 20 should read **B$="RUN:"**. (If this were a machine language program stored at 49152, you would substitute SYS 49152 in line 20.) Save this under the name "SSLOADER" as the second program on the disk containing *SpeedScript*.
3. Go into 128 mode by pressing the reset switch, or turning it off and then back on. Load Program 5-4 and insert the disk containing the two programs. Run Program 5-4 and answer SSLOADER for the program name. Then answer SS.BOOT for the name of the boot program. There are now three programs on the disk.
4. While still in 128 mode, type NEW. Enter this one-line program:

   10 BLOAD"SS.BOOT":GO 64

   Save this to the disk as "HI"—the fourth program on the disk.
5. Finally, load Program 5-2. Switch disks, inserting the dedicated *SpeedScript* disk. Run the program, and an autoboot sector will be created. Now you have a disk which contains four programs and a boot sector.

Turn everything off. Turn on the disk drive and insert the *SpeedScript* disk. Turn on the 128. You don't need to do anything else. The boot sector loads the program called "HI". It loads the cartridge emulator and goes into 64 mode. The code at 32768 makes the 64 think a cartridge is in place, so SSLOADER is run. It, in turn, loads and runs *SpeedScript*.

It may seem like a lot of work, but the results are worth

it. To load and run a 64 program, just turn on your 128. Everything is handled by the boot sector (and the related files).

Program 5-2. Create 128 Autoboot Sector

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
QF 10 REM *** PROGRAM CREATES A C-128 BOOT DISK TO
       LOAD & RUN 'HI' FILE ***
BD 20 DATA67,66,77,0,0,0,0,0,0,120,32,132,255,234,
      234,234,234,234,169,195
HD 30 DATA141,238,255,169,8,141,239,255,169,0,141,
      240,255,169,15,141,32,208,169,6
RQ 40 DATA141,33,208,165,213,201,72,208,1,96,169,7
      2,141,233,7,169,73,141,234,7,169
QA 50 DATA1,162,8,160,255,32,186,255,32,192,255,16
      9,2,162,233,160,7
HR 60 DATA32,189,255,169,0,32,213,255,142,16,18,14
      0,17,18,32,231,255,169,4,133,208
EQ 70 DATA169,82,141,74,3,169,85,141,75,3,169,78,1
      41,76,3,169,13,141,77,3,96,-1
DS 80 DIMA(121)
CA 90 FORMX=1TO121:READS:X=X+S:IFS=-1THEN110
EP 100 A(MX)=S:NEXT
CF 110 IFX<>14733THENPRINT"ERROR IN DATA STATEMENT
       S.":STOP
GA 120 PRINT"{CLR}E6]{2 DOWN}ENTER YOUR PREFERRED
       {SPACE}COLOR CHOICES"
SF 130 PRINT"{2 DOWN}# OF SCREEN COLOR";:INPUTA(40
        ):POK$E53281,A(40)
EA 140 PRINT"{DOWN}# OF BORDER COLOR";:INPUTA(35):
       POKE53280,A(35)
KD 150 PRINT"{DOWN}ARE THESE ACCEPTABLE? (Y/N)";:I
       NPUTA$:IFA$<>"Y"THEN120
DH 160 POKE53280,15:POKE53281,6:PRINT"{CLR}{WHT}"
DR 170 PRINT"{2 DOWN}{RVS}INSERT A FORMATTED DISK
       {SPACE}TO RECEIVE DATA "
QF 180 PRINT"{RVS}{10 SPACES}THEN PRESS [RETURN]
       {11 SPACES}"
FA 190 GETR$:IFR$<>CHR$(13)THEN190
FG 200 PRINT"{2 DOWN}WORKING...":OPEN15,8,15:OPEN5
       ,8,5,"#":PRINT#15,"B-P:5,0"
JM 210 FORD=1TOMX:PRINT#5,CHR$(A(D));:NEXTD:PRINT#
       5
GH 220 PRINT#15,"M-W:"CHR$(0)CHR$(5);1;CHR$(67)
KK 230 PRINT#15,"B-P:5,0":PRINT#15,"U2:5,0,1,0":CL
       OSE5:CLOSE15
FH 240 OPEN15,8,15:INPUT#15,A$,B$,C$,D$:CLOSE15
MC 250 IFA$="00"THENPRINT"BOOT TRACK WRITTEN":END
MB 260 PRINT"{RVS}AN ERROR HAS OCCURRED.":PRINTA$,
       B$,C$,D$
GH 270 END
```

Program 5-3. Menu

Note: Save as "HI" if you want the menu to boot.
For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
HQ 10 REM *** AUTO-BOOT "HI" PROGRAM ***
HP 20 PRINT"{CLR}{2 DOWN}"TAB(8)"E5}--------------
      ---------"
CS 30 PRINTTAB(11)"{YEL}C-128 SYSTEM MENUE5}":PRIN
      TTAB(8)"------------------------E6}"
XS 40 DATA64 MODE - BASIC,64 MODE - RUN FILE,BOOT
      {SPACE}CP/M DISK,128 MODE - BASIC
FF 50 DATA128 MODE - RUN FILE
MH 60 MX=5:FORD=1TOMX:READA$(D):NEXT
JE 70 FORD=1TOMX:CHARØ,2Ø-(LEN(A$(D))/2),5+D*2,A$(
      D),Ø:NEXT
FS 80 NW=1:CHARØ,Ø,2Ø,"USE CURSOR KEYS TO MOVE-RET
      URN TO SELECT"
EQ 90 PRINT"{WHT}":CHARØ,2Ø-(LEN(A$(NW))/2),5+NW*2
      ,A$(NW),1:PRINT"E6}"
HP 100 GETR$:IFR$="{DOWN}"THEN14Ø
HB 110 IFR$="{UP}"THEN17Ø
HQ 120 IFR$=CHR$(13)THEN2ØØ
EJ 130 GOTO1ØØ
DP 140 CHARØ,2Ø-(LEN(A$(NW))/2),5+NW*2,A$(NW),Ø
AE 150 NW=NW+1:IFNW=MX+1THENNW=1
BA 160 GOTO9Ø
HF 170 CHARØ,2Ø-(LEN(A$(NW))/2),5+NW*2,A$(NW),Ø
CG 180 NW=NW-1:IFNW=.THENNW=MX
RD 190 GOTO9Ø
QF 200 ONNWGOTO22Ø,29Ø,23Ø,28Ø,26Ø
PR 210 GOTO1ØØ
SF 220 GO64
SH 230 PRINT"{CLR}{DOWN}INSERT CP/M SYSTEM DISK, P
       RESS [RETURN]"
BH 240 GETR$:IFR$<>CHR$(13)THEN24Ø
HR 250 BOOT
JA 260 PRINT"{CLR}{DOWN}FILENAME";:INPUTF$
EC 270 RUN(F$)
RF 280 PRINT"{CLR}{DOWN}THIS DISK CONTAINS:{DOWN}"
       :DIRECTORY:NEW
MC 290 PRINT"{CLR}DOES FILE CONTAIN AUTORUN CODE (
       Y/N)";
PK 300 INPUTF$:IFF$<>"Y"ANDF$<>"N"THEN29Ø
CM 310 IFF$="Y"THEN44Ø
DQ 320 BLOAD"AUTORUN.C64"
PA 330 PRINT"{CLR}{DOWN}FILE NAME";:INPUTF$
KA 340 IF LEN(F$)=.ORLEN(F$)>16THEN29Ø
ER 350 FORD=1TOLEN(F$):POKE32888+D,ASC(MID$(F$,D,1
       )):NEXT:POKE32888+D,42
ED 360 PRINT"IS THIS A SELF STARTING FILE? (Y/N)"
```

```
SG 370 GETKEYA$
RS 380 IF A$<>"N"THEN 430
MH 390 PRINT"{DOWN}ENTER FILE START ADDRESS";:INPU
       TF$
AJ 400 IFLEN(F$)=.ORLEN(F$)>5THEN390
HQ 410 F$="SYS"+F$
SK 420 FORD=1TOLEN(F$):POKE32915+D,ASC(MID$(F$,D,1
       )):NEXT
HC 430 GO64
PP 440 PRINT"{2 DOWN}FILE NAME";:INPUTF$
DQ 450 BLOAD(F$)
PF 460 GO64
```

Program 5-4. Create 64 Autorun Program

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
CK 10 REM ** CREATES AN AUTO RUN FILE WHEN YOU ENT
      ER 64 MODE **
ED 20 REM ** MUST BE RUN IN 128 MODE. THE PBASIC P
      ROGRAM MUST BE BASIC 2.0 ! **
JA 30 DATA15,128,9,128,195,194,205,56,48,104,168,1
      04,170
AK 40 DATA104,64,162,255,120,154,216,142,22,208,32
      ,163,253,32,80
DP 50 DATA253,32,21,253,32,24,229,88,32,83,228,32,
      191,227,32
JX 60 DATA34,228,162,251,154,169,113,133,43,169,12
      8,133,44,234,234
DQ 70 DATA169,0,133,45,169,0,133,46
JK 80 DATA162,160,134,56,169
GE 90 DATA82,141,119,2,169,213,141,120,2,169,13,14
      1,121,2,169
DQ 100 DATA3,133,198,162,128,138,76,116,164,0,0,0,
       0,0,0,51,47
FD 110 DATA49,53,22,49,46,49,0,0,0,0
GR 120 PRINTCHR$(147):DN$=CHR$(17)
KA 130 PRINTDN$"ENTER FILENAME TO AUTORUN";:INPUTF
       1$
SF 140 OPEN2,8,2,F1$+",P,R":GOSUB300:CLOSE2
HH 150 PRINTCHR$(147)DN$DN$"POKE45,113:POKE46,128:
       DLOAD"CHR$(34)F1$CHR$(34)
PJ 160 PRINTDN$DN$DN$DN$"PP=FRE(0):POKE45,1:POKE46
       ,28:GOTO170{HOME}";
XJ 165 POKE208,2:POKE842,13:POKE843,13:END
XA 170 PRINTCHR$(147)CHR$(17)"ENTER AUTORUN FILENA
       ME: ";:INPUTF$
ER 180 PRINTCHR$(17)"["F$"] FILESIZE:"118+(32397-P
       P)"BYTES"
QD 190 XX=(32397-PP)+32768+118
```

```
MQ 200 BANK0:RESTORE:FORD=.TO112:READS:POKE32768+D
       ,S:NEXT
GR 210 HI=INT(XX/256):LO=256*(XX/256-INT(XX/256))
QC 220 POKE32827,LO:POKE32831,HI
CC 230 BSAVE(F$),B0,P32768TOP(XX)
BQ 240 GOSUB300
MX 250 PRINTCHR$(17)"AUTORUN FILE CREATED."
AK 260 PRINTCHR$(17)"CREATE ANOTHER? (Y/N)"
FX 270 GETKEYA$:IFA$="Y"THENRUN
XR 280 IF A$="N"THENPRINTCHR$(147):NEW
DH 290 GOTO270
KE 300 CLOSE15:OPEN15,8,15:INPUT#15,A$,B$:IFA$<>"0
       0"THENPRINT"{DOWN}{RIGHT}{RVS}ERROR :"B$:EN
       D
PK 310 RETURN
```

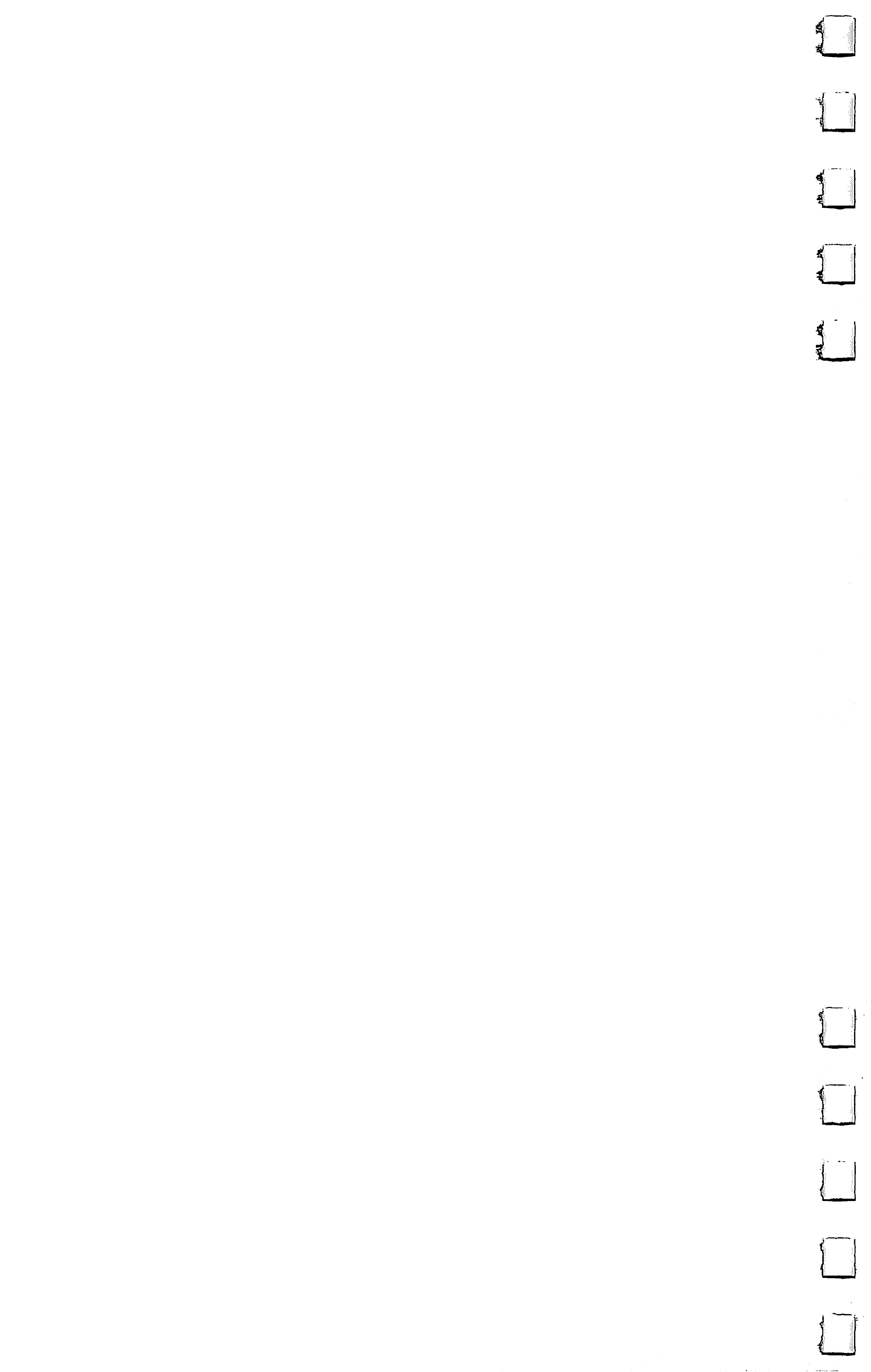
Program 5-5. 64 Loader Program

Enter and save from 64 mode.

For mistake-proof program entry, be sure to use "The Automatic Proofreader," Appendix B.

```
KR 10 A$="................."
DQ 20 B$="RUN:...."
BS 30 PRINT"{CLR}{BLU}{2 DOWN}PO43,1:PO44,8:NEW":P
      RINT"{2 DOWN}LOAD"CHR$(34)A$CHR$(34)",8,1
EG 40 PRINT"{4 DOWN}?CHR$(5):"B$"{HOME}";
SX 50 POKE808,237:POKE809,246
MG 60 POKE198,3:POKE631,13:POKE632,13:POKE633,13:E
      ND
```

Appendices



Appendix A
How to Type In Programs

In order to make it as easy as possible to enter the programs in this book, we've included two program entry aids written in BASIC: "The Automatic Proofreader" and "MLX." To assist you in understanding how to enter these programs, COMPUTE! has established the following listing conventions.

Generally, BASIC program listings like the one for MLX will contain words within braces which spell out any special characters: {DOWN} means to press the cursor-down key; {5 SPACES} means to press the space bar five times.

To indicate that a key should be *shifted* (press the key while holding down the SHIFT key), the key will be underlined in our listings. For example, S means to type the S key while holding down the SHIFT key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces, for example, {10 N}, you should type the key as many times as indicated. In this case, you would enter ten shifted *N*'s.

If a key is enclosed in special brackets, [< >], you should hold down the Commodore key while pressing the key inside the special brackets. (The Commodore key is the key in the lower left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as indicated; [<9@>] means type Commodore-@ nine times.

Refer to Figure A-1 when entering cursor and color control keys:

Figure A-1. Keyboard Conventions

When You Read:	Press:	See:
{CLR}	SHIFT CLR/HOME	
{HOME}	CLR/HOME	
{UP}	SHIFT ↑ CRSR ↓	
{DOWN}	↑ CRSR ↓	
{LEFT}	SHIFT ← CRSR →	
{RIGHT}	← CRSR →	
{RVS}	CTRL 9	
{OFF}	CTRL 0	
{BLK}	CTRL 1	
{WHT}	CTRL 2	
{RED}	CTRL 3	
{CYN}	CTRL 4	
{PUR}	CTRL 5	
{GRN}	CTRL 6	
{BLU}	CTRL 7	
{YEL}	CTRL 8	

When You Read:	Press:	See:
{1}	COMMODORE 1	
{2}	COMMODORE 2	
{3}	COMMODORE 3	
{4}	COMMODORE 4	
{5}	COMMODORE 5	
{6}	COMMODORE 6	
{7}	COMMODORE 7	
{8}	COMMODORE 8	
{ F1 }	f1	
{ F2 }	SHIFT f1	
{ F3 }	f3	
{ F4 }	SHIFT f3	
{ F5 }	f5	
{ F6 }	SHIFT f5	
{ F7 }	f7	
{ F8 }	SHIFT f7	
←	←	
↑	SHIFT ↑	

Appendix B
The Automatic Proofreader

Philip I. Nelson

"The Automatic Proofreader" helps you type in program listings without typing mistakes. It's a short error-checking program that conceals itself in memory and adheres to your Commodore's operating system. Each time you press RETURN to enter a program line, the Proofreader displays a two-letter checksum in reverse video at the top of your screen. If the checksum on your screen doesn't match the one in the printed listing, you've typed the line incorrectly—it's that simple. You don't have to use the Proofreader to enter printed listings, but doing so greatly reduces your chances of making a typo.

Getting Started

First, type in the Automatic Proofreader program *exactly* as it appears in the listing. Since the Proofreader can't check itself, type carefully to avoid mistakes. Don't omit any lines, even if they contain unfamiliar commands. As soon as you're finished typing the Proofreader, save at least two copies on disk or tape before running it the first time. This is very important because the Proofreader erases the BASIC portion of itself when you run it, leaving only the machine language portion in memory.

When that's done, type RUN and press RETURN. After announcing which computer it's running on, the Proofreader installs the ML routine in memory, displays the message PROOFREADER ACTIVE, erases the BASIC portion of itself, and ends. If you type LIST and press RETURN, you'll see that no BASIC program remains in memory. The computer is ready for you to type in a new BASIC program.

Entering Programs

Once the Proofreader is active, you can begin typing in a BASIC program as usual. Every time you finish typing a line and press RETURN, the Proofreader displays a two-letter checksum (reverse-video letters) in the upper left corner of the screen. Compare this checksum with the two-letter checksum

printed to the left of the corresponding line in the program listing. If the letters match, it's almost certain the line was typed correctly. If the letters don't match, check for your mistake and correct the line.

The Proofreader ignores spaces that aren't enclosed in quotation marks, so you can omit spaces (or add extra ones) between keywords and still see a matching checksum. For example, these two lines generate the same checksum:

```
10 PRINT"THIS IS BASIC"
10 PRINT          "THIS IS BASIC"
```

However, since spaces inside quotation marks are almost always significant, the Proofreader pays attention to them. For instance, these two lines generate different checksums:

```
10 PRINT"THIS IS BASIC"
10 PRINT"THIS ISBA     SIC"
```

A common typing mistake is transposition—typing two successive characters in the wrong order, like PIRNT instead of PRINT or 64378 instead of 64738. A checksum program that adds up the values of all the characters in a line can't possibly detect transposition errors (it can only tell whether the right characters are present, regardless of what order they're in). Because the Proofreader computes the checksum with a more sophisticated formula, it is also sensitive to the *position* of each character within the line and thus catches transposition errors.

The Proofreader does *not* accept keyword abbreviations (for example, ? instead of PRINT). If you prefer to use abbreviations, you can still check the line with the Proofreader: Simply LIST the line after typing it, move the cursor back onto the line, and press RETURN. LISTing the line substitutes the full keyword for the abbreviation and allows the Proofreader to work properly. The same technique works for rechecking a program you've already typed in: Reload the program, LIST several lines on the screen, and press RETURN over them.

Do not use any GRAPHIC commands while the Proofreader is active. When you activate a command like GRAPHIC 1, the computer moves everything at the start of BASIC program space—including the Proofreader—to another memory area, causing the Proofreader to crash. The same thing happens if you run any program that contains a GRAPHIC command. The Proofreader deallocates any graphics areas before install-

ing itself in memory, but you are responsible for seeing that the computer remains in this configuration.

Though the Proofreader doesn't interfere with other BASIC operations, it's always a good idea to disable it before running any other program. Some programs may need the space occupied by the Proofreader's ML routine or may create other memory conflicts. However, the Proofreader is purposely made difficult to dislodge: It's not affected by tape or disk operations, or by pressing RUN/STOP–RESTORE. The simplest way to disable it is to turn the computer off, then on again.

Program B-1. The Automatic Proofreader

```
10 VEC=PEEK(772)+256*PEEK(773):LO=43:HI=44
20 PRINT "AUTOMATIC PROOFREADER FOR ";:IF VEC=4236
   4 THEN PRINT "C-64"
30 IF VEC=50556 THEN PRINT "VIC-20"
40 IF VEC=35158 THEN GRAPHIC CLR:PRINT "PLUS/4 & 1
   6"
50 IF VEC=17165 THEN LO=45:HI=46:GRAPHIC CLR:PRINT
   "128"
60 SA=(PEEK(LO)+256*PEEK(HI))+6:ADR=SA
70 FOR J=0 TO 166:READ BYT:POKE ADR,BYT:ADR=ADR+1:
   CHK=CHK+BYT:NEXT
80 IF CHK<>20570 THEN PRINT "*ERROR* CHECK TYPING
   {SPACE}IN DATA STATEMENTS":END
90 FOR J=1 TO 5:READ RF,LF,HF:RS=SA+RF:HB=INT(RS/2
   56):LB=RS-(256*HB)
100 CHK=CHK+RF+LF+HF:POKE SA+LF,LB:POKE SA+HF,HB:N
   EXT
110 IF CHK<>22054 THEN PRINT "*ERROR* RELOAD PROGR
   AM AND CHECK FINAL LINE":END
120 POKE SA+149,PEEK(772):POKE SA+150,PEEK(773)
130 IF VEC=17165 THEN POKE SA+14,22:POKE SA+18,23:
   POKESA+29,224:POKESA+139,224
140 PRINT CHR$(147);CHR$(17);"PROOFREADER ACTIVE":
   SYS SA
150 POKE HI,PEEK(HI)+1:POKE (PEEK(LO)+256*PEEK(HI)
   )-1,0:NEW
160 DATA 120,169,73,141,4,3,169,3,141,5,3
170 DATA 88,96,165,20,133,167,165,21,133,168,169
180 DATA 0,141,0,255,162,31,181,199,157,227,3
190 DATA 202,16,248,169,19,32,210,255,169,18,32
200 DATA 210,255,160,0,132,180,132,176,136,230,180
210 DATA 200,185,0,2,240,46,201,34,208,8,72
220 DATA 165,176,73,255,133,176,104,72,201,32,208
230 DATA 7,165,176,208,3,104,208,226,104,166,180
240 DATA 24,165,167,121,0,2,133,167,165,168,105
```

```
250 DATA Ø,133,168,2Ø2,2Ø8,239,24Ø,2Ø2,165,167,69
260 DATA 168,72,41,15,168,185,211,3,32,21Ø,255
270 DATA 1Ø4,74,74,74,74,168,185,211,3,32,21Ø
280 DATA 255,162,31,189,227,3,149,199,2Ø2,16,248
290 DATA 169,146,32,21Ø,255,76,86,137,65,66,67
300 DATA 68,69,7Ø,71,72,74,75,77,8Ø,81,82,83,88
310 DATA 13,2,7,167,31,32,151,116,117,151,128,129,
   167,136,137
```

Appendix C
Machine Language Editor, MLX

Ottis R. Cowper

"MLX" is a new way to enter long machine language programs without a lot of fuss. MLX lets you enter the numbers from a special list that looks similar to BASIC DATA statements. It checks your typing on a line-by-line basis. It won't let you enter invalid characters or let you continue if there's a mistake in a line. It won't even let you enter a line or digit out of sequence.

Using MLX

Type in and save some copies of MLX (you'll want to use it to enter future ML programs from other COMPUTE! publications). When you're ready to enter the machine language part of "Orbitron" or "MetaBASIC", load and run MLX. It asks you for a starting address and an ending address. These addresses are

Program 3-2. ORB.OBJ
Starting address: 7530
Ending address: 7997
Save as: ORB.OBJ

Program 4-1. MetaBASIC
Starting address: 1300
Ending address: 18BF

If you're unfamiliar with machine language, the addresses (and all other values you enter in MLX) may appear strange. Instead of the usual decimal numbers you're accustomed to, these numbers are in *hexadecimal*—a base 16 numbering system commonly used by ML programmers. Hexadecimal—hex for short—includes the numbers 0–9 and the letters *A–F*. But don't worry—even if you know nothing about ML or hex, you should have no trouble using MLX.

After you enter the starting and ending addresses, MLX will offer you the option of clearing the workspace. Choose

this option if you're starting to enter a program for the first time. If you're continuing to enter a program that you partially typed from a previous session, don't choose this option.

It's not necessary to know more about this option to use MLX, but here's an explanation if you're interested: When you first run MLX, the workspace area contains random values. Clearing the workspace fills it with zeros. This makes it easier to find where you left off if you enter the listing in multiple sittings. However, clearing the workspace is useful only before you first begin entering a listing; there's no need to clear it before you reload to continue entering a partially typed listing.

When you save your work with MLX, it stores the entire contents of the data buffer. If you clear the workspace before starting, the incomplete portion of the listing is filled with zeros when saved and thus refilled with zeros when reloaded. If you don't clear the workspace when first starting, the incomplete portion of the listing is filled with random data. Whether or not you clear the workspace before you reload, this random data will refill the unfinished part of the listing when you load your previous work. The rule, then, is to use the clear workspace feature before you begin entering data from a listing and not to bother with it afterward.

At this point, MLX presents a menu of commands:

Enter data
Display data
Load data
Save file
Catalog disk
Quit

Entering a Listing

To begin entering data, press E. You'll be asked for the address at which you wish to begin entering data. (If you pressed E by mistake, you can return to the command menu by pressing RETURN.) When you begin typing, you should enter the starting address here. If you're typing a program in multiple sittings, you should enter the address where you left off typing at the end of the previous session. In any case, make sure the address you enter corresponds to the address of a line of the MLX listing. Otherwise, you'll be unable to enter the data correctly.

After you enter the address, you'll see that address appear as a prompt with a nonblinking cursor. Now you're ready to enter data. Type in all nine numbers on that line, beginning with the first two-digit number after the colon (:). Each line represents eight data bytes and a checksum. Although an MLX-format listing resembles the "hex dump" machine language listings you may be accustomed to, the extra checksum number on the end allows MLX to check your typing. (You *can* enter the data from an MLX listing using the built-in monitor if the rightmost column of data is omitted, but we recommend against it. It's much easier to let MLX do the proofreading and error checking for you.)

Only the numbers 0–9 and the letters *A–F* can be typed in. If you press any other key (with some exceptions noted below), you'll hear a warning buzz. To simplify typing, MLX redefines the function keys and the + and – keys on the numeric keypad so that you can enter data one-handed. Figure C-1 shows the keypad configuration supported by MLX.

Figure C-1. Keypad for 128 MLX

A	B	C	D

7	8	9	E
4	5	6	F
1	2	3	ENTER
0		.	

MLX checks for transposed characters. If you're supposed to type in *A0* and instead enter *0A*, MLX will catch your mistake. To correct typing mistakes before finishing a line, use the INST/DEL key to delete the character to the left of the cursor.

(The cursor-left key also deletes.) If you mess up a line really badly, press CLR/HOME to start the line over.

The RETURN key is also active, but only before any data is typed on a line. Pressing RETURN at this point returns you to the command menu. After you type a character of data, MLX disables RETURN until the cursor returns to the start of a line. Remember, you can press CLR/HOME to get to a line number prompt quickly.

Beep or Buzz

When you enter a line, MLX recalculates the checksum from the eight bytes and the address and compares this value to the number from the ninth column. If the values match, you'll hear a pleasant beep to indicate that the line was entered correctly. The data is then added to the workspace area, and the prompt for the next line of data appears. But if MLX detects a typing error, you'll hear a low buzz and see an error message. MLX will then redisplay the line for editing.

To make corrections in a line that MLX has redisplayed for editing, compare the line on the screen with the one printed in the listing, then move the cursor to the mistake and press the correct key. The cursor-left and -right keys provide the normal cursor controls. (The INST/DEL key now works as an alternative cursor-left key.) You cannot move left beyond the first character in the line. If you try to move beyond the rightmost character, you'll reenter the line. During editing, RETURN is active; pressing it tells MLX to recheck the line. You can press the CLR/HOME key to clear the entire line if you want to start from scratch, or if you want to get to a line number prompt to use RETURN to get back to the menu.

After you have entered the last number on the last line of the listing, MLX automatically moves to the Save option.

Other MLX Functions

The second menu choice, DISPLAY DATA, examines memory and shows the contents in the same format as the program listing (including the checksum). When you press D, MLX asks you for a starting address. Be sure that the starting address you give corresponds to a line number in the listing. Otherwise, the display will be meaningless. MLX displays program lines until it reaches the end of the program, at which point

the menu is redisplayed. You can pause the scrolling display by pressing the space bar. (MLX finishes printing the current line before halting.) To resume scrolling, press the space bar again. To break out of the display and return to the menu before the ending address is reached, press RETURN.

Two more menu selections let you save programs and load them back into the computer. These are SAVE FILE and LOAD DATA; their operation is quite straightforward. When you press S or L, MLX asks you for the filename. (Again, pressing RETURN at this prompt without entering anything returns you to the command menu.) Next, MLX asks you to press either D or T to select disk or tape.

You'll notice the disk drive starting and stopping several times during a save. Don't panic; this is normal behavior. MLX opens and writes to the file instead of using the usual SAVE command. (Loads, on the other hand, operate at normal speed—thanks to the relocating feature of BASIC 7.0's BLOAD command.) Remember that MLX saves the entire workspace area from the starting address to the ending address, so the save or load may take longer than you might expect if you've entered only a small amount of data from a long listing. When saving a partially completed listing, make sure to note the address where you stopped typing so that you'll know where to resume entry when you reload.

Error Alert

MLX reports any errors detected during the save or load and displays the standard error messages. (Tape users should bear in mind that the Commodore 128 is never able to detect errors when saving to tape.) MLX also has three special load error messages:

- INCORRECT STARTING ADDRESS, which means the file you're trying to load does not have the starting address you specified when you ran MLX. In this case, no data will be loaded.
- LOAD ENDED AT *address*, which means the file you're trying to load ends before the ending address you specified when you started MLX. The data from the file is loaded, but it ends at the address specified in the error message.
- TRUNCATED AT ENDING ADDRESS, which means the file you're trying to load extends beyond the ending address you

specified when you started MLX. The data from the file is loaded, but only up to the specified ending address.

If you see one of these messages and feel certain that you've loaded the right file, exit and rerun MLX, being careful to enter the correct starting and ending addresses.

If you wish to check which programs are on a disk, select the C option from the command menu for a directory. You can use the 128's NO SCROLL key to pause the display. Afterward, press any key to return to the menu.

The Quit menu option has the obvious effect—it stops MLX and enters BASIC. The RUN/STOP key is trapped, so the Q option lets you exit the program without turning off the computer. (Of course, RUN/STOP–RESTORE also gets you out.) You'll be asked for verification; press Y to exit to BASIC or any other key to return to the menu. After quitting, you can type RUN again and reenter MLX without losing your data as long as you don't use the clear workspace option.

The Finished Product

When you've finished typing all the data for an ML program and saved your work, you're ready to see the results. The instructions for loading and using the finished product vary from program to program. So check the instructions in the article about the program.

An Ounce of Prevention

By the time you finish typing in the data for a long ML program, you'll have many hours invested in the project. Don't take chances—use our "Automatic Proofreader" (Appendix B) to type MLX, and then test your copy *thoroughly* before first using it to enter any significant amount of data. Make sure all the menu options work as they should. Enter fragments of the program starting at several different addresses, then use the Display option to verify that the data has been entered correctly. And be sure to test the Save and Load options several times to insure that you can recall your work from disk or tape. Don't let a simple typing error in MLX cost you several nights of hard work.

Program C-1. MLX

```
AE 100 TRAP 960:POKE 4627,128:DIM NL$,A(7)
XP 110 Z2=2:Z4=254:Z5=255:Z6=256:Z7=127:BS=256*PEE
       K(4627):EA=65280
FB 120 BE$=CHR$(7):RT$=CHR$(13):DL$=CHR$(20):SP$=C
       HR$(32):LF$=CHR$(157)
KE 130 DEF FNHB(A)=INT(A/256):DEF FNLB(A)=A-FNHB(A
       )*256:DEF FNAD(A)=PEEK(A)+256*PEEK(A+1)
JB 140 KEY 1,"A":KEY 3,"B":KEY 5,"C":KEY 7,"D":VOL
        15:IF RGR(0)=5 THEN FAST
FJ 150 PRINT"{CLR}"CHR$(142);CHR$(8):COLOR 0,15:CO
       LOR 4,15:COLOR 6,15
GQ 160 PRINT TAB(12)"{RED}{RVS}{2 SPACES}{9 @}
       {2 SPACES}"RT$;TAB(12)"{RVS}{2 SPACES}{OFF}
       {BLU} 128 MLX {RED}{RVS}{2 SPACES}"RT$;TAB(
       12)"{RVS}{13 SPACES}{BLU}"
FE 170 PRINT"{2 DOWN}{3 SPACES}COMPUTE!'S MACHINE
       {SPACE}LANGUAGE EDITOR{2 DOWN}"
DK 180 PRINT"{BLK}STARTING ADDRESS{4}";:GOSUB 260:
       IF AD THEN SA=AD:ELSE 180
FH 190 PRINT"{BLK}{2 SPACES}ENDING ADDRESS{4}";:GO
       SUB 260:IF AD THEN EA=AD:ELSE 190
MF 200 PRINT"{DOWN}{BLK}CLEAR WORKSPACE [Y/N]?{4}"
       :GETKEY A$:IF A$<>"Y" THEN 220
QH 210 PRINT"{DOWN}{BLU}WORKING...";:BANK 0:FOR A=
       BS TO BS+(EA-SA)+7:POKE A,0:NEXT A:PRINT"DO
       NE"
DC 220 PRINT TAB(10)"{DOWN}{BLK}{RVS} MLX COMMAND
       {SPACE}MENU {4}{DOWN}":PRINT TAB(13)"{RVS}E
       {OFF}NTER DATA"RT$;TAB(13)"{RVS}D{OFF}ISPLA
       Y DATA"RT$;TAB(13)"{RVS}L{OFF}OAD FILE"
HB 230 PRINT TAB(13)"{RVS}S{OFF}AVE FILE"RT$;TAB(1
       3)"{RVS}C{OFF}ATALOG DISK"RT$;TAB(13)"{RVS}
       Q{OFF}UIT{DOWN}{BLK}"
AP 240 GETKEY A$:A=INSTR("EDLSCQ",A$):ON A GOTO 34
       0,550,640,650,930,940:GOSUB 950:GOTO 240
SX 250 PRINT"STARTING AT";:GOSUB 260:IF(AD<>0)OR(A
       $=NL$)THEN RETURN:ELSE 250
BG 260 A$=NL$:INPUT A$:IF LEN(A$)=4 THEN AD=DEC(A$
       )
PP 270 IF AD=0 THEN BEGIN:IF A$<>NL$ THEN 300:ELSE
        RETURN:BEND
MA 280 IF AD<SA OR AD>EA THEN 300
PM 290 IF AD>511 AND AD<65280 THEN PRINT BE$;:RETU
       RN
SQ 300 GOSUB 950:PRINT"{RVS} INVALID ADDRESS
       {DOWN}{BLK}":AD=0:RETURN
RD 310 CK=FNHB(AD):CK=AD-Z4*CK+Z5*(CK>Z7):GOTO 330
```

```
DD 32Ø CK=CK*Z2+Z5*(CK>Z7)+A
AH 33Ø CK=CK+Z5*(CK>Z5):RETURN
QD 34Ø PRINT BE$;"{RVS} ENTER DATA ":GOSUB 25Ø:IF
       {SPACE}A$=NL$ THEN 22Ø
JA 35Ø BANK Ø:PRINT:F=Ø:OPEN 3,3
BR 36Ø GOSUB 31Ø:PRINT HEX$(AD)+":";:IF F THEN PRI
       NT L$:PRINT"{UP}{5 RIGHT}";
QA 37Ø FOR I=Ø TO 24 STEP 3:B$=SP$:FOR J=1 TO 2:IF
        F THEN B$=MID$(L$,I+J,1)
PS 38Ø PRINT"{RVS}"B$+LF$;:IF I<24 THEN PRINT"
       {OFF}";
RC 39Ø GETKEY A$:IF (A$>"/" AND A$<":") OR(A$>"@"
       {SPACE}AND A$<"G") THEN 47Ø
AC 4ØØ IF A$="+" THEN A$="E":GOTO 47Ø
QB 41Ø IF A$="-" THEN A$="F":GOTO 47Ø
FB 42Ø IF A$=RT$ AND ((I=Ø) AND (J=1) OR F) THEN P
       RINT B$;:J=2:NEXT:I=24:GOTO 48Ø
RD 43Ø IF A$="{HOME}" THEN PRINT B$:J=2:NEXT:I=24:
       NEXT:F=Ø:GOTO 36Ø
XB 44Ø IF (A$="{RIGHT}") AND F THEN PRINT B$+LF$;:
       GOTO 47Ø
JP 45Ø IF A$<>LF$ AND A$<>DL$ OR ((I=Ø) AND (J=1))
        THEN GOSUB 95Ø:GOTO 39Ø
PS 46Ø A$=LF$+SP$+LF$:PRINT B$+LF$;:J=2-J:IF J THE
       N PRINT LF$;:I=I-3
GB 47Ø PRINT A$;:NEXT J:PRINT SP$;
HA 48Ø NEXT I:PRINT:PRINT"{UP}{5 RIGHT}";:L$="
       {27 SPACES}"
DP 49Ø FOR I=1 TO 25 STEP 3:GET#3,A$,B$:IF A$=SP$
       {SPACE}THEN I=25:NEXT:CLOSE 3:GOTO 22Ø
BA 5ØØ A$=A$+B$:A=DEC(A$):MID$(L$,I,2)=A$:IF I<25
       {SPACE}THEN GOSUB 32Ø:A(I/3)=A:GET#3,A$
AR 51Ø NEXT I:IF A<>CK THEN GOSUB 95Ø:PRINT:PRINT"
       {RVS} ERROR: REENTER LINE ":F=1:GOTO 36Ø
DX 52Ø PRINT BE$:B=BS+AD-SA:FOR I=Ø TO 7:POKE B+I,
       A(I):NEXT I
XB 53Ø F=Ø:AD=AD+8:IF AD<=EA THEN 36Ø
CA 54Ø CLOSE 3:PRINT"{DOWN}{BLU}** END OF ENTRY **
       {BLK}{2 DOWN}":GOTO 65Ø
MC 55Ø PRINT BE$;"{CLR}{DOWN}{RVS} DISPLAY DATA ":
       GOSUB 25Ø:IF A$=NL$ THEN 22Ø
JF 56Ø BANK Ø:PRINT"{DOWN}{BLU}PRESS: {RVS}SPACE
       {OFF} TO PAUSE, {RVS}RETURN{OFF} TO BREAK
       {4}{DOWN}"
XA 57Ø PRINT HEX$(AD)+":";:GOSUB 31Ø:B=BS+AD-SA
DJ 58Ø FOR I=B TO B+7:A=PEEK(I):PRINT RIGHT$(HEX$(
       A),2);SP$;:GOSUB 32Ø:NEXT I
XB 59Ø PRINT"{RVS}";RIGHT$(HEX$(CK),2)
GR 6ØØ F=1:AD=AD+8:IF AD>EA THEN PRINT"{BLU}** END
        OF DATA **":GOTO 22Ø
```

```
EB 610 GET A$:IF A$=RT$ THEN PRINT BE$:GOTO 220
QK 620 IF A$=SP$ THEN F=F+1:PRINT BE$;
XS 630 ON F GOTO 570,610,570
RF 640 PRINT BE$"{DOWN}{RVS} LOAD DATA ":OP=1:GOTO
        660
BP 650 PRINT BE$"{DOWN}{RVS} SAVE FILE ":OP=0
DM 660 F=0:F$=NL$:INPUT"FILENAME{4}";F$:IF F$=NL$
       {SPACE}THEN 220
RF 670 PRINT"{DOWN}{BLK}{RVS}T{OFF}APE OR {RVS}D
       {OFF}ISK: {4}";
SQ 680 GETKEY A$:IF A$="T" THEN 850:ELSE IF A$<>"D
       " THEN 680
SP 690 PRINT"DISK{DOWN}":IF OP THEN 760
EH 700 DOPEN#1,(F$+",P"),W:IF DS THEN A$=D$:GOTO 7
       40
JH 710 BANK 0:POKE BS-2,FNLB(SA):POKE BS-1,FNHB(SA
       ):PRINT"SAVING ";F$:PRINT
MC 720 FOR A=BS-2 TO BS+EA-SA:PRINT#1,CHR$(PEEK(A)
       );:IF ST THEN A$="DISK WRITE ERROR":GOTO 75
       0
GC 730 NEXT A:CLOSE 1:PRINT"{BLU}** SAVE COMPLETED
        WITHOUT ERRORS **":GOTO 220
RA 740 IF DS=63 THEN BEGIN:CLOSE 1:INPUT"{BLK}REPL
       ACE EXISTING FILE [Y/N]{4}";A$:IF A$="Y" TH
       EN SCRATCH(F$):PRINT:GOTO 700:ELSE PRINT"
       {BLK}":GOTO 660:BEND
GA 750 CLOSE 1:GOSUB 950:PRINT"{BLK}{RVS} ERROR DU
       RING SAVE: {4}":PRINT A$:GOTO 220
FD 760 DOPEN#1,(F$+",P"):IF DS THEN A$=DS$:F=4:CLO
       SE 1:GOTO 790
PX 770 GET#1,A$,B$:CLOSE 1:AD=ASC(A$)+256*ASC(B$):
       IF AD<>SA THEN F=1:GOTO 790
KB 780 PRINT"LOADING ";F$:PRINT:BLOAD(F$),B0,P(BS)
       :AD=SA+FNAD(174)-BS-1:F=-2*(AD<EA)-3*(AD>EA
       )
RQ 790 IF F THEN 800:ELSE PRINT"{BLU}** LOAD COMPL
       ETED WITHOUT ERRORS **":GOTO 220
ER 800 GOSUB 950:PRINT"{BLK}{RVS} ERROR DURING LOA
       D: {4}":ON F GOSUB 810,820,830,840:GOTO220
QJ 810 PRINT"INCORRECT STARTING ADDRESS (";HEX$(AD
       );")":RETURN
DP 820 PRINT"LOAD ENDED AT ";HEX$(AD):RETURN
EB 830 PRINT"TRUNCATED AT ENDING ADDRESS ("HEX$(EA
       )")":RETURN
FP 840 PRINT"DISK ERROR ";A$:RETURN
KS 850 PRINT"TAPE":AD=POINTER(F$):BANK 1:A=PEEK(AD
       ):AL=PEEK(AD+1):AH=PEEK(AD+2)
XX 860 BANK 15:SYS DEC("FF68"),0,1:SYS DEC("FFBA")
       ,1,1,0:SYS DEC("FFBD"),A,AL,AH:SYS DEC("FF9
       0"),128:IF OP THEN 890
```

```
FG 87Ø PRINT:A=SA:B=EA+1:GOSUB 92Ø:SYS DEC("E919")
       ,3:PRINT"SAVING ";F$
AB 88Ø A=BS:B=BS+(EA-SA)+1:GOSUB 92Ø:SYS DEC("EA18
       "):PRINT"{DOWN}{BLU}** TAPE SAVE COMPLETED
       {SPACE}**":GOTO 22Ø
CP 89Ø SYS DEC("E99A"):PRINT:IF PEEK(2816)=5 THEN
       {SPACE}GOSUB 95Ø:PRINT"{DOWN}{BLK}{RVS} FIL
       E NOT FOUND ":GOTO 22Ø
GQ 9ØØ PRINT"LOADING ...{DOWN}":AD=FNAD(2817):IF A
       D<>SA THEN F=1:GOTO 8ØØ:ELSE AD=FNAD(2819)-
       1:F=-2*(AD<EA)-3*(AD>EA)
SH 91Ø A=BS:B=BS+(EA-SA)+1:GOSUB 92Ø:SYS DEC("E9FB
       "):IF ST THEN 8ØØ:ELSE 79Ø
XB 92Ø POKE193,FNLB(A):POKE194,FNHB(A):POKE 174,FN
       LB(B):POKE 175,FNHB(B):RETURN
CP 93Ø CATALOG:PRINT"{DOWN}{BLU}** PRESS ANY KEY F
       OR MENU **":GETKEY A$:GOTO 22Ø
MM 94Ø PRINT BE$"{RVS} QUIT {4}";RT$;"ARE YOU SURE
        [Y/N]?":GETKEY A$:IF A$<>"Y" THEN 22Ø:ELSE
        PRINT"{CLR}":BANK 15:END
JE 95Ø SOUND 1,5ØØ,1Ø:RETURN
AF 96Ø IF ER=14 AND EL=26Ø THEN RESUME 3ØØ
MK 97Ø IF ER=14 AND EL=5ØØ THEN RESUME NEXT
KJ 98Ø IF ER=4 AND EL=78Ø THEN F=4:A$=DS$:RESUME 8
       ØØ
DQ 99Ø IF ER=3Ø THEN RESUME:ELSE PRINT ERR$(ER);"
       {SPACE}ERROR IN LINE";EL
```

Index

@ *See* at sign
? *See* question mark
/ *See* slash
accumulator 44
addresses, changing 69–71
addressing scheme 46
ADSR 84, 86
AID 144, 145
AND 5
APPEND 5, 175
ASC 90
ASCII code 34, 64, 76, 145, 175
asterisk 59
at sign (@) 145
attack rate 83, 84
AUTO 4
autoboot 172, 186–90
autoboot sector 186, 187
AUTOI 22
"Automatic Proofreader, The" program 201–2
automatic sprite movement 110, 111, 114
BACKUP 5, 178
BAM 17, 21–24, 177
BANK 5, 46, 70–72
bank numbers 46
"Bar Chart" program 15
BASIC 7.0 3
BASIC 7.0 keywords 13, 14
batch files 54, 55
BEGIN 6, 115
BEGIN–BEND 6
BEND 6, 115
binary search method 37–39
"Blick" program 158
BLOAD 5, 167, 170–72, 175
block allocation map. *See* BAM
blocks of data 167
BOOT 5, 170, 172
boot disk 186
bootstraps 33
BOX 9
BRK 44, 45
BSAVE 4, 5, 174, 175
BUMP routine 4, 115
burst mode 167
bus 166
CATALOG 5
"Cataloger" program 182–85
CBM 172
chaining 29, 33, 173
CHANGE 144, 145
CHAR 6, 11, 112
character set 69–71
CHR$ code 90, 176
chroma 168
CIA chip 167
CIRCLE 11
CLEAR 64
clock 7
CLOSE 18, 175
CLR 70
CLSDIR 24
"Coder-Decoder" program 41
coding 40
COLLECT 5, 177
COLLISION 4
collision routine 115
colon 114, 115
color 9, 53
COLOR 9
COM files 54
compare 43
compatibility (of 128) 12
complex interface adapter. *See* CIA chip
CONCAT 5, 177
CONT 28
COPY 5, 178
CP/M 51–60
"Create 128 Autoboot Sector" program 191
"Create 64 Autorun Program" program 193
DATA 34, 40
dBASE program 57
DCLEAR 5, 177
DCLOSE 5, 175
DEC 5
decay rate 83, 84
decimal addresses 46
decoding 40, 41
DEFAULT 144, 146
default volume 82
DELETE 4, 148
DELFIL 24
development system 143
device number 20
DF 113
direct mode 28–33
DIRECTORY 5, 176, 178
disassembler 45

disk controller 22
disk drive number 20
disk drives 12, 17, 20, 53, 165–67
  1541 mode 165, 166, 179
  1571 mode 165–67, 179
disk operating system. *See* DOS
DLIST 144, 147
DLOAD 5, 17, 20, 167, 170, 171, 173, 174
DO-LOOP 6
DOPEN 5, 175, 176
DOS 17, 20, 55, 177
DOS V2.6 25
DOS Wedge 25
DO-UNTIL-LOOP structure 10
DRAW 7, 9
DRIVE NOT READY error 21, 22
DS 177
DS$ 177
DSAVE 5, 17, 20, 173, 174
DUR 77
duration 83
DVERIFY 5, 178
dynamic keyboard technique 28–36, 40, 76, 155
editing 34–36
EL 5, 8
ELSE 6
END 30
ENVELOPE 4, 81, 83–86, 91
EPROM (Erasable-Programmable Read Only Memory) 22
ER 5
ERR$ 5, 8
EXIT 10
FAST 7, 179
fast serial mode 166
FETCH 5
FFST ($C49D) 22
fill 43
FILTER 4, 81, 87, 88, 91
  SID 88
"FILTER Editor" program 92–94
FIND 144, 147
FOR-NEXT loop 6, 37, 41
frequency 76
FRETS 24
FRQ 77
GET 28
GET# 28, 175, 176
GETBUF 22
GETKEY 9
GO 44
GOTO 31
GRAPHIC 9, 66, 178
GRAPHIC CLR 187
GSHAPE 12
HEADER 5
HELP 5
"HELP" program 60
HEX 5
hexadecimal addresses 46, 203
hi-res graphics 7–12
horizontal resolution 8
hunt 44
IF-THEN 6
INPUT 28
INPUT# 28, 175, 176
input buffer 28
INSTR 6
instrument number 83
INT 8
internal buffers 21, 22
internal channel 22
jiffy 6
JOYstick 115
JSR routine 44
jump 44
"Jump Search" program 39
KEY 4
keyboard buffer 29–32
keyboard buffer counter 32
keyboard conventions 198
keyscan values 74
L 170
LEFT$ 77, 90
LEN 90
"Lexitron" program 136–40
LIST 28, 173, 178
"Litter Patrol" program 116–19
LOAD 17, 20, 29, 33, 44, 167, 170, 171
loading a program 170–74
LRUTBL,Y 23
luma 168
MAPOUT 24
"Map with Flag" program 15
mask value 46
MBASIC-80 program 57
megahertz 7
memory 6, 28, 29
MEMORY 44
memory map 46–50
MENU 77
menu, bypassing 190
"Menu" program 192
MERGE 144, 147
"MetaBASIC" program 149–53
MID$ 6, 77, 90
"MLX" program 209–12
mnemonics 45, 144

monitor 43–45, 168, 169
composite 168
monochrome 168
RGBI 169
MONITOR 5
monophonic 75
MOVSPR 4, 113
multivoice music 90
music 74–78, 81–92
"Musical Keyboard" program 78
NEW 88
NewWord program 56
NO CHANNEL error 22
NOP 45
ONEDRV 22
ON-GOTO 31
OPEN 17, 22, 175
operating system (OS) 51
dedicated 51
RAM-based 52
ROM-based 52
transportable 52
OPTSCH 22
OR 5
"Orbitron" program 103–9
PAINT 11
pi character 11
pie chart 10
"Pie Graph" program 15
pitch 76
PLAY 4, 74–78, 81–84, 86, 87, 90
"PLAY Demonstrator" program 96
POLYGON 11
polyphonic 75
PRINT 5, 6, 77, 157
PRINT# 175, 176
PRINT DS,DS$ 5
PRINT FRE(*x*) 6
PRINT USING 5
programming 4, 5
program mode 28–33
programs
compiled 7
interpreted 7
PUDEF 5
pulsewidth 83, 86
PUTBAM 24
question mark 44, 59
QUIT 144, 146, 148
radio frequency. *See* RF
RAM 46
READ 144, 148
READ-DATA pointer 6
RECORD 5
RECORD# 176
registers 5, 44
A 5
P 5
X 5
Y 5
relative file 176
release rate 83, 84
relocating the screen 71
REM 146, 154
"REM Highlighter" program 155, 156
RENAME 5, 177
RENUMBER 4, 177
RESAVE 144, 148
reserved variables 5
RESTORE 6, 9
RESUME 5
RETURN 28, 77, 175
RF (radio frequency) output 168
RIGHT$ 90
ring modulation 85
ROM locations 44, 46, 52
RREG 5
RUN 28, 170, 171, 173, 174
running programs 59
RUN/STOP key 144
RUN/STOP-RESTORE 75, 88, 146
S 174
SAVE 17, 20, 21, 44
SAVE@ 17, 20
"SAVE@ Bug Demonstration" program 25–27
save-with-replace 17–27, 148
saving a program 174
SAVSPR 4
SCALE 9
SCNCLR 9, 10
SCRATCH 5, 17, 21, 147, 177
screen editing 30
SEEK 22
sequential files 175
sequential search 37
serial 166
serial communication bus 166
service request (SRQ) 166
SHIFT-RUN/STOP 170, 171, 178
SID chip. *See* sound interface device
slash (/) 145
SLOW 7
"Song Player" program 79
sorting 37
sound 81–92
SOUND 4, 75–77, 81, 82, 88, 89, 91
"SOUND Editor" program 94–96
sound effects 74–78
sound interface device (SID) 3, 81, 92

"Soundmaker" program 78
sound pattern 84
speed 6, 7
SPRCOLOR 113
SPRDEF 4, 174
SPRITE 4, 113
sprites 3
SPRSAV 113
SSHAPE 4, 12
START 144, 149
STASH 5
STLBUT 22, 23
STOP 44
STR$ 90
SUBMIT 54
SuperCalc program 57
sustain rate 83, 84
SWAP 5, 23
sweep direction 76
"Switchbox" program 132–34
synchronization 85
SYS 5
system utilities 55
TAB 173
TEMPO 4, 75, 81, 83, 91
text editor 54
THEN 114, 115
TO 7
tone generators 81
transfer 44
TRAP 5, 8, 70
TRON 5
tune 83
Turbo Pascal 57
UNNEW 144, 149
user areas 56
utilities 176–78
VAL 90
validate 24
variables 179
VERIFY 178
video displays 168, 169
video outputs 12
VOC 77
voices 81
VOL 76, 81, 82, 88, 91
waveform 76, 81, 83, 85
  noise 85
  pulse 85
  sawtooth 85
  triangle 85
WIDTH 10
WINDOW 63–66
"Window Demo" program 66
windows 63–68
  adding 64
"Window Save for 80 Columns" program 68
"Window Save for 40 Columns" program 67
"Word Counter" program 160, 161
"Word Search" program 121–25
WordStar program 56
writing/debugging tool 143
WUSED 23
XOR 5

To order your copy of *First Book of 128 Disk*, call our toll-free US order line: 1-800-346-6767 (in NY 212-887-8525) or send your prepaid order to:

First Book of 128 Disk
COMPUTE! Publications
P.O. Box 5038
F.D.R. Station
New York, NY 10150

All orders must be prepaid (check, charge, or money order). NC residents add 4.5% sales tax.

Send ____ copies of *First Book of 128 Disk* at $12.95 per copy.

Subtotal $________

Shipping and Handling: $2.00/disk $________

Sales tax (if applicable) $________

Total payment enclosed $________

☐ Payment enclosed
☐ Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No. ________________ Exp. Date ______ (Required)

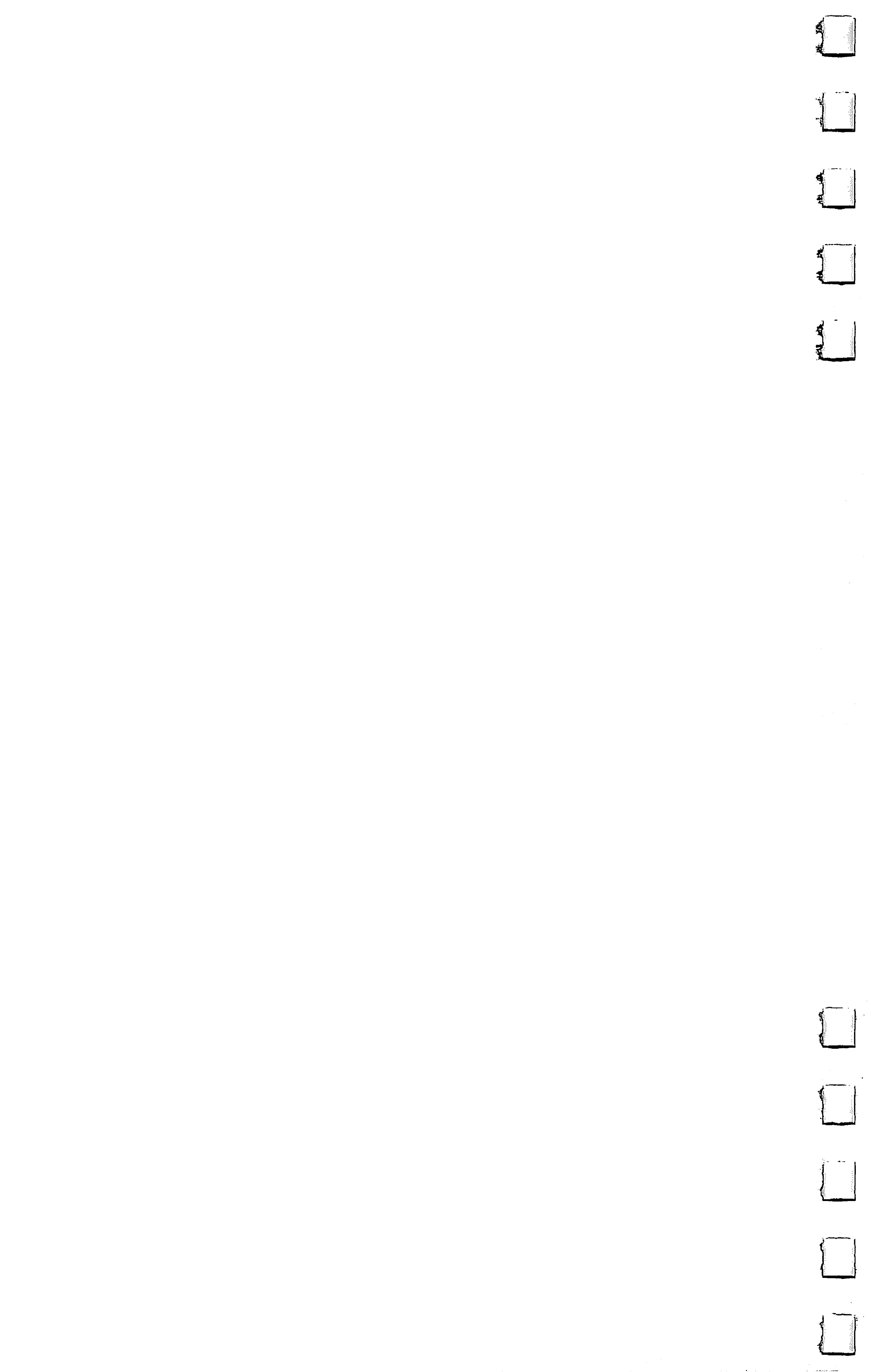
Name ____________________

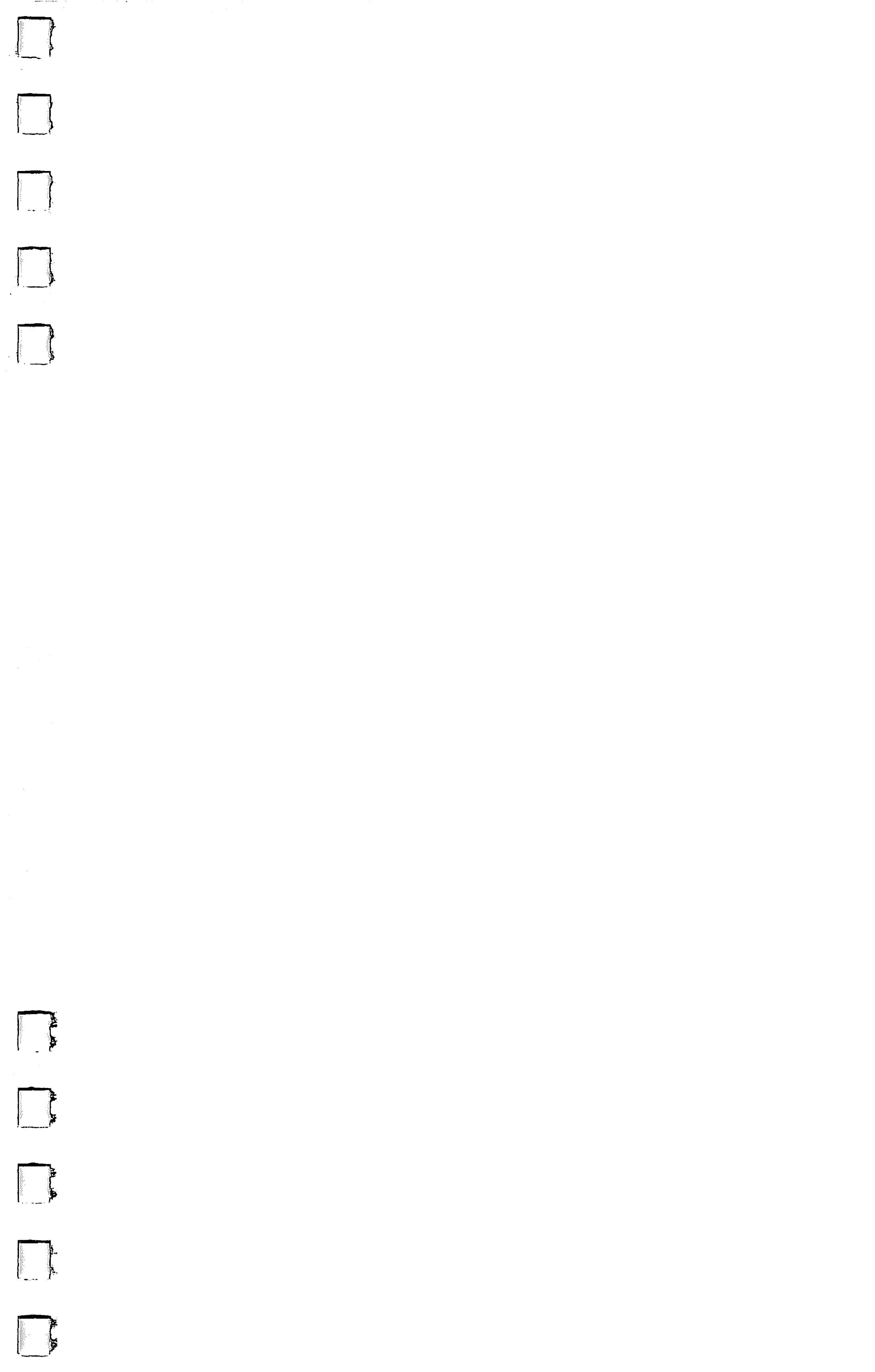
Address ____________________

City ____________ State ______ Zip ______

Please allow 4-5 weeks for delivery.

46559923





COMPUTE!'s First Book of the Commodore 128

The Commodore 128 is the latest and most powerful in a series of popular eight-bit computers from Commodore. The 128 can run all Commodore 64 programs, but it's more than a late-model 64—it's really three computers in one. And since the programs in this book run in 128 mode, *COMPUTE!'s First Book of the Commodore 128* will help you take full advantage of the 128's advanced features.

Since the 128 has a more extensive BASIC than any previous Commodore computer, we've included information and examples so you can utilize the advanced commands in BASIC 7.0. Games like "Litter Patrol" and "Switchbox" show just what can be achieved with 128 BASIC.

COMPUTE!'s First Book of the Commodore 128 offers ready-to-type programs and useful articles for your 128. Here's some of what is included:

- "MetaBASIC," an all machine language utility that adds powerful commands to BASIC 7.0 like Merge, Find, and RESAVE.
- Programs to autoboot disks which permit you to decide which program on the disk should be loaded and run; works with the 1541 or 1571 drive.
- Clear explanations of the 128's disk commands.
- A concise map of important memory locations.
- Example programs that illustrate how to add windows to your programs.
- "Cataloger," which will organize your disk library.
- Articles and sample programs to help you program sound and music.

All the articles in *COMPUTE!'s First Book of the Commodore 128* are written in the clear and concise style you've come to expect from COMPUTE! Publications, the leading publisher of Commodore programs and information. Each program in this book has been fully tested and is ready to enter using our error-checking utility, "The Automatic Proofreader."

ISBN 0-87455-059-9

All the programs in this book are available on a companion disk. See the coupon in the back for details.

COMPUTE!'s First Book of Commodore 128

COMPUTE! Books