COMPUTE!'s Machine Language Routines for the Commodore 64

Dozens of easy-to-use routines and programs which make your Commodore 64 even more powerful and versatile. Includes programming aids, game enhancements, and high-speed graphics utilities.

A **COMPUTE! Books** Publication

$14.95

COMPUTE!'s Machine Language Routines for the Commodore 64

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

The following articles were originally published in *COMPUTE!* magazine, copyright 1983, COMPUTE! Publications, Inc.: "64 Escape Key" (August — originally titled "VIC and 64 Escape Key"); "Ultrasort" (September — originally titled "Ultrasort for Commodore"); "Variable Lister" (November).

The following article was originally published in *COMPUTE!* magazine, copyright 1984, COMPUTE! Publications, Inc.: "Dr. Video 64" (February).

The following articles were originally published in *COMPUTE!'s Gazette*, copyright 1983, COMPUTE! Publications, Inc.: "The Four-Speed Brake" (August); "RAMtest" (August — originally titled "Machine Language for Beginners: The Easy Way"); "Disassembling" (September — originally titled "Machine Language for Beginners: Disassembling"); "64 Searcher" (September); "64 Paddle Reader" (October — originally titled "Improved Paddle Reader Routine"); "Windows and Pages" (October — originally titled "Machine Language for Beginners: Windows and Pages"); "Disk Defaulter" (November — originally titled "VIC/64 Disk Defaulter"); "One-Touch Commands" (November — originally titled "One-Touch Commands for the 64"); "The Assembler" (November — originally titled "Machine Language for Beginners: The Assembler"); "Foolproof INPUT" (December — originally titled "Foolproof INPUT for VIC and 64").

The following articles were originally published in *COMPUTE!'s Gazette*, copyright 1984, COMPUTE! Publications, Inc.: "Auto Line Numbering" (February); "ASCII/POKE Printer" (March — originally titled "ASCII/POKE Printer for VIC and 64"); "Numeric Keypad" (April); "Step Lister" (May); "Scroll 64" (June); "Ultrafont +" (July); "Sprite Magic" (August); "String Search" (August).

The following article was originally published in *COMPUTE!'s First Book of Commodore 64 Games*, copyright 1983, COMPUTE! Publications, Inc.: "Maze Generator."

The following program was originally published in *Creating Arcade Games on the Commodore 64*, copyright 1984, COMPUTE! Publications, Inc.: "Two-Sprite Joystick."

Printed in the United States of America

ISBN 0-942386-48-5

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403 (919) 275-9809, is one of the ABC Publishing Companies and is not associated with any manufacturer of personal computers. Commodore 64 is a trademark of Commodore Electronics Limited.

Contents

Foreword vii

Chapter 1: Introduction 1
Another Toolbox
Gregg Peele 3
The Assembler
Richard Mansfield 6
Disassembling
Richard Mansfield 14
Windows and Pages
Richard Mansfield 21
Stealing Characters from ROM
J.R. Chaffer 26

Chapter 2: Programming Aids 43
BASIC Aid
Brent Anderson and Sheldon Leemon 45
Auto Line Numbering
Jeff Young 69
Numeric Keypad
Charles Kluepfel 71
One-Touch Commands
David W. Martin 74
Dr. Video
David W. Martin 78
Step Lister
E.A. Cottrell 80
Foolproof INPUT
Charles Brannon 83
64 Searcher
John Krause and David W. Martin 87
The Four-Speed Brake
Dan Carmichael 89
ASCII/POKE Printer
Todd Heimarck 91
64 Escape Key
Thomas Henry 97

Variable Lister
E.A. Cottrell 102
Disk Defaulter
Eric Brandon 106

Chapter 3: High-Speed Graphics 109
Ultrafont + Character Editor
Charles Brannon 111
Sprite Magic: An All-Machine-Language Sprite Editor
Charles Brannon 131
The Graphics Package
Chris Metcalf 150

Chapter 4: Game Programming 165
Two-Sprite Joystick
Gregg Peele 167
Scroll 64
Peter Marcotty 171
64 Paddle Reader
Dan Carmichael and Tom R. Halfhill 176
Maze Generator
Charles Bond (Translated to machine language by Gary E. Marsa and for the 64 by Gregg Peele) 178
Multiple-Key Scanner
Chris Metcalf 186

Chapter 5: Applications and Utilities 189
String Search
Glen Colbert 191
Ultrasort
John W. Ross 196
64 Freeze
Dan Carmichael 202
64 Merge
Harold D. Vanderpool 204
RAMtest
Richard Mansfield 209

Appendices 213
A: A Beginner's Guide to Typing In Programs 215
B: How to Type In Programs 217
C: The Automatic Proofreader
Charles Brannon 219
D: Using the Machine Language Editor: MLX
Charles Brannon 223
E: The 6502 Instruction Set 230
F: Number Tables 246

Index 253

Foreword

Machine language. If you're a BASIC programmer, these words may seem mysterious. Perhaps intimidating. After all, the best programs — from word processors to arcade games — are written in it. But why is machine language so special?

Your Commodore 64 doesn't really speak in BASIC. You may be well versed in that programming language, but your computer isn't. It has to look up everything you type in and translate it into the language it does understand — machine language (ML).

ML is simply a series of 0's and 1's. Bits off or on. When you type something into the computer — LET A = 2 + 2, for example — the computer has to use its internal dictionary to look up what that means in numbers. Then it has to return the answer to you in a way you can understand.

This constant referring and translating, called BASIC, is just not as fast as machine language. If you could somehow speak to your 64 in its native tongue, ML, you would be able to execute commands and programs a hundred, even a thousand, times faster.

Fortunately, you don't have to know how to write machine language programs to use them. BASIC and ML can work together. The routines and programs in this book can be added to your own BASIC programs. All you have to do is type them in.

There are several routines which make it easier to write your own BASIC programs, from automatically adding line numbers to slowing down listings on the screen. "BASIC Aid," an all machine language program, gives you 20 tools that simplify BASIC programming. Graphics utilities let you create custom characters, sprites, or impressive designs, all at machine language speed. If you enjoy writing games, you'll find the ML joystick, paddle, and keyboard routines helpful. You can insert other routines to make your screen scroll horizontally or vertically, or use the maze generator to create random mazes almost instantly. Other applications let you freeze the display, sort thousands of things quickly, merge files, search for strings, and even test your computer's RAM chip.

You don't need to know how or why machine language works to use these utilities. As long as you know BASIC, you'll be able to speed up, simplify, and amplify your own programs with the routines in this book. What once took minutes can take only seconds with ML. You'll be amazed at how powerful your programs can become.

Chapter 1
Introduction

Gregg Peele

Another Toolbox

Machine language is fast and powerful. This simple introduction helps you understand what machine language can and should do. There's even a short routine which compares the speeds of BASIC and machine language.

Computers are fast. I'm sure you've heard that before. The processor inside your Commodore 64, called the 6510 chip, takes only one millionth of a second (one microsecond) to complete one work cycle. The computer can process over one hundred thousand instructions in one second. Speedy.

Unfortunately, the 64 is only that quick when it's executing instructions written in its native tongue, machine language (ML). Machine language is essentially a series of numbers stored in the computer's memory. Remembering those numbers, and what they do, is sometimes hard for humans. That's why most of us use a different language to speak to the computer. Called BASIC, it's much slower because it has to translate what you enter into code that the computer can understand. There's nothing inherently wrong with this; if you didn't have BASIC to use, you would have to talk to the 64 in machine language. BASIC may be easier for the novice programmer to remember, but it *is* slower.

Quick Clear

To see the difference in speed between BASIC and ML, let's look at two programs, both of which do the same thing. The high-resolution screen on the Commodore 64 contains 8000 bytes of information. Clearing that screen with a BASIC program takes at least 30 seconds. Here's what the BASIC version would look like:

```
5 PRINT"{CLR}"
10 POKE 53272,PEEK(53272)OR8
20 POKE 53265,PEEK(53265)OR32
30 FOR T=8192 TO 8192+8000:POKE T,0:NEXT
40 GET A$:IF A$="" THEN 40: REM HIT SPACE TO CONTI
   NUE
50 POKE 53265,PEEK(53265)AND223
60 POKE 53272,21
```

Type it in and RUN it. It does the job, but it's slow. Using machine language, however, clears the screen almost instantly. Type in the following program, RUN it, then enter SYS 49152 to see the quickness of ML.

```
10 I=49152:IFPEEK(49152)=169THENSYS49152:END
20 READ A:IF A=256 THEN SYS49152:END
30 POKE I,A:I=I+1:GOTO 20
49152 DATA 169,147,32,210,255,173,24
49159 DATA 208,9,8,141,24,208,173
49166 DATA 17,208,9,32,141,17,208
49173 DATA 169,0,168,133,252,169,32
49180 DATA 133,253,169,0,145,252,200
49187 DATA 208,249,230,253,165,253,201
49194 DATA 64,144,241,165,197,201,60
49201 DATA 208,250,173,17,208,41,223
49208 DATA 141,17,208,169,21,141,24
49215 DATA 208,96,256
```

Speed is often essential to a program. Arcade games which need fast action and smooth animation use ML to add realism to movement and to provide precise player control. Most professional word processors are written in machine language, since moving text in BASIC is especially slow. And many short routines, such as the ones in this book, use ML's speed to enhance particular aspects of programs written in BASIC.

Finding the Right Mix

Even though machine language is fast, it's sometimes more appropriate to use another language. Programs which include complex formulas, or which don't need ML's speed, can sometimes best be written in BASIC. Such programs may involve complex bit-manipulation techniques that are easier or faster to write in BASIC than in machine language. In fact, it's often most efficient to combine the features of BASIC and ML. You can use BASIC for sections where speed is not vital, and ML for parts which require quick execution. Using the SYS command, you can call these machine language routines when they're needed. For instance, the high-resolution screen-clearing routine might be used to accompany a BASIC graphics program.

Since the operating system of your 64 is written in machine language, there are several routines already included in the BASIC ROM that you can use for your own programs. Some of these routines print to the screen or other device, move the cursor to any place on the screen, or save/load to disk or tape. Many programmers use these routines as part of their programs, saving them considerable time and effort. Several of the programs in this book do just that.

Being able to mix BASIC and ML has other advantages. A machine language program can add new features and commands to BASIC.

These additions can speed up programming in BASIC by providing high-speed line renumbering, search and replace functions, and other helpful utilities. Take a look at the "Programming Aids" section for some good examples of these kinds of routines.

Machine language can also help you create your own ML programs. To make it easier to write machine language, a series of *mnemonics*, each representing a certain operation, helps you keep track of what you write. A program which reads these mnemonics and changes them to numbers (which, after all, is what the computer wants to see) is a special kind of ML program called an assembler.

Assemblers are important tools to ML programmers, since they allow you to use easy-to-remember symbols instead of hard-to-recall numbers. A simple assembler is included in this book, but a more complex one can be found in Richard Mansfield's book, *The Second Book of Machine Language*. There are also commercially available assemblers for the 64 on the market.

Tools

This book is not meant to teach you how to write machine language. There are other books for that. You'll find something different here: powerful machine language tools. These tools supplement BASIC with added commands and features, or replace functions of BASIC with speedier ML versions. Some of the tools are written in machine language and are meant to be used by themselves. Others are in the form of a BASIC loader which POKEs machine language data into memory. Most of these can be easily added to your own BASIC programs. Whatever the form, you'll find these ML utilities fast, powerful, and versatile. Just what a tool should be.

Richard Mansfield

The Assembler

One of the basic tools of machine language (ML) programming is an assembler. Richard Mansfield, senior editor of COMPUTE! Publications, offers his assembler along with some elementary explanations and examples on how it's used.

People often use the words *machine language* and *assembly language* interchangeably. However, *machine language* is becoming the more common term. It's more accurate — when you program in this language, you're speaking directly to your computer in its native tongue.

Unfortunately, the computer's internal language is almost impossible for humans to work with. These machines communicate only with numbers, and very odd numbers at that. They're binary, consisting of only 1's and 0's, grouped together in eight-digit clusters called *bytes*: 01100111, 11110001, and so on. Humans find it easier to work with words. That's where an *assembler* comes in.

The Primary Tool

We first need to build the basic tool for machine language (ML) programming. Type in the program and you'll have your own working assembler.

The assembler works like this: You type in a wordlike, three-letter code, and the assembler looks up the correct number (in the computer's language) and POKEs it into RAM memory to start forming an ML program. In a minute we'll create a simple ML program to show you how ML programming is done. But let's clear up a few possible sources of confusion first.

These wordlike codes are called *mnemonics*, which means they've been designed to be simple to remember. It's easy enough to remember what USA stands for. Likewise, you can quickly pick up the essential ML words. There are 56 of these commands available to you, roughly as many words as there are in BASIC. But, like BASIC, there is a core group of about 20 important ones. They are the only ones you need to use to get almost anything accomplished. What's more, the ML words *are* easy to learn and remember. For example, BRK stands for BReaK (like BASIC's STOP), JSR is Jump to SubRoutine (GOSUB), and RTS is ReTurn from Subroutine (RETURN). The command which does the same thing as BASIC's GOTO is called JMP, for JuMP.

A Kind of Swing

ML programming involves a kind of swing between Command and Target. First you give a command, then you give the specific target for that command. Then another command, another target. These paired-event phenomena are called by many names and appear in many disguises in programming as well as in real life. They're called Operator/Operand, Instruction/Argument, Mnemonic/Address, Analyst/Analysand, Shopper/Apples, Thief/Victim.

Notice that the first half of the pair is the more general, the second more specific. At a given moment, the apple is the specific thing the shopper's involved with, but the shopper will be buying other things during this visit to the store. Similarly, a thief is always a thief, but a victim is a victim only that once (we hope). Also, the transaction which all these pairs have in common is that the first half of the pair is *doing something* to the second half. Together they form a complete action in the sense that Open/Envelope or Eat/Peach are paired (command/target) actions.

A Robot Dinner

If you think about it, you can see this do-it-to-it rhythm throughout BASIC programming: PEEK (8), PRINT "HELLO", SAVE "PROGRAM", X = 15, X = X + 1, GOTO 1500, and so on. The reason we're stressing this distinction, this rhythmic swing between actor and acted-upon, is because an ML program is constructed in precisely this way — you make a list of tiny, elementary actions for the computer to later carry out. It's like a robot dinner: spear/meat, raise/arm, insert/food, chew/morsel, lower/arm, spear .... List enough of these mini-instructions and you can do amazing things.

One result of all this is that an ML program doesn't look like a BASIC program. BASIC tends to spread these pairs out along a line:

```
100 Y=3:X=X+1:POKE 63222,Y:Y=PEEK(1200)
```

ML lists each tiny action-pair on its own line:

```
100 LDY #3
110 INX
120 STY 63222
130 LDY 1200
```

These two programs are doing exactly the same thing, but in different ways. STY and LDY mean STore Y and LoaD Y (it's like a variable in BASIC). INX means INcrement X (raise it by one). The # sign means to think of the number as *literally* the number three, not

address three. Without the #, the computer assumes you mean a memory location.

Take a look at the mnemonics here. They're all three-letter words. They are always the first thing on each line. And they usually have their target right next to them (the INX doesn't because the mnemonic itself already contains the specific information required). The other half of the pair, those numbers, is called *addressing modes* in ML. In In general, that's because numbers are usually being sent to and from addresses in the computer's memory while an ML program is running. That, plus simple arithmetic, is the essence of what a computer does to accomplish any given task.

We won't get into the addressing modes (there are about ten) right now, but you can already recognize two of them: Line 100's mode is called *immediate addressing* (the number is immediately after the instruction, not in some memory location elsewhere in the computer), and line 110's mode is called *implied addressing* (because the instruction contains its own target).

Putting the Assembler to Work

Enough theory, let's do something. Let's assemble a small program. If you've typed in the program, the first thing to do is to change line 10 so that the assembler will accept ordinary decimal numbers. It's designed to work with either decimal or hexadecimal, but we've not yet touched on hex so we'll stick with the familiar. Change the line to:

```
1Ø H=Ø
```

Then RUN the assembler and type in 830 when it asks you where you want to put the ML program. That's a safe place until you next load in a program from cassette. ML can be put into a variety of places in RAM. BASIC, of course, has a computer-determined starting location in memory, but *you* specify the start of an ML program. Now you'll see that address printed onscreen. The addresses where the instructions are being stored will function as the "line numbers" for your reference when programming. Unlike BASIC, you can't go back up and change a line. If you make a mistake, start over. (There are easier ways to fix errors, but that, too, is more complicated, so we'll stick to simpler methods for now.)

Now type LDY #0, hit RETURN, and you've written a line of ML which will put a zero into the Y register. (You'll see the numbers forming the ML version of your program appear to the right of the mnemonic/address you've typed.) Then the assembler will furnish

you with the next available "line number" address in RAM, 832. The mnemonic/address pair LDY #0 uses up two bytes.

You are ready to type in your next pair: LDA #66. Hit RETURN on this line and you've put the code for the letter B into the A register. Then type in the rest of our ML program, one pair per line:

JSR 65490
DEY
BNE 834
RTS

That's it. To let the assembler know that you're through with your program, type END instead of a normal mnemonic and it will tell you the start and end addresses of your ML program. Then, having done its job, the assembler quits. The mnemonics and addresses were all POKEd into their proper places after being translated into the machine's language. To see what happens when this RUNs, you can type SYS 830 and see the effect of the small ML loop we wrote. You'll get 256 B's onscreen in record time. Not something you've been anxious to do? More useful things are on their way.

In the next few chapters we'll look at some other aspects of machine language, including a disassembler that you can use to pull apart ML programs. You'll see how a simple ML program is created and then find valuable utilities that will allow you to use the speed and power of machine language in your own programs.

You're on your way to using machine language.

The Assembler

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

```
10 H=1:REM IF H = 0 THEN ASSEMBLY IS IN DECIMAL
                                          :rem 42
50 HE$="0123456789ABCDEF":SZ=1:ZO$="000"  :rem 166
100 PRINT"{3 SPACES}SIMPLE{3 SPACES}ASSEMBLER
    {2 SPACES}CONVENTIONS:"               :rem 90
110 DIMM$(56),TY(56),OP(56)               :rem 181
120 FORI=1TO56:READM$(I)                  :rem 160
122 ROP$=MID$(M$(I),4,1):TY(I)=VAL(ROP$)    :rem 5
124 OP$=RIGHT$(M$(I),3):OP(I)=VAL(OP$)    :rem 155
126 M$(I)=LEFT$(M$(I),3)                  :rem 235
140 NEXTI: PRINT                          :rem 228
150 PRINT"IMMEDIATE{5 SPACES}LDA #15       :rem 46
155 PRINT"ABSOLUTE{6 SPACES}LDA 1500       :rem 64
160 PRINT"ZERO PAGE{5 SPACES}LDA 15       :rem 218
165 PRINT"ACCUMULATOR{3 SPACES}ASL        :rem 107
170 PRINT"INDIRECT X{4 SPACES}LDA (15X)   :rem 209
175 PRINT"INDIRECT Y{4 SPACES}LDA (15)Y   :rem 216
```

```
177 PRINT"ZERO PAGE X{3 SPACES}LDA 15X     :rem 146
179 PRINT"ZERO PAGE Y{3 SPACES}LDX 15Y     :rem 173
180 PRINT"ABSOLUTE X{4 SPACES}LDA 1500X    :rem 238
185 PRINT"ABSOLUTE Y{4 SPACES}LDA 1500Y    :rem 245
189 PRINT:PRINT"{4 SPACES}ENTER ALL NUMBERS IN ";
                                           :rem 127
190 IFH=1 THENPRINT"HEX":GOTO200           :rem 201
195 PRINT"DECIMAL"                          :rem 95
200 PRINT:PRINT"INPUT STARTING ADDRESS FOR ML PROG
    RAM:":INPUTSA$                         :rem 128
210 IFH=1THENH$=SA$:GOSUB5000:SA=DE:GOTO220
                                           :rem 130
215 SA=VAL(SA$)                             :rem 85
220 TA=SA:PRINT"{CLR}":REM CLEAR THE SCREEN
                                           :rem 190
230 IFH=1THENDE=SA:SZ=3:GOSUB4000:PRINTH$;:GOTO240
                                           :rem 175
235 PRINTSA" ";                             :rem 58
240 INPUTMN$:PRINT"{UP}"SPC(20);:REM GO UP ONE LIN
    E AND OVER 20 SPACES                   :rem 232
241 REM ADD NEW PSEUDO-OPS HERE             :rem 65
242 IFRIGHT$(MN$,7)="FORWARD"THENFB=SA      :rem 90
243 IFRIGHT$(MN$,7)="RESOLVE"THENFR=SA-FB:POKEFB+1
    ,FR-2:PRINT"{2 SPACES}OK":GOTO230      :rem 72
244 IFRIGHT$(MN$,4)="POKE"THEN246          :rem 182
245 GOTO 250                               :rem 107
246 PRINT"ADDR,NUMBER(DEC)";:INPUTADR,NUM:POKEADR,
    NUM:GOTO230                            :rem 246
250 IFMN$="END"THENPRINT:PRINT"{6 SPACES}PROGRAM I
    S FROM"TA"TO"SA:END                     :rem 13
260 L=LEN(MN$):L$=LEFT$(MN$,3)             :rem 181
270 FORI=1TO56:IFL$=M$(I)THEN300           :rem 136
280 NEXTI                                   :rem 34
290 GOTO850                                :rem 113
300 REM PRIMARY OPCODE CATEGORIES           :rem 59
301 TY=TY(I):OP=OP(I)                       :rem 20
305 IFFB=SATHENTN=0:GOTO2010               :rem 244
310 IFTY=0THENGOTO1000                     :rem 102
320 IFTY=3THENTY=1:IFL=3THENOP=OP+8:GOTO1000
                                           :rem 81
330 R$=RIGHT$(MN$,L-4):IFH=1THENGOSUB6000 :rem 200
340 LR$=LEFT$(R$,1):LL=LEN(R$):IFLR$="#"THEN480
                                           :rem 184
350 IFLR$="("THEN520                        :rem 88
360 IFTY=8THEN600                           :rem 15
370 IFTY=3THENOP=OP+8:GOTO1000             :rem 135
380 IFRIGHT$(R$,1)="X"ORRIGHT$(R$,1)="Y"THEN630
                                           :rem 210
390 IFLEFT$(L$,1)="J"THEN820                :rem 44
```

```
400 TN=VAL(R$):IFTN>255THEN430                    :rem 40
410 IFTY=1ORTY=3ORTY=4ORTY=5THENOP=OP+4          :rem 133
420 GOTO2000                                      :rem 145
430 H%=TN/256:L%=TN-256*H%:IFTY=2ORTY=7THENOP=OP+8
    :GOTO470                                      :rem 92
440 IFTY=1ORTY=3ORTY=4ORTY=5THENOP=OP+12:GOTO470
                                                  :rem 197
450 IFTY=6ORTY=9THEN470                           :rem 214
460 GOTO850                                       :rem 112
470 GOTO3000                                      :rem 151
480 TN=VAL(RIGHT$(R$,LL-1))                       :rem 58
490 IFTY=1THENOP=OP+8:GOTO2000                    :rem 137
500 IFTY=4ORTY=5THENGOTO2000                      :rem 44
510 GOTO850                                       :rem 108
520 IFRIGHT$(R$,2)=")Y"THEN540                    :rem 184
530 IFRIGHT$(R$,2)="X)"THEN570                    :rem 187
540 TN=VAL(MID$(R$,2,LL-3))                       :rem 243
550 IFTY=1THENOP=OP+16:GOTO2000                   :rem 181
560 GOTO850                                       :rem 113
570 TN=VAL(MID$(R$,2,LL-3))                       :rem 246
580 IFTY=1THENGOTO2000                            :rem 113
590 GOTO850                                       :rem 116
600 TN=VAL(R$):TN=TN-SA-2:IFTN<-128ORTN>127THENPRI
    NT"TOO FAR ";:GOTO850                         :rem 154
610 IFTN<0THENTN=TN+256                           :rem 172
620 GOTO2000                                      :rem 147
630 IFRIGHT$(R$,2)=")Y"THEN540                    :rem 186
640 IFRIGHT$(R$,1)="X"THEN720                     :rem 144
650 REM *ZERO Y                                   :rem 66
660 TN=VAL(LEFT$(R$,LL-1)):IFTN>255THEN680:rem 249
670 IFTY=2ORTY=5THEN730                           :rem 209
675 IFTY=1THEN760                                 :rem 24
680 GOSUB770:IFTY=1THENOP=OP+24:GOTO710          :rem 230
690 IFTY=5THENOP=OP+28:GOTO710                    :rem 151
700 GOTO850                                       :rem 109
710 GOTO3000                                      :rem 148
720 TN=VAL(LEFT$(R$,LL-1)):IFTN>255THENGOSUB770:GO
    TO780                                         :rem 136
730 IFTY=2THENOP=OP+16:GOTO760                    :rem 145
740 IFTY=1ORTY=3ORTY=5THENOP=OP+20:GOTO760 :rem 10
750 GOTO850                                       :rem 114
760 GOTO2000                                      :rem 152
770 H%=TN/256:L%=TN-256*H%:RETURN                 :rem 187
780 IFTY=2THENOP=OP+24:GOTO810                    :rem 145
790 IFTY=1ORTY=3ORTY=5THENOP=OP+28:GOTO810 :rem 19
800 GOTO850                                       :rem 110
810 GOTO3000                                      :rem 149
820 TN=VAL(R$)                                    :rem 35
830 GOSUB770                                      :rem 185
```

```
840 GOTO710                                          :rem 109
850 PRINT"{RVS} ERROR ":GOTO230                       :rem 18
1000 REM 1 BYTE INSTRUCTIONS                         :rem 191
1010 POKESA,OP:SA=SA+1:IFH=1THEN 1030                :rem 189
1020 PRINTOP:GOTO230                                 :rem 247
1030 DE = OP:GOSUB4000:PRINTH$:GOTO230               :rem 226
2000 REM 2 BYTE INSTRUCTIONS                         :rem 193
2005 IFTN>255THENPRINT" INCORRECT ARGUMENT. (#5 IN
     HEX IS #05)":GOTO230                            :rem 93
2010 POKESA,OP:POKESA+1,TN:SA=SA+2:IFH=1THEN2030
                                                     :rem 231
2020 PRINTOP;TN:GOTO230                              :rem 213
2030 DE = OP:GOSUB4000:PRINTH$" ";                    :rem 90
2040 DE = TN:GOSUB4000:PRINTH$:GOTO230               :rem 231
3000 REM 3 BYTE INSTRUCTIONS                         :rem 195
3010 POKESA,OP:POKESA+1,L%:POKESA+2,H%:SA=SA+3:IFH
    =1THEN3030                                       :rem 172
3020 PRINTOP;L%;H%:GOTO230                            :rem 77
3030 DE = OP:GOSUB4000:PRINTH$" ";                    :rem 91
3040 DE = L%:GOSUB4000:PRINTH$" ";                    :rem 46
3050 DE = H%:GOSUB4000:PRINTH$:GOTO230               :rem 180
4000 REM{2 SPACES}DECIMAL TO HEX (DE TO H$) :rem 8
4010 H$="":FORM=SZTO0STEP-1:N%=DE/(16↑M):DE=DE-N%*
    16↑M:H$=H$+MID$(HE$,N%+1,1)                      :rem 179
4020 NEXT:SZ=1:RETURN                                :rem 116
5000 REM{2 SPACES}HEX TO DECIMAL (H$ TO DE) :rem 9
5010 D=0:Q=3:FORM=1TO4:FORW=0TO15:IFMID$(H$,M,1)=M
    ID$(HE$,W+1,1)THEN5030                           :rem 221
5020 NEXTW                                            :rem 93
5030 D1=W*(16↑(Q)):D=D+D1:Q=Q-1:NEXTM:DE=INT(D):RE
    TURN                                             :rem 41
6000 REM ACCEPT HEX OPCODE INPUT AND TRANSLATE IT
    {SPACE}TO DECIMAL                                :rem 57
6010 IFLEFT$(R$,1)="#"THENH$="00"+RIGHT$(R$,2):GOS
    UB5000:R$="#"+STR$(DE):RETURN                    :rem 234
6020 LS=LEN(R$):AZ$=LEFT$(R$,1):ZA$=MID$(R$,LS,1):
    IFAZ$<>"("THEN6050                               :rem 126
6030 IFZA$="Y"THENH$="00"+MID$(R$,2,2):GOSUB5000:R
    $="("+STR$(DE)+")Y":RETURN                       :rem 30
6040 IFZA$=")"THENH$="00"+MID$(R$,2,2):GOSUB5000:R
    $="("+STR$(DE)+"X)":RETURN                       :rem 238
6050 IFZA$="X"ORZA$="Y"THEN6070                       :rem 40
6060 H$=LEFT$(ZA$,4-LS)+R$:GOSUB5000:R$=STR$(DE):R
    ETURN                                            :rem 30
6070 IFLS=5THENH$=LEFT$(R$,4):GOTO6090               :rem 253
6080 H$="00"+LEFT$(R$,2)                             :rem 186
6090 GOSUB5000:R$=STR$(DE)+ZA$:RETURN                :rem 252
20000 DATAADC1097,AND1033,ASL3002,BCC8144,BCS8176,
     BEQ8240,BIT7036,BMI8048                         :rem 96
```

```
20010 DATABNE8208,BPL8016,BRK0000,BVC8080,BVS8112,
      CLC0024,CLD0216,CLI0088                :rem 114
20020 DATACLV0184,CMP1193,CPX4224,CPY4192,DEC2198,
      DEX0202,DEY0136,EOR1065                :rem 184
20030 DATAINC2230,INX0232,INY0200,JMP6076,JSR9032,
      LDA1161,LDX5162,LDY5160                :rem 200
20040 DATALSR3066,NOP0234,ORA1001,PHA0072,PHP0008,
      PLA0104,PLP0040,ROL3034                :rem 185
20050 DATAROR3098,RTI0064,RTS0096,SBC1225,SEC0056,
      SED0248,SEI0120,STA1129                :rem 216
20060 DATASTX2134,STY2132,TAX0170,TAY0168,TSX0186,
      TXA0138,TXS0154,TYA0152                 :rem 79
```

Richard Mansfield

Disassembling

Knowing how to assemble machine language (ML) programs isn't enough; you also need to know how to convert the numbers the computer uses into things you can understand. A disassembler does this. The simple-to-use disassembler included here makes ML programs less confusing and easier to study.

A *disassembler* is the second of two major tools you'll be using when you work with machine language. In order to understand what it does, we'll need to briefly review the other major tool, an *assembler*, which was described in the previous section.

An assembler is used to write an ML (machine language) program in the same way that BASIC is used to write a BASIC program. An assembler lets you type in ML instructions like LDA #8 and then translates the instructions into numbers and POKEs them into memory for you. Take a look at Program 3. The first line, numbered 884, says LDA (LoaD the Accumulator) with the number eight. This same instruction appears in different form in line 882 of Program 2: DATA 169,8. An assembler would translate your LDA instruction into the number 169. If you're just starting out with ML, these instructions won't mean much to you yet, but for now all we want to do is get a feel for the broad concepts of ML.

To look at "assembling" another way, it helps to realize that there's a similar process going on when you write a BASIC program. After you type in a BASIC command, the BASIC interpreter translates it into a *token*, a single-byte representation of the command, and stores the token in memory. So, a line of BASIC is stored inside the computer in a different form than you would see on the screen when you type it in. The *word* LIST would be stored in four bytes, but the *command* LIST would be crunched down by BASIC into only one byte. Similarly, an assembler takes your LDA and turns it into the number 169, which can be stored in a single byte. These words — LDA and LIST — are for our convenience. They are easier for us to work with. The computer only needs numbers and so BASIC and its ML equivalent, an assembler, accepts the words, but stores numbers.

An Understandable Version

Of course, you need to go the opposite way sometimes, from the numbers back to the words. If the computer stores, interprets, and executes programs as pure numbers, how can we examine or

modify a program? We don't want to study a list of numbers, however efficient they are for the computer's internal use. Program 2 is a good example of this. Program 1, a disassembler, does for ML what the LIST command does for BASIC programs. It takes a look at a compressed, numeric, machine-readable program in memory and prints out an understandable, human-readable version.

Let's look at an example. A fragment of a program which puts every possible number (0–255) into every memory cell in your computer's RAM memory appears in Programs 2 and 3 here. Type in and SAVE Program 1, then type in and RUN Program 2. Next, LOAD Program 1 again, and when the disassembler asks you for START ADDRESS, type 884. That's the address where the fragment starts in RAM memory. You'll then see your screen fill with the disassembly of the ML fragment. It should look something like Program 3.

At this point, you will probably find it difficult to understand this disassembly listing. As you begin to learn the meaning of ML instructions, however, the purpose of this fragment will become clear. As a quick explanation: Line 898 copies a number from the accumulator into a cell in RAM memory. Then line 900 compares the RAM memory cell against the accumulator. If they are the same (BEQ means Branch if EQual), we are sent down to lines 925 and 926, where the number in the accumulator is raised by one. We go back and test the same cell over and over, raising the number each time so we can see if that cell will hold all the possible numbers.

Just the way that IF-THEN tests in BASIC, if we had a bad memory cell and the number was found to be *not* equal in line 900, we would "fall through" the BEQ to line 904, which would print out an error message on the screen to alert us about the bad memory. By the way, we've been calling the numbers on the left side of Program 3 "line numbers." In fact, they're memory addresses where the instructions were found in RAM. It's useful, though, to think of them as similar to BASIC's line numbers. They serve the same purpose.

Don't be concerned if this is difficult to follow. We're jumping into ML to get our feet wet. It's likely that you learned BASIC the way I did: by working with the language and making lots of mistakes and not fully understanding what was going on at first. So we'll plunge into ML by starting off with the main tools, the assembler and the disassembler. You won't be able to use them with very much skill to begin with, but just working with them is probably the fastest way to learn.

Trying It Out

We can conclude with a few comments about the disassembler. There are several graphics features of this disassembler which can make it easier to visualize the programs it disassembles. All branching instructions (like BEQ), JSRs, and JMPs (the equivalents of GOSUB and GOTO) are offset on the screen to indicate that the flow of the program might be taking a new course at that point. Likewise, the RTS instruction (ReTurn from Subroutine, the equivalent of BASIC's RETURN) causes a line to be drawn, marking the end of a subroutine.

Line 210 of Program 1 PEEKs the ML command from memory. If it cannot make a match against the array containing all legal ML instructions (lines 820–960), a ? is printed on screen in line 230. When you see a series of question marks during a disassembly, it means that you are not disassembling an ML program, but rather have come across a "data table." This would be a list of numbers or words which might be *used* by an ML program, but is not actually ML code.

You can use the disassembler to look into the heart of your BASIC language. Just give an address between 40960–49151 as the START ADDRESS, and you can see the insides of one of the most complex ML programs ever written: your BASIC. The next article will show you how to go directly into BASIC and access some of its ML subroutines.

Program 1. A Disassembler

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

```
100 HE$="0123456789ABCDEF"                    :rem 101
110 L$="----------------------------------------":J
    $="{4 SPACES}--->"                         :rem 76
120 PRINT"{2 SPACES}DISASSEMBLER":PRINT:DIMM$(15,1
    5)                                         :rem 119
130 FORI=0TO15:FORB=0TO14:READM$(I,B):NEXTB:NEXTI
                                               :rem 94
140 REM START MAIN LOOP                        :rem 102
150 PRINT"STARTING ADDRESS (DECIMAL)";:INPUTSA:TA=
    SA                                         :rem 82
160 PRINT"START ADDRESS HEX{2 SPACES}";:DE=SA:ZX=3
    :GOSUB970:PRINTH$"{2 SPACES}"              :rem 221
170 IFSA<0THENEND                              :rem 45
180 I=SA                                       :rem 179
190 REM PRINT ADDRESS                          :rem 17
200 PRINTI" ";                                 :rem 231
210 X=PEEK(I)                                  :rem 231
220 GOSUB1040                                  :rem 217
230 IFL%=15ORM$(H%,L%)="0"THENPRINT" ?{5 SPACES}"X
    :CK=0:LN=LN+1:GOTO260                      :rem 52
```

```
240 PRINTM$(H%,L%);                                    :rem 42
250 GOSUB1070:IFEQTHENEQ=0                            :rem 112
260 I=I+1                                              :rem 195
270 IFLN=20THENLN=0:GOTO1000                           :rem 203
280 GOTO190                                            :rem 109
290 IFCK=12THEN320                                      :rem 28
300 B=PEEK(I+1):IFB>127THENB=((NOTB)AND255)+1:B=-B
                                                       :rem 134
310 BAD=I+2+B:PRINT"{8 SPACES}"BAD:I=I+1:RETURN
                                                       :rem 252
320 IFH%>8THEN730                                      :rem 208
330 IFH%=2THENPRINT"{6 SPACES}";:J=1:GOTO750
                                                        :rem 61
340 IFH%=6THENPRINT:PRINTL$:EQ=1:RETURN                 :rem 81
350 IFH%=6THENRETURN                                    :rem 22
360 PRINT                                               :rem 38
370 RETURN                                             :rem 122
380 IFCK=12THEN410                                      :rem 28
390 PRINT" ("PEEK(I+1)"),Y"                            :rem 162
400 I=I+1:RETURN                                       :rem 217
410 PRINT" ("PEEK(I+1)",X)"                            :rem 154
420 I=I+1:RETURN                                       :rem 219
430 IFCK=12THEN460                                      :rem 29
440 PRINT" "PEEK(I+1)",X"                               :rem 76
450 I=I+1:RETURN                                       :rem 222
460 PRINT"{2 SPACES}"PEEK(I+1)                         :rem 134
470 I=I+1:RETURN                                       :rem 224
480 IFCK=12THEN510                                      :rem 30
490 PRINT" "PEEK(I+1)",X"                               :rem 81
500 I=I+1:RETURN                                       :rem 218
510 PRINT"{2 SPACES}"PEEK(I+1)                         :rem 130
520 I=I+1:RETURN                                       :rem 220
530 IFCK=12THEN510                                      :rem 26
540 IFH%=9ORH%=11THENPRINT" "PEEK(I+1)",Y":rem 156
550 IFH%=7ORH%=15ORH%=5ORH%=3THEN480                   :rem 132
560 IFH%=13THEN440                                     :rem 255
570 PRINT:GOTO500                                       :rem 49
580 PRINT: RETURN                                       :rem 68
590 IFCK=12THEN730                                      :rem 36
600 I$="Y":GOTO750                                     :rem 236
610 IFCK=12THEN630                                      :rem 28
620 I$="X":GOTO750                                     :rem 237
630 IFH%=6THENPRINT"{6 SPACES}";:I$="Z":GOTO750
                                                       :rem 212
640 IFH%=2THEN750                                      :rem 208
650 IFH%=4THENPRINTJ$;:GOTO750                         :rem 124
660 IFH%=8ORH%=10ORH%=12ORH%=14THEN750                 :rem 226
670 GOTO380                                            :rem 113
680 IFCK=12THEN750                                      :rem 38
```

```
690 I$="X":GOTO750                                    :rem 244
700 IFCK=12THEN750                                    :rem 31
710 IFH%=11THENI$="Y":GOTO750                         :rem 184
720 I$="X":GOTO750                                    :rem 238
730 PRINT"{3 SPACES}#"PEEK(I+1)                       :rem 169
740 I=I+1:RETURN                                      :rem 224
750 N=PEEK(I+1)+PEEK(I+2)*256                         :rem 80
760 IFI$=""THEN800                                    :rem 225
770 IFI$="X"THENPRINT"{2 SPACES}"N",X"                :rem 137
780 IFI$="Y"THENPRINT"{2 SPACES}"N",Y"                :rem 140
785 IFI$="Z"THENPRINT" ("N")"                         :rem 94
790 I$="":I=I+2:RETURN                                :rem 14
800 PRINT"{2 SPACES}"N:I=I+2                          :rem 29
810 RETURN                                            :rem 121
820 DATABRK,ORA,0,0,0,ORA,ASL,0,PHP,ORA,ASL,0,0,OR
    A,ASL,BPL,ORA,0,0,0,ORA,ASL                       :rem 113
830 DATA0,CLC,ORA,0,0,0,ORA,ASL,JSR,AND,0,0,BIT,AN
    D,ROL,0,PLP,AND,ROL,0,BIT                         :rem 7
840 DATAAND,ROL,BMI,AND,0,0,0,AND,ROL,0,SEC,AND,0,
    0,0,AND,ROL,RTI,EOR,0,0,0                         :rem 160
850 DATAEOR,LSR,0,PHA,EOR,LSR,0,JMP,EOR,LSR,BVC,EO
    R,0,0,0,EOR,LSR,0,CLI,EOR,0                       :rem 18
860 DATA0,0,EOR,LSR,RTS,ADC,0,0,0,ADC,ROR,0,PLA,AD
    C                                                 :rem 12
870 DATAROR,0,JMP,ADC,ROR,BVS,ADC,0,0,0               :rem 77
880 DATAADC,ROR,0,SEI,ADC,0,0,0,ADC,ROR,0,STA
                                                      :rem 149
890 DATA0,0,STY,STA,STX,0,DEY,0,TXA,0,STY,STA
                                                      :rem 45
900 DATASTX,BCC,STA,0,0,STY,STA,STX,0,TYA,STA,TXS,
    0,0,STA,0,LDY,LDA,LDX,0                           :rem 60
910 DATALDY,LDA,LDX,0,TAY,LDA,TAX,0,LDY,LDA,LDX,BC
    S,LDA,0,0,LDY,LDA,LDX,0                           :rem 247
920 DATACLV,LDA,TSX,0                                 :rem 30
930 DATALDY,LDA,LDX,CPY,CMP,0,0,CPY,CMP,DEC,0,INY,
    CMP,DEX,0,CPY,CMP,DEC                             :rem 177
940 DATABNE,CMP,0,0,0,CMP,DEC,0,CLD,CMP,0,0,0,CMP,
    DEC,CPX,SBC,0,0,CPX,SBC,INC                       :rem 35
950 DATA0,INX,SBC,NOP,0,CPX,SBC,INC,BEQ,SBC,0,0,0,
    SBC,INC,0,SED,SBC,0,0,0,SBC                       :rem 67
960 DATAINC                                           :rem 147
970 REM MAKE DECIMAL INTO HEX                         :rem 176
980 H$="":FORM=ZXTO0STEP-1:N%=DE/(16↑M):DE=DE-N%*1
    6↑M:H$=H$+MID$(HE$,N%+1,1)                        :rem 148
990 NEXT:RETURN                                       :rem 251
1000 PRINT"TYPE C TO CONTINUE FROM" I                 :rem 156
1010 GETK$:IFK$=""THEN1010                            :rem 187
1020 IFK$="C"THENSA=I:TA=SA:GOTO170                   :rem 121
1030 INPUTSA:TA=SA:GOTO170                            :rem 147
```

```
1040 REM ANALYZE H & L OF OPCODE.          :rem 198
1050 H%=X/16:L%=X-H%*16                    :rem 201
1060 :RETURN                               :rem 225
1070 REM FIND ADDRESS TYPE & GOSUB         :rem 187
1080 CK=H%/2:IFCK=INT(CK)THENCK=12          :rem 29
1090 L%=L%+1                                :rem 69
1100 ONL%GOSUB290,380,730,1130,480,480,530,1130,58
     0,590,580,1130,610,680,700             :rem 75
1110 CK=0                                  :rem 190
1120 LN=LN+1                               :rem 145
1130 RETURN                                :rem 165
```

Program 2. Fragment

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

```
800 FOR ADRES=884 TO 930:READ DATTA:POKE ADRES,DAT
    TA:NEXT ADRES                          :rem 44
882 DATA 169,8,133,58                      :rem 28
888 DATA 169,0,133,57,160,0                :rem 56
894 DATA 24,141,0,4,145,57                  :rem 1
900 DATA 209,57,240,21,152,72              :rem 144
906 DATA 165,58,72,32,179,3                 :rem 64
912 DATA 104,133,58,104,168,169            :rem 255
918 DATA 0,230,57,208,7,230                 :rem 47
924 DATA 58,24,105,1,208,221,96            :rem 249
```

Program 3. Fragment Disassembly

```
884 LDA   # 8
886 STA   58
888 LDA   # 0
890 STA   57
892 LDY   # 0
894 CLC
895 STA 1024
898 STA ( 57 ),Y
900 CMP ( 57 ),Y
902 BEQ   925
904 TYA
905 PHA
906 LDA   58
908 PHA
909 JSR 947
912 PLA
913 STA   58
915 PLA
916 TAY
917 LDA   # 0
919 INC   57
```

```
921 BNE    930
923 INC   58
925 CLC
926 ADC   # 1
928 BNE    895
930 RTS
```

Richard Mansfield

Windows and Pages

Comparing short BASIC programs and their machine language equivalents is one of the best ways to see how machine language works. Using a "window" into your Commodore 64's memory, this article takes you step by step through the process of LOADing and manipulating an ML routine.

One of the most effective ways to learn machine language (ML) is to examine short routines in BASIC and then see how the same thing is accomplished in ML. After all, there are a fairly limited number of basic programming techniques in any language: looping, branching, comparing, counting, and a few others. And long programs are not created in a furious burst of nonstop programming. Rather, they are built by knitting together many small subprograms, short routines which are as understandable in ML as they are in BASIC. Looking at side-by-side BASIC-ML examples is the best way to learn ML. Before you know it, you'll be able to think in both languages, and you'll have a working knowledge of machine language.

Peering into Memory

To start things off, type in Program 1. This is called a *BASIC loader*, and its function is to POKE a machine language program into RAM memory. The numbers in the DATA statements are the ML program. SAVE the program in case things go awry, then type RUN. Nothing seems to happen. You can then type NEW because the little loader has done its job: A short ML program is now in memory from address 864 to 875.

Program 1. 64 Loader

```
800 FOR ADRES=864 TO 875:READ DATTA:POKE ADRES,DAT
    TA:NEXT ADRES
864 DATA 162,0,189,0,0,157
870 DATA 0,4,232,208,247,96
```

Because of the color memory problem (you have to POKE values into the entire color memory before you can see things), enter this short BASIC line in direct mode (without line numbers) before trying out the ML program and after each time you clear the screen:

FOR I = 55296 TO 55552: POKE I,1: NEXT

To see what this ML program does, you can just SYS to the start of it by typing SYS 864. If you typed in the DATA statements correctly, you'll see a collection of strange symbols on the screen. Now clear the screen and type in Program 2. When RUN, it allows you to see things happening. Some characters are flashing rapidly, some change only in response to things you typed on the keyboard, some do nothing. What you're looking at is the first 256 memory addresses in your computer. The flashing characters (160, 161, and 162, counting down from the top-left corner of the screen) are your computer's clock. They're what you get when you ask for TI$.

Program 2. The Window

```
10 SYS 864
20 GETK$
30 GOTO 10
```

The computer divides its memory into groupings of 256 bytes, called *pages*. What you're seeing onscreen is *zero page*. Stop the program with the RUN/STOP key and type POKE 868,1. Then RUN Program 2 again. Now you've changed the zero to a one, and you've changed the ML program so that it puts page 1 onscreen. You can see any of the 256 pages in the computer by just POKEing the page number into address 868.

Comparing BASIC and Machine Language

How does this little ML program send the contents of RAM memory to the screen? Let's show how we could do it in BASIC and then see how ML does it:

```
10 X=0
20 A=0+PEEK(X)
30 POKE 1024+X,A
40 X=X+1
50 IF X<>256 THEN 20
```

Of course you would probably write a program like this using a FOR-NEXT loop, but we've distorted normal BASIC style a bit to more closely reflect the approach used in ML. Look at Program 3. It's the kind of disassembly you would see if you used a "monitor" program like Tinymon (published in the January 1982 issue of *COMPUTE!* magazine). The first column, starting with 0360 and containing a series of four-digit hexadecimal (hex) numbers, represents the addresses in memory where each ML instruction resides. Don't worry if you don't know hexadecimal notation. For now, it's enough that

you understand that, in ML, memory addresses perform the same function as BASIC's line numbers do in a program LISTing.

Program 3. Monitor Disassembly

```
Ø36Ø A2 ØØ     LDX #$ØØ
Ø362 BD ØØ ØØ LDA $ØØØØ,X
Ø365 9D ØØ Ø4 STA $Ø4ØØ,X
Ø368 E8        INX
Ø369 DØ F7     BNE $Ø362
Ø36B 6Ø        RTS
```

Program 4. Disassembler

```
STARTING ADDRESS (DECIMAL)? 864
START ADDRESS HEX  Ø36Ø
864  LDX   # Ø
866  LDA  Ø ,X
869  STA   1Ø24 ,X
872  INX
873  BNE         866
875  RTS
```

After the "line number" address, you see some groupings of two-digit hex numbers. The first group is A2 00. These are the actual numbers that the computer reads when it executes the ML program. These numbers are the most elemental form of machine language and sit in memory at the addresses indicated to their left. Finally, the LDX #$00 is the disassembly (the *translation*) of the ML A2 00. LDX means "LoaD the X register." The X register is like a variable, and the # symbol tells the computer to load the *number* zero into the X — as opposed to the number found at *address* zero. Without the #, X would be given whatever number was currently in address zero. The $ means that the number is in hex notation, not decimal.

Now take a look at Program 4. If you have saved a copy of the disassembler from the previous section, LOAD it. When it asks you for "starting address (decimal)," type in 864 and you'll see the same disassembly as illustrated in Program 4. Notice that 864 is translated into hex (0360). Program 4 is nearly identical to Program 3 except that the numbers between the address and the disassembly are not shown.

The second "line" in Program 4 "LoaDs the Accumulator" with the item in *address* zero + X. That is, if X is 5, the item in 0 + 5 (address 5) is loaded into the accumulator. The accumulator is another "variable" in ML, used to hold things temporarily. Since we're trying to send all the items in zero page (addresses 0–255) up to the screen

memory so we can see them, our next job is to "STore the Accumulator" at address 1024 + X. (1024 is the starting address of screen memory on the 64.) As you can see, we're making X do double duty here as an "index." It's acting as an offset for both the source items (in zero page) as well as the target to which we're sending those items (screen memory).

The next line raises, or *increments*, X by 1. INcrement X causes X = X + 1 to take place, so this first time through the loop, X goes up from 0 to 1. The BNE means "Branch if Not Equal to zero." Branch is like GOTO, but instead of giving a line number as its target, it gives an address in memory (866 in this case, the start of our loop). How does X ever get to zero if it's being INXed each time through the loop? No single-byte variable in ML can go higher than 255. (Likewise, no individual memory address in the computer can "hold" a number beyond 255. This is similar to the fact that no decimal digit can ever go higher than 9. After that, the digits "reset" to zero.) As soon as you've raised X to 255, the next time you INX, it resets itself to zero and starts over. So line 873 in Program 4 will throw us back to line 866 until we've been through the loop 256 times. Then we'll finally get to line 875, where REturn from Subroutine sends us out of ML and back into BASIC where we left off. Notice that SYS-RTS has the same effect as GOSUB-RETURN, except that the former moves between BASIC and ML.

Making a Loader Automatically

Program 5 is another useful tool when you're working with ML. The loader in Program 1 POKEs an ML program into RAM memory for you; Program 5, "Datamaker," goes in the other direction and translates an ML program from RAM into a BASIC loader.

After you type it in and SAVE it, try an experiment. To make a loader out of "The Window," change line 1 in Program 5 to read S = 864:F = 875:L = 10. S is the starting address of your ML and F is the finish. Then type RUN. You'll see the loader created for you onscreen.

The Datamaker destroys itself after it's finished. Notice that the line numbers created in the loaders made by Datamaker are also the *addresses* where the ML will be POKEd. And don't forget to change the starting and ending addresses in line 800 before SAVEing a finished loader.

Program 5. Datamaker

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

```
1 S=826:F=1Ø23:L=1Ø:REM S&F=ADRES          :rem 145
2 PRINT"{CLR}{2 DOWN}":FORI=STOS+48STEP6:IFI>FTHEN
  NEXT:PRINT"GOTO7":GOTO6                  :rem 217
3 PRINTI;"DATA";:FORJ=ØTO5:A$=STR$(PEEK(I+J))
                                           :rem 11
4 PRINTRIGHT$(A$,LEN(A$)-1)",";:NEXTJ:PRINTCHR$(2Ø
  ):NEXTI                                  :rem 227
5 PRINT"S="S+48":F="F":L="L":GOTO2         :rem 225
6 POKE198,L:FORK=1TOL:POKE63Ø+K,13:NEXT:PRINT"
  {HOME}":END                              :rem 248
7 PRINT"{CLR}{2 DOWN}":FORI=1TO7:PRINTI:NEXT:L=7:G
  OTO6                                     :rem 155
8ØØ FOR ADRES=826 TO 1Ø23:READ DATTA:POKE ADRES,DA
    TTA:NEXT ADRES                         :rem 82
```

You've seen a little of how ML can be assembled, disassembled, and compared with BASIC. The next article will show you how a simple ML program is created.

J.R. Chaffer

Stealing Characters from ROM

Plotting bitmapped data or graphics with machine language? Want to use standard ROM characters on the same screen, without *tediously bit-mapping each one? Here's a shortcut to do just that. More importantly, this article shows you how a simple machine language routine is created and how it works.*

Jim Butterfield dropped an intriguing hint in the May 1983 issue of *COMPUTE!* magazine when he used the phrase which became the title of this article. Normally, ROM characters cannot be displayed on the same screen as high-resolution graphics unless each byte is carefully mapped out exactly as it exists in ROM. However, the eight bytes which create each character *are* contained permanently in ROM at locations 53248–57343 ($D000–$DFFF). They can easily be read and transferred to your bitmap area with a subroutine. This *could* be done in BASIC; but if you're doing bitmapping, you've probably already discovered that BASIC is too slow for this application. As long as you know the address of each of the desired characters in ROM you'll find this method of "stealing" characters from ROM just what you need. Many of these addresses are listed in Table 1.

Plotstring and Dummy Plot

This technique works best if you already have a machine language subroutine which plots a byte of data by storing it at some specific address in your 8K block of high-resolution memory. The dummy routine Plot, called by the example program ("Plotstring"), theoretically does just this; it stores the byte contained in the accumulator at the RAM address specified by PTR1A and PTR1B (low byte, high byte format). When you are in bitmap mode, that byte then appears on the screen at the location corresponding to its address in RAM.

Plotstring initially reads the first data byte in a specified two-byte character address in ROM, and stores it at your prearranged RAM location. It then does the same for each of the seven remaining bytes of the character. You will, presumably, already have figured out the correct RAM address for the desired screen location. The only other thing you need is the ROM address from which to start reading the eight-byte character. (Note that this technique will work for all the Commodore graphics characters as well.)

If you're plotting a string of characters, you should place the string on one of the 25 standard screen-text lines (numbered 0–24). Also, each character should be located in one of the 40 standard columns (numbered 0–39). Then, if you simply increment target RAM pointers between each byte read and stored, after eight bytes the pointers indicate the correct starting location for the next character. Plotstring does this by incrementing PTR1A and PTR1B. PTR2A and PTR2B are pointers to the character addresses in ROM. ROMADS1 and ROMADS2, as well as the byte following each, point to the start of a table of double-byte ROM addresses for your intended string. This table must be built somewhere into your code. BYTECNT is the counter which insures that you read and store eight bytes at a time. CHRCOUNT is a count of the number of characters plotted so far, and is compared to CHRS (the total number of characters to be plotted) to test for string completion. Note that the timer IRQ is turned off before switching in ROM for reading.

Here's the subroutine Plotstring, which reads the characters in ROM, and stores the eight-byte blocks of data, one byte at a time, in the RAM bitmap. Remember that this routine doesn't actually store the data in a bitmap. The line JSR PLOT (Jump to SubRoutine Plot) would do that *only* if there really was a Plot subroutine. You'll have to create a Plot subroutine yourself, or refer to Source Code 3, "Demonote," to see how I created such a subroutine.

This listing is the *source code* for the Plotstring subroutine; that is, it shows the program code I entered before I ran it through an assembler. The assembler, in turn, created the machine language routine that runs on my Commodore 64. Only through a source code listing can you see the labels for the various routines called, as well as comments such as *turn off timer IRQ*.

Source Code 1. Plotstring

```
PLOTSTRING: LDA  $DC0E
            AND  #$FE
            STA  $DC0E        ; turn off timer IRQ.
            LDA  $01
            AND  #$FB
            STA  $01          ; switch character ROM in.
            LDX  #$00
            STX  CHRCOUNT     ; number of characters
                              ; plotted so far.
PLOTCHR     LDA  CHRCOUNT
            ASL  A            ; multiply by two, to get
            TAY               ; address table offsets.
```

```
                LDA  ROMADS1, Y  ; lo-byte and hi-byte of char
                LDX  ROMADS2, Y  ; address in ROM (see text).
                STA  PTR2A
                STX  PTR2B       ; store in pointers.
                LDY  #$08
                STY  BYTECNT     ; counter for eight bytes.
PLOTBYTE        LDY  #$00        ; set zero offset.
                LDA  (PTR2A), Y  ; get next byte of character
                                   from ROM,
                JSR  PLOT        ; and store in RAM bitmap.
                INC  PTR1A
                BNE  NEXTBYTE
                INC  PTR1B       ; increment bitmap pointers.
NEXTBYTE        INC  PTR2A       ; increment ROM pointer.
                DEC  BYTECNT     ; decrement eight-byte
                                   counter.
                BNE  PLOTBYTE    ; if not done, plot next byte.
                INC  CHRCOUNT    ; another character plotted.
                LDA  CHRCOUNT
                CMP  CHRS        ; all characters plotted?
                BNE  PLOTCHR     ; no, plot another.
                LDA  $01         ; all done now.
                ORA  #$04
                STA  $01         ; switch character ROM out.
                LDA  $DC0E
                ORA  #$01
                STA  $DC0E       ; timer IRQ back on.
                RTS              ; back to calling routine.
```

Routine Mechanics

I continually use this subroutine to get standard numbers and characters on my bitmap. Of course, the characters are still bitmapped, but the data is taken directly from the ROM.

If you're new to machine language listings, the labeled variables, such as CHRS, BYTECNT, and so on, are simply single-byte storage locations defined by the user. They can be anywhere in free RAM, with the exception of PTR1A, PTR1B, PTR2A, and PTR2B, which must be in page 0. You can use locations 251–254 (hex $FB–$FE) and location 2 which are not used by BASIC. You can also use locations 139–143 ($8B–$8F) if you're not using BASIC's RND function.

This routine can be used repeatedly if the ROMADS1 and ROMADS2 addresses are set by each calling sequence. Note that ROMADS1 *is* the first byte of the two-byte address immediately fol-

lowing the *LDA* in the line, *LDA ROMADS1,Y*; the LDA here is byte 185 ($B9). Likewise, ROMADS2 is the byte immediately following the *LDX* in the next line, and the LDX is coded as byte 190 ($BE). So what we are doing here is modifying the subroutine itself as part of the calling sequence. Actually, four bytes must be set by the calling routine: (1) The ROMADS1 byte; (2) the byte just following ROMADS1; (3) ROMADS2; and (4) the byte immediately after ROMADS2.

Let's say you want to plot the string, *Note 1:* near the upper-left corner of your bitmapped screen. And let's say your bitmap storage area is from 57344 to 65343 ($E000–$FF3F), which is under the Kernal ROM. You're starting the string in text column one, on line one. (Remember that column and line numbers run from 0 to 39 and from 0 to 24, respectively.) If you work out the address for this screen location, you will find that the starting byte is 57672 ($E148). Let's also say your character ROM address table for the string *Note 1:* has been stored in RAM as in Table 2, starting at location 49152 ($C000). A functional calling sequence would be as follows:

Source Code 2. Calling Sequence

```
PLOTNOTE1   LDA  #$48          ; lo-byte, screen location.
            LDX  #$E1          ; hi-byte, screen location.
            STA  PTR1A
            STX  PTR1B
            LDA  #$C0          ; hi-byte, Table 2 start.
            LDX  #$00          ; lo-byte, Table 2 start.
            STA  ROMADS1 + 1   ; address = ROMADS1 + 1
            STA  ROMADS2 + 1   ; address = ROMADS2 + 1
            STX  ROMADS1       ; lo-byte of address.
            INX
            BNE  CONT
            INC  ROMADS2 + 1
CONT        STX  ROMADS2       ; ROMADS pointers set.
            LDY  #$07
            STY  CHRS          ; seven characters in string.
            JSR  PLOTSTRING    ; go plot "Note 1:"
                 .
                 .
                 .
```

Each string you plot must have its ROM character address table stored somewhere in RAM, similar to Table 2. Note that the address bytes are stored in standard lo-byte, hi-byte format. Thus, they appear to be backwards when compared to the addresses given in Table 1. This can be switched, if you modify Plotstring accordingly.

Faster and Easier

If you bitmapped each character directly, you would still need a subroutine to do it, and your data table would have eight bytes for each character instead of two. In addition, you would have to work out, or copy, the bitmap for each individual character. With a routine like Plotstring, all you need is a table of their ROM addresses. The addresses of the most common characters are listed in Table 1.

The calling sequence can be summarized as follows:

1. Set screen location pointers (PTR1A and PTR1B).
2. Set the ROMADS pointers (four different bytes) to indicate the start of the string address table.
3. Set the number of characters in the string.
4. JSR to Plotstring.

It's not necessary to use an exact copy of Plotstring as it appears in this article. If you write your own routine, however, be sure to switch the timer IRQ out before switching the ROM in for reading. Also, make sure you turn the timer on again after switching the ROM out, or you'll probably not be able to use the keyboard.

You don't need to worry about the video bank fouling up the ROM addresses, either. If you use the absolute addresses given in Table 1, you'll read ROM characters correctly with this scheme regardless of which video bank you're using.

Source Code 3. Demonote

```
                0065 ;              ZERO PAGE LOCATIONS:
                0070
                0075 TEXTMODE       .DE   $02 : UPPER/LOWERCASE.
                0080
                0085 PTR1A          .DE   $FB : POINTERS FOR BITMAP
                0090 PTR1B          .DE   $FC : STORAGE.
                0095
                0100 PTR2A          .DE   $FD : POINTERS FOR
                0105 PTR2B          .DE   $FE : ROM ADDRESSES.
                0110
                0115 CHRS           .DE   $8B : NR. CHARS IN STRING.
                0120 CHRCOUNT       .DE   $8C : NR. CHARS PLOTTED SO FAR.
                0125 BYTECNT        .DE   $8D : COUNTER FOR 8 BYTES.
                0130 BYTEVAL        .DE   $8E : BYTE VALUE TO PLOT.
                0135 NDX            .DE   $C6 : NR CHARS IN KEY BUF.
                0140
                0145
                0150 ;              STORE ROM ADDRESS TABLE
                0155 ;                FOR STRING, "NOTE 1:"
                0160
                0165                .BA   $C000
                0170
                0175                .OS
                0180
C000- 70 D0 78  0185                .BY   $70 $D0 $78 $D8 $A0 $D8 $28 $D8
C003- D8 A0 D8
```

```
C006- 28 D8
C008- 00 D1 88   0190                .BY   $00 $D1 $88 $D1 $D0 $D1
C00B- D1 D0 D1
                 0195
                 0200 ;          SET BITMAP MODE, ETC:
                 0205
C00E- 20 A6 C0   0210 SETUP          JSR   CLRMAP : CLEAR BITMAP AREA
C011- 20 C3 C0   0215                JSR   SETCOLOR : SET COLOR CODE IN SCREEN 1K.
C014- 20 E2 C0   0220                JSR   SETBMM : SET BITMAP MODE.
                 0225
                 0230
                 0235
                 0240 ;          SUPERVISOR TO CALL STRING
                 0245 ;                PLOT ROUTINE:
                 0250
C017- A9 48      0255 PLOTNOTE1      LDA   #$48 : LO-BYTE, SCREEN LOCATION
C019- A2 E1      0260                LDX   #$E1 : HI-BYTE, SCREEN LOCATION
C01B- 85 FB      0265                STA   PTR1A
C01D- 86 FC      0270                STX   PTR1B
C01F- A9 C0      0275                LDA   #$C0 : HI-BYTE, ADR TABLE START
C021 - A2 00     0280                LDX   #$00 : LO-BYTE, ADR TABLE START
C023- 8D 57 C0   0285                STA   ROMADS1 + 1 : BYTE AT LINE 0455.
C026- 8D 5A C0   0290                STA   ROMADS2 + 1 : BYTE AT LINE 0490.
C029- 8E 56 C0   0295                STX   ROMADS1 : BYTE AT LINE 0450.
C02C- E8         0300                INX
C02D- D0 03      0305                BNE   CONT
C02F- EE 5A C0   0310                INC   ROMADS2 + 1 : BYTE AT LINE 0490.
C032- 8E 59 C0   0315 CONT           STX   ROMADS2 : BYTE AT LINE 0485.
```

```
                0320
                0325 ;          ROMADS POINTERS ARE NOW SET
                0330
C035- A0 07     0335                LDY  #$07
C037- 84 8B     0340                STY  CHRS : SEVEN CHARS IN STRING.
C039- 20 3F C0  0345                JSR  PLOTSTRING : GO PLOT "NOTE 1:"
C03C- 4C 23 C1  0350                JMP  ENDPLOT
                0355
                0360
                0365 ;          STRING PLOTTER:
                0370
C03F- AD 0E DC  0375 PLOTSTRING     LDA  $DC0E
C042- 29 FE     0380                AND  #$FE
C044- 8D 0E DC  0385                STA  $DC0E : TURN OFF TIMER IRQ
C047- A5 01     0390                LDA  $01
C049- 29 FB     0395                AND  #$FB
C04B- 85 01     0400                STA  $01 : SWITCH CHAR ROM IN
C04D- A2 00     0405                LDX  #$00
C04F- 86 8C     0410                STX  CHRCOUNT : NR CHARS PLOTTED SO FAR
C051- A5 8C     0415 PLOTCHR        LDA  CHRCOUNT
C053- 0A        0420                ASL  A : MULT BY 2 TO GET
C054- A8        0425                TAY  : ADR TABLE OFFSETS.
C055- B9        0430                .BY  $B9
                0435
                0440 ; NOTE: $B9 IS "LDA ABS, Y"
                0445
C056-           0450 ROMADS1        .DS  1 : LO-BYTE, TABLE ADR
C057-           0455                .DS  1 : HI-BYTE, TABLE ADR
```

```
                0460
C058- BE        0465                .BY   $BE
                0470
                0475 ; NOTE: $BE IS "LDX ABS, Y"
                0480
C059-           0485 ROMADS2        .DS   1 : LO-BYTE, TABLE ADR + 1
C05A-           0490                .DS   1 : HI-BYTE, TABLE ADR + 1
C05B- 85 FD     0495                STA   PTR2A
C05D- 86 FE     0500                STX   PTR2B : STORE IN POINTERS.
C05F- A0 08     0505                LDY   #$08
C061- 84 8D     0510                STY   BYTECNT
C063- A0 00     0515 PLOTBYTE       LDY   #$00 : SET ZERO OFFSET.
C065- B1 FD     0520                LDA   (PTR2A),Y : GET NEXT BYTE FROM ROM,
C067- 20 8D C0  0525                JSR   PLOT : AND STORE IN BITMAP.
C06A- E6 FB     0530                INC   PTR1A
C06C- D0 02     0535                BNE   NEXTBYTE
C06E- E6 FC     0540                INC   PTR1B : INCREMENT BITMAP POINTERS.
C070- E6 FD     0545 NEXTBYTE       INC   PTR2A : INC. ROM POINTER.
                0550
                0555 ; NOTE: ROM CHARACTERS DON'T CROSS
                0560 ; PAGES, SO NO NEED TO INC. PTR2B.
                0565
C072- C6 8D     0570                DEC   BYTECNT : DECREMENT 8-BYTE COUNTER.
C074- D0 ED     0575                BNE   PLOTBYTE : IF NOT DONE, PLOT NEXT BYTE.
C076- E6 8C     0580                INC   CHRCOUNT : ANOTHER CHAR PLOTTED.
C078- A5 8C     0585                LDA   CHRCOUNT
C07A- C5 8B     0590                CMP   CHRS : ALL CHARS PLOTTED?
C07C- D0 D3     0595                BNE   PLOTCHR : NO; PLOT ANOTHER.
```

```
C07E- A5 01   0600                LDA  $01 : ALL DONE NOW.
C080- 09 04   0605                ORA  #$04
C082- 85 01   0610                STA  $01 : CHAR ROM BACK OUT.
C084- AD 0E DC 0615               LDA  $DC0E
C087- 09 01   0620                ORA  #$01
C089- 8D 0E DC 0625               STA  $DC0E : TIMER IRQ BACK ON.
C08C- 60      0630                RTS  : BACK TO SUPERVISOR.
              0635
              0640
              0645 ;          PLOT A BYTE BY STORING
              0650 ;              IN BITMAP AREA:
              0655 ;              ($E000-$FF3F)
              0660
C08D- 78      0665 PLOT           SEI  : TURN OFF IRQ
              0670
              0675 ; NOTE: MUST SWITCH KERNAL OUT SO WON'T
              0680 ; READ IT WITH ORA, LINE 0720.
              0685
C08E- 85 8E   0690                STA  BYTEVAL : STORE VALUE TO PLOT.
C090- A5 01   0695                LDA  $01
C092- 29 FD   0700                AND  #$FD
C094- 85 01   0705                STA  $01 : REMOVE KERNAL ROM.
C096- A5 8E   0710                LDA  BYTEVAL
C098- A0 00   0715                LDY  #$00 : SET ZERO OFFSET.
C09A- 11 FB   0720                ORA  (PTR1A),Y : DON'T ERASE ANYTHING.
C09C- 91 FB   0725                STA  (PTR1A),Y : PLOT THE BYTE.
C09E- A5 01   0730                LDA  $01
C0A0- 09 02   0735                ORA  #$02
```

```
C0A2- 85 01   0740                STA  $01 : RESTORE KERNAL.
C0A4- 58      0745                CLI  : IRQ BACK ON.
C0A5- 60      0750                RTS
              0755
              0760
              0765 ;          CLEAR BITMAP AREA:
              0770
C0A6- A9 E0   0775 CLRMAP         LDA  #$E0
C0A8- 85 FC   0780                STA  PTR1B
C0AA- A9 00   0785                LDA  #$00
C0AC- 85 FB   0790                STA  PTR1A : SET POINTERS.
C0AE- A2 1F   0795                LDX  #$1F : 31+ TIMES THROUGH LOOP.
C0B0- A8      0800                TAY  : SET ZERO OFFSET.
C0B1- 91 FB   0805 CLRSTO         STA  (PTR1A),Y
C0B3- C8      0810                INY
C0B4- D0 FB   0815                BNE  CLRSTO : CLEAR PAGE.
C0B6- E6 FC   0820                INC  PTR1B : INC. PAGE.
C0B8- CA      0825                DEX  : DEC PAGE COUNTER.
C0B9- D0 F6   0830                BNE  CLRSTO : (Y IS AT ZERO.)
C0BB- 91 FB   0835 LASTMAP        STA  (PTR1A),Y : PART OF LAST PAGE.
C0BD- C8      0840                INY
C0BE- C0 40   0845                CPY  #$40
C0C0- D0 F9   0850                BNE  LASTMAP
C0C2- 60      0855                RTS  : ALL DONE CLEARING MAP.
              0860
              0865
              0870 ;          STORE COLOR CODE:
              0875 ;          (COLOR STARTS AT $C800)
```

```
                0880
C0C3- A9 C8     0885  SETCOLOR     LDA  #$C8
C0C5- 85 FC     0890                STA  PTR1B
C0C7- A9 00     0895                LDA  #$00
C0C9- 85 FB     0900                STA  PTR1A : SET 1K COLOR POINTERS.
C0CB- A8        0905                TAY  : SET ZERO OFFSET.
C0CC- A2 03     0910                LDX  #$03 : 3+ TIMES THROUGH LOOP.
C0CE- A9 CB     0915                LDA  #$CB : LIGHT & DARK GRAY.
C0D0- 91 FB     0920  STOCOLR       STA  (PTR1A),Y
C0D2- C8        0925                INY
C0D3- D0 FB     0930                BNE  STOCOLR
C0D5- E6 FC     0935                INC  PTR1B
C0D7- CA        0940                DEX
C0D8- D0 F6     0945                BNE  STOCOLR
C0DA- 91 FB     0950  LASTSCRN      STA  (PTR1A),Y
C0DC- C8        0955                INY
C0DD- C0 E8     0960                CPY  #$E8
C0DF- D0 F9     0965                BNE  LASTSCRN
C0E1- 60        0970                RTS  : COLORS SET.
                0975
                0980
                0985  ;             SET BITMAP MODE, ETC:
                0990
C0E2- AD 18 D0  0995  SETBMM        LDA  $D018
C0E5- 85 02     1000                STA  TEXTMODE : SAVE MODE.
C0E7- A9 28     1005                LDA  #$28
C0E9- 8D 18 D0  1010                STA  $D018 : SET MAP & SCREEN.
C0EC- AD 02 DD  1015                LDA  $DD02
```

```
C0EF- 09 03     1020                ORA  #$03
C0F1- 8D 02 DD  1025                STA  $DD02 : SET VIDEO BANK TO 3.
C0F4- AD 00 DD  1030                LDA  $DD00
C0F7- 29 FC     1035                AND  #$FC
C0F9- 8D 00 DD  1040                STA  $DD00
C0FC- AD 11 D0  1045                LDA  $D011
C0FF- 09 20     1050                ORA  #$20
C101- 8D 11 D0  1055                STA  $D011 : SET BITMAP MODE.
C104- 60        1060                RTS
                1065
                1070
                1075 ;          CLEAR BITMAP MODE, GO TO TEXT:
                1080
C105- AD 11 D0  1085 CLRBMM         LDA  $D011
C108- 29 DF     1090                AND  #$DF
C10A- 8D 11 D0  1095                STA  $D011 : CLEAR BITMAP MODE.
C10D- A5 02     1100                LDA  TEXTMODE
C10F- 8D 18 D0  1105                STA  $D018 : RESTORE TEXT MODE.
C112- AD 02 DD  1110                LDA  $DD02 : RESTORE VIDEO BANK TO 0:
C115- 09 03     1115                ORA  #$03
C117- 8D 02 DD  1120                STA  $DD02
C11A- AD 00 DD  1125                LDA  $DD00
C11D- 09 03     1130                ORA  #$03
C11F- 8D 00 DD  1135                STA  $DD00
C122- 60        1140                RTS
                1145
                1150
                1155 ;          WHILE IN BITMAP MODE, WAIT FOR KEY
                1160 ;          TO BE PRESSED; THEN RETURN TO TEXT.
```

```
                1165
C123- A9 00     1170 ENDPLOT    LDA  #$00
C125- 85 C6     1175            STA  NDX
C127- A5 C6     1180 CHKBUF     LDA  NDX
C129- F0 FC     1185            BEQ  CHKBUF : WAIT FOR KEY !
C12B- 20 05 C1  1190            JSR  CLRBMM : CLEAR BITMAP MODE.
C12E- 60        1195            RTS  : BACK TO TEXT.
                1200
                1205
                1210 ;          END OF PROGRAM.
                1215
                1220                 .EN
```

— LABEL FILE: —

```
BYTECNT = 008D      BYTEVAL = 008E      CHKBUF = C127
CHRCOUNT = 008C     CHRS = 008B         CLRBMM = C105
CLRMAP = C0A6       CLRSTO = C0B1       CONT = C032
ENDPLOT = C123      LASTMAP = C0BB      LASTSCRN = C0DA
NDX = 00C6          NEXTBYTE = C070     PLOT = C08D
PLOTBYTE = C063     PLOTCHR = C051      PLOTNOTE1 = C017
PLOTSTRING = C03F   PTR1A = 00FB        PTR1B = 00FC
PTR2A = 00FD        PTR2B = 00FE        ROMADS1 = C056
ROMADS2 = C059      SETBMM = C0E2       SETCOLOR = C0C3
SETUP = C00E        STOCOLR = C0D0      TEXTMODE = 0002
```

To actually see this ML subroutine in action, you need to enter two programs. First, type in and RUN Program 1, "Plotstring Loader." This program POKEs the required values into the appropriate memory locations. If you've entered some of the DATA incorrectly, the checksum included will tell you there's an error. Next, type NEW, enter Program 2, "Note 1:," and RUN it. What you'll see is a message telling you to press the space bar. As soon as you do, the machine language subroutine places the characters spelling *Note 1:* on the screen. It may not seem impressive, but what you've just done is steal characters from ROM and place them on a bitmapped screen. Painless machine language.

Program 1. Plotstring Loader

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

```
10 FOR I=49152 TO 49454:READ A                    :rem 23
20 CK=CK+A                                        :rem 39
30 POKE I,A:NEXT I                                :rem 10
40 IF CK<>41733 THEN PRINT"ERROR IN DATA"         :rem 56
50 END                                            :rem 60
49152 DATA 112,208,120,216,160,216               :rem 133
49158 DATA 040,216,000,209,136,209               :rem 142
49164 DATA 208,209,032,166,192,032               :rem 150
49170 DATA 195,192,032,226,192,169               :rem 161
49176 DATA 072,162,225,133,251,134               :rem 147
49182 DATA 252,169,192,162,000,141               :rem 146
49188 DATA 087,192,141,090,192,142               :rem 161
49194 DATA 086,192,232,208,003,238               :rem 156
49200 DATA 090,192,142,089,192,160               :rem 149
49206 DATA 007,132,139,032,063,192               :rem 143
49212 DATA 076,035,193,173,014,220               :rem 142
49218 DATA 041,254,141,014,220,165               :rem 137
49224 DATA 001,041,251,133,001,162               :rem 122
49230 DATA 000,134,140,165,140,010               :rem 119
49236 DATA 168,185,000,000,190,000               :rem 133
49242 DATA 000,133,253,134,254,160               :rem 134
49248 DATA 008,132,141,160,000,177               :rem 139
49254 DATA 253,032,141,192,230,251               :rem 140
49260 DATA 208,002,230,252,230,253               :rem 132
49266 DATA 198,141,208,237,230,140               :rem 153
49272 DATA 165,140,197,139,208,211               :rem 155
49278 DATA 165,001,009,004,133,001               :rem 134
49284 DATA 173,014,220,009,001,141               :rem 133
49290 DATA 014,220,096,120,133,142               :rem 135
49296 DATA 165,001,041,253,133,001               :rem 136
49302 DATA 165,142,160,000,017,251               :rem 130
49308 DATA 145,251,165,001,009,002               :rem 136
49314 DATA 133,001,088,096,169,224               :rem 154
```

```
49320 DATA 133,252,169,000,133,251          :rem 135
49326 DATA 162,031,168,145,251,200          :rem 142
49332 DATA 208,251,230,252,202,208          :rem 137
49338 DATA 246,145,251,200,192,064          :rem 151
49344 DATA 208,249,096,169,200,133          :rem 159
49350 DATA 252,169,000,133,251,168          :rem 146
49356 DATA 162,003,169,203,145,251          :rem 148
49362 DATA 200,208,251,230,252,202          :rem 132
49368 DATA 208,246,145,251,200,192          :rem 154
49374 DATA 232,208,249,096,173,024          :rem 161
49380 DATA 208,133,002,169,040,141          :rem 139
49386 DATA 024,208,173,002,221,009          :rem 143
49392 DATA 003,141,002,221,173,000          :rem 124
49398 DATA 221,041,252,141,000,221          :rem 133
49404 DATA 173,017,208,009,032,141          :rem 140
49410 DATA 017,208,096,173,017,208          :rem 150
49416 DATA 041,223,141,017,208,165          :rem 142
49422 DATA 002,141,024,208,173,002          :rem 128
49428 DATA 221,009,003,141,002,221          :rem 127
49434 DATA 173,000,221,009,003,141          :rem 128
49440 DATA 000,221,096,169,000,133          :rem 134
49446 DATA 198,165,198,240,252,032          :rem 165
49452 DATA 005,193,096,013,013,013          :rem 139
```

Program 2. Note 1:

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

```
90 POKE 53272,23                                    :rem 44
100 PRINT"{CLR}{DOWN}{RIGHT} {OFF}{27 SPACES}":PRI
    NT" {RVS} DEMO TO SHOW PLOTSTRING "    :rem 113
110 PRINT" {RVS} IN ACTION.{14 SPACES}"    :rem 248
120 PRINT:PRINT" {RVS} PRESS SPACE BAR TO SEE
    {2 SPACES}":PRINT" {RVS} 'NOTE 1:'{15 SPACES}"
                                                    :rem 151
130 POKE 198,0:WAIT 198,1:SYS 49166:GOTO 120
                                                    :rem 120
140 END                                             :rem 108
```

Table 1. Common ROM Character Addresses

Character	Address	Character	Address	Character	Address
A	$D008	space	$D100	a	$D808
B	$D010	"	$D110	b	$D810
C	$D018	%	$D128	c	$D818
D	$D020	&	$D130	d	$D820
E	$D028	'	$D138	e	$D828
F	$D030	(	$D140	f	$D830
G	$D038	)	$D148	g	$D838
H	$D040	+	$D158	h	$D840
I	$D048	,	$D160	i	$D848
J	$D050	-	$D168	j	$D850
K	$D058	.	$D170	k	$D858
L	$D060	/	$D178	l	$D860
M	$D068	0	$D180	m	$D868
N	$D070	1	$D188	n	$D870
O	$D078	2	$D190	o	$D878
P	$D080	3	$D198	p	$D880
Q	$D088	4	$D1A0	q	$D888
R	$D090	5	$D1A8	r	$D890
S	$D098	6	$D1B0	s	$D898
T	$D0A0	7	$D1B8	t	$D8A0
U	$D0A8	8	$D1C0	u	$D8A8
V	$D0B0	9	$D1C8	v	$D8B0
W	$D0B8	:	$D1D0	w	$D8B8
X	$D0C0	;	$D1D8	x	$D8C0
Y	$D0C8	=	$D1E8	y	$D8C8
Z	$D0D0	?	$D1F8	z	$D8D0

Table 2. ROM Addresses for Note 1:

Memory Location	Address	Character
$C000	$70 $D0	N
$C002	$78 $D8	o
$C004	$A0 $D8	t
$C006	$28 $D8	e
$C008	$00 $D1	space
$C00A	$88 $D1	1
$C00C	$D0 $D1	:

Chapter 2

Programming Aids

Brent Anderson and Sheldon Leemon

BASIC Aid

Twenty of the most useful programming aids, from searching and replacing strings to line renumbering, are included in this one package. Several DOS support commands also make it simple to see the directory (without affecting the program in memory), rename files, or load and run programs. Mistake-proof entry is easy when you use MLX to type in this program.

We'd all like to customize BASIC to fit our individual needs. It's not difficult to think of some valuable programming aids that were omitted from Commodore's BASIC. Renumbering, automatic line numbering, searching and replacing, merging, and deleting line ranges are just a few. Some of the short routines elsewhere in this book add just one or two of these functions. "BASIC Aid," however, gives you 20 new ways to help you write BASIC programs. Even more importantly, they're all in one package. You don't have to search and somehow splice together several different programs.

If you do any BASIC programming at all, this package will be one of your most valuable utilities. It takes up nearly 4K bytes of memory, but it's safe from BASIC. You can still program normally. And by using MLX, the Machine Language Editor found in Appendix D, you can be assured it's entered correctly.

Some History

The program was originally written for the PET/CBM computers by James Strasma, and later modified by F. Arthur Cochrane. It's become one of the most popular programs available for Commodore computers. Until now, unfortunately, there hasn't been a version written expressly for the 64. Brent Anderson, who translated this program to the 64, has been a user of Commodore computers since 1981, when he joined Strasma and others to form the Central Illinois PET Users Group. He currently heads ATUG, a Commodore users group dedicated to exchanging information dealing with machine language programs.

MLX and BASIC Aid

You'll use MLX to enter BASIC Aid. Before you begin to type in this program, make sure you read Appendix D. You'll also need to type in and save a copy of the MLX program to tape or disk. Once you've done that, load and run MLX. It will ask for the starting and ending

addresses for BASIC Aid. They are:

Starting address: 49152
Ending address: 52997

Once you've typed in those numbers, you're ready to begin entering BASIC Aid. You don't have to enter it in one session; you can stop and pick up where you left off any number of times. Refer to the instructions in Appendix D for details.

Once you've got BASIC Aid typed in (which may take you several sessions — it's a long program), make sure you save it to tape or disk. It's then ready to use.

Load BASIC Aid as you would any other completely machine language program by typing LOAD "filename",8,1. Then type NEW to reset the BASIC pointers. You start it by entering SYS 49152. BASIC Aid uses the wedge technique to add the following commands to BASIC. Since the wedge can slow execution of a program, all of these commands work only in direct mode (without using line numbers).

An additional feature lets you pause or stop any BASIC Aid command that displays to the screen. To pause the display, simply hold down the SHIFT key. To stop it, hit the SHIFT LOCK key or the space bar. To release, unlock the SHIFT, or hit the Commodore key. Pressing the STOP key aborts the feature entirely.

Programming Aids

Each of the aids supported by BASIC Aid is listed below. The syntax of the command is printed, and a short explanation of what each does is also included. Many of the features are obvious, and little information is necessary. Others, such as the CHANGE option, are more complex and are explained in greater detail. At the end of the program, you'll find a quick reference card you can cut out and place next to your computer.

AUTO

Syntax: AUTO *line increment* (to turn on)
AUTO (to turn off)

AUTO automatically numbers the lines of your BASIC program. You start the process by typing AUTO, followed by a line increment from 1 to 127. The first line number must be entered manually. The next line number in the sequence automatically appears from that point on. For example, if you enter *AUTO 10,* and then the line *10 REM THIS IS THE FIRST LINE,* the line number *20* appears under the *10,* ready for you to type in the next line. To turn this feature off, simply enter AUTO without a line number increment.

BREAK

Syntax: BREAK

The BREAK command is used to enter a machine language monitor like Micromon or Supermon, if such a program has been previously installed.

CHANGE

Syntax: CHANGE/*search string/replacement string/, line number range*
or
CHANGE*"search string"replacement string", line number range*

CHANGE replaces one string of characters (the search string) with another string of characters (the replacement string). Each line in which the string is replaced is displayed on the screen. If the optional range of line numbers is specified, the search string is replaced within that range. Otherwise, the search string is changed to the replacement string at every occurrence within the program. Line number ranges should be specified as they are in the LIST command. For instance, CHANGE*/FIRST STRING/SECOND STRING/, -100* changes the string in lines 0–100; CHANGE*/FIRST STRING/ SECOND STRING/, 100-200* changes it within lines 100–200; and CHANGE*/FIRST STRING/SECOND STRING/,200-* changes it from line 200 to the end of the program.

Two different delimiters can be used to set off the search and replacement strings. This is because BASIC stores keyword commands (such as PRINT) differently than it does the same combination of characters when they appear in quotes or as DATA. If it didn't, it would get terribly confused by statements such as PRINT "PRINT". It would try to execute the second PRINT, rather than printing it on the screen. The CHANGE command lets you decide which type of string you wish to change. If the backslash character is used to separate the two strings, BASIC reserved keywords are recognized as such. If quotation marks are used, all text is treated as strings of characters. For example, if the program in memory is:

```
10 PRINT"ALL THAT THIS PROGRAM DOES IS PRINT"
```

then the command CHANGE*/PRINT/REM/* alters the program to:

```
10 REM"ALL THAT THIS PROGRAM DOES IS PRINT"
```

while the command CHANGE*"PRINT"OUTPUT CHARACTERS"* changes the program to:

```
10 PRINT"ALL THAT THIS PROGRAM DOES IS OUTPUT CHAR
   ACTERS"
```

In the first case, only the tokenized PRINT command is recognized, while in the second, only the string literal "PRINT" is changed.

Keep in mind that every occurrence of the string is changed, even if that string appears as a substring in the middle of a word. For example, CHANGE/*TO/FROM*/ converts the line

```
100 PRINT"GET THE MESSAGE TO TONY"
```

to

```
100 PRINT"GET THE MESSAGE FROM FROMNY"
```

Such an error could be avoided by including the spaces around the word in the search and replacement strings (CHANGE/ *TO* / *FROM* /would have the desired effect). When in doubt, look for all occurrences of the search string with the FIND command before using CHANGE, so that no unwanted substrings crop up.

COLD

Syntax: COLD

The COLD command is used to cold start the computer. This means that the computer goes through all the steps it normally would when you turn the power off and on again, except that the contents of memory remain intact. Sometimes, when certain memory locations that BASIC uses have been changed, commands no longer function correctly. By entering COLD, then initializing BASIC Aid with a SYS 49152, and entering the OLD command, you can get a fresh start. COLD can also be used to disengage BASIC Aid and all other machine language programs like monitors that affect the operation of the computer.

CRT

Syntax: CRT

This command sends text and graphics characters to the printer exactly as they appear on the screen. This version accommodates only a Commodore printer connected as device number 4.

DELETE

Syntax: DELETE *line number range*

This command deletes a number of BASIC program lines at once. The line number range uses the same format as the LIST command. For example, DELETE -100 deletes all lines up to and including line

100, DELETE 100-200 deletes those two lines and all lines in between, and DELETE 200- deletes lines 200 and up.

DUMP

Syntax: DUMP

DUMP lists the variables used in a BASIC program in the order in which they were created, as well as shows their current value. Only scalar (nonarray) variables are displayed. DUMP can be handy for testing the effect of changing the value of a variable in a running program. Just hit the STOP key, type DUMP to check the current value of a variable, edit and enter a new value for the variable, and type CONT to continue the program using the new value.

FIND

Syntax: FIND/*search string/, line number range*
or
FIND *"search string", line number range*

The FIND command searches the BASIC program for a string of characters, and displays the program lines in which the string appears. This command displays every occurrence of the string, unless a limiting range of line numbers is specified. The format of this number range is the same used by the LIST command. If the backslash character is used to enclose the string, BASIC keyword tokens within the string are recognized as such, but if the string is enclosed in quotes, such words are treated as their literal string of characters. For examples of this distinction, see the CHANGE command.

FLIST

Syntax: FLIST *"BASIC program filename"*

This reads a BASIC program file on disk and lists the program to the screen without entering it into memory or otherwise affecting the program currently in memory. FLIST allows you to make certain you've got the right program before you try to LOAD or MERGE it.

HELP

Syntax: HELP

HELP displays the BASIC program line that was executing when the program was stopped, and highlights in reverse video the last character read by the program. It's helpful for finding which statement of a multistatement program line caused an error. On the 64 it's particularly useful when the screen has been changed from text to high-resolution graphics, and error messages cannot be read. Since changing the screen back to text with the RUN/STOP–RESTORE

combination also erases the error message, HELP can show where the error occurred.

HEX

Syntax: HEX $hexadecimal number
or
HEX decimal number

You can convert decimal numbers to hexadecimal notation and vice versa, with this feature. If the number entered is preceded by a dollar sign, it's considered a hexadecimal number in the range $0000 to $FFFF, and its decimal equivalent is displayed. If no dollar sign is entered, a decimal number in the range 0 to 65535 is converted to hex and then displayed.

KILL

Syntax: KILL

This disables BASIC Aid. To restart the program, type SYS 49152.

MERGE

Syntax: MERGE *"BASIC program filename"*

This command reads a BASIC program file from disk, lists each line on the screen, and enters the line just as if it had been typed in from the keyboard. To use MERGE, first load one program into the computer's memory. Do not run it. Then type MERGE *"filename"*, using the filename of the program you want to merge into the first. The program lines using numbers not already found in the first program (the one in memory) are added, while those that duplicate numbers already in use will replace those lines.

OLD

Syntax: OLD

As you might have guessed, the OLD command is used to undo the effects of an inadvertent NEW command. As long as no program lines are entered after NEW has been entered, OLD can recover the program. It can also be used to restore the program after a cold start (either from the COLD command, or using a reset switch connected to the user port for recovering from a crash).

OFF

Syntax: OFF

This command restores the normal IRQ vector, and turns off the interrupt-driven functions, namely, program scrolling, quote mode/

insert mode escape, and keyprint. (See the description of these functions below.)

READ

Syntax: READ *"sequential filename"*

The READ command reads a sequential file from disk and prints its contents to the screen. It can be used for viewing text or data files.

RENUMBER

Syntax: RENUMBER
or
RENUMBER *starting line number*
or
RENUMBER *starting line number, line increment*

RENUMBER completely renumbers the BASIC program in memory, including line number references in GOTO, GOSUB, and IF-THEN statements. If no numbers are entered after RENUMBER, the program will be renumbered starting at line 100, with line increments of 10. You can specify a different starting line number or a different increment value.

REPEAT

Syntax: REPEAT

This command is used to toggle the repeat key function. When BASIC Aid is started, all keys repeat if held down. Entering REPEAT disables this feature for all but the cursor and space keys; typing it again reenables it.

SCROLL

Syntax: SCROLL

SCROLL enables all of the interrupt-driven keystroke commands. These are:

1. Program scrolling. When you move the cursor to the bottom-left corner of the screen and press the cursor down key, the listing will roll up, and the next program line will be printed at the bottom of the screen. If you move the cursor to the top-left corner of the screen and press the cursor up key combination, the listing will roll down, and the previous program line will be displayed at the top.
2. Quote mode/Insert mode escape. By pressing the f1 key, quote mode and insert mode are canceled, allowing you to move the cursor normally.

3. Keyprint. This function allows you to send the characters currently on the text screen to a printer by pressing the f8 key (SHIFTed f7). In effect, this executes the CRT command (see above), but can be used while a program is running. With these you can make a hard copy of output normally only printed on the screen, such as program instructions. You must be careful, however, not to try to use this function while serial bus operations such as disk accesses are taking place, since this will lock up the system. This means that the keyprint feature cannot be used to print the directory displayed by the DOS wedge, or to print programs or text displayed by the READ or FLIST commands. Likewise, trying to use it without a printer connected may lock up the system or abort the program that is running.

When BASIC Aid is started, all of these interrupt-driven functions are enabled. During the course of programming, however, there are several ways in which the normal interrupt can be restored (such as by hitting the RUN/STOP–RESTORE combination, or by using the OFF command). To restart these functions, use the SCROLL command.

START

Syntax: Start *"program filename"*

The loading address for program files is located in the first two bytes of the file. This command reads those bytes from the specified program file on the disk and displays the starting address in decimal and hexadecimal. START is handy for finding where a nonrelocatable machine language program (one that is loaded with the LOAD "NAME",8,1 format) starts.

DOS Support Commands

Syntax: > *disk command* or *@disk command*
> $ or @$ (directory)
/ program name (LOAD)
↑ *program name* (LOAD and RUN)

BASIC Aid also supports many of the commands found in the DOS support program. The greater than (>) and commercial at (@) signs are used to communicate with the disk on the command channel. Using the symbol alone reads the error channel and prints it to the screen. Entering the symbol followed by a dollar sign prints the directory on the screen without altering the program in memory. Typing in the symbol followed by a command string sends that command to the disk, just as if you had typed in the BASIC line OPEN 1,8,15: PRINT#1,"command string": CLOSE 1. These commands include:

1. > *S0:filename.* Erases the named file from the disk. If wildcards such as * or ? appear in the filename, more than one file may be erased.
2. > *R0:new filename = old filename.* Changes the name of the disk file from *old filename* to *new filename.*
3. > *C0:new filename = old filename.* This command copies disk file *old filename* to the file *new filename.*
4. > *V0.* Performs a disk validation or collection, which reclaims disk blocks marked as in use, but which are in fact not used.
5. > *N0:disk name, ID.* Formats the disk for use, erasing all information that it previously contained, and giving it the title and disk ID number entered in the command.

Two additional DOS support functions are included. Entering the backslash (/) followed by the program filename loads that program from disk. Typing the up arrow (↑) followed by the program filename loads and runs the program.

BASIC Aid

Be sure to read "Using the Machine Language Editor: MLX," Appendix D, before typing in this program.

```
49152 :169,000,141,136,003,141,078
49158 :137,003,173,243,206,174,174
49164 :244,206,224,160,176,007,005
49170 :133,055,134,056,032,089,005
49176 :166,162,015,189,043,192,023
49182 :149,115,202,016,248,162,154
49188 :019,032,059,192,076,089,247
49194 :193,230,122,208,002,230,003
49200 :123,173,255,001,076,154,062
49206 :192,234,076,191,192,189,104
49212 :047,206,240,006,032,210,033
49218 :255,232,208,245,096,208,030
49224 :014,120,162,023,189,162,230
49230 :227,149,115,202,016,248,011
49236 :076,186,199,076,217,205,019
49242 :173,141,002,041,001,208,144
49248 :249,032,125,192,208,020,154
49254 :032,125,192,240,251,201,119
49260 :255,240,247,032,125,192,175
49266 :201,255,208,249,169,000,172
49272 :133,198,076,225,255,173,156
49278 :001,220,205,001,220,208,213
49284 :248,201,239,096,133,098,123
49290 :162,144,056,032,073,188,025
49296 :076,221,189,230,135,208,179
```

```
49302 :002,230,136,096,133,131,110
49308 :134,151,186,189,001,001,050
49314 :201,140,240,035,166,123,043
49320 :224,002,240,011,134,138,149
49326 :166,122,134,137,076,183,224
49332 :192,164,133,166,151,165,127
49338 :131,201,058,176,200,201,129
49344 :032,240,003,076,179,227,181
49350 :076,115,000,189,002,001,069
49356 :201,164,208,231,032,183,199
49362 :192,144,077,165,131,016,167
49368 :002,230,122,162,000,134,098
49374 :127,132,133,164,122,185,061
49380 :000,002,056,253,086,206,063
49386 :240,019,201,128,240,019,057
49392 :230,127,232,189,085,206,029
49398 :016,250,189,086,206,208,177
49404 :228,240,182,232,200,208,006
49410 :224,132,122,104,104,165,085
49416 :127,010,170,189,192,206,134
49422 :072,189,191,206,072,032,008
49428 :181,192,076,115,000,032,104
49434 :207,195,141,136,003,076,016
49440 :002,194,104,104,165,131,220
49446 :032,107,169,240,041,173,032
49452 :136,003,240,036,024,165,136
49458 :020,109,136,003,133,099,038
49464 :165,021,105,000,032,136,003
49470 :192,162,000,189,001,001,095
49476 :240,006,157,119,002,232,056
49482 :208,245,169,032,157,119,236
49488 :002,232,134,198,076,159,113
49494 :164,208,020,120,173,245,248
49500 :206,141,020,003,173,246,113
49506 :206,141,021,003,032,031,020
49512 :206,088,076,116,164,076,062
49518 :217,205,044,141,002,240,191
49524 :045,032,133,198,162,000,174
49530 :134,199,134,198,134,197,094
49536 :228,212,240,004,134,212,134
49542 :208,004,228,216,240,016,022
49548 :134,216,230,198,164,211,013
49554 :136,169,032,145,209,169,238
49560 :157,141,119,002,165,197,165
49566 :201,003,240,206,201,004,245
49572 :240,210,173,248,206,072,033
49578 :173,247,206,072,008,072,180
49584 :072,072,076,049,234,032,199
49590 :061,195,165,095,166,096,192
```

```
49596 :133,036,134,037,032,019,067
49602 :166,165,095,166,096,144,002
49608 :010,160,001,177,095,240,115
49614 :004,170,136,177,095,133,153
49620 :122,134,123,165,036,056,080
49626 :229,122,170,165,037,229,146
49632 :123,168,176,030,138,024,115
49638 :101,045,133,045,152,101,039
49644 :046,133,046,160,000,177,030
49650 :122,145,036,200,208,249,178
49656 :230,123,230,037,165,046,055
49662 :197,037,176,239,032,051,218
49668 :165,165,034,166,035,024,081
49674 :105,002,133,045,144,001,184
49680 :232,134,046,032,089,166,203
49686 :076,116,164,032,121,165,184
49692 :032,115,000,133,131,162,089
49698 :000,134,073,032,253,194,208
49704 :165,127,201,003,208,007,239
49710 :162,002,134,073,032,253,190
49716 :194,032,115,000,240,003,124
49722 :032,253,174,032,061,195,037
49728 :165,095,166,096,133,122,073
49734 :134,123,032,215,170,208,184
49740 :011,200,152,024,101,122,174
49746 :133,122,144,002,230,123,068
49752 :032,201,197,240,005,032,027
49758 :103,195,176,003,076,002,137
49764 :194,132,085,230,085,164,222
49770 :085,166,049,165,050,133,242
49776 :131,177,122,240,216,221,195
49782 :000,002,208,237,232,200,229
49788 :198,131,208,241,136,132,146
49794 :011,132,151,165,073,240,134
49800 :091,032,122,195,165,052,025
49806 :056,229,050,133,158,240,240
49812 :040,200,240,202,177,122,105
49818 :208,249,024,152,101,158,022
49824 :201,002,144,064,201,075,079
49830 :176,060,165,158,016,002,231
49836 :198,131,024,101,011,133,002
49842 :151,176,005,032,180,195,149
49848 :240,003,032,156,195,165,207
49854 :151,056,229,052,168,200,022
49860 :165,052,240,015,133,133,166
49866 :166,051,189,000,002,145,243
49872 :122,232,200,198,133,208,021
49878 :245,024,165,045,101,158,184
49884 :133,045,165,046,101,131,073
```

```
49890 :133,046,165,122,166,123,213
49896 :133,095,134,096,166,067,155
49902 :165,068,032,018,196,032,237
49908 :090,192,240,158,164,151,215
49914 :076,101,194,164,122,200,083
49920 :148,049,169,000,149,050,053
49926 :185,000,002,240,047,197,165
49932 :131,240,005,246,050,200,116
49938 :208,242,132,122,096,208,002
49944 :033,141,003,002,142,004,093
49950 :002,140,005,002,008,104,035
49956 :141,002,002,169,164,072,074
49962 :169,116,072,056,108,022,073
49968 :003,201,171,240,004,201,100
49974 :045,208,001,096,076,217,185
49980 :205,144,005,240,003,032,177
49986 :049,195,032,107,169,032,138
49992 :019,166,032,121,000,240,138
49998 :011,032,049,195,032,115,000
50004 :000,032,107,169,208,224,056
50010 :165,020,005,021,208,006,003
50016 :169,255,133,020,133,021,059
50022 :096,032,201,197,133,067,060
50028 :032,201,197,133,068,165,136
50034 :020,197,067,165,021,229,045
50040 :068,096,165,122,133,034,226
50046 :165,123,133,035,165,045,024
50052 :133,036,165,046,133,037,170
50058 :096,165,034,197,036,208,106
50064 :004,165,035,197,037,096,166
50070 :230,034,208,002,230,035,121
50076 :164,011,200,177,034,164,138
50082 :151,200,145,034,032,139,095
50088 :195,208,235,096,165,036,079
50094 :208,002,198,037,198,036,085
50100 :164,011,177,036,164,151,115
50106 :145,036,032,139,195,208,173
50112 :235,096,201,034,208,008,206
50118 :072,165,015,073,128,133,016
50124 :015,104,096,032,107,169,215
50130 :165,021,208,004,165,020,025
50136 :016,002,169,127,096,076,190
50142 :217,205,208,251,032,232,087
50148 :195,076,116,164,032,215,002
50154 :170,133,073,166,043,165,216
50160 :044,134,095,133,096,160,134
50166 :000,177,095,170,200,177,041
50172 :095,208,003,076,115,000,237
50178 :197,138,144,235,228,137,057
```

```
50184 :144,231,200,177,095,170,001
50190 :200,177,095,044,160,000,178
50196 :132,127,132,015,032,205,151
50202 :189,169,032,164,127,041,236
50208 :127,032,210,255,032,194,114
50214 :195,169,000,133,199,200,166
50220 :036,073,016,023,166,096,198
50226 :152,056,101,095,144,001,087
50232 :232,228,138,144,010,197,237
50238 :137,144,006,169,001,133,140
50244 :073,133,199,177,095,240,217
50250 :017,016,212,201,255,240,247
50256 :208,036,015,048,204,132,211
50262 :127,032,122,196,048,193,036
50268 :076,215,170,096,162,160,203
50274 :160,157,134,136,132,135,184
50280 :056,233,127,170,160,000,082
50286 :202,240,238,032,147,192,137
50292 :177,135,016,249,048,244,217
50298 :032,096,196,200,177,135,190
50304 :048,221,032,210,255,208,078
50310 :246,032,107,169,164,020,104
50316 :166,021,152,005,021,208,201
50322 :002,160,100,132,053,134,215
50328 :054,162,000,161,122,208,091
50334 :004,169,010,208,010,032,079
50340 :253,174,032,107,169,165,040
50346 :020,166,021,133,051,134,183
50352 :052,032,142,166,032,201,033
50358 :197,032,201,197,208,033,026
50364 :032,171,197,032,201,197,250
50370 :032,201,197,208,003,076,143
50376 :002,194,032,201,197,165,223
50382 :099,145,122,032,201,197,234
50388 :165,098,145,122,032,182,188
50394 :197,240,226,032,201,197,031
50400 :032,201,197,032,201,197,060
50406 :201,034,208,011,032,201,149
50412 :197,240,197,201,034,208,033
50418 :247,240,238,170,240,188,029
50424 :016,233,162,004,221,081,197
50430 :206,240,005,202,208,248,083
50436 :240,221,165,122,133,059,176
50442 :165,123,133,060,032,115,126
50448 :000,176,211,032,107,169,199
50454 :032,080,197,165,060,133,177
50460 :123,165,059,133,122,160,022
50466 :000,162,000,189,000,001,130
50472 :201,048,144,017,072,032,042
```

```
50478 :115,000,144,003,032,129,213
50484 :197,104,160,000,145,122,012
50490 :232,208,232,032,115,000,109
50496 :176,008,032,144,197,032,141
50502 :121,000,144,248,201,044,060
50508 :240,184,208,150,032,171,037
50514 :197,032,201,197,032,201,174
50520 :197,208,008,169,255,133,034
50526 :099,133,098,048,014,032,006
50532 :201,197,197,020,208,015,170
50538 :032,201,197,197,021,208,194
50544 :011,032,209,189,169,032,242
50550 :076,210,255,032,201,197,065
50556 :032,182,197,240,210,032,249
50562 :161,197,230,151,032,180,057
50568 :195,230,045,208,002,230,022
50574 :046,096,032,161,197,198,104
50580 :151,032,156,195,165,045,124
50586 :208,002,198,046,198,045,083
50592 :096,032,122,195,160,000,253
50598 :132,011,132,151,096,165,085
50604 :053,133,099,165,054,133,041
50610 :098,076,142,166,165,099,156
50616 :024,101,051,133,099,165,245
50622 :098,101,052,133,098,032,192
50628 :201,197,208,251,096,160,029
50634 :000,230,122,208,002,230,226
50640 :123,177,122,096,076,116,150
50646 :164,208,089,165,045,133,250
50652 :095,165,046,133,096,165,152
50658 :095,197,047,165,096,229,031
50664 :048,176,233,160,000,132,213
50670 :036,200,177,095,010,102,090
50676 :036,074,153,069,000,136,200
50682 :016,244,036,036,240,030,084
50688 :016,051,080,089,032,112,124
50694 :198,162,037,169,061,032,153
50700 :001,206,160,002,177,095,141
50706 :072,200,177,095,168,104,066
50712 :032,145,179,076,044,198,186
50718 :032,112,198,169,061,032,122
50724 :210,255,032,133,177,032,107
50730 :162,187,032,215,189,076,135
50736 :090,198,076,217,205,032,098
50742 :112,198,162,036,169,061,024
50748 :032,001,206,169,034,032,022
50754 :210,255,160,004,177,095,199
50760 :133,035,136,177,095,133,013
50766 :034,136,177,095,032,036,076
```

```
50772 :171,169,034,032,210,255,187
50778 :032,215,170,032,090,192,053
50784 :240,032,024,165,095,105,245
50790 :007,133,095,144,002,230,201
50796 :096,076,225,197,165,069,168
50802 :032,210,255,165,070,240,062
50808 :003,032,210,255,096,208,156
50814 :179,032,133,198,076,116,092
50820 :164,169,004,133,035,169,038
50826 :000,133,034,169,004,170,136
50832 :160,255,032,186,255,032,040
50838 :192,255,032,183,255,208,251
50844 :107,162,004,032,201,255,149
50850 :169,025,133,037,169,013,196
50856 :133,015,032,210,255,169,214
50862 :017,174,024,208,224,021,074
50868 :208,002,169,145,032,210,178
50874 :255,160,000,177,034,041,085
50880 :127,170,177,034,069,015,016
50886 :016,011,177,034,133,015,072
50892 :041,128,073,146,032,210,066
50898 :255,138,201,032,176,004,248
50904 :009,064,208,014,201,064,008
50910 :144,010,201,096,176,004,085
50916 :009,128,208,002,073,192,072
50922 :032,210,255,200,192,040,139
50928 :144,203,165,034,105,039,162
50934 :133,034,144,002,230,035,056
50940 :198,037,208,166,169,013,019
50946 :032,210,255,032,210,255,228
50952 :169,004,032,195,255,076,227
50958 :204,255,076,217,205,076,023
50964 :002,194,208,248,165,043,112
50970 :133,135,133,034,165,044,158
50976 :133,136,133,035,160,003,120
50982 :177,135,145,034,136,016,169
50988 :249,200,132,015,177,034,083
50994 :200,017,034,240,220,160,153
51000 :004,177,135,201,058,208,071
51006 :006,200,177,135,240,048,100
51012 :136,177,135,201,143,240,076
51018 :041,036,015,112,004,201,227
51024 :058,240,008,201,032,208,059
51030 :009,036,015,048,005,032,231
51036 :147,192,208,229,170,169,183
51042 :064,005,015,133,015,138,212
51048 :145,034,200,201,000,240,156
51054 :046,032,194,195,208,209,226
51060 :136,036,015,112,013,160,076
```

```
51066 :000,024,165,135,105,004,043
51072 :133,135,144,002,230,136,140
51078 :177,135,240,005,032,147,102
51084 :192,208,247,036,015,080,150
51090 :005,145,034,200,016,005,039
51096 :032,147,192,208,135,152,250
51102 :170,160,001,177,034,133,065
51108 :136,136,177,034,133,135,147
51114 :024,138,101,034,133,034,122
51120 :144,002,230,035,076,036,187
51126 :199,208,010,120,162,012,125
51132 :032,184,252,088,076,116,168
51138 :164,076,217,205,208,251,035
51144 :160,001,165,044,145,043,246
51150 :076,002,194,160,000,132,002
51156 :134,201,000,240,032,201,252
51162 :036,240,064,169,008,133,100
51168 :186,032,177,255,169,111,130
51174 :032,147,255,177,122,240,179
51180 :006,032,168,255,200,208,081
51186 :246,032,174,255,076,116,117
51192 :164,169,008,133,186,032,172
51198 :180,255,169,111,032,150,127
51204 :255,032,165,255,201,013,157
51210 :240,005,032,210,255,208,192
51216 :244,032,171,255,076,128,154
51222 :200,169,094,133,134,160,144
51228 :000,200,177,122,208,251,218
51234 :132,183,165,122,133,187,188
51240 :165,123,133,188,169,008,058
51246 :133,186,165,134,208,082,186
51252 :169,096,133,185,032,213,112
51258 :243,165,186,032,180,255,095
51264 :165,185,032,150,255,032,115
51270 :215,170,169,000,133,144,133
51276 :160,003,132,183,032,165,239
51282 :255,170,164,144,208,037,036
51288 :032,165,255,164,144,208,032
51294 :030,198,183,208,237,032,214
51300 :205,189,032,063,171,032,024
51306 :165,255,240,005,032,210,245
51312 :255,208,246,032,215,170,214
51318 :160,002,032,090,192,208,034
51324 :209,032,066,246,032,215,156
51330 :170,076,116,164,169,000,057
51336 :133,144,133,147,032,213,170
51342 :255,165,144,041,191,208,122
51348 :029,165,175,133,046,165,093
51354 :174,133,045,032,089,166,025
```

```
51360 :032,051,165,165,134,201,140
51366 :094,240,003,076,116,164,091
51372 :032,142,166,076,174,167,161
51378 :076,004,247,189,240,236,146
51384 :133,122,032,195,200,133,231
51390 :123,032,040,203,096,181,097
51396 :217,041,003,013,136,002,096
51402 :096,076,129,234,165,198,076
51408 :240,249,165,211,201,002,252
51414 :176,243,173,119,002,041,200
51420 :127,201,017,208,234,173,156
51426 :001,008,013,002,008,240,242
51432 :226,169,000,141,132,003,135
51438 :141,134,003,169,039,141,097
51444 :135,003,169,024,141,133,081
51450 :003,032,025,203,173,119,037
51456 :002,048,092,166,214,224,234
51462 :024,208,083,142,131,003,085
51468 :142,129,003,202,048,059,083
51474 :180,217,016,249,032,181,125
51480 :200,176,244,032,107,169,184
51486 :230,020,208,002,230,021,229
51492 :032,019,166,176,016,208,141
51498 :014,032,049,202,032,049,164
51504 :202,169,000,133,020,133,193
51510 :021,240,235,032,049,202,065
51516 :206,129,003,165,217,016,028
51522 :246,032,096,202,165,217,000
51528 :048,003,032,049,202,162,056
51534 :000,189,120,002,157,119,153
51540 :002,232,228,198,208,245,173
51546 :198,198,076,129,234,166,067
51552 :214,208,249,142,129,003,017
51558 :142,131,003,202,232,224,012
51564 :025,176,222,180,217,016,176
51570 :247,032,181,200,176,242,168
51576 :032,107,169,032,019,166,133
51582 :165,095,166,096,197,043,120
51588 :208,018,228,044,208,014,084
51594 :032,019,202,032,019,202,132
51600 :169,255,133,020,133,021,107
51606 :208,227,133,187,202,134,217
51612 :188,160,255,200,177,187,043
51618 :170,208,250,200,177,187,074
51624 :197,095,208,246,200,177,011
51630 :187,197,096,208,239,136,213
51636 :152,024,101,187,133,095,104
51642 :165,188,105,000,133,096,105
51648 :165,241,048,003,032,019,188
```

```
51654 :202,032,019,202,076,067,028
51660 :201,189,240,236,133,036,215
51666 :032,195,200,133,037,181,220
51672 :217,009,128,096,048,002,204
51678 :041,127,149,217,172,134,038
51684 :003,136,096,032,197,200,124
51690 :133,035,200,177,034,145,190
51696 :036,165,035,072,041,003,080
51702 :009,216,133,035,165,037,073
51708 :072,041,003,009,216,133,214
51714 :037,177,034,145,036,104,023
51720 :133,037,104,133,035,204,142
51726 :135,003,144,218,096,174,016
51732 :133,003,232,202,032,205,059
51738 :201,180,216,032,220,201,052
51744 :236,132,003,240,042,189,106
51750 :239,236,133,034,181,216,053
51756 :032,231,201,176,230,174,064
51762 :132,003,202,232,032,205,088
51768 :201,180,218,032,220,201,084
51774 :236,133,003,176,012,189,043
51780 :241,236,133,034,181,218,087
51786 :032,231,201,176,230,169,089
51792 :032,200,145,036,204,135,064
51798 :003,144,248,181,217,009,120
51804 :128,149,217,096,162,000,076
51810 :134,015,142,130,003,174,184
51816 :129,003,189,240,236,133,010
51822 :187,032,195,200,133,188,021
51828 :032,076,203,133,099,032,179
51834 :079,203,032,136,192,162,158
51840 :000,189,001,001,240,006,053
51846 :032,188,202,232,208,245,217
51852 :169,032,041,127,032,188,217
51858 :202,032,079,203,008,032,190
51864 :194,195,040,048,004,240,105
51870 :069,208,239,201,255,240,090
51876 :235,036,015,048,231,032,249
51882 :096,196,200,177,135,048,254
51888 :221,132,138,032,188,202,065
51894 :164,138,208,242,169,032,111
51900 :032,090,203,160,000,145,050
51906 :187,165,188,072,041,003,082
51912 :009,216,133,188,173,134,029
51918 :002,145,187,104,133,188,197
51924 :230,187,208,002,230,188,233
51930 :238,130,003,173,130,003,127
51936 :201,040,240,001,096,173,207
51942 :131,003,240,017,172,129,154
```

```
51948 :003,192,024,240,039,138,104
51954 :072,206,133,003,032,049,225
51960 :202,176,008,138,072,238,058
51966 :132,003,032,019,202,032,162
51972 :197,200,133,188,041,127,122
51978 :149,217,189,240,236,133,150
51984 :187,169,000,141,130,003,134
51990 :104,170,096,165,207,240,236
51996 :010,160,000,132,207,164,189
52002 :211,165,206,145,209,096,042
52008 :160,000,140,130,003,240,201
52014 :016,230,122,208,002,230,086
52020 :123,238,130,003,173,130,081
52026 :003,201,040,176,232,177,119
52032 :122,201,058,176,226,201,024
52038 :032,240,230,076,179,227,030
52044 :032,079,203,230,095,208,155
52050 :002,230,096,160,000,177,235
52056 :095,096,133,137,041,127,205
52062 :201,032,008,041,063,040,223
52068 :176,002,009,128,036,137,076
52074 :016,002,009,064,096,104,141
52080 :104,032,215,170,032,204,101
52086 :255,169,001,032,195,255,001
52092 :169,002,032,195,255,076,085
52098 :116,164,032,207,255,032,168
52104 :207,255,240,227,165,144,094
52110 :208,223,162,255,032,207,205
52116 :255,133,020,032,207,255,026
52122 :133,021,232,224,078,176,250
52128 :008,032,207,255,157,000,051
52134 :002,208,243,232,232,157,216
52140 :000,002,232,232,232,134,236
52146 :011,096,032,227,204,162,142
52152 :002,134,152,032,198,255,189
52158 :032,132,203,032,017,205,043
52164 :032,019,166,144,068,160,017
52170 :001,177,095,133,035,165,040
52176 :045,133,034,165,096,133,046
52182 :037,165,095,136,241,095,215
52188 :024,101,045,133,045,133,189
52194 :036,165,046,105,255,133,198
52200 :046,229,096,170,056,165,226
52206 :095,229,045,168,176,003,186
52212 :232,198,037,024,101,034,102
52218 :144,003,198,035,024,177,063
52224 :034,145,036,200,208,249,104
52230 :230,035,230,037,202,208,180
52236 :242,032,089,166,032,051,112
```

```
52242 :165,173,000,002,208,003,057
52248 :076,183,203,024,165,045,208
52254 :133,090,101,011,133,088,074
52260 :164,046,132,091,144,001,102
52266 :200,132,089,032,184,163,074
52272 :165,020,164,021,141,254,045
52278 :001,140,255,001,165,049,153
52284 :164,050,133,045,132,046,118
52290 :164,011,136,185,252,001,047
52296 :145,095,136,016,248,032,232
52302 :089,166,032,051,165,076,145
52308 :183,203,162,001,032,198,095
52314 :255,032,207,255,133,137,085
52320 :032,207,255,133,138,005,098
52326 :137,201,048,240,022,166,148
52332 :137,165,138,032,001,206,019
52338 :032,207,255,032,210,255,081
52344 :201,013,208,246,104,104,228
52350 :076,116,203,032,207,255,247
52356 :201,013,208,249,076,204,059
52362 :255,240,068,201,036,240,154
52368 :023,032,121,000,032,243,083
52374 :188,032,247,183,165,021,218
52380 :133,194,165,020,133,193,226
52386 :032,066,205,076,116,164,053
52392 :169,000,133,098,133,099,032
52398 :169,004,133,100,032,201,045
52404 :197,240,020,032,224,205,074
52410 :162,004,006,099,038,098,081
52416 :202,208,249,005,099,133,064
52422 :099,198,100,208,231,032,042
52428 :209,189,076,128,200,076,058
52434 :217,205,032,207,255,133,235
52440 :194,162,000,032,059,192,087
52446 :104,104,076,053,205,032,028
52452 :148,205,032,207,255,133,184
52458 :193,201,001,208,229,032,074
52464 :207,255,133,194,076,215,040
52470 :170,032,227,204,032,132,019
52476 :203,032,017,205,208,248,141
52482 :104,104,160,000,185,129,172
52488 :163,240,045,032,210,255,185
52494 :200,208,245,169,255,133,200
52500 :095,169,001,133,096,133,135
52506 :073,166,020,165,021,032,247
52512 :018,196,032,090,192,240,032
52518 :219,096,032,148,205,032,002
52524 :207,255,133,193,032,207,047
52530 :255,133,194,032,059,205,160
```

```
52536 :076,116,203,166,193,165,207
52542 :194,032,205,189,162,032,108
52548 :169,036,032,001,206,032,032
52554 :235,205,076,215,170,240,191
52560 :022,162,254,134,193,232,053
52566 :134,194,032,148,205,032,063
52572 :207,255,032,020,206,164,208
52578 :144,240,246,208,206,056,174
52584 :165,045,229,043,133,193,144
52590 :165,046,229,044,133,194,153
52596 :032,059,205,076,116,164,000
52602 :032,152,205,032,207,255,237
52608 :164,144,008,032,210,255,173
52614 :040,208,008,032,090,192,192
52620 :208,239,076,004,205,076,180
52626 :113,203,169,000,240,002,105
52632 :169,002,133,134,032,231,085
52638 :255,032,087,226,166,183,083
52644 :240,051,134,015,169,001,006
52650 :133,184,169,008,133,186,215
52656 :169,015,133,185,169,000,079
52662 :133,183,032,192,255,032,241
52668 :204,255,165,015,133,183,119
52674 :169,002,133,184,169,008,091
52680 :133,186,165,134,133,185,112
52686 :032,192,255,032,086,204,239
52692 :162,002,076,198,255,169,050
52698 :255,133,058,076,008,175,155
52704 :201,058,008,041,015,040,075
52710 :144,002,105,008,096,165,238
52716 :194,032,242,205,165,193,243
52722 :072,074,074,074,074,032,130
52728 :010,206,170,104,041,015,026
52734 :032,010,206,072,138,032,232
52740 :210,255,104,076,210,255,090
52746 :024,105,246,144,002,105,124
52752 :006,105,058,096,230,193,192
52758 :208,006,230,194,208,002,102
52764 :230,190,096,173,138,002,089
52770 :073,128,141,138,002,096,100
52776 :032,031,206,076,116,164,153
52782 :167,013,078,079,084,032,243
52788 :066,065,083,073,067,044,194
52794 :032,083,084,065,082,084,232
52800 :061,000,147,066,065,083,230
52806 :073,067,045,065,073,068,205
52812 :032,050,013,017,000,000,188
52818 :137,138,141,167,072,069,038
52824 :076,208,065,085,084,207,045
```

```
52830 :066,082,069,065,203,067,134
52836 :072,065,078,071,197,068,139
52842 :069,076,069,084,197,070,159
52848 :076,073,083,212,068,085,197
52854 :077,208,070,073,078,196,052
52860 :072,069,216,067,082,212,074
52866 :075,073,076,204,077,069,192
52872 :082,071,197,082,069,078,203
52878 :085,077,066,069,210,079,216
52884 :070,198,080,065,067,203,063
52890 :082,069,065,196,083,067,204
52896 :082,079,076,204,083,084,000
52902 :065,082,212,082,069,080,244
52908 :069,065,212,190,192,175,051
52914 :222,083,073,090,197,079,154
52920 :076,196,067,079,076,196,106
52926 :000,223,195,024,193,022,079
52932 :195,024,194,180,193,246,204
52938 :204,214,197,024,194,138,149
52944 :204,124,198,070,192,179,151
52950 :203,134,196,182,199,021,125
52956 :199,121,205,086,193,039,039
52962 :205,039,206,208,199,208,011
52968 :199,024,200,022,200,078,187
52974 :205,197,199,225,252,000,036
52980 :192,156,193,206,200,083,250
52986 :076,049,057,048,052,056,076
52992 :052,255,013,013,013,013,103
```

"Basic Aid" Clip-Out Quick Reference Card

Command and Syntax	Function
AUTO ***line increment***	Auto line numbering
BREAK	Enter monitor
CHANGE/*string/string/,line#*	Search and replace string
or	
CHANGE ***"string"string",line#***	
COLD	Cold start
CRT	Screen print
DELETE ***line# range***	Block delete
DUMP	List variables and values
FIND/*string/string/line#*	Find string
or	
FIND ***"string"string",line#***	
FLIST ***"filename"***	List to screen
HELP	Last line executed
HEX ***number***	Convert from decimal to hexadecimal; vice versa
KILL	Disable BASIC Aid
MERGE ***"filename"***	Merge two programs
OLD	Reclaim NEWed programs
OFF	Restore normal IRQ vector
READ ***"sequential filename"***	Read sequential file
RENUMBER	Renumber program lines
REPEAT	Toggle repeat key function
SCROLL	Enable interrupt-driven commands
START ***"filename"***	Starting address of program
$ or @$	List directory
/filename	LOAD program
↑filename	LOAD and RUN program

Jeff Young

Auto Line Numbering

This short routine is a handy, time-saving utility for programmers.

"Auto Line Numbering" is a utility which automatically generates a line number for the current BASIC program statement you're entering. As written, the routine numbers programs beginning with line 100 and increments by tens (100, 110, 120, and so on). This can be easily modified.

How to Use the Program

Auto Line Numbering consists of a BASIC loader which places a machine language subroutine into a free block of memory at location 49152 ($C000). This area of memory is not used by BASIC, so the program should be safe.

Type in the program and SAVE it. After LOADing, type RUN, press RETURN, type NEW, press RETURN, then type SYS 49152. If you wish to leave the program for any reason, just press RETURN immediately after you see a new line number. To return to the program, type SYS 49160. This will continue generating line numbers from where you left off.

Although the program will always begin numbering with 100 and increment by tens, you can modify either of these if you wish. If you want to begin with a number other than 100, determine the number with which you want to start, then subtract ten. POKE this number in low-byte/high-byte format into 251 and 252, then SYS 49160.

For example, if you wish to begin with line 1000, subtract ten. The number you are now working with is 990. To determine low-byte/high-byte, divide 990 by 256. The result, 3, is the number you POKE into location 252—POKE 252,3. The remainder of the division is 222. POKE 251,222. The *low byte* is location 251, and the *high byte*, 252.

If you wished to begin the line numbering with 1000 then, you'd enter:

```
POKE 251,222:POKE 252,3
SYS 49160
```

To change the increment from ten, POKE the desired number into location 49179. If you want to increment by fives, for example, you'd enter:

```
POKE 49179,5
```

This utility program can save you a lot of time when programming, and it provides a neat, structured sequence for program line numbers.

Auto Line Numbering

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

```
1 X=49152                                          :rem 203
2 READY:IFY=-1THEN4                                :rem 199
3 POKEX,Y:X=X+1:Z=Z+Y:GOTO2                        :rem 22
4 IFZ<>12374THENPRINT"ERROR IN DATA STATEMENTS":EN
  D                                                :rem 236
100 DATA169,90,133,251,169,0,133,252,169,19,141,2,
    3,169,192,141,3,3,96,32,25                     :rem 203
110 DATA192,76,134,164,24,169,10,101,251,133,251,1
    44,2,230,252,165,251,133,99                    :rem 246
120 DATA165,252,133,98,162,144,56,32,73,188,32,221
    ,189,162,0,189,1,1,240,9,32                      :rem 4
130 DATA210,255,157,0,2,232,208,242,32,18,225,201,
    13,240,3,76,105,165,56,165                     :rem 182
140 DATA251,233,20,176,2,198,252,169,131,141,2,3,1
    69,164,141,3,3,76,118,165,-1                   :rem 36
```

Charles Kluepfel

Numeric Keypad

Turn your keyboard into a "Numeric Keypad" for more efficient numeric input. The program lets you toggle to standard or numeric keypad.

You could type in numbers much faster and with fewer errors if the Commodore 64 had a numeric keypad. This program offers this handy feature by redefining a set of keys to represent numbers instead of letters.

When you run "Numeric Keypad," your computer will behave normally until CTRL-N is pressed. The cursor disappears until you press another key. Then the M, J, K, L, U, I, and O keys will be 0, 1, 2, 3, 4, 5, and 6. By using these along with the numeric keys 7, 8, and 9, you have a numeric keypad. Pressing CTRL-N toggles the keyboard back into its normal mode (again causing the cursor to disappear until you press a key). If you press RUN/STOP–RESTORE, however, you won't be able to use the keypad option. You'll need to reRUN the program to restore the feature.

You can put press-apply transfer numbers on the affected keys to help you remember which number each key represents. You should use very small ones, so they won't interfere with the normal identification of the keys. (Transfer letters and numbers are available at many art supply stores.)

Use Numeric Keypad in a Program

You also can activate and deactivate the numeric keypad from a program, in anticipation of numeric or nonnumeric input, by POKEing location 50216 with 255 or 0 respectively. The user can always override this with CTRL-N. (CTRL-N is never passed to the program, but serves only as the toggle function.) Just don't POKE any value other than 0 or 255, because that would prevent you from toggling with CTRL-N.

If you prefer that the keypad start out as activated, change the next-to-last DATA item in line 520 from 0 to 255.

Redefining the Keys

To redefine the 64 keys, we transfer the Kernal from ROM into RAM, change it to intercept the M, J, K, L, U, I, and O keys, and convert the data to the appropriate numbers.

Lines 3 and 4 POKE the machine language into an unused area of memory from the DATA statements in lines 500–560.

Lines 10 and 20 transfer the BASIC interpreter *and* the Kernal from ROM to RAM with the same addresses, so we can modify them.

The *Commodore 64 Programmer's Reference Guide,* page 261, states that turning off bit 1 in location 1 switches only the Kernal addresses to RAM; actually it affects both the Kernal and BASIC address ranges.

Line 25 merely signals that the transfer is complete (it takes about a minute).

The Intercept Routine

Line 30 sets up the routine which intercepts keyboard characters. It is put at the end of the routine that pulls a character from the keyboard buffer.

Finally, line 40 activates the modified Kernal by turning off bit 1 of location 1 (changing the value in location 1 from 55 to 53). Once this is done, the change has been made, and pressing CTRL-N toggles between a numeric keypad and the normal usage of the M, J, K, L, U, I, and O keys.

A Color Memory Bonus

A couple of bonuses have been included in lines 31 and 32. Line 31 changes the portion of the Kernal on newer 64s that puts the background color into the color memory for screen locations being cleared. Instead of putting the background color there, it will now put 1 (for white), so that if addresses 1024 to 2023 (decimal) are POKEd, a character will appear. (See "Commodore 64 Video Update," *COMPUTE!'s Gazette,* July 1983, page 44.)

POKEing 1000 locations as suggested there takes a few seconds — not something to do for every PRINT of a screen clear.

Choose a Color

In the normal mode, printed characters will be light blue on a dark blue background, while POKEd characters will be white. Change the POKE to location 58587 in line 31 to some other number if you would like a color different from white for POKEd screen characters. Of course, if you have an older 64 which does not clear color memory to the background color, leave out this patch (line 31).

Line 32 eliminates the printing of a question mark and space in an INPUT statement prompt. This makes it possible to write:

```
100 INPUT "TITLE:";T$
```

and have the resulting screen look like

```
TITLE:COMPUTE!
```

In any place where you really want the ? and the space, you can put them inside the quotes.

Using this INPUT feature, and calling the keypad routine from within your own program, you could create something like:

```
10 POKE 50216,255
20 INPUT "NUMBERS";N
30 PRINT N
40 POKE 50216,0
50 INPUT "WORDS";N$
60 PRINT N$
```

As long as the keypad routine has been placed in memory, you can activate and deactivate it by POKEing location 50216 with 255 or 0 respectively. Then, using the INPUT statement, you can enter numbers *or* words. This makes the keypad routine even more versatile.

Numeric Keypad

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

```
3 FORI=50176TO50261:READX:POKEI,X            :rem 40
4 NEXT                                       :rem 115
10 FORI=40960TO49151:POKEI,PEEK(I):NEXT      :rem 142
20 FORI=57344TO65535:POKEI,PEEK(I):NEXT      :rem 151
25 PRINT"TRANSFERRED"                        :rem 120
30 POKE 58823,76:POKE58824,0:POKE58825,196 :rem 70
31 POKE58586,169:POKE58587,1:POKE58588,234:rem 134
32 FORI=44029TO44034:POKEI,234:NEXT          :rem 103
40 POKE 1,53                                  :rem 88
500 DATA201, 14, 240, 65, 44, 40, 196, 240, 28, 20
    1, 85, 240, 40, 201                      :rem 221
510 DATA73, 240, 40, 201, 79, 240, 40, 201, 74, 24
    0, 16, 201, 75, 240                      :rem 221
520 DATA16, 201, 76, 240, 16, 201, 77, 240, 28, 88
    , 24, 96, 0, 169                         :rem 103
530 DATA 49, 208, 248, 169, 50, 208, 244, 169, 51,
     208, 240, 169, 52, 208                  :rem 163
540 DATA 236, 169, 53, 208, 232, 169, 54, 208, 228
    , 169, 48, 208, 224, 169                 :rem 224
550 DATA 255, 77, 40, 196, 141, 40, 196, 88, 165,
    {SPACE}198, 240, 252, 120, 76            :rem 117
560 DATA 180, 229                             :rem 23
```

David W. Martin

One-Touch Commands

You can put the normally unused function keys on the 64 to work with this programmer's utility. An entire command can be typed with a single keypress. You'll quickly appreciate the time saved with this technique as you enter programs.

Unlike most people, computers excel at performing boring, repetitive tasks. What's more, time-consuming tasks which annoy us can be performed by an uncomplaining computer in a fraction of a second. It only makes sense to let computers handle the things they do best.

One of these jobs is the routine typing of frequently used commands. During a session with your computer, how many times do you type RUN, LIST, SAVE, or LOAD? Probably more times than you think. If you're a hunt-and-peck typist new to typewriter-style keyboards, this can be a major annoyance. Even if you're a fast touch-typist, you probably stumble over such often-used commands as POKE 53281,1:PRINT"{BLK}" (which sets up an easier-to-read white screen background with contrasting black characters).

The utility presented here can free you from all that. It redefines the special function keys (f1 through f8 to the right of the keyboard) so that a single keypress enters a whole command. The short while it takes to type in this program can pay for itself many times over.

One-Touch Commands

Be sure to type the program carefully. As always, save it twice on tape or disk before running it for the first time. The program is in the familiar form of a BASIC loader—a BASIC program which includes a machine language program encoded in DATA statements. A mistyped number can "crash" the computer when the program is first run, forcing you to switch it off, then on again to clear the machine. Saving the program beforehand can prevent you from losing all your work.

Actually, this BASIC loader contains two machine language programs. Neither program consumes any memory normally used by BASIC (see Programmer's Notes below). After activating the utility, it erases the BASIC loader from memory and allows you to load your own programs. The utility keeps working "in the background," so to speak, until you turn off the computer or reset it by pressing RUN/STOP–RESTORE.

The utility is very easy to use. First, enter and run the BASIC loader. You'll see a screen prompt which asks:

F1?

Now, type in whatever command you'd like to have available at a stroke of the f1 key. Then press RETURN. For instance, if you answer the prompt by typing LIST and pressing RETURN, hitting f1 after the utility is activated will print the command LIST on the screen.

There's a way to save even more keystrokes. If you answer the prompt by typing the command followed by a back arrow — using the back arrow key in the upper-left corner of the keyboard — the utility will press RETURN for you, when activated. Otherwise, it's up to you to hit RETURN when using each command. In other words, answering the prompt like this:

F1? LIST [*Press RETURN*]

means that when the utility is working, it will type the command LIST on the screen for you, but you'll still have to press RETURN yourself to actually execute the command. But if you answer the prompt like this:

F1? LIST ← [*Press RETURN*]

it means the utility, when working, will type LIST *and* press RETURN for you when you hit the f1 key. The back arrow makes the command *self-executing*. Pressing the function key executes the command instantly. Depending on the command, this may or may not be desirable. For instance, you probably wouldn't want the command NEW to execute instantly because it would be too easy to accidentally wipe out a BASIC program. (In fact, you probably wouldn't want to program a function key with NEW at all.)

You can also answer the prompt with more than one command. An example might be:

F1? LOAD ←RUN ← [*Press RETURN*]

which means f1 will automatically load and run the next program from tape.

After answering the F1? prompt, the utility asks for F2, F3, and so on through F8. After F8, the utility immediately activates itself and erases the BASIC loader from memory.

The function keys are now programmed. They will remain so until you shut off the computer or trigger a warm start by pressing RUN/STOP–RESTORE.

Programmer's Notes

The one-touch command utility consists of two machine language programs tucked away in different parts of the Commodore 64's memory. The first part is in the cassette buffer, starting at memory location 828 ($033C hexadecimal). This program asks for the key definitions. Each time RETURN is pressed, it stores the ASCII values of the characters into high memory.

Since the cassette buffer is used only temporarily, as you program the function keys, you can load or save programs from or to tape and not erase this routine. There may be problems, however, if you're trying to use another machine language program which stores data starting at location 49152. That's because *this* routine uses that area for its second program (see below).

After entering f8, control jumps to the second program, stored in high memory at location 49152 ($C000 hex). This is a 4K block of unused memory in the 64. The first two POKEs in the first line of the BASIC loader fool BASIC into thinking that memory ends at 53248. To restore normal vectors, you can enter POKE 56,160:POKE 55,0.

The first machine language program also sets up an interrupt. Every sixtieth of a second, the computer checks the second program to see if a function key has been pressed. If so, the key's definition is printed on the screen. If a back arrow was defined after the command, the program forces a RETURN to execute the command also.

One-Touch Commands

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

```
1 POKE56,208:POKE55,0:F=0:C=PEEK(55)-120:IFC<0THEN
  C=C+256:F=-1                                    :rem 84
2 D=PEEK(56)+F:POKE55,C:POKE56,D                  :rem 139
3 S=828:I=146:GOSUB100                            :rem 11
10 DATA32,198,3,165,55,133,251,133,253,165,56,133,
   252,133,254,169                                :rem 184
15 DATA49,133,167,169,133,133,168,169,13,32,210,25
   5,169,70,32,210                                :rem 186
20 DATA255,165,167,32,210,255,169,61,32,210,255,16
   9,63,32,210,255                                :rem 178
25 DATA169,32,32,210,255,32,207,255,72,160,0,165,1
   68,145,55,104                                  :rem 76
30 DATA32,198,3,201,13,240,14,201,95,208,2,169,13,
   145,55,32                                      :rem 119
35 DATA207,255,76,124,3,230,167,165,167,41,1,208,1
   0,24,165,168                                   :rem 33
40 DATA105,4,133,168,76,170,3,56,165,168,233,3,133
   ,168,165,167                                   :rem 43
45 DATA201,57,144,163,120,169,L0,141,20,3,169,H0,1
   41,21,3,88                                     :rem 215
```

```
50 DATA169,0,133,167,32,68,166,76,116,164,166,55,2
   08,2,198,56                                    :rem 8
55 DATA198,55,96                                  :rem 87
56 S=PEEK(55)+256*PEEK(56):I=120:GOSUB100         :rem 76
57 SYS(828)                                       :rem 94
58 END                                            :rem 68
60 DATA165,167,240,59,160,0,177,251,32,L99,H0,176,
   12,165,55,197                                  :rem 147
65 DATA251,208,21,165,56,197,252,208,15,169,0,133,
   167,165,253,133                                :rem 193
70 DATA251,165,254,133,252,76,49,234,166,198,177,2
   51,157,119,2,230                               :rem 252
75 DATA198,32,L111,H0,165,198,201,11,144,204,230,1
   67,76,49,234,165                               :rem 28
80 DATA215,32,L99,H0,176,3,76,49,234,165,8,41,1,20
   8,247,160                                      :rem 197
85 DATA0,177,251,197,215,208,6,32,L111,H0,76,L6,H0
   ,32,L111,H0                                    :rem 106
90 DATA76,L81,H0,201,133,144,6,201,141,176,2,56,96
   ,24,96,166                                     :rem 239
95 DATA251,208,2,198,252,198,251,96,0,0           :rem 188
100 F=0:FORD=STOS+I:READA$:IFASC(A$)<58THENA=VAL(A
    $):GOTO115                                    :rem 173
105 IFASC(A$)=76THENA=VAL(RIGHT$(A$,LEN(A$)-1))+PE
    EK(55):IFA>255THENA=A-256:F=1                 :rem 73
110 IFASC(A$)=72THENA=VAL(RIGHT$(A$,LEN(A$)-1))+PE
    EK(56)+F:F=0                                  :rem 22
115 POKED,A:NEXT:RETURN                           :rem 10
```

David W. Martin

Dr. Video

The cursor control keys on your 64 already give you some of the most powerful screen-editing capabilities of any home computer, but this utility adds even more: clear screen below the cursor, clear screen above the cursor, and "home" the cursor to the bottom left of the screen, all at machine language speed.

While revising long programs or doing repeated numerical calculations in immediate mode, it's often useful to be able to clear a portion of the screen display while leaving the rest intact. It's also useful at times to be able to "home" the cursor to the lower left of the screen instead of the usual upper-left position.

Although Commodore built excellent screen-editing features into the 64, "Dr. Video" adds even more flexibility by giving you three additional cursor control keys. A special technique allows Dr. Video to function even while you are typing or running another program. Since the program is written entirely in machine language, it doesn't take up any of the memory normally used for BASIC programming.

The new cursor control features are assigned to three of the 64's function keys. The assignments are as follows:

f1 Clear display to the top of the screen starting with the line containing the cursor.

f3 Clear display to the bottom of the screen starting with the line containing the cursor.

f5 Move the cursor to the lower-left corner of the screen.

How the Doctor Operates

Every 1/60 second your 64 stops whatever it is doing and takes some time to read the keyboard and perform other housekeeping tasks. These breaks are called *interrupts,* and the machine language program which runs during this interrupt period is called the interrupt service routine. When the microprocessor receives the interrupt request (IRQ) signal, it looks at a pair of memory locations to find the starting address (called the IRQ vector) of the interrupt service routine to be executed. On the 64, the IRQ vector is contained in locations 788 and 789, which normally point to address 59953, the beginning of the standard IRQ service routine in ROM (unchanging memory). However, since the IRQ vector is stored in RAM, changeable memory, we can substitute the address of our own machine language subroutine and add it to the normal interrupt service routine.

Like all interrupt-driven routines, Dr. Video continues to run until you reset the computer (by hitting the RUN/STOP and RESTORE combination, for example). It is not disabled by hitting just the STOP key. After a reset, you can reactivate the new screen-editing keys by typing SYS 49152.

Typing In the Program

Dr. Video is a machine language program which uses a BASIC loader to POKE the data into memory and issue the SYS to start it running. A checksum is calculated to assist in detecting typing errors in the DATA statements, but since the loader program NEWs itself out of the BASIC memory area, you should be careful to save a copy before running for the first time.

Dr. Video

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

```
200 FORI=0TO148                                    :rem 111
210 READJ:POKE49152+I,J:X=X+J:NEXTI                :rem 175
230 IFX<>17525THENPRINT"ERROR IN DATA":STOP
                                                   :rem 181
240 SYS49152:NEW                                   :rem 190
300 DATA120,169,13,141,20,3,169,192,141,21,3,88,96
    ,165,197,41                                    :rem 236
310 DATA127,201,4,208,27,169,0,133,25,169,4,133,26
    ,216,24,165                                    :rem 226
320 DATA209,105,40,133,27,165,210,133,28,144,2,230
    ,28,24,144,46                                  :rem 59
330 DATA165,197,41,127,201,5,208,19,165,209,133,25
    ,165,210,133,26                                :rem 175
340 DATA169,232,133,27,169,7,133,28,24,144,19,201,
    6,208,67,169                                   :rem 45
350 DATA192,133,209,169,7,133,210,169,24,133,214,2
    4,144,44,216,56                                :rem 180
360 DATA165,27,229,25,133,29,165,28,229,26,133,30,
    169,32,166,30                                  :rem 92
370 DATA240,12,160,0,145,25,200,208,251,230,26,202
    ,208,246,166,29                                :rem 156
380 DATA240,8,160,0,145,25,200,202,208,250,169,0,1
    33,211,169,32                                  :rem 54
390 DATA133,197,76,49,234                          :rem 17
```

E. A. Cottrell

Step Lister

"Step Lister" is a timesaving programming aid which lets you look at your BASIC program lines without repeatedly typing LIST. It's short, and safe from BASIC.

As you're programming, you probably use the LIST command often. It's the only way you can turn the pages of your program when it's in your computer's memory. But LIST can be clumsy, for you have to either constantly type LIST and hit the RUN/STOP key, or type LIST and a line number or range. You can easily miss the line you want to examine. No matter what, you have to type LIST a lot. "Step Lister" is a machine language *wedge* (explained below) which allows you to step through a BASIC listing one line at a time.

Type in and SAVE Step Lister. It's in the familiar form of a BASIC loader which POKEs a machine language program into high memory starting at location 49152 and issues a SYS command to run it. Since even a single error in typing it in can lock up your computer, forcing you to turn it off, then back on, to restore control, make sure you save Step Lister before RUNning it. That way, you won't lose all your typing.

RUN the program. It's now safe from BASIC. You can LOAD another program and use Step Lister to look through it.

To see the first line of your program, just type:

@0.

(Entering any other number after the @ will start the listing at that line. There should be no spaces between the @ and the line number, and the @ must be on the left edge of the screen.)

Then, press any key and the next line will be displayed. Press the space bar and hold it down, and the listing will continue scrolling until the space bar is released.

If you wish to stop Step Lister, press RUN/STOP. The cursor returns to the screen and you can edit a line or lines. Step Lister is still available; enter the @ symbol and a line number to see another part of the program.

What Is a Wedge?

To understand a wedge, you must first have some knowledge of how BASIC works. When you press RETURN, one of two things happens. If the entered line has a number as the first character, the computer assumes that a BASIC line is being entered. This line is then converted to BASIC *tokens* and put in its proper place in memory. (Tokens

are single-byte symbols which represent BASIC commands. To save space and time, the computer stores PRINT, for example, as 153.)

No interpretation of the characters following the line number is made until the program is run. If the first character is not numeric, the line is tokenized and placed in the BASIC input buffer at locations 512–600 ($0200–$0258). The interpreter then calls the CHRGET subroutine to get the characters from the buffer and return them for interpretation.

To implement a wedge, the CHRGET subroutine located at 115–138 ($73–$8A) must be altered to go to your machine language program before returning to the interpreter. At the entry point of the wedge, a check is made to see if the special character (in this case, @) has been entered. If it has, the special routine is executed. Otherwise, the character is sent to the interpreter for normal BASIC interpretation and execution.

Using ROM Routines

Step Lister uses many of the subroutines which are part of the BASIC ROM in the 64. Analyzing some of the subroutines already in the machine can prove useful.

The wedge can be a powerful tool. If you decide to write a wedge program of your own, heed one word of caution: Do not try to alter the CHRGET subroutine with BASIC. You will be changing the way BASIC gets its instructions in the middle of a BASIC program, and this will crash your computer.

Step Lister

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

```
10 TM=49152                                          :rem 68
20 FOR I = TM TO TM + 241                           :rem 118
30 READ A: POKE I,A: CHK = CHK + A: NEXT I          :rem 48
40 X = 828: FOR I = X TO X + 23                      :rem 37
50 READ A: POKE I,A: CHK = CHK + A: NEXT I          :rem 50
60 IF CHK <> 32456 THEN PRINT "DATA ERROR": END
                                                     :rem 254
70 SYS49152                                          :rem 107
100 DATA162,0,189,60,3,149,115,232,224,23,208,246,
    0,201,64,240,22,201                               :rem 84
120 DATA58,176,10,201,32,240,11,56,233,48,56,233,2
    08,96,76,116,164,234                             :rem 169
140 DATA76,115,0,160,0,185,0,2,201,64,208,243,200,
    185,0,2,201,0,240,9,201                           :rem 1
160 DATA45,208,244,169,171,153,0,2,169,1,133,122,1
    60,1,24,185,0,2,32,107                           :rem 238
180 DATA169,32,19,166,160,0,32,121,0,32,107,169,16
    5,20,5,21,208,6,169                              :rem 106
```

```
200 DATA255,133,20,133,21,160,1,132,198,160,1,132,
    15,177,95,240,175,32                    :rem 144
220 DATA44,168,32,215,170,134,25,132,26,173,198,0,
    240,251,169,0,141,198                   :rem 214
230 DATA0,166,25,164,26,200,177,95,170,200,177,95,
    197,21,208,4,228,20                     :rem 120
250 DATA240,2,176,44,132,73,32,205,189,169,32,164,
    73,41,127,32,71,171                     :rem 120
270 DATA201,34,208,6,165,15,73,255,133,15,200,240,
    17,177,95,208,16,168                    :rem 171
290 DATA177,95,170,200,177,95,134,95,133,96,208,16
    3,108,6,3,16,218,201                    :rem 188
310 DATA255,240,214,36,15,48,210,56,233,127,170,13
    2,73,160,255,202,240,8                  :rem 251
320 DATA200,185,158,160,16,250,48,245,200,185,158,
    160,48,181,32,71,171,208                :rem 112
350 DATA245,0,230,122,208,2,230,123,173,0,2,201,58
    ,240,10,201,32,240,239                  :rem 214
370 DATA76,13,192,234,234,234,96             :rem 99
```

Charles Brannon

Foolproof INPUT

Machine language routines, even short ones, can perform some impressive functions. Overcoming some of the problems of the INPUT statement is relatively easy in machine language. This routine reprograms BASIC's own INPUT routine and can be added to your own program. Since it's in the form of a BASIC loader, you don't need any special knowledge of machine language.

Problems with INPUT

You are probably familiar with some of the problems with the INPUT statement. First, it will not properly handle input with commas and colons. If you entered the previous sentence, the computer would accept only the word "First" and ignore the rest of the line (as the computer warns you with ?EXTRA IGNORED). This is because the comma is used to separate multiple INPUTs on the same line, as in this example:

```
INPUT "ENTER NAME: FIRST,LAST";A$,B$
```

The colon, too, triggers an ?EXTRA IGNORED message. Yet it cannot be used to separate INPUT items, so it appears to be some kind of a bug (error) in the BASIC language itself.

You can get around these problems somewhat, but they become especially annoying when you are trying to read a file on tape or disk containing these characters. In a mailing list program, for instance, you need commas for address fields such as "Greensboro, NC, 27403".

There are other difficulties with the INPUT statement. Quotation marks are not handled correctly. Leading and trailing spaces are stripped away. INPUT also allows people to use all the cursor and color control keys. Theoretically, you can place the cursor anywhere on the screen where there is something you want to INPUT, and press RETURN. In effect, this is what happens when you edit a program (the same INPUT routine is used by both the system and BASIC). But it just makes no sense to allow cursor moves all over the screen when you simply want the user to answer a question. If the user accidentally presses a cursor key and then tries to move the cursor back, the entire line, including any prompts, is read.

This can also be a problem when you have carefully laid out a screen format with blanks or boxes into which the user is supposed to enter information. You have no way to control how many characters the user can type, so if your blank space is only ten characters long, there is nothing to prevent someone from typing more. Not

only that, but with the standard INPUT routine, someone can move the cursor out of the box you want them to use, clear the screen entirely, or otherwise destroy your carefully planned format.

Improving on INPUT

What we need, then, is a new INPUT routine that will not allow cursor moves. The DEL key should still let the user delete characters to make corrections, however. Additionally, the ideal INPUT routine should let your program limit the number of characters typed, yet allow commas and colons.

The usual solution is to write your own INPUT routine using the GET statement, which fetches one key at a time from the keyboard. With such a simple statement as GET, however, you have to reinvent the wheel anytime you need such a protected INPUT routine. And it certainly isn't as easy to use as a simple INPUT statement.

Well, I certainly wouldn't bring such gloom to the scene without a solution. The accompanying program is the key. It's a machine language routine that replaces the standard Commodore INPUT with a protected INPUT like the one described above. The beauty of it is that after you GOSUB 60000, all INPUT (and INPUT#) statements are redefined. You don't have to understand how the machine language works in order to use it, and you don't have to rewrite any existing programs, other than to insert the GOSUB. You still have all the flexibility of the standard INPUT statement. Just add the subroutine to the end of your program.

The machine language program has a couple of niceties. After you GOSUB 60000, you can change the maximum number of characters allowed by POKEing memory location 251 with the length (don't POKE with zero, or more than 88). The cursor is an underline by default, but you can change the character used by POKEing its ASCII value into memory location 2. For example, to change the cursor into an asterisk, enter:

```
POKE 2,ASC("*")
```

or

```
POKE 2,42
```

When you use the routine to INPUT data from files, just remember that it strips away all control characters, from CHR$(0) to CHR$(31) and CHR$(128) to CHR$(159). This includes all special codes such as cursor controls, function keys, color codes, etc. You'll rarely write these to a standard data file, anyway.

Cautions

Curiously, "Foolproof INPUT" does not work properly in direct mode. To make BASIC accept commas and colons, an invisible quote is added to the start of each line that is input. Naturally, direct mode doesn't like statements such as RUN or LIST. If you want the special INPUT routine out of your way, just press RUN/STOP–RESTORE.

The invisible quote also prevents you from using something like INPUT A. Only string variables work with this routine; use INPUT A$ instead. If you want, you can include a line such as:

INPUT A$:A = VAL(A$)

instead of INPUT A.

To display the contents of A$ (or any other string variable set with INPUT using this routine), PRINT it from within a program. For instance:

```
10 GOSUB 60000:INPUT A$
20 PRINT A$:END
```

would PRINT the contents of A$. If you INPUT something, then use the PRINT statement in direct mode, you'll get a SYNTAX ERROR. Pressing RUN/STOP–RESTORE before printing, however, displays the contents of the string variable correctly.

Foolproof INPUT

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

```
60000 IFPEEK(830)=133THEN60020                :rem 145
60010 FORI=828TO983:READA:POKEI,A:NEXT        :rem 124
60020 SYS828:RETURN                           :rem 179
60030 DATA 169,000,133,252,169,080            :rem 135
60040 DATA 133,251,169,164,133,002            :rem 131
60050 DATA 169,083,141,036,003,169            :rem 142
60060 DATA 003,141,037,003,096,152            :rem 127
60070 DATA 072,138,072,165,252,208            :rem 144
60080 DATA 007,032,116,003,169,000            :rem 123
60090 DATA 133,253,166,253,189,000            :rem 143
60100 DATA 002,133,254,198,252,230            :rem 129
60110 DATA 253,104,170,104,168,165            :rem 133
60120 DATA 254,096,160,000,169,034            :rem 135
60130 DATA 141,000,002,132,252,165            :rem 115
60140 DATA 002,032,210,255,169,157            :rem 132
60150 DATA 032,210,255,032,228,255            :rem 131
60160 DATA 240,251,164,252,133,254            :rem 135
60170 DATA 169,032,032,210,255,169            :rem 141
60180 DATA 157,032,210,255,165,254            :rem 141
60190 DATA 201,013,240,043,201,020            :rem 111
60200 DATA 208,013,192,000,240,211            :rem 114
```

```
60210 DATA 136,169,157,032,210,255          :rem 138
60220 DATA 076,123,003,041,127,201          :rem 120
60230 DATA 032,144,196,196,251,240          :rem 141
60240 DATA 192,165,254,153,001,002          :rem 129
60250 DATA 032,210,255,169,000,133          :rem 126
60260 DATA 212,200,076,123,003,230          :rem 118
60270 DATA 252,230,252,153,001,002          :rem 120
60280 DATA 169,032,032,210,255,096          :rem 142
```

John Krause and David W. Martin

64 Searcher

"64 Searcher" is a time-saving utility that searches through your BASIC program and locates any character or string of characters.

When working on a long BASIC program, it pays to plan ahead. But it seems that no matter how hard you try, you can't keep track of everything in your program. Can I use *H* to store the high score, or is that variable already being used for something else? Where is this subroutine called from? You probably end up searching for a number or word hidden among scores of program lines.

"64 Searcher" allows you to spend less time searching and more time programming. Simply give it the string of characters to search for, and it tells you the numbers of all lines in which the string appears. It can search a hundred lines faster than it takes you to search one. It's fast because it's machine language. But you don't have to know machine language to use it.

Searching

Enter the program carefully and save it on tape or disk before running it for the first time. In the form of a BASIC loader and a series of DATA statements, the program must be entered exactly as it appears. By using "The Automatic Proofreader" in Appendix C, you should be able to type it in correctly the first time. If there's even one error, the computer may lock up (not respond to keypresses). You'll have to turn it off, then on again, to regain control. If you've saved the program, you can load it again and begin looking for the typing mistake.

To use 64 Searcher, load and run it, then load your BASIC program. 64 Searcher doesn't use any BASIC memory, so you can work on your program normally. To initiate a search, type 0 followed by the string you want to find. The string must be enclosed within either slashes or quotes. Hit the RETURN key and the string is stored in your program as line 0. If your program already has a line 0, you'll have to change that line number because the string must be the first line in the program.

Type SYS49152 and press RETURN. Instantly you'll see numbers appear on the screen. These are the line numbers that contain the string you specified. If no match is found, no numbers will be printed. If the string occurs more than once in a line, the line number is printed only once.

Once you're done searching through a BASIC program, remember to delete line 0 before saving or running it.

Quotes and Slashes

Because BASIC commands are stored differently than other characters in a program, there are two ways of specifying the search string. If the string is enclosed within slashes (/), BASIC commands are recognized as such. If the string is within quotes (""), however, it will be treated as a literal string of characters.

For example, to find the BASIC statement AND, line 0 should be:

```
Ø /AND/
```

After entering SYS49152, 64 Searcher will find the AND in this line:

```
1Ø IF X=1 AND Y=2 THEN 5Ø
```

but not in this line:

```
2Ø PRINT "THIS AND THAT"
```

To find the AND in line 20 above, you'd use quotes instead of slashes. 64 Searcher is an excellent debugging aid, especially for long programs, and will be a valuable addition to your toolbox of machine language utilities.

64 Searcher

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

```
1Ø FORI=49152TO49255:READJ:K=K+J:POKEI,J:NEXT
                                          :rem 66
2Ø IFK<>16302THENPRINT"ERROR IN DATA STATEMENTS":S
   TOP                                   :rem 117
3Ø PRINT"{CLR}SYS49152 TO SEARCH"         :rem 36
1ØØ DATA169,1,133,251,169,8,133,252,16Ø,Ø,177,251,
    56,229,251,56                         :rem 8Ø
11Ø DATA233,5,141,1Ø4,192,233,2,141,1Ø5,192,16Ø,Ø,
    177,251,17Ø,2ØØ                      :rem 142
12Ø DATA177,251,24Ø,67,133,252,134,251,16Ø,Ø,177,2
    51,56,229,251,17Ø                     :rem 17
13Ø DATA2Ø2,134,2,198,2,165,2,2Ø5,1Ø4,192,48,222,1
    33,253,173,1Ø5                       :rem 11Ø
14Ø DATA192,133,254,164,253,177,251,164,254,217,5,
    8,2Ø8,229,198,253                     :rem 45
15Ø DATA198,254,2Ø8,239,16Ø,2,177,251,17Ø,2ØØ,177,
    251,32,2Ø5,189,169                    :rem 88
16Ø DATA32,32,21Ø,255,76,26,192,96       :rem 19Ø
```

Dan Carmichael

The Four-Speed Brake

Not only does this machine language routine vary the speed of your listings, but it can also select the speed at which your BASIC programs execute. And it doesn't use any of your BASIC programming memory.

One small inconvenience of programming with the Commodore 64 (if you don't have a printer) is the limitation of being able to display only relatively small sections of your programs on the screen at one time. If you have a large BASIC program, listings can be hard to follow. Even slowing down the LIST command with the CTRL key isn't much help at times; the BASIC lines still pass by at a reasonably fast rate, and because of the way the lines "jump," they can be hard to follow.

The short program that follows will help slow things down for you. It's a "Four-Speed Brake" that lets you vary the speed of your listings from reasonably slow to a complete stop. The program is written in machine language, and normally sits undisturbed in an area of available memory called the *cassette buffer.* Once it is POKEd into memory, it uses none of your available BASIC programming memory.

How to Use the Program

First, load your BASIC program into the computer, then either append this program to it, or type it in after your program. Before running the program for the first time, verify it carefully, and save it to tape or disk. An error in this (or any) machine language program can cause your system to crash, forcing you to turn your computer off and then on to reset. After verification, type RUN 60000 and press RETURN to POKE the machine language program into the cassette buffer. Then type SYS828 and press RETURN. The Four-Speed Brake is now running.

The program is controlled by the special function keys. The chart illustrates what the function keys do.

To stop Four-Speed Brake, press RUN/STOP–RESTORE; to restart, enter SYS828.

After the Four-Speed Brake has been successfully POKEd into memory and tested, you may, if you wish, delete lines 60000–60040. Also, the CTRL key will still work as it normally does in slowing down your listings, and might be considered a "fifth speed," a little faster than the f1 key.

The Four-Speed Brake also has another important benefit. It will slow down or stop the running of your BASIC program just as it slows the LIST command. This can be a very useful tool for debugging your BASIC program. To do this, use the Four-Speed Brake in the same manner as you would for the LIST command; enter SYS828, then RUN your program. The function keys will slow down or stop your BASIC program.

Special Function Keys

f1	**fastest speed**
f3	**medium speed**
f5	**slowest speed**
f7	**complete stop**

Words of Caution

First, this program runs in the cassette buffer, and as is true with all programs in this buffer, you cannot use the tape cassette while this program is running. Second, because of the way the computer outputs the lines while listing programs, you will encounter a glitch every now and then. It will appear as if one line repeats itself. If you continue to hold down the function key and let the screen scroll, it will take care of itself. You can observe how this happens if you list a program while holding down the f5 key.

If you're a machine language programmer, the Four-Speed Brake will also work, both in listing and running your ML programs. To use, enter SYS828, then SYSXXX into your ML monitor as usual, or SYSXXX into your machine language program. However, a word of caution is needed here. The Four-Speed Brake uses all three registers (A, X, and Y), so you'll have to be careful when using these registers in your own program.

Four-Speed Brake

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

```
60000 FORA=828TO894:READB:POKEA,B:NEXT:END:rem 127
60010 DATA120,169,73,141,20,3,169,3,141,21,3,88,96
      ,162,0,160,0,165,197,201              :rem 91
60020 DATA4,208,10,232,208,253,200,192,48,208,248,
      160,0,201,5,208,6,232,208            :rem 134
60030 DATA253,200,208,250,201,6,208,8,232,234,234,
      208,251,200,208,248,32               :rem 245
60040 DATA159,255,165,197,201,3,240,247,76,49,234
                                           :rem 176
```

Todd Heimarck

ASCII/POKE Printer

The computer does the work for you in this short routine, which automatically calculates ASCII and POKE values. A handy programming utility, there's even a BASIC-machine language comparison included. A disassembly of the code is also listed.

Reference Tables

Chances are, PRINTing to the screen was one of the first things you learned to do in BASIC. You probably also learned how to control where the computer prints by putting cursor commands within strings or by using SPC and TAB commands. The PRINT command is common, primarily because it is so easy to use. But in certain situations, you may need to find out a character's ASCII number. And sometimes it is quicker to simply POKE a character onto the screen.

But before you can POKE, you have to know the character number. Let's put a row of hearts at the top of the screen. So we need to POKE a bunch of 81's. Wait, those are solid circles. What's the number for hearts? I know that list is somewhere.

If you use POKEs or ASCII values in programming, you know how annoying it is to flip back and forth through the reference book, losing time and patience. Even worse, you could lose the book and end up typing the character and PEEKing screen memory to get the POKE value.

Let the Computer Do the Work

Your computer already knows the POKE values and ASCII numbers, so why not let it do the work?

This short machine language program, "ASCII/POKE Printer," does not use any BASIC memory. Its 52 bytes remain in the cassette buffer, ready to convert letters and graphics characters to POKE and ASCII numbers whenever you want.

Note that if you write a program that POKEs any of the address locations of the cassette buffer (828–1019), you may lose ASCII/POKE Printer. Also, if you use a cassette player for SAVEs, LOADs, or tape files, you will erase the machine language program. Fortunately, it is entirely relocatable, so if you want to use the cassette buffer, you can change line 10 to move it to another part of memory. On the 64, it is usually safe to use any of the memory locations from 49152 to 53247.

LOADing and Using the Program

Type in ASCII/POKE Printer. Make sure the DATA statements are exactly as printed. SAVE it to tape or disk and VERIFY (if you have a cassette drive). RUN the program and type NEW. The program is now in your cassette buffer. BASIC memory was cleared when you typed NEW, but it did not touch the cassette buffer.

Anytime you want to use ASCII/POKE Printer, type SYS 828. The computer will wait for you to type a character and then display that character in the upper-left corner with the ASCII value to the right and the POKE value below. Type another character and you get two new values.

To exit (back to BASIC), hold down SHIFT and press RETURN. This returns you to your program. SYS 828 will send you back to ASCII/POKE Printer. You can toggle back and forth as the need arises.

Special Cases

There are some ASCII numbers that have no equivalent POKE. For example, adding CHR$(13) to a string will force a RETURN after the string is printed. But ASCII 13 cannot be POKEd to the screen (what would a RETURN look like?). ASCII/POKE Printer will give you the correct ASCII numbers, but for certain characters, like RETURN, it will print a blank space and list a POKE of 32 (which is the number for a blank space). In the case of function keys, CLR/HOME, INST/DEL, and color commands, it will print a reverse video character, as if in quote mode, and the correct ASCII number. But the POKE number will be wrong. Keys that perform a function — clearing the screen, for example — are not characters that can be POKEd to the screen.

Also note that you cannot get values for reverse video characters, which do not have separate ASCII numbers. To program a reverse character, precede it with a CHR$(18). To POKE a reverse video character, *add* 128 to the POKE value of the regular character.

This machine language utility will be most helpful when you are writing BASIC programs. By letting the computer tell you ASCII and POKE values, you can really save time. The program was written to be short and simple, but if you are familiar with machine language, you could modify it to do much more.

Machine Language Vs. BASIC

This utility was originally written to avoid the problems of trying to figure out the ASCII and screen codes. The *Commodore 64 Programmer's Reference Guide* contains the ASCII and screen codes,

but why look in the book when the computer already knows the numbers? Let the computer do the work.

Writing the machine language routine was fairly easy, because once again the computer can do the work. There are a number of useful built-in ROM routines. Call the routines a few times and you have the answer.

The first routine is GETIN. When you jump to this subroutine (JSR $FFE4), it checks to see if a key has been pressed. If so, the ASCII value of the key is put into the accumulator. If not, a zero is put in the accumulator. Then it returns from the subroutine.

Another useful Kernal routine is CHROUT. When you JSR $FFD2, the computer checks the value in the accumulator. The number is translated from ASCII to a character and it is printed on the screen (wherever the cursor happens to be at the time).

The final routine is at $BDCD. Among other things, it's used by BASIC's LIST routine to print line numbers. It takes the number in the accumulator (the most significant byte or MSB) and multiplies it by 256. Then the number in the X register (the least significant byte, or LSB) is added. The result is converted into ASCII numbers and printed on the screen. For example, if you load the accumulator with number two and put 88 into X, JSR $BDCD calculates 2*256 + 88 = 600 and prints CHR$(54) + CHR$(48) + CHR$(48). In other words, it prints the characters for the number 600.

The disassembly of the object code is listed as Program 2 at the end of this article.

Here's a line-by-line explanation of the disassembly:

Line	Function
033C	Check to see if a key has been pressed.
033F	If not, go back to 033C. If so, the accumulator holds the ASCII value and the program continues.
0341	Save the value in X.
0342	Compare A to the number $8D (the ASCII for shifted return, to exit to BASIC).
0344	If not equal (to $8D), continue at 0347.
0346	A does equal SHIFT-RETURN, return to BASIC.
0347	Put $93 into the accumulator (ASCII for clear screen).
0349	Print it.
034C	Load A with $FF (255)
034E	Store it in the quote flag (which turns on quote mode). Now if you push the clear screen key, you will see a reverse heart.
0350	Transfer the number in X back to A.
0351	Print the character.
0354	Load A with $20 (ASCII for a blank space).
0356	Print it.

0359 Load A with $00 (the MSB of the number to print in the HEXDECPRNT routine).
035B Do the HEXDECPRNT — print the ASCII value (MSB in A is always zero and LSB in X was transferred up at $0341).
035E Load A with $0D (ASCII for carriage return).
0360 Print it. Cursor position is now at the beginning of the second screen line.
0363 Load A with zero (for HEXDECPRNT) again.
0365 Store A in the quote flag; turn off quote mode.
0367 Load X with the screen code of $0400 (1024). In other words, X = PEEK(1024).
036A Go to HEXDECPRNT, to print the screen code.
036D Increment X (or X = X + 1).
036E If X is not equal to zero, branch back to the start. Because this line is a branch (which is relative), the routine can be moved to other memory locations. If it were a JMP, it would *not* be relocatable.

The program *could* be written completely in BASIC and would look like:

```
828 GET G$:A=ASC(G$+CHR$(Ø)): REM START
831 IF G$="" THEN 828
833 X=A
834 IF G$=CHR$(141) THEN EQ=1
836 IF EQ<>1 THEN 839
838 RETURN
839 A=147
841 PRINT CHR$(A);
844 A=255
846 POKE 212,A
848 A=X
850 PRINT CHR$(A);
853 A=32
855 PRINT CHR$(A);
858 A=Ø
860 PRINTSTR$(A*256+X);
863 A=13
864 PRINT CHR$(A);
867 A=Ø
869 POKE 212,A
871 X=PEEK(1Ø24)
874 PRINT(A*256+X);
877 X=X+1
878 IF X<>Ø THEN 828
```

You can do a line-by-line comparison between the machine language and BASIC versions of this program. For instance, line 033C in the ML version is the same as line 828 in the BASIC program. Both lines check to see if a key has been pressed. Comparing BASIC and machine language versions in this way is one of the best ways to see how ML operates, especially if you're unfamiliar with machine language programming.

Program 1. ASCII/POKE Printer

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

```
10 FORJ=828TO879:READK:POKEJ,K:NEXT          :rem 0
15 READY:IFY<>999THENSTOP                  :rem 151
20 DATA32,228,255,240,251,170,201,141,208,1,96,169
   ,147                                    :rem 142
21 DATA32,210,255,169,255,133,212,138,32,210,255,1
   69,32                                   :rem 190
22 DATA32,210,255,169,0,32,205,189,169,13,32,210,2
   55                                       :rem 40
23 DATA169,0,133,212,174,0,4,32,205,189,232,208,20
   4                                       :rem 240
25 DATA999                                  :rem 44
```

Program 2. Disassembly

```
033C JSR  $FFE4
033F BEQ  $033C
0341 TAX
0342 CMP  #$8D
0344 BNE  $0347
0346 RTS
0347 LDA  #$93
0349 JSR  $FFD2
034C LDA  #$FF
034E STA  $D4
0350 TXA
0351 JSR  $FFD2
0354 LDA  #$20
0356 JSR  $FFD2
0359 LDA  #$00
035B JSR  $BDCD
035E LDA  #$0D
```

```
0360 JSR  $FFD2
0363 LDA  #$00
0365 STA  $D4
0367 LDX  $0400
036A JSR  $BDCD
036D INX
036E BNE  $033C
```

Thomas Henry

64 Escape Key

While programming, there are lots of ways to get trapped inside quotes and be unable to use the cursor controls. Until now, your only recourse was to hit RETURN and try the line again. With this handy utility, you can escape from "quote mode" traps by just hitting the pound sign key. The routine also serves as an example of machine language programming for those who are interested in trying their hand at it.

How many times has this happened to you? You're sitting at your Commodore 64, entering or editing a program, and through a series of keystrokes that you probably don't even remember, get into the following trap. When you push a cursor movement key, instead of the cursor actually moving, you get a reverse video field symbol on the screen. Frustrating, isn't it? As you have probably learned, about the only way to get free of the trap is to hit RETURN to get out of the line, and then start over.

Here's an easier way: a program that adds a valuable escape option to your computer. With this feature, the seldom-used British pound symbol (£) becomes an escape key. When you are stuck in the cursor trap mentioned above, simply push the key; you will be released from what's called *the quote mode* and will be free to move the cursor as desired. Before looking at the program, let's examine the problem in greater detail.

Store or Perform the Action

Some of the computer's keys are able to perform two distinct jobs, depending on whether the computer is in the immediate or program mode. These keys include the four cursor keys, RVS ON, RVS OFF, CLR, HOME, INST (Insert), DEL (Delete), and all of the color selection keys. In the immediate mode, you push one of these keys and the action is performed immediately. For example, depress the cursor right key and the cursor moves one space to the right.

But one of the truly impressive features of your Commodore computer is its ability to store or save the action implied by the key. For example, here's a one-line program:

```
10 PRINT"{RIGHT}HELLO"
```

The string contains the word HELLO preceded by a cursor-right. When you type this line into the computer, the cursor-right movement

is not performed; instead it is stored in the string. The cursor-right will be performed only when the program is run. We are storing a cursor movement to be executed later in the program mode. To indicate that a cursor-right movement is stored in the string, the computer will leave a reverse video field brace symbol inside the quotes. In fact, every one of the keys mentioned above has a reverse video field character which stands for it when it's inside quotes.

The trouble comes when the computer thinks you're trying to *store* an action, but you want to *perform* it. There are a number of ways this can happen. One way is if you've typed in an odd number of quote marks while entering a line. Another way is pushing the insert key more times than you expected.

Escape by Machine Language

Having defined the problem, let's look at a program that will take care of it. Examine Program 1. This is the source code of the Escape Key program. Since assemblers are now becoming quite common for the 64, enterprising users might wish to enter the source code in directly and assemble their own version. If you're an experimenter, you'll find that this is a great program to begin with. It's not too long, and yet not so short as to be just a trivial exercise. And it has a practical use too.

Examine Program 1. The first part shows the "equates" for the program. These equates give names or labels to the various internal addresses that are used by the program. For example, NOKEYS stands for location $C6, and this location always contains the number of keystrokes stored in the keyboard buffer. IRQVEC stands for the IRQ vector stored in RAM (Random Access Memory). And so it goes for all of the labels. Each stands for a location, and usually the label suggests the meaning of the location in question.

The IRQ Routine

The escape key initialization occurs next. A new vector is stuffed into RAM, and this vector directs the computer to always jump to the start of the new IRQ routine. This routine occurs next in the listing. As this is the heart of the whole program, let's examine it in greater detail.

The first thing that happens here is that all of the registers are saved temporarily. Next, the last key depressed is examined. If it wasn't the British pound symbol (which is used for the escape key), the registers are restored and the normal IRQ is finished. But if it is the desired key, a zero is stored in three important locations. These are CMODE, REVERS, and NOINST. Stuffing a zero in CMODE turns

off the quote mode, a zero in REVERS turns off the reverse screen mode, and a zero in NOINST nulls out the number of inserts pending. Turning off these three locations allows you to escape from all of the "offending" modes.

Blanking the Pound

Recall that a British pound symbol has been printed to the screen. A true escape key shouldn't print anything; it should simply "escape." So the next block of code deposits a blank on top of the British pound character and backs the cursor up one space. The net effect is that no residual character is printed. So a true escape key has been implemented.

Before going on to the rest of the normal IRQ routine (called IRQRTN in Program 1), the registers are restored. We have kept the new routine transparent to the normal Commodore 64 operating system.

You might wish to assemble your own version of this program. Most users, however, will want to use the BASIC loader in Program 2. This loader puts the program into the top of memory.

Make an Escape

To prepare a copy of this program for use, follow these steps:

1. Type in Program 2.
2. Check for errors.
3. SAVE the program first.
4. Now try it out. Type RUN and hit RETURN.
5. Almost instantly, the program will relocate to the top of memory and perform a self-initialization. You may leave the program in place for the duration of a programming session; it will not interfere with normal BASIC operation.

Typing NEW will not affect the escape key program, but if you hit the RUN/STOP–RESTORE key combination, the program is disabled. You can reenable it quite easily by typing:

```
SYS 256*PEEK(56)+PEEK(55)
```

Since cassette operations affect the IRQ loop, you may wish to disable the escape option with a RUN/STOP–RESTORE before doing any loading or saving and reenable it afterwards with the SYS 256*PEEK(56)+PEEK(55).

If you have the program in place, try it out. For example, type a quote mark. Now hit the cursor right key a number of times. Do you see the reverse video field brace? Now hit the British pound

key. Then hit the cursor key once more. Notice that this time you actually move to the right. Think of the most outlandish combination of keystrokes that you can, then try the escape. The quote mode is powerless to hold your cursor.

Program 1. Disassembly of 64 Escape Key

```
NOKEYS   = $C6              ;NUMBER OF KEYS IN BUFFER.
REVERS   = $C7              ;SCREEN REVERSE FLAG.
ROW      = $D1              ;CURRENT CURSOR ROW.
COLUMN   = $D3              ;CURRENT CURSOR COLUMN.
CMODE    = $D4              ;CURSOR MODE:0 = DIRECT.
INKEY    = $D7              ;LAST KEYSTROKE IN.
NOINST   = $D8              ;NUMBER OF INSERTS PENDING.
KEYBRD   = $0277            ;KEYBOARD BUFFER.
IRQVEC   = $0314            ;IRQ VECTOR.
IRQRTN   = $EA31            ;NORMAL IRQ ROUTINE.
;
         SEI
         LDX #<NEWIRQ       ;SET UP NEW IRQ VECTOR.
         LDY #>NEWIRQ
         STX IRQVEC
         STY IRQVEC+1
         CLI
         RTS                ;RETURN TO BASIC.
;
NEWIRQ   PHA                ;SAVE ALL REGISTERS.
         TXA
         PHA
         TYA
         PHA
         LDA INKEY          ;GET LAST KEY PUSHED.
         CMP #$5C           ;IS IT BRITISH POUND SIGN?
         BNE MOVEON         ;BRANCH IF NOT.
         LDX #$00           ;YES.
         STX CMODE          ;TURN QUOTE MODE OFF.
         STX REVERS         ;TURN REVERSE MODE OFF.
         STX NOINST         ;TURN INSERT MODE OFF.
         INX                ;TELL THE KBD BUFFER THAT
         STX NOKEYS         ;IT CONTAINS ONE KEYSTROKE.
         LDY COLUMN
         DEY                ;MOVE CURSOR BACK ONE SPACE.
         LDA #$20           ;THEN DEPOSIT A BLANK.
         STA (ROW), Y
         LDA #$9D           ;FINALLY, PUT A CURSOR LEFT
         STA KEYBRD         ;IN THE KEYBOARD BUFFER.
```

```
MOVEON  PLA             ;RESTORE ALL REGISTERS.
        TAY
        PLA
        TAX
        PLA
        JMP  IRQRTN     ;FINISH NORMAL INTERRUPT.
```

Program 2. Escape Key BASIC Loader

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

```
100 T=256*PEEK(56)+PEEK(55)-55:GOSUB160    :rem 189
110 POKE56,HI%:POKE55,LO                   :rem 168
120 FORA=TTOT+54:READD:POKEA,D:NEXT          :rem 4
130 X=T:T=T+13:GOSUB160:POKEX+2,LO:POKEX+4,HI%
                                            :rem 99
140 SYS(X)                                  :rem 61
150 NEW                                    :rem 128
160 HI%=T/256:LO=T-HI%*256:RETURN          :rem 212
170 DATA120,162,13,160,16,142,20,3         :rem 160
180 DATA140,21,3,88,96,72,138,72            :rem 95
190 DATA152,72,165,215,201,92,208,23        :rem 26
200 DATA162,0,134,212,134,199,134,216       :rem 66
210 DATA232,134,198,164,211,136,169,32     :rem 128
220 DATA145,209,169,157,141,119,2,104       :rem 76
230 DATA168,104,170,104,76,49,234          :rem 139
```

E. A. Cottrell

Variable Lister

You almost always use variables when you write a BASIC program. Sometimes, though, in a long program, you can lose track of them. This utility lists all your variables in order, including variable type, such as simple or array. It's an especially helpful tool for writing program documentation.

There are two types of variables, *simple* and *array*, and three categories in each type, *floating point numeric, integer numeric*, and *string*. All of these variables are stored in the 64 immediately above the BASIC program.

The simple variables are stored below the arrays starting at the address pointed to by memory locations 45 and 46 (see box). Each of these simple variables occupies seven bytes of memory. The first two bytes contain the first two characters (in ASCII code) of the name of the variable, with coding to indicate which type of variable it is. This coding is accomplished by adding 128 to *both* characters if it is an integer variable and by adding 128 to the second character if it is a string variable. No coding indicates a floating point variable. The remaining bytes in numeric variables contain the value of the variable. In the case of string variables, the remaining bytes contain the length of the string and the location at the top of memory which contains the first character of the string.

Arrays are quite different in that the length of the variable is determined by the number of elements in the array. The information which must be stored for an array variable includes the name of the variable, which is coded the same as for a simple variable, a pointer to the location of the next variable, the number of dimensions in the array, and the number of elements in the array.

In addition, the value of each element in numeric arrays, or the pointer to the string and its length for string arrays, must be stored. As you can see, array variables can eat up a lot of memory in a hurry. It is best to use the lowest possible number of elements in your arrays. If you do not specify the size of an array, the computer will set it at eleven elements. If you need less than eleven, you'll save a minimum of five bytes per element if you establish the size of the array with a DIMension statement. Although a simple integer variable takes up the same amount of memory as a simple floating point variable, three bytes per element can be saved if you use integer instead of floating point variables in arrays.

Address Pointers

Now and then you'll see a reference to "pointers" within the computer's memory. These are two-byte long numbers, usually located in the first 256 memory cells of the computer, which hold an important address.

Things change while a program is running or being written. For example, if you add a line to a BASIC program, you've expanded the amount of space that the program is taking up in RAM memory. Obviously, when you go to save the program, the computer has to know where the BASIC program ends. So, it keeps track of the "current top of BASIC program" in a pointer. This pointer is located in the 64 in addresses 45 and 46. The number held in cell 46 is multiplied by 256 and then added to the number in cell 45. To see at which address in RAM memory your current BASIC program ends, you can type: ? PEEK (45) + PEEK (46) * 256.

There are a number of other pointers as well, including "limit of memory," "start of arrays," "string storage," and "start of BASIC." The locations of these pointers are listed in *memory maps* for each computer. The best memory map for the Commodore 64 is *Mapping the 64*, by Sheldon Leemon, published by COMPUTE! Books.

There are some interesting things you can do by manipulating these pointers with POKEs. For one thing, you could fool the computer into reserving space for programs in odd places, or even partitioning memory so that two independent BASIC programs could run simultaneously. In any event, pointers hold information essential to the computer, and their values can be accessed using the formula above.

LOADing the Lister

"Variable Lister" is a machine language (ML) program which is loaded by BASIC POKEs, thus eliminating the need for an assembler. The ML is automatically loaded into the top of memory and protected from your BASIC program. Before you run the program, be sure to save a copy since it self-destructs after it is run. When the machine language is loaded, the loader program gives you the location to SYS to when you want to list your variables. For example, when you first use this on the 64, you would type SYS 40704 to list your variables. The program then lists the simple variables in the order of appearance in the program, with indicators of their type. Next the array variables are listed with proper indicators.

To use Variable Lister, first load and run it. Note the SYS number you'll later enter. Next, load the program whose variables you want to list. Type RUN. The BASIC program has to be run before you give the SYS to start the Lister. This is because the variables of a BASIC program are not set up in memory until it's run. Break out of the program by pressing the RUN/STOP–RESTORE keys, then enter the correct SYS (which Lister gave you earlier).

What you'll see on the screen is a list of the simple variables in the order of appearance in the program, along with indicators of their type. Next the array variables are listed with proper indicators. Function variables (in the form fn(x)) are noted by an asterisk (*).

Variable Lister is especially useful when you write programs with many variables and have to find new names. It is also valuable for documenting programs once they're completed.

The variables are listed across the screen to prevent them from scrolling out of view. If you have a printer, the following changes may be made to give you a listing which may be easier to read.

```
160 IF PA<>35126 THEN PRINT"DATA ERROR":END
260 DATA 32,210,255,169,13,32,210
420 DATA 41,32,210,255,169,13,32
```

To send the list to your printer, simply OPEN a file to your printer:

```
OPEN1,4 :CMD1 :SYSXXXXX
```

Variable Lister

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

```
120 ME=PEEK(55)+256*PEEK(56)                    :rem 21
130 VS=ME-256:PA=0                              :rem 14
140 POKE 56,PEEK(56)-1                         :rem 154
150 FOR I=VS TO VS+253:READ A:POKE I,A:PA=PA+A:NEX
    T                                          :rem 241
160 IF PA<>35164 THEN PRINT"DATA ERROR":END
                                               :rem 233
170 PRINT"SYS" VS "TO START":NEW               :rem 170
180 DATA 165,45,197,47,240,106,133             :rem 144
190 DATA 253,165,46,133,254,160,0               :rem 85
200 DATA 169,0,141,61,3,177,253                :rem 237
210 DATA 41,128,208,73,177,253,41               :rem 87
220 DATA 127,32,210,255,200,173,61             :rem 120
230 DATA 3,201,0,208,8,177,253                 :rem 184
240 DATA 41,128,208,59,240,11,177               :rem 87
245 DATA 253,41,128,208,5,169,42,141,61,3,177,253,
    41                                           :rem 2
250 DATA 127,32,210,255,173,61,3                :rem 28
260 DATA 32,210,255,169,32,32,210               :rem 75
270 DATA 255,152,24,105,6,144,5                :rem 239
```

```
280 DATA 164,254,200,132,254,168,101      :rem 230
290 DATA 253,197,47,240,17,208,173        :rem 150
300 DATA 96,169,37,141,61,3,208           :rem 248
310 DATA 176,169,36,141,61,3,208           :rem 39
320 DATA 203,165,49,197,47,240,114        :rem 141
330 DATA 165,47,133,253,165,48,133        :rem 143
340 DATA 254,160,0,169,0,141,61           :rem 232
350 DATA 3,177,253,240,216,41,128          :rem 85
360 DATA 208,77,177,253,41,127,32          :rem 96
370 DATA 210,255,200,173,61,3,201          :rem 69
380 DATA 0,208,6,177,253,41,128           :rem 246
390 DATA 208,63,177,253,41,127,32          :rem 94
400 DATA 210,255,173,61,3,32,210           :rem 18
410 DATA 255,169,40,32,210,255,169        :rem 139
420 DATA 41,32,210,255,169,32,32           :rem 27
430 DATA 210,255,200,177,253,24,101       :rem 171
440 DATA 253,197,49,240,39,177,253        :rem 157
450 DATA 24,101,253,170,200,177,253       :rem 176
460 DATA 101,254,133,254,134,253,208      :rem 231
470 DATA 165,96,169,37,141,61,3             :rem 2
480 DATA 208,172,169,36,141,61,3           :rem 43
490 DATA 208,186,165,48,197,50,208        :rem 160
500 DATA 136,96,200,234,177,253,101       :rem 182
510 DATA 254,197,50,240,224,16,222        :rem 132
520 DATA 136,208,202                      :rem 209
```

Eric Brandon

Disk Defaulter

Short and simple, this machine language routine saves typing if you regularly use a disk drive instead of a cassette recorder. SAVEs, LOADs, and VERIFYs automatically default to disk rather than tape when you have this routine in memory.

When Commodore designed the operating system used in the Commodore 64, the designers assumed that most people would be using a cassette recorder for storage instead of the more expensive disk drive. That's why, when you type LOAD or SAVE, the computer responds by prompting "Press Play On Tape" or "Press Record & Play On Tape." It *defaults* to the tape recorder.

If you're using a disk drive, you have to type the device number — ,8 — after each command (as in LOAD"filename",8). This can become bothersome after a while.

"Disk Defaulter" is a short utility, written in machine language, that modifies the computer's operating system to recognize the disk drive, instead of the cassette recorder, as the default device. As long as the utility is activated, you no longer have to append ,8 to the LOAD, SAVE, and VERIFY commands.

To use Disk Defaulter, enter the program. When you type RUN, this BASIC loader POKEs the machine language into some free memory space and activates the utility. To turn it off (for instance, if you want to use cassette), press RUN/STOP–RESTORE. To turn it back on, type SYS 679.

To load machine language programs, you still must type LOAD "filename",8,1. Also, pressing SHIFT–RUN/STOP will not access the disk drive because it results in a "Missing Filename Error." But otherwise, all LOAD, SAVE, and VERIFY commands will refer to disk.

The only program we've found that interferes with Disk Defaulter is the PAL Assembler for the Commodore 64.

Disk Defaulter

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

```
10 I=679                                :rem 141
20 READ A:IF A=256 THEN 1000            :rem 147
30 POKE I,A:I=I+1:GOTO 20               :rem 130
679 DATA 169,188,141,48,3,169,2          :rem 16
686 DATA 141,49,3,169,195,141,50         :rem 54
693 DATA 3,169,2,141,51,3,96            :rem 103
```

```
700 DATA 162,8,134,186,76,165,244          :rem 100
707 DATA 162,8,134,186,76,237,245,256       :rem 53
1000 PRINT"{CLR}DISK DEFAULTER ACTIVATED" :rem 129
1010 PRINT"USE RUN/STOP-RESTORE TO DEACTIVATE"
                                           :rem 184
1020 PRINT"TYPE SYS 679 TO REACTIVATE"       :rem 6
1030 SYS 679                               :rem 105
```

Chapter 3

High-Speed Graphics

Charles Brannon

Ultrafont + Character Editor

This fast, feature-packed, machine language utility makes custom characters a breeze. Its unique features let you concentrate on your artwork instead of programming.

Anyone who has used graph paper to plot out characters, then tediously converted the rows into decimal numbers, can appreciate a character editor. Instead of drawing and erasing on paper, you can draw your characters freehand with a joystick. "Ultrafont + " has been written to offer almost every conceivable aid to help you design whole character sets.

Typing It In

Ultrafont + is written entirely in machine language, giving you speed and efficiency that BASIC can't match. Although this gives you a product of commercial quality, it carries the liability of lots of typing. Ultrafont + is actually rather short, using less than 4K of memory at hexadecimal location $C000 (49152), which is reserved for programs like Ultrafont + . Therefore, you don't lose one byte of BASIC programming space.

However, 4000 characters require three times as much typing, since each byte must be represented by a three-digit number (000–255). With that much typing, mistakes are inevitable. To make things manageable, we've prepared Ultrafont + to be typed in using MLX, the Machine Language Editor. Full instructions are provided in Appendix D. So despite the typing, rest assured that a few afternoons at the keyboard will yield a substantial reward.

Once you've entered, saved, and run MLX, answer the two questions, *starting address* and *ending address*, with 49152 and 52409, respectively. After you've saved the program with MLX, you can load it from disk with LOAD "filename", 1,1 from tape or LOAD "filename",8,1 from disk. After it's loaded, enter NEW, then SYS 49152. This command runs the machine language program at $C000 (12*4096 = 49152).

The Display

After you SYS to Ultrafont + , you should see the work area. At the bottom of the screen are eight lines of characters. These are the 256 characters you can customize, arranged in eight rows of 32 characters. A flashing square is resting on the @ symbol, the home position

of the character set. Above the eight rows is the main grid, a blown-up view of ten characters. The last row of the screen is reserved for messages. The first time you SYS 49152, you'll be asked whether you want to edit the uppercase/graphics character set, or the lowercase set.

About the Grid

The grid is like a large-sized window on the character set. You see the first five characters and the five beneath them. A large red cursor shows you which character you are currently editing, and a smaller flashing square is the cursor you use to set and clear pixels in order to draw a character.

Moving Around

You can use the cursor keys (up, down, left, right) to move the large red cursor to any character you want to edit. If you move to a character not on the large grid (out of the window), the window will automatically scroll to make the character appear. You can also look at the bottom of the screen to move the larger cursor, as the flashing square on the character set moves with the main grid.

The HOME key moves the small cursor to the upper-left corner of the screen. If you press it twice, it will take you back to the top of the character set — to @.

A joystick (plugged into port 2) moves the small cursor within the grid. If you move the cursor out of the current character, the red cursor will jump to the next character in whatever direction you want to move. The display at the bottom will adjust, and the grid will scroll as necessary. This means that you can ignore the traditional boundaries between characters, and draw shapes as big as the entire character set (256 × 64 pixels — a pixel is a picture element, or dot). You can still edit one character at a time, or make a shape within a 2 × 2 box of characters. There is no wraparound for the cursor in the bottom section of the screen. When it hits an edge, it will go no further in that direction.

The fire button is used to set and clear points. When you press fire, if the cursor is resting on a solid square, it will be turned off. If the square is off, it will be turned on. If you hold down the fire button while you move the joystick, you can stay in the same drawing mode. If you set a point, you will continue to draw as you move. If you clear a point, you can move around and erase points all over the screen.

If the drawing cursor is too fast or too slow to use, just press V to set the cursor velocity (speed). Answer the prompt with a speed from 0 (slow) to 9 (too fast for practical use).

Manipulations

There are several functions that affect the current character (where the red box is). You can rotate, shift, mirror, reverse, erase, replace, and copy characters. The best way to learn is to play with the functions. It's really a lot of fun. The following keys control each function:

Function Keys

f1: Scroll character right. All pixels move right. The rightmost column of pixels wraps around to the left.

f2: Scroll character left. Wraparound is like f1.

f3: Scroll character down. All pixels move down. The last row of pixels wraps around to the top.

f4: Scroll character up. Wraparound is like f3.

R: Rotate. Rotates the character 90 degrees. Press twice to flip the character upside down.

M: Mirror. Creates a mirror image of the character left to right.

CLR (SHIFT–CLR/HOME): Erases the current character.

CTRL-R or CTRL-9: Reverses the character. All set dots are clear, and all empty dots are set. The bottom half of the character set is the reversed image of the top half.

CTRL-back arrow (←): This causes the lower half of the character set to be the inverse of the upper half. This way, you only have to redraw the normal characters, then use CTRL-back arrow to create the inverse set.

F: Fix. Use this if you want to restore the normal pattern for the character. If you've redefined A, and press F while the red cursor is on the character, the Commodore pattern for A will be copied back from ROM.

T: Type. This lets you try out your character set. The screen clears, with a copy of the character set provided for reference. You can type and move the cursor around, just as in BASIC. This is handy for envisioning sample screens, and fitting together multiple-character shapes. If you want to change the background color while in Type mode, press the f1 key. Cycle through the colors by repeatedly hitting the f1 key. Press the RUN/STOP key to exit from Type and return to Ultrafont + .

Saving and Loading Character Sets

To save your creation to tape or disk, press S. Then press either T for tape or D for disk. When requested, enter the filename, up to 16

characters. Don't use the 0: prefix if you're using a disk drive (it's added for you). The screen will clear, display the appropriate messages, and then return to the editing screen if there are no errors. If there *are* errors, such as the disk being full, Ultrafont + will read the disk error message and display it at the bottom of the screen.

Press a key after you've read the message and try to correct the cause of the error before you save again. The computer cannot detect an error during a tape SAVE.

To load a character set previously saved, press L and answer the TAPE OR DISK message. Enter the filename. If you're using tape, be sure the tape is rewound and ready. After the load, you will be returned to the editing screen, and a glance is all it takes to see that the set is loaded. If an error is detected on tape load, you will see the message ERROR ON SAVE/LOAD. Once again, if you are using disk, the error message will be displayed. Press a key to return to editing so you can try again.

Copying and Moving Characters

You can copy one character to another with function keys 7 and 8. When you press f7, the current character will flash briefly, and it will be copied into a little buffer. Ultrafont + will remember that character pattern. You can then position the cursor where you want to copy the character and press f8. The memorized character will then replace the character the cursor is resting on. You can also use the buffer as a fail-safe device. Before you begin to edit a character you've already worked on, press f7 to store it safely away. That way, if you accidentally wipe it out or otherwise garble the character, you can press f8 to bring back your earlier character.

Creating DATA Statements

A very useful command, CTRL-D, allows you to create DATA statements for whatever characters you've defined. Ultrafont + doesn't make DATA statements for all the characters, just the ones you've changed. After you press CTRL-D, Ultrafont + adds the DATA statements to the end of whatever program you have in BASIC memory. If there is no program, the DATA statements exist alone.

You can LOAD Ultrafont + , enter NEW to reset some BASIC pointers, LOAD a program you are working on, then SYS 49152 to Ultrafont + to add DATA to the end of the program. The DATA statements always start at line 63000, so you may want to renumber them. If you press CTRL-D twice, another set of DATA statements will be appended, also numbered from line numbers 63000 and up. Since the keys repeat if held down, just tap CTRL-D. If you hold it down,

you may find a hundred DATA statements have been created! See the notes at the end of this article for more details on using the DATA statements in your own programs.

Exiting Ultrafont +

After you create the DATA, you'll still be in Ultrafont + . If you want to exit to see the DATA statements or go on to other things, press CTRL-X. The screen will reset to the normal colors and you'll see READY. If you've made DATA, a LIST will dramatically reveal it. I recommend you enter the command CLR to make sure BASIC is initialized properly after creating DATA statements. One thing to watch out for: Don't use RUN/STOP–RESTORE to exit Ultrafont + . Ultrafont + moves screen memory from the default area at 1024, and the RUN/STOP–RESTORE combination does not reset the operating system pointers to screen memory. If you do press it, you will not be able to see what you are typing. To fix it, blindly type POKE 648,4 or SYS 49152 to reenter Ultrafont + so you can exit properly.

Reentering Ultrafont +

To restart Ultrafont + within the program, press SHIFT–RUN/STOP. After you've exited to BASIC, you can rerun Ultrafont + with SYS 49152. You'll see the character set you were working on previously, along with the message USE ROM SET? (Y/N). Usually, Ultrafont + will copy the ROM character patterns into RAM where you can change them. If you press N, however, the set you were working on previously is left untouched. Press any other key, like RETURN, to reset the characters to the ROM standard. You can copy either the uppercase/graphics set from ROM, or the lowercase set.

A Whole New World of Multicolor

We're not finished yet. There is a whole other mode of operation within Ultrafont + : the multicolor mode. In multicolor mode, any character can contain up to four colors (one has to be used for the background) simultaneously. Multicolor changes the way the computer interprets character patterns. Instead of a 1 bit representing a solid pixel and 0 representing a blank, the eight bits are organized as four *pairs* of bits. Each pair can represent four possibilities: 00, 01, 10, and 11. Each of these is also a number in decimal from 0 to 3. Each two-bit pattern represents one of the four colors. Programming and using multicolor characters are described in my article "Advanced Use of Character Graphics," found in *COMPUTE!'s First Book of 64 Sound and Graphics.*

Ultrafont + makes multicolor easy. You don't have to keep track of bit-pairs any more than you have to convert binary to decimal. Just press the f5 function key. Presto! The whole screen changes. The normal characters are rather unrecognizable, and the drawing cursor is twice as wide (since eight bits have been reduced to four pixel-pairs, making each dot twice as wide). You have only four dots horizontally per character, but you can easily combine many characters to form larger shapes.

Multicolor also redefines the way the joystick and fire button work. The fire button always lays down a colored rectangle in the color you are currently working with. That color is shown in the center of the drawing cursor. Press the number keys 1, 2, 3, or 4 to choose one of the four different colors to draw with. The number of the key is one more than the bit pattern, so color 1 is bit pattern 00, and color 4 is bit pattern 11. When you first SYS to Ultrafont + , the four colors show up distinctly on a color TV or monitor.

You can easily change the colors. Just hold down SHIFT and press the appropriate number key to change that number's color. You will see the message PRESS COLOR KEY. Now press one of the color keys, from CTRL-1 to CTRL-8 or from Commodore-1 to Commodore-8. Hold down CTRL or the Commodore key as you do this. Instantly, that color, and everything previously drawn in that color, is changed.

Three of the colors (including 1, the background color) can be any of the 16 colors. But because of the way multicolor works, color 4 (represented by bit pattern 11, or 3 in decimal) can only be one of the 8 CTRL-colors. Assigning it one of the Commodore colors just picks the color shown on the face of the color key. Incidentally, it is the color of bit pattern 3 (color 4) that changes according to the character color as set in color memory. The other colors are programmed in multicolor registers 1 and 2 (POKE 53282 and 53283), so all characters share these two colors. When you want to vary a certain color without affecting the rest of the characters, you'll want to draw it in color 4.

Some of the commands in the multicolor mode aren't as useful as others. You have to press f1 and f2 twice to shift a character, since they only shift one bit, which causes all the colors to change. You can use CTRL-R, Reverse, to reverse all the colors (color 1 becomes color 4, color 2 becomes color 3, and color 3 becomes color 2). R: (Rotate) changes all the colors and is rather useless unless you press it twice to just turn the character upside down. M: (Mirror) will switch colors 2 and 3, since bit pattern 01 (color 2) becomes 10 (color 3). You can still copy characters using f7 and f8 (see above).

Returning to Normal

You can switch back instantly to the normal character mode by pressing f6. If you were drawing in multicolor, you can see the bit patterns that make up each color. Multicolor characters look just as strange in normal mode as normal characters look in multicolor.

If you changed colors in the multicolor mode, some of the colors in the normal mode may have changed. You can change these colors as in multicolor mode. Press SHIFT-1 to change the color of the empty pixels, and SHIFT-4 to change the color of the eight rows of characters. Use SHIFT-2 to change the color of the on pixels.

Programming

You'll find the article "Advanced Use of Character Graphics" in *COMPUTE!'s First Book of 64 Sound and Graphics* quite informative.

It shows you how you can make the most of characters. The article includes several short machine language utilities that you can use when writing games or other programs using these custom characters. It shows how your program can read the SAVEd files directly, without having to POKE from DATA statements. You should still have a good grasp of the essentials of programming characters (see Orson Scott Card's "Make Your Own Characters," also in *COMPUTE!'s First Book of 64 Sound and Graphics*). Ultrafont + is intended as an artistic aid in your creations, letting the computer take over the tedious tasks it is best suited for.

Notes: How to Use the DATA Statements

The DATA statements are created from lines 63000 and up. Each line of data has nine numbers. The first number is the internal code of the character (the code you use when POKEing to the screen). It represents an offset into the table of character patterns. The eight bytes that follow are the decimal numbers for the eight bytes it takes to define any character. A sample program to read and display them could be:

```
10 POKE 56,48:CLR
50 READ A:IF A=-1 THEN 70
60 FOR I=0 TO 7:READ B:POKE 12288+A*8+I,B:NEXT:GOT
   O50
70 PRINT CHR$(147);"{10 DOWN}":REM TEN CURSOR DOWN
   S
80 FOR I=0TO7:FORJ=0TO31:POKE1028+J+I*40,I*32+J:PO
   KE55300+J+I*40,1:NEXT:NEXT
90 POKE 53272,(PEEK(53272)AND240)OR 12:END
```

You'll also need to add the following line to the end of your DATA statements:

```
63999 DATA -1
```

If you want to have your cake and eat it, too — that is, also have the normal ROM patterns — copy the normal patterns from ROM down to RAM by adding:

```
20 POKE 56334,PEEK(56334)AND254:POKE 1,PEEK(1)AND2
   51
30 FOR I=0 TO 2047:POKE 12288+I,PEEK(53248+I):NEXT
40 POKE 1,PEEK(1)OR4:POKE 56334,PEEK(56334)OR1
```

Quick Reference: Ultrafont + Commands

Cursor keys:	Move to next character
HOME(CLR/HOME):	Move the cursor to upper-left corner. Press twice to go back to start
V:	Cursor velocity; answer from 0 (slow) to 9 (fast)
f1:	Scroll right with wraparound
f2(SHIFT-f1):	Scroll left
f3:	Scroll down
f4(SHIFT-f3):	Scroll up
R:	Rotate 90 degrees; press twice to invert
M:	Mirror image
SHIFT–CLR/HOME:	Erase current character
CTRL-R or CTRL-9:	Reverse pixels
CTRL-back arrow (←) or CTRL-F:	Copy first four rows, inverted, to last four
F:	Fix character from ROM pattern
L:	Load. Tape or Disk, Filename
S:	Save. Tape or Disk, Filename
T:	Typing mode; RUN/STOP to exit
f5:	Switch to multicolor character mode
f6(SHIFT-f5):	Return to normal character mode
f7:	Memorize character (keep)
f8(SHIFT-f7):	Recall character (put)
CTRL-D:	Make DATA statements
SHIFT–RUN/STOP:	Restart Ultrafont +
CTRL-X:	Exit Ultrafont + to BASIC

Ultrafont +

Be sure to read "Using the Machine Language Editor: MLX," Appendix D, before typing in this program.

```
49152 :076,247,196,000,001,003,011
49158 :004,000,001,003,004,000,018
49164 :173,048,002,072,173,045,013
49170 :002,141,048,002,141,079,175
49176 :002,032,047,193,104,141,031
49182 :048,002,169,100,133,252,222
49188 :169,000,133,251,133,167,121
49194 :169,216,133,168,169,008,137
49200 :141,040,002,169,002,141,031
49206 :042,002,169,005,141,041,198
49212 :002,174,003,192,173,079,171
49218 :002,205,048,002,208,002,021
49224 :162,002,142,080,002,160,108
49230 :000,177,253,170,173,063,146
49236 :002,240,003,076,233,192,062
49242 :169,207,145,251,138,010,242
49248 :170,176,008,173,080,002,193
49254 :145,167,076,112,192,173,199
49260 :004,192,145,167,200,192,240
49266 :008,208,221,024,165,251,223
49272 :105,008,133,251,133,167,149
49278 :165,252,105,000,133,252,009
49284 :105,116,133,168,024,165,075
49290 :253,105,008,133,253,165,031
49296 :254,105,000,133,254,056,178
49302 :238,079,002,206,041,002,206
49308 :173,041,002,208,156,056,024
49314 :173,079,002,233,005,141,027
49320 :079,002,056,165,253,233,188
49326 :039,133,253,165,254,233,227
49332 :000,133,254,206,040,002,047
49338 :173,040,002,240,003,076,208
49344 :056,192,206,042,002,173,095
49350 :042,002,240,030,169,008,177
49356 :141,040,002,024,173,079,151
49362 :002,105,032,141,079,002,059
49368 :024,165,253,105,248,133,120
49374 :253,165,254,105,000,133,108
49380 :254,076,056,192,096,134,012
49386 :097,169,000,141,043,002,174
49392 :006,097,046,043,002,006,184
49398 :097,046,043,002,174,043,139
49404 :002,169,207,145,251,200,202
49410 :169,247,145,251,136,189,115
49416 :003,192,145,167,200,145,092
49422 :167,200,192,008,208,215,236
```

```
49428 :076,117,192,169,000,141,203
49434 :026,208,165,001,041,251,206
49440 :133,001,096,165,001,009,181
49446 :004,133,001,169,001,141,231
49452 :026,208,096,169,000,133,164
49458 :254,173,048,002,010,133,158
49464 :253,038,254,006,253,038,130
49470 :254,006,253,038,254,169,012
49476 :112,005,254,133,254,096,154
49482 :032,047,193,160,000,177,171
49488 :253,073,255,145,253,200,235
49494 :192,008,208,245,032,012,015
49500 :192,096,169,102,133,252,012
49506 :169,218,133,168,173,058,249
49512 :002,174,063,002,240,002,075
49518 :009,008,141,080,002,169,007
49524 :132,133,251,133,167,162,070
49530 :008,169,000,133,097,160,177
49536 :000,165,097,145,251,230,248
49542 :097,173,080,002,145,167,030
49548 :200,192,032,208,240,024,012
49554 :165,251,105,040,133,251,067
49560 :133,167,165,252,105,000,206
49566 :133,252,105,116,133,168,041
49572 :202,208,216,096,032,082,232
49578 :203,173,044,002,141,024,245
49584 :208,169,200,013,063,002,063
49590 :141,022,208,169,000,141,095
49596 :032,208,141,033,208,032,074
49602 :094,193,173,058,002,174,120
49608 :063,002,240,002,009,008,012
49614 :141,134,002,165,209,133,222
49620 :243,024,165,210,105,116,051
49626 :133,244,164,211,177,209,076
49632 :073,128,145,209,177,243,175
49638 :072,173,134,002,145,243,231
49644 :032,228,255,240,251,201,163
49650 :133,208,006,238,032,208,043
49656 :238,033,208,170,164,211,248
49662 :177,209,073,128,145,209,171
49668 :104,145,243,138,032,210,108
49674 :255,032,225,255,208,193,154
49680 :032,114,203,169,000,141,163
49686 :134,002,169,012,141,032,000
49692 :208,076,141,196,032,023,192
49698 :193,169,112,133,252,173,042
49704 :082,002,133,254,162,008,169
49710 :169,000,133,253,133,251,217
49716 :168,177,253,145,251,200,222
```

```
49722 :208,249,230,254,230,252,201
49728 :202,208,242,165,252,201,054
49734 :128,240,007,169,208,133,187
49740 :254,076,044,194,032,035,199
49746 :193,162,004,189,006,192,060
49752 :157,002,192,202,208,247,072
49758 :096,169,112,133,252,169,001
49764 :116,133,254,169,000,133,137
49770 :253,133,251,168,162,004,053
49776 :177,251,073,255,145,253,242
49782 :200,208,247,230,254,230,207
49788 :252,202,208,240,096,032,130
49794 :047,193,160,000,177,253,192
49800 :010,008,074,040,042,145,199
49806 :253,200,192,008,208,242,221
49812 :076,012,192,032,047,193,188
49818 :160,000,177,253,074,008,058
49824 :010,040,106,145,253,200,146
49830 :192,008,208,242,076,012,136
49836 :192,032,047,193,160,000,028
49842 :177,253,133,097,200,177,191
49848 :253,136,145,253,200,200,091
49854 :192,008,208,245,165,097,081
49860 :136,145,253,076,012,192,242
49866 :032,047,193,160,007,177,050
49872 :253,133,097,136,177,253,233
49878 :200,145,253,136,016,247,187
49884 :200,165,097,145,253,076,132
49890 :012,192,032,047,193,160,094
49896 :000,169,000,133,097,162,025
49902 :008,177,253,010,102,097,117
49908 :202,208,250,165,097,145,031
49914 :253,200,192,008,208,233,064
49920 :076,012,192,032,047,193,040
49926 :160,008,169,000,153,048,032
49932 :002,136,208,250,169,007,016
49938 :133,097,152,170,169,000,227
49944 :133,007,177,253,074,145,045
49950 :253,038,007,202,016,251,029
49956 :166,097,165,007,029,049,037
49962 :002,157,049,002,198,097,035
49968 :165,097,016,224,200,192,174
49974 :008,208,215,136,185,049,087
49980 :002,145,253,136,016,248,092
49986 :076,012,192,032,047,193,106
49992 :160,000,152,145,253,200,214
49998 :192,008,208,249,076,012,055
50004 :192,120,169,127,141,013,078
50010 :220,169,001,141,026,208,087
```

```
50016 :169,177,141,018,208,169,210
50022 :027,141,017,208,169,118,014
50028 :141,020,003,169,195,141,009
50034 :021,003,088,096,173,018,001
50040 :208,201,177,208,039,169,098
50046 :242,141,018,208,173,044,184
50052 :002,141,024,208,173,022,190
50058 :208,041,239,013,063,002,192
50064 :141,022,208,173,057,002,235
50070 :141,033,208,169,001,141,075
50076 :025,208,104,168,104,170,167
50082 :104,064,169,177,141,018,067
50088 :208,169,158,141,024,208,052
50094 :173,032,208,141,033,208,201
50100 :169,200,141,022,208,238,134
50106 :037,208,169,001,141,025,255
50112 :208,076,049,234,085,064,140
50118 :000,064,064,000,076,064,210
50124 :000,076,064,000,076,064,228
50130 :000,076,064,000,064,064,222
50136 :000,085,064,000,000,000,109
50142 :085,080,000,064,016,000,211
50148 :064,016,000,064,016,000,132
50154 :064,016,000,064,016,000,138
50160 :064,016,000,064,016,000,144
50166 :064,016,000,085,080,000,235
50172 :000,000,000,255,255,255,249
50178 :000,001,001,001,000,255,004
50184 :001,000,000,255,001,000,009
50190 :000,255,001,018,085,076,193
50196 :084,082,065,070,079,078,222
50202 :084,032,043,032,086,046,093
50208 :050,146,095,069,082,082,044
50214 :079,082,032,079,078,032,164
50220 :083,065,086,069,047,076,214
50226 :079,065,068,095,018,084,203
50232 :146,065,080,069,032,079,015
50238 :082,032,018,068,146,073,225
50244 :083,075,063,095,070,073,015
50250 :076,069,078,065,077,069,252
50256 :058,095,069,078,084,069,021
50262 :082,032,067,079,076,079,245
50268 :082,032,075,069,089,095,022
50274 :085,083,069,032,082,079,016
50280 :077,032,083,069,084,063,000
50286 :032,040,089,047,078,041,181
50292 :095,018,085,146,080,080,108
50298 :069,082,067,065,083,069,045
50304 :032,079,082,032,018,076,191
```

```
50310 :146,079,087,069,082,063,148
50316 :095,169,017,160,196,133,142
50322 :251,132,252,160,040,169,126
50328 :032,153,191,103,136,208,207
50334 :250,177,251,200,201,095,052
50340 :208,249,136,132,097,152,114
50346 :074,073,255,056,105,020,241
50352 :168,162,024,024,032,240,058
50358 :255,160,000,177,251,032,033
50364 :210,255,200,196,097,144,010
50370 :246,096,133,251,132,252,024
50376 :160,040,169,032,153,191,177
50382 :103,136,208,250,162,024,065
50388 :160,000,024,032,240,255,155
50394 :160,000,177,251,201,095,078
50400 :240,006,032,210,255,200,143
50406 :208,244,096,174,076,002,006
50412 :240,008,160,000,200,208,028
50418 :253,202,208,250,096,173,144
50424 :002,221,009,003,141,002,114
50430 :221,173,000,221,041,252,138
50436 :009,002,141,000,221,169,034
50442 :100,141,136,002,169,147,193
50448 :032,210,255,169,000,141,055
50454 :134,002,169,008,032,210,065
50460 :255,160,000,152,153,128,108
50466 :099,200,016,250,168,185,184
50472 :196,195,153,128,099,200,243
50478 :192,023,208,245,160,000,106
50484 :185,219,195,153,192,099,071
50490 :200,192,032,208,245,169,080
50496 :156,141,044,002,169,012,076
50502 :141,032,208,169,128,141,121
50508 :138,002,032,085,195,169,185
50514 :048,141,076,002,169,011,017
50520 :141,057,002,169,007,169,121
50526 :000,141,048,002,141,045,215
50532 :002,141,063,002,173,006,231
50538 :192,009,008,141,058,002,004
50544 :173,004,192,141,034,208,096
50550 :173,005,192,141,035,208,104
50556 :032,012,192,032,094,193,167
50562 :169,203,205,011,192,240,126
50568 :017,141,011,192,162,208,099
50574 :142,082,002,032,032,194,114
50580 :032,012,192,076,170,197,059
50586 :169,098,160,196,032,145,186
50592 :196,032,228,255,240,251,082
50598 :201,078,240,029,169,117,232
```

```
50604 :160,196,032,145,196,032,165
50610 :228,255,240,251,162,208,242
50616 :201,076,208,002,162,216,025
50622 :142,082,002,032,032,194,162
50628 :032,012,192,032,141,196,033
50634 :169,142,141,248,103,169,150
50640 :143,141,249,103,169,003,248
50646 :141,021,208,169,024,141,150
50652 :000,208,169,000,141,016,242
50658 :208,169,051,141,001,208,236
50664 :169,176,141,003,208,169,074
50670 :053,141,002,208,169,000,043
50676 :141,029,208,141,023,208,226
50682 :141,038,208,169,003,141,182
50688 :028,208,169,000,141,059,093
50694 :002,141,060,002,173,000,128
50700 :220,072,041,015,073,015,192
50706 :141,061,002,104,041,016,127
50712 :141,062,002,032,228,255,232
50718 :240,006,032,169,199,076,240
50724 :010,198,032,233,196,173,110
50730 :062,002,208,003,032,060,153
50736 :199,173,062,002,073,016,061
50742 :141,075,002,173,061,002,252
50748 :240,204,174,061,002,189,162
50754 :251,195,172,063,002,240,221
50760 :001,010,024,109,059,002,021
50766 :141,059,002,024,173,060,025
50772 :002,125,006,196,141,060,102
50778 :002,174,059,002,016,027,114
50784 :162,000,142,059,002,173,122
50790 :048,002,041,031,240,015,223
50796 :206,045,002,162,007,173,191
50802 :063,002,240,002,162,006,077
50808 :142,059,002,174,059,002,046
50814 :224,040,144,022,162,039,245
50820 :142,059,002,173,048,002,046
50826 :041,031,201,031,240,008,178
50832 :238,045,002,162,032,142,253
50838 :059,002,172,060,002,016,205
50844 :026,160,000,140,060,002,032
50850 :173,048,002,201,032,144,250
50856 :014,056,173,045,002,233,179
50862 :032,141,045,002,160,007,049
50868 :140,060,002,172,060,002,104
50874 :192,016,144,026,160,015,227
50880 :140,060,002,173,048,002,105
50886 :201,224,176,014,024,173,242
50892 :045,002,105,032,141,045,062
```

```
50898 :002,160,008,140,060,002,070
50904 :173,059,002,172,060,002,172
50910 :074,074,074,192,008,144,020
50916 :002,105,031,109,045,002,010
50922 :141,048,002,041,224,074,252
50928 :074,105,176,141,003,208,179
50934 :173,048,002,041,031,010,039
50940 :010,010,105,053,141,002,061
50946 :208,169,000,105,000,133,105
50952 :097,173,060,002,010,010,104
50958 :010,105,051,141,001,208,018
50964 :173,059,002,010,010,010,028
50970 :038,097,105,024,141,000,175
50976 :208,165,097,105,000,141,236
50982 :016,208,173,048,002,205,178
50988 :081,002,240,009,032,012,164
50994 :192,173,048,002,141,081,175
51000 :002,076,010,198,032,047,165
51006 :193,173,060,002,041,007,026
51012 :168,173,059,002,041,007,006
51018 :073,007,170,232,134,097,019
51024 :056,169,000,042,202,208,245
51030 :252,174,063,002,208,048,065
51036 :133,097,173,075,002,208,012
51042 :022,169,000,141,064,002,240
51048 :141,038,208,177,253,037,190
51054 :097,208,008,169,001,141,222
51060 :064,002,141,038,208,165,222
51066 :097,073,255,049,253,174,255
51072 :064,002,240,002,005,097,026
51078 :145,253,032,012,192,096,096
51084 :133,098,074,005,098,073,109
51090 :255,049,253,166,097,202,144
51096 :133,097,173,066,002,074,185
51102 :042,202,208,252,005,097,196
51108 :145,253,076,012,192,141,215
51114 :065,002,174,197,199,221,004
51120 :197,199,240,004,202,208,202
51126 :248,096,202,138,010,170,022
51132 :189,233,199,072,189,232,022
51138 :199,072,096,034,133,137,097
51144 :134,138,077,082,147,018,028
51150 :145,017,157,029,070,135,247
51156 :139,049,050,051,052,019,060
51162 :136,140,033,034,035,036,120
51168 :086,083,076,024,004,006,247
51174 :131,084,150,194,128,194,087
51180 :201,194,172,194,227,194,138
51186 :002,195,068,195,073,193,200
```

```
51192 :052,200,074,200,096,200,046
51198 :118,200,142,200,177,200,011
51204 :212,200,224,200,224,200,240
51210 :224,200,224,200,241,200,019
51216 :010,201,032,201,050,201,199
51222 :050,201,050,201,050,201,007
51228 :124,201,175,202,059,203,224
51234 :075,203,199,203,094,194,234
51240 :043,200,167,193,162,255,036
51246 :154,032,129,255,076,247,171
51252 :196,173,060,002,041,007,019
51258 :133,097,056,173,060,002,067
51264 :233,008,056,229,097,141,060
51270 :060,002,076,138,200,173,207
51276 :060,002,041,007,133,097,160
51282 :024,173,060,002,105,008,198
51288 :056,229,097,141,060,002,161
51294 :076,138,200,173,059,002,230
51300 :041,007,133,097,056,173,095
51306 :059,002,233,008,056,229,181
51312 :097,141,059,002,076,138,113
51318 :200,173,059,002,041,007,088
51324 :133,097,024,173,059,002,100
51330 :105,008,056,229,097,141,254
51336 :059,002,104,104,076,091,060
51342 :198,032,047,193,032,023,155
51348 :193,160,007,024,173,082,019
51354 :002,101,254,105,143,133,124
51360 :252,165,253,133,251,177,111
51366 :251,145,253,136,016,249,192
51372 :032,035,193,076,012,192,200
51378 :169,016,141,063,002,169,226
51384 :001,141,029,208,032,012,095
51390 :192,032,094,193,169,050,152
51396 :141,065,002,032,225,200,093
51402 :173,059,002,041,254,141,104
51408 :059,002,076,138,200,169,084
51414 :000,141,063,002,141,029,078
51420 :208,032,012,192,096,056,048
51426 :173,065,002,233,049,141,121
51432 :066,002,170,189,003,192,086
51438 :141,038,208,096,173,059,185
51444 :002,013,060,002,208,003,020
51450 :141,045,002,169,000,141,236
51456 :059,002,141,060,002,032,040
51462 :012,192,076,138,200,032,144
51468 :074,193,032,074,193,032,098
51474 :047,193,160,000,177,253,080
51480 :153,067,002,200,192,008,134
```

```
51486 :208,246,096,032,047,193,084
51492 :160,000,185,067,002,145,083
51498 :253,200,192,008,208,246,125
51504 :076,012,192,169,082,160,227
51510 :196,032,145,196,032,228,115
51516 :255,240,251,162,000,221,165
51522 :218,232,240,008,232,224,196
51528 :016,208,246,076,141,196,187
51534 :056,173,065,002,233,033,128
51540 :168,138,153,003,192,192,162
51546 :003,240,010,192,000,240,007
51552 :022,153,033,208,076,119,195
51558 :201,174,063,002,240,002,016
51564 :041,007,141,058,002,153,254
51570 :003,192,032,094,193,032,148
51576 :012,192,076,141,196,169,138
51582 :161,160,201,032,145,196,253
51588 :032,228,255,056,233,048,216
51594 :048,248,201,010,176,244,041
51600 :133,097,056,169,009,229,069
51606 :097,010,010,010,010,141,172
51612 :076,002,076,141,196,067,202
51618 :085,082,083,079,082,032,093
51624 :086,069,076,079,067,073,106
51630 :084,089,032,040,048,045,000
51636 :057,041,063,095,160,000,084
51642 :140,078,002,169,164,032,003
51648 :210,255,169,157,032,210,201
51654 :255,032,228,255,240,251,179
51660 :172,078,002,133,097,169,087
51666 :032,032,210,255,169,157,041
51672 :032,210,255,165,097,201,152
51678 :013,240,039,201,020,208,175
51684 :013,192,000,240,209,136,250
51690 :169,157,032,210,255,076,109
51696 :186,201,041,127,201,032,004
51702 :144,194,192,020,240,190,202
51708 :165,097,153,000,002,032,189
51714 :210,255,200,076,186,201,106
51720 :169,095,153,000,002,152,067
51726 :096,032,231,255,169,054,083
51732 :160,196,032,145,196,032,013
51738 :228,255,240,251,162,001,139
51744 :201,084,240,011,162,008,226
51750 :201,068,240,005,104,104,248
51756 :076,141,196,141,077,002,165
51762 :160,001,169,001,032,186,087
51768 :255,169,072,160,196,032,172
51774 :196,196,032,184,201,208,055
```

```
51780 :007,173,077,002,201,084,100
51786 :208,237,173,077,002,201,204
51792 :068,208,066,169,064,141,028
51798 :020,002,169,048,141,021,231
51804 :002,169,058,141,022,002,230
51810 :160,000,185,000,002,153,086
51816 :023,002,200,204,078,002,101
51822 :208,244,169,044,153,023,183
51828 :002,169,080,153,024,002,034
51834 :173,065,002,201,083,208,086
51840 :012,169,044,153,025,002,021
51846 :169,087,153,026,002,200,003
51852 :200,200,200,200,200,200,060
51858 :076,163,202,160,000,185,164
51864 :000,002,153,020,002,200,017
51870 :204,078,002,208,244,152,022
51876 :162,020,160,002,032,189,217
51882 :255,169,160,133,178,096,137
51888 :032,015,202,032,082,203,230
51894 :169,000,133,253,133,251,097
51900 :169,112,133,252,162,255,247
51906 :160,119,169,251,032,216,117
51912 :255,176,011,032,183,255,088
51918 :208,006,032,114,203,076,077
51924 :141,196,032,114,203,032,162
51930 :231,255,173,077,002,201,133
51936 :068,240,015,169,035,160,143
51942 :196,032,145,196,032,228,035
51948 :255,240,251,076,141,196,115
51954 :169,000,032,189,255,169,032
51960 :015,162,008,160,015,032,128
51966 :186,255,032,192,255,162,056
51972 :015,032,198,255,160,000,152
51978 :032,207,255,201,013,240,190
51984 :007,153,000,002,200,076,198
51990 :010,203,169,095,153,000,140
51996 :002,032,204,255,169,000,178
52002 :160,002,032,145,196,162,219
52008 :015,032,201,255,169,073,017
52014 :032,210,255,169,013,032,245
52020 :210,255,032,231,255,076,087
52026 :234,202,032,015,202,032,007
52032 :082,203,169,000,032,213,251
52038 :255,176,141,076,114,203,011
52044 :169,004,141,136,002,000,016
52050 :120,169,000,141,026,208,234
52056 :169,255,141,013,220,169,031
52062 :049,141,020,003,169,234,198
52068 :141,021,003,169,000,141,063
```

```
52074 :021,208,169,147,088,076,047
52080 :210,255,032,085,195,169,034
52086 :003,141,021,208,032,012,023
52092 :192,032,094,193,076,141,084
52098 :196,248,169,000,141,000,116
52104 :001,141,001,001,224,000,248
52110 :240,021,202,024,173,000,034
52116 :001,105,001,141,000,001,141
52122 :173,001,001,105,000,141,063
52128 :001,001,076,140,203,216,029
52134 :173,001,001,009,048,141,027
52140 :002,001,173,000,001,041,134
52146 :240,074,074,074,074,009,211
52152 :048,141,001,001,173,000,036
52158 :001,041,015,009,048,141,189
52164 :000,001,096,096,056,165,098
52170 :045,233,002,133,045,165,057
52176 :046,233,000,133,046,169,067
52182 :024,133,057,169,246,133,208
52188 :058,169,000,141,079,002,157
52194 :133,251,133,253,169,112,253
52200 :133,254,173,082,002,133,241
52206 :252,032,023,193,160,000,130
52212 :177,251,209,253,208,062,124
52218 :200,192,008,208,245,238,061
52224 :079,002,024,165,253,105,116
52230 :008,133,253,133,251,165,181
52236 :254,105,000,133,254,109,099
52242 :082,002,105,143,133,252,223
52248 :173,079,002,208,213,169,100
52254 :000,168,145,045,200,145,221
52260 :045,024,165,045,105,002,166
52266 :133,045,165,046,105,000,024
52272 :133,046,032,035,193,076,051
52278 :051,165,160,000,024,165,107
52284 :045,105,041,145,045,200,129
52290 :165,046,105,000,145,045,060
52296 :200,165,057,145,045,200,116
52302 :165,058,145,045,200,169,092
52308 :131,145,045,174,079,002,148
52314 :032,131,203,200,173,002,063
52320 :001,145,045,200,173,001,149
52326 :001,145,045,200,173,000,154
52332 :001,145,045,200,132,097,216
52338 :160,000,132,098,177,253,166
52344 :170,032,131,203,164,097,149
52350 :169,044,145,045,200,173,134
52356 :002,001,145,045,173,001,243
52362 :001,200,145,045,173,000,190
```

```
52368 :001,200,145,045,200,132,099
52374 :097,164,098,200,192,008,141
52380 :208,214,164,097,169,000,240
52386 :145,045,160,000,177,045,222
52392 :072,200,177,045,133,046,073
52398 :104,133,045,230,057,208,183
52404 :002,230,058,076,255,203,236
```

Charles Brannon

Sprite Magic:
An All-Machine-Language Sprite Editor

Sprites make animation on the 64 fun and easy to program. But actually drawing and creating sprites with graph paper can be tedious. "Sprite Magic" simplifies their creation, and lets you concentrate on the artistic aspects of sprite design. You can even animate minimovies!

What Is a Sprite Editor?

Most of what you've read about sprites covers how to program them: setting them up, protecting memory, moving and animating them, and using them in games. But sprite design is usually left up to you.

A sprite is defined by 63 binary numbers. The one bits (on) represent solid pixels. Zeros (off) represent blank areas, through which the screen background is visible. Normally, you sketch a sprite on a grid 24 squares across and 21 squares high. This is three bytes per row (8 bits * 3 bytes = 24 bits) and 21 rows of bytes (3 * 21 = 63 bytes). But after you've drawn the sprite, you have to convert the squares into binary, and then into decimal so that you can put the numbers in DATA statements.

There are utility programs that will do the conversion for you, even editors that let you clear and set squares with a joystick. Since you're using a computer, other functions can be supported to let you clear, invert, reflect, reverse, shift, and test out your sprite. The more work the computer does, the less you have to think in terms of binary numbers. Having used many sprite editors, I craved a utility that would make sprites easy to draw and fun to use. Although there are many good sprite editors available for the 64, none had all the features I wanted. So I wrote "Sprite Magic."

Sprite Magic includes the best features of most sprite editors, including true multicolor mode, and pulls it off with the speed and power of an all machine language program. Sprite Magic's style (and even some of the coding) came from "Ultrafont + ," an all machine language character editor also in this book. As a matter of fact, many of the commands are the same, which lets you get up to speed quickly. If you've learned how to use Ultrafont + , it won't take much to become comfortable with Sprite Magic.

Typing It In

Since Sprite Magic is an all machine language program, you cannot enter it as you do a BASIC program. Machine language is basically a bunch of numbers; the numbers make no sense in themselves. Only the 6510 microprocessor in your machine can interpret and execute these numbers. Since typing in numbers is no fun, we've tried to make it as painless as possible with MLX, the Machine Language Editor. You'll find MLX, and the explanation of its use and commands, in Appendix D of this book. If you haven't already typed in MLX, do so before you try to enter Sprite Magic. MLX is used for two other machine language programs in this book, as well as in *COMPUTE!* magazine and *COMPUTE!'s Gazette*, so save it for future use.

After you've typed in MLX, RUN it and answer the prompts of Starting Address and Ending Address with 49152 and 51821, respectively. You'll then be ready to start typing in Sprite Magic. Enter each line from the listing at the end of this article. The last number in each line is a checksum, so type it carefully. If the checksum you've typed matches the checksum computed from the line you typed, a pleasant bell tone tells you you've typed the line correctly. If the number doesn't match, a buzzer warns you to reenter the line. This way, you should be able to type in Sprite Magic correctly the first time.

Assuming you've typed and saved Sprite Magic, here's how you get it up and running. If you used the filename "SPRITE MAGIC", type:

LOAD "SPRITE MAGIC",8,1 (for disk)
or
LOAD "SPRITE MAGIC",1,1 (for tape)

Be sure to add the (,1) to the end. After the computer comes back with the READY message, type NEW and press RETURN. This resets some important memory locations, but leaves Sprite Magic in its protected cubbyhole at $C000.

Doodle!

Activate Sprite Magic with SYS 49152. Instantly, the main screen should appear, with a large 24 × 21 grid. The grid is a blowup of the sprite you're editing. The actual sprite will be seen to the right of the grid. The flashing square within the large grid is your cursor. You move the cursor with either the cursor keys on the keyboard, or with a joystick plugged into port 2. To light up a blank spot (in other words, to turn that pixel on), press either the space bar or the joystick fire button. If the square is already lit, it will turn dark. This signifies that the pixel has been turned off. The button or space bar thus *toggles* points on or off. You can draw your sprite quite easily in this

manner. One fine point: With the joystick, you can hold down the fire button and move the cursor. If the first point you change was set, the fire button will continue to set points as you move the joystick, regardless of the other points' original state. If the first point you change was empty, you can hold down the fire button and move about, clearing anything the cursor passes over. Notice how any changes are immediately visible in the actual sprite.

If you've just entered Sprite Magic, the grid is probably full of garbage pixels. To clear out the grid for a new picture, press SHIFT-CLR/HOME. You now have an empty area (a fresh canvas, so to speak) to draw upon. You can press CLR/HOME without holding down SHIFT to home the cursor to the upper left-hand corner of the grid.

Does the cursor move too slow or too fast? To change the velocity (speed) of the cursor, press V. Answer the prompt with a number key from 0 (slow) to 9 (very fast).

Shift, Expansion, and Symmetry

Sometimes when you're drawing, it's necessary to reposition the shape within the grid. The first two function keys let you shift the sprite shape around within the grid. If you shift something out of the grid, it wraps around to the opposite side. The f1 key shifts right, f3 shifts down. Use the SHIFT key along with the function key to move in the opposite direction: f2 moves the sprite shape left; f4, up.

After you've drawn something, press the F key. Instantly, the sprite is flipped upside down. Press it again to flip it back over. Remember F as the command for Flip. Now try M, for Mirror. The shape you've drawn is mirrored left to right. Of course, if you've drawn something symmetrical, you may not see any change.

Now try CTRL-R or CTRL-9. The sprite will become reversed. Every square that was on is now turned off, and vice versa.

A sprite can also be expanded or contracted either horizontally or vertically, or *both* horizontally and vertically. The X and Y keys on the keyboard let you do this. Press X to switch from wide to narrow, or vice versa. Press Y to switch from tall to short, or vice versa. The main grid will not change size or proportion (there's not enough room on the screen).

An unusual command is Symmetry. I added this command after some suggestions that many shapes are symmetrical from left to right, as if a mirror were put in the middle of the grid. To enter the Symmetry mode, press the back arrow key (found in the upper left-hand corner of the keyboard). Now, every square drawn on one side will be instantly mirrored to the left. Blank squares are not copied

over, though, so you cannot erase in this mode. This command is not only quite useful, but also a great deal of fun to play with. To return to normal editing, press the back arrow key again.

Notice the number in the upper right-hand corner of the screen. This is the sprite page number, which can range from 0 to 255. You start out at the top of the sprite memory. The + and – keys are used to go forward or backward through sprite shapes. Press the minus key and see how you now have a new shape in the grid. There is a limit to how far back you can go. If you have no BASIC program in memory, you can step back to sprite page number 36. However, character information resides in sprite pages below 128. You can still clear the page and draw a sprite shape on pages below 128, but it won't really register. To be safe, use only the sprite pages from 128 on up. If you have a program in memory, Sprite Magic will not let you step back past its end. This protects your program from being accidentally overwritten by a sprite shape. If you want maximum space available for sprite shapes, be sure to NEW out any BASIC program before you SYS 49152. You'll sometimes want to keep a program in memory, however. We'll show you why a bit later.

Programming note: The sprite page number, when multiplied by 64, gives you the starting memory location for the 63 numbers representing the sprite.

Put It in the Buffer

You might use Flip to design two views of a shape, such as a spaceship pointing in two directions. Draw one freehand, then do the other with Flip. Mirror can be used to design separate left and right views as well. But what you first need is a way to copy the original shape to another sprite area. One way to do this is to copy the sprite shape to an area of memory (a buffer). You can use + or – to step to another sprite page, then copy the buffer to the sprite. This, if you remember, is the way you copy characters with Ultrafont + . The same keys are used in Sprite Magic. Press f7 to copy the sprite to the buffer. The grid flashes to affirm this. Then go to the sprite page where you want to put the copy and press f8 (SHIFT-f7). The shape in the buffer replaces any shape already in the sprite grid. You can also use the buffer as a fail-safe device. Before modifying an existing sprite, press f7 to save it in the buffer. Then, if you mangle the sprite, or accidentally erase it, you can recall the previous shape from the buffer.

Computer Disney?

The buffer is also useful for animation. Since you can change sprite pages so easily, you can also use Sprite Magic as an animation design tool. Cartoons make only minor changes between frames. Too much change makes the animation jerky. So put the first frame into the buffer, copy it to the next area, then make a change. Put the new image into the buffer, copy it again to a new area, then make another small change. Continue in this fashion as you build up a whole series of frames. Put different but similar shapes on adjacent pages, then hold down plus or minus to step through the shapes. As with cartoon animation, you will get the illusion of motion. Use a cursor velocity of 9 for maximum speed. Even if you don't care to program sprites, Sprite Magic is a fun tool for making moving cartoons.

A Bit of Color

The normal drawing mode lets you set or clear points, but in only one color. If you're willing to give up half as many horizontal points, you can have four colors to work with. Multicolor mode lets any square be one of four colors, but gives you only 12 pixels across instead of 24. This is because two dots are grouped together to give four combinations. The colors come from four memory locations:

Pattern	Color location	
00	53281	Background color register
01	53285	Sprite multicolor register 0
10	53287–53294	Sprite color registers
11	53286	Sprite multicolor register 1

There are two multicolor sprite registers, which are shared between all sprites (in programming, but not in Sprite Magic, you can have eight sprites on the screen at the same time). The bit pattern marked 10 is unique to each sprite and comes from that sprite's own color register. 00 is blank, and whatever is underneath the sprite shape will show through.

The reason for this sojourn into bits and addresses is that only the 10 bit pattern has a unique color for that sprite. If you're designing several sprites for a game, remember that anything drawn in that color can be changed individually for each sprite. Squares drawn with bit pattern 01 or 11 will be colored from two locations shared by all sprites.

Many sprite editors let you see how the sprite would look in multicolor, but you still have to pair up the pixels yourself and keep

track of binary bit pairs. No fun! Instead, Sprite Magic offers a multicolor mode. When you press f5, the screen instantly changes. Each square in the grid is now rectangular, two squares wide. The cursor has also been enlarged, and can be moved about as before in the new grid. But the way you set and clear points has been changed, since you are now working with four colors.

Multicolor Palette

The fire button or the space bar always sets a point, but you have to tell Sprite Magic which color you are currently drawing in. The number keys 1 to 4 select the drawing color. The number you press is one number higher than the binary value of the bit-pairs in the table above. The 1 key, for instance, chooses the 00 bit-pair, which represents the background color. In practice, you are choosing from a palette of four colors. The 1 key is normally used when you want to erase.

When you press a number key from 1 to 4, the border color changes to remind you which color you're drawing with. If you want to change one of the four colors, hold down SHIFT while you type the number. The prompt ENTER COLOR KEY appears. Now you have to enter another key combination. Press CTRL and one of the number keys from 1 to 8, or hold down the Commodore key and one of the number keys from 1 to 8. These are the same key combinations you use to change the text color in BASIC. You can also change the screen background color by pressing the letter B on the keyboard until the color you want appears.

Some Sprite Magic commands act strangely in multicolor mode. For example, a shift left or shift right (done with the f1 or f2 key respectively) moves the sprite over by only one bit, which changes the color assignments. In general, you must press f1 or f2 twice to preserve the same colors. Pressing the M key (for Mirror) reverses the bit-pairs, so that every 01 becomes a 10. The effect is that colors 2 and 3 are exchanged. The R key (Reverse) also inverts the bits, so that 01 becomes 10, 10 becomes 01, 00 becomes 11, and 11 becomes 00. Colors 2 and 3 are switched, as well as colors 1 and 4. The Symmetry command (back arrow) also does not work in multicolor mode.

If you want to go back to normal (nonmulticolor) mode, press the f6 key (SHIFT-f5). There's nothing to prevent you from designing both normal and multicolor sprites on different pages.

If you changed colors in the multicolor mode, some of the colors in the normal mode may have been changed. You can alter these colors as in multicolor mode. Press SHIFT-1 to change the color of

the empty pixels, and SHIFT-2 to alter the color of the on pixels. (You'll be prompted to press a color key after each SHIFT-1 or SHIFT-2 combination.)

Action!

If you want to try out your sprite in action, press J (for Joystick). You can now move the actual sprite around with the joystick. The speed of movement depends on the current cursor velocity. When you've finished putting your sprite through its paces, press the fire button to return to Sprite Magic. Also, if you want to test the animation while you are moving about, hold down the SHIFT key to step forward, or the Commodore key to step backward. You can lock the SHIFT key to keep the animation happening while you move around.

Saving Your Sprites

After all your work, you surely want to save your creations on tape or disk for future use. You can save an individual shape, or all the sprites. Press S (for Save), then either D (Disk) or T (Tape). Next, enter the filename. You'll be asked if you want to "Save all from here?" If you press N for No, only the current sprite you are working on is saved. If you press Y for Yes, every sprite from the current to sprite 255 will be saved. Thus, if you want to save a range of sprites, be sure to use the minus key to step back to the first sprite you want saved.

To recall your sprites, press L. The Load command loads everything that was saved. If you're loading in more than one sprite, be sure you step backward far enough with the minus key so that all the sprites will fit between the current sprite and sprite 255. The sprites load starting at the current sprite page number. After you press L, enter T or D for Tape or Disk.

Let There Be DATA

If you're a programmer, you're probably more interested in DATA statements. That way, you can use BASIC to READ and POKE the numbers into memory. If you have some kind of DATAmaker, you can run it on the memory used by the sprite in Sprite Magic (again, the memory location is the sprite number times 64). But Sprite Magic has a special DATAmaker of its own. It's similar to the Create DATA option in Ultrafont + , but it's been enhanced.

Press CTRL-D to create a series of DATA statements from the current sprite in memory. Just tap the key, or you'll get hundreds of DATA statements as the key repeats. Sprite Magic will create eight DATA statements, with eight bytes per line. The last byte is not strictly used. Sprite shapes are made from 63 bytes, but the sprite areas are

padded out so they will conveniently fall in 64-byte ranges. To create DATA statements for another sprite, use the + or – key to move to the correct sprite page, then press CTRL-D again.

If you have a program already in memory, the DATA statements are appended to the end of the program, starting with the next available line number. To add DATA statements to an existing program, then, first load Sprite Magic. Type NEW. Load your BASIC program, and SYS 49152 to enter Sprite Magic. You can then load in sprite shapes and use CTRL-D to add those DATA statements to the end of the BASIC program in memory.

You can check to see that these DATA statements were added by exiting Sprite Magic (press CTRL-X) and typing LIST. Your program should have eight new DATA lines for each sprite pattern. If there was no program in memory, the DATA statements form a program all their own, starting with line 1. If you want, you can save just the DATA statements to tape or disk, using the normal SAVE command.

To exit Sprite Magic and return to BASIC, press CTRL-X. You can also use RUN/STOP–RESTORE.

We're quite pleased that we can offer you such a powerful, easy-to-use utility. We also hope that it encourages more beginning programmers to learn about sprites. Now that you have a sprite editor, how about using it to write a fantastic game?

Quick Reference Chart: Sprite Magic

B:	Cycle through background colors
F:	Flip sprite upside down
J:	Move sprite with joystick. Press button when done.
L:	Load sprites from tape or disk
M:	Mirror sprite from left to right
S:	Save sprite(s) to tape or disk
V:	Set cursor velocity
X:	Toggle X expansion on/off
Y:	Toggle Y expansion on/off
CTRL-D:	Create DATA statements
CTRL-R or CTRL-9:	Reverse sprite
CTRL-X:	Exit to BASIC
+	Next sprite page
–	Previous sprite page
CLR/HOME:	Home cursor
SHIFT–CLR/HOME:	Erase grid
Space bar or fire button:	Set/clear points
CRSR keys or joystick in port 2:	Move cursor
Back arrow:	Symmetry mode (only in normal mode)
Keys 1–4:	Select drawing color for multicolor mode
SHIFT 1–4:	Change a drawing color
f1:	Shift right
f2:	Shift left
f3:	Shift down
f4:	Shift up
f5:	Multicolor mode
f6:	Normal mode
f7:	Store sprite to buffer
f8:	Recall sprite from buffer

Sprite Magic

Be sure to read "Using the Machine Language Editor: MLX," Appendix D, before typing in this program.

```
49152 :076,032,195,000,001,003,051
49158 :004,032,184,192,169,004,079
49164 :133,252,169,000,133,251,182
49170 :133,167,169,216,133,168,236
49176 :169,021,141,040,002,169,054
49182 :003,141,041,002,160,000,121
49188 :177,253,170,173,048,002,091
49194 :240,003,076,138,192,169,092
49200 :207,145,251,138,010,170,201
49206 :176,008,173,003,192,145,239
49212 :167,076,069,192,173,004,229
49218 :192,145,167,200,192,008,202
49224 :208,221,024,165,251,105,022
49230 :008,133,251,133,167,165,167
49236 :252,105,000,133,252,105,163
49242 :212,133,168,230,253,208,014
49248 :002,230,254,206,041,002,063
49254 :173,041,002,208,183,024,221
49260 :165,251,105,016,133,251,005
49266 :133,167,165,252,105,000,168
49272 :133,252,105,212,133,168,099
49278 :206,040,002,173,040,002,077
49284 :240,003,076,029,192,096,000
49290 :134,097,169,000,141,042,209
49296 :002,006,097,046,042,002,083
49302 :006,097,046,042,002,174,005
49308 :042,002,169,207,145,251,204
49314 :200,169,247,145,251,136,030
49320 :189,003,192,145,167,200,040
49326 :145,167,200,192,008,208,070
49332 :215,076,074,192,169,000,138
49338 :133,254,173,043,002,133,156
49344 :253,006,253,038,254,006,234
49350 :253,038,254,006,253,038,016
49356 :254,006,253,038,254,006,247
49362 :253,038,254,006,253,038,028
49368 :254,096,032,184,192,160,110
49374 :000,177,253,073,255,145,101
49380 :253,200,192,064,208,245,110
49386 :096,032,184,192,160,062,192
49392 :136,136,177,253,010,008,192
49398 :200,200,162,003,177,253,217
49404 :040,042,008,145,253,136,108
49410 :202,208,245,040,192,255,120
49416 :208,230,096,032,184,192,182
49422 :160,000,200,200,177,253,236
```

```
49428 :074,008,136,136,162,003,027
49434 :177,253,040,106,008,145,243
49440 :253,200,202,208,245,040,156
49446 :192,063,208,230,096,032,091
49452 :184,192,160,000,177,253,242
49458 :153,173,202,200,192,003,205
49464 :208,246,177,253,136,136,188
49470 :136,145,253,200,200,200,172
49476 :200,192,063,208,241,162,110
49482 :000,160,060,189,173,202,090
49488 :145,253,200,232,224,003,113
49494 :208,245,096,032,184,192,019
49500 :160,060,162,000,177,253,136
49506 :157,173,202,200,232,224,006
49512 :003,208,245,160,060,177,189
49518 :253,200,200,200,145,253,081
49524 :136,136,136,136,016,243,151
49530 :160,000,185,173,202,145,219
49536 :253,200,192,003,208,246,206
49542 :096,032,184,192,160,000,030
49548 :152,170,232,232,169,003,074
49554 :133,097,169,008,141,055,237
49560 :002,177,253,074,145,253,032
49566 :062,173,202,206,055,002,090
49572 :173,055,002,208,240,200,018
49578 :202,198,097,165,097,208,113
49584 :227,192,063,144,215,160,153
49590 :000,185,173,202,145,253,116
49596 :200,192,063,208,246,096,169
49602 :169,147,032,210,255,173,156
49608 :000,220,133,097,041,015,194
49614 :073,015,170,173,000,208,077
49620 :024,125,066,194,141,000,250
49626 :208,173,016,208,125,077,001
49632 :194,141,016,208,173,001,189
49638 :208,024,125,088,194,141,242
49644 :001,208,032,018,195,173,095
49650 :141,002,041,001,024,109,048
49656 :248,007,141,248,007,173,048
49662 :141,002,041,002,074,073,075
49668 :255,056,109,248,007,141,052
49674 :248,007,165,097,041,016,072
49680 :208,181,173,000,220,041,071
49686 :016,240,249,173,043,002,233
49692 :141,248,007,032,059,196,199
49698 :169,255,141,000,208,169,208
49704 :000,141,016,208,169,128,190
49710 :141,001,208,076,177,194,075
49716 :032,184,192,160,000,152,004
```

```
49722 :145,253,200,192,063,208,095
49728 :249,096,000,000,000,000,153
49734 :255,255,255,000,001,001,069
49740 :001,000,000,000,000,255,076
49746 :255,255,000,000,000,000,080
49752 :000,255,001,000,000,255,087
49758 :001,000,000,255,001,018,113
49764 :083,080,082,073,084,069,059
49770 :032,077,065,071,073,067,235
49776 :146,095,069,082,082,079,153
49782 :082,032,079,078,032,083,248
49788 :065,086,069,047,076,079,034
49794 :065,068,095,018,084,146,094
49800 :065,080,069,032,079,082,031
49806 :032,018,068,146,073,083,050
49812 :075,063,095,070,073,076,088
49818 :069,078,065,077,069,058,058
49824 :095,069,078,084,069,082,125
49830 :032,067,079,076,079,082,069
49836 :032,075,069,089,095,169,189
49842 :099,160,194,133,251,132,123
49848 :252,160,040,169,032,153,222
49854 :191,007,136,208,250,177,135
49860 :251,200,201,095,208,249,120
49866 :136,132,097,152,074,073,098
49872 :255,056,105,020,168,162,206
49878 :024,024,032,240,255,169,190
49884 :146,032,210,255,160,000,255
49890 :177,251,032,210,255,200,071
49896 :196,097,144,246,096,133,120
49902 :251,132,252,160,040,169,218
49908 :032,153,191,007,136,208,203
49914 :250,162,024,160,000,024,102
49920 :032,240,255,160,000,177,096
49926 :251,201,095,240,006,032,063
49932 :210,255,200,208,244,096,201
49938 :174,053,002,240,008,160,143
49944 :000,200,208,253,202,208,071
49950 :250,096,169,147,032,210,166
49956 :255,169,000,141,134,002,225
49962 :141,056,002,169,008,032,194
49968 :210,255,169,128,141,138,065
49974 :002,169,048,141,053,002,213
49980 :169,255,141,043,002,169,071
49986 :000,141,048,002,173,006,180
49992 :192,141,038,208,173,004,060
49998 :192,141,037,208,141,039,068
50004 :208,032,007,192,169,255,179
50010 :141,000,208,169,128,141,109
```

```
50016 :001,208,173,043,002,141,152
50022 :248,007,169,001,141,021,177
50028 :208,169,000,141,028,208,094
50034 :169,012,141,033,208,141,050
50040 :032,208,141,044,002,141,176
50046 :045,002,032,177,194,032,096
50052 :059,196,032,007,192,032,138
50058 :030,196,173,000,220,072,061
50064 :041,015,073,015,141,046,219
50070 :002,104,041,016,141,047,245
50076 :002,032,228,255,240,006,151
50082 :032,208,196,076,134,195,235
50088 :032,018,195,173,047,002,123
50094 :208,003,032,089,196,032,222
50100 :030,196,173,047,002,073,189
50106 :016,141,052,002,173,046,104
50112 :002,240,195,174,046,002,083
50118 :189,066,194,172,048,002,101
50124 :240,001,010,024,109,044,120
50130 :002,141,044,002,024,173,084
50136 :045,002,125,088,194,141,043
50142 :045,002,174,044,002,016,249
50148 :017,162,000,142,044,002,083
50154 :162,023,173,048,002,240,114
50160 :002,162,022,142,044,002,102
50166 :174,044,002,224,024,144,090
50172 :005,162,000,142,044,002,095
50178 :172,045,002,016,005,160,146
50184 :020,140,045,002,172,045,176
50190 :002,192,021,144,005,160,026
50196 :000,140,045,002,032,030,013
50202 :196,076,134,195,174,045,078
50208 :002,172,044,002,032,240,012
50214 :255,164,211,173,048,002,123
50220 :208,005,169,032,145,209,044
50226 :096,169,032,145,209,200,133
50232 :145,209,096,162,000,160,060
50238 :030,024,032,240,255,169,044
50244 :018,032,210,255,174,043,032
50250 :002,142,248,007,169,000,130
50256 :032,205,189,169,032,032,227
50262 :210,255,096,032,184,192,031
50268 :173,045,002,010,109,045,220
50274 :002,133,097,173,044,002,037
50280 :074,074,074,024,101,097,036
50286 :168,173,044,002,041,007,033
50292 :073,007,170,232,134,097,061
50298 :056,169,000,042,202,208,031
50304 :252,174,048,002,208,047,091
```

```
50310 :133,097,173,052,002,208,031
50316 :016,169,000,141,049,002,005
50322 :177,253,037,097,208,005,155
50328 :169,001,141,049,002,165,167
50334 :097,073,255,049,253,174,035
50340 :049,002,240,002,005,097,047
50346 :145,253,173,056,002,240,015
50352 :003,032,000,202,096,133,130
50358 :098,074,005,098,073,255,017
50364 :049,253,166,097,202,133,064
50370 :097,173,051,002,074,042,121
50376 :202,208,252,005,097,145,085
50382 :253,096,141,050,002,174,154
50388 :236,196,221,236,196,240,001
50394 :004,202,208,248,096,202,154
50400 :138,010,170,189,021,197,181
50406 :072,189,020,197,072,096,108
50412 :039,133,137,134,138,077,126
50418 :074,147,018,145,017,157,032
50424 :029,135,139,049,050,051,189
50430 :052,019,136,140,033,034,156
50436 :035,036,086,083,076,024,088
50442 :088,089,066,032,160,043,232
50448 :045,004,095,070,010,193,177
50454 :234,192,088,193,042,193,196
50460 :134,193,193,193,051,194,218
50466 :217,192,097,197,107,197,017
50472 :113,197,127,197,161,197,008
50478 :214,197,232,197,232,197,035
50484 :232,197,232,197,249,197,076
50490 :004,198,032,198,064,198,240
50496 :064,198,064,198,064,198,082
50502 :144,198,254,199,165,200,206
50508 :188,200,143,197,152,197,129
50514 :103,197,088,196,088,196,182
50520 :202,198,216,198,035,201,114
50526 :051,202,060,202,206,045,092
50532 :002,076,139,197,238,033,017
50538 :208,096,238,045,002,076,003
50544 :139,197,206,044,002,173,105
50550 :048,002,240,017,206,044,163
50556 :002,076,139,197,238,044,052
50562 :002,173,048,002,240,003,086
50568 :238,044,002,104,104,076,192
50574 :224,195,173,029,208,073,020
50580 :001,141,029,208,096,173,028
50586 :023,208,073,001,141,023,111
50592 :208,096,169,016,141,048,070
50598 :002,169,001,141,028,208,203
```

```
50604 :032,007,192,162,001,142,196
50610 :051,002,189,003,192,141,244
50616 :032,208,173,004,192,141,166
50622 :037,208,173,005,192,141,178
50628 :039,208,173,006,192,141,187
50634 :038,208,173,044,002,041,196
50640 :254,141,044,002,076,139,096
50646 :197,169,000,141,048,002,003
50652 :141,032,208,141,028,208,210
50658 :173,004,192,141,039,208,215
50664 :096,056,173,050,002,233,074
50670 :049,141,051,002,170,189,072
50676 :003,192,141,032,208,096,148
50682 :169,000,141,044,002,141,235
50688 :045,002,076,139,197,032,235
50694 :218,192,032,007,192,032,167
50700 :218,192,032,007,192,032,173
50706 :184,192,160,000,177,253,216
50712 :153,109,202,200,192,064,176
50718 :208,246,096,032,184,192,220
50724 :160,000,185,109,202,145,069
50730 :253,200,192,064,208,246,181
50736 :096,144,005,028,159,156,124
50742 :030,031,158,129,149,150,189
50748 :151,152,153,154,155,169,226
50754 :161,160,194,032,181,194,220
50760 :032,103,202,162,000,221,024
50766 :049,198,240,008,232,224,005
50772 :016,208,246,076,177,194,233
50778 :056,173,050,002,233,033,125
50784 :168,138,153,003,192,173,155
50790 :048,002,208,009,173,004,034
50796 :192,141,039,208,076,133,129
50802 :198,173,004,192,141,037,091
50808 :208,173,005,192,141,039,110
50814 :208,173,006,192,141,038,116
50820 :208,174,051,002,189,003,247
50826 :192,141,032,208,076,177,196
50832 :194,169,180,160,198,032,053
50838 :181,194,032,228,255,056,072
50844 :233,048,048,248,201,010,176
50850 :176,244,133,097,056,169,013
50856 :009,229,097,010,010,010,021
50862 :141,053,002,076,177,194,049
50868 :067,085,082,083,079,082,146
50874 :032,086,069,076,079,067,083
50880 :073,084,089,032,040,048,046
50886 :045,057,041,063,095,173,160
50892 :043,002,201,255,240,006,183
```

```
50898 :238,043,002,032,059,196,012
50904 :096,206,043,002,032,184,011
50910 :192,165,046,197,254,144,196
50916 :004,238,043,002,096,032,131
50922 :059,196,096,160,000,140,117
50928 :055,002,169,164,032,210,104
50934 :255,169,157,032,210,255,044
50940 :032,103,202,172,055,002,050
50946 :133,097,169,032,032,210,163
50952 :255,169,157,032,210,255,062
50958 :165,097,201,013,240,043,005
50964 :201,020,208,013,192,000,142
50970 :240,211,136,169,157,032,203
50976 :210,255,076,239,198,041,027
50982 :127,201,032,144,196,192,162
50988 :020,240,192,165,097,153,143
50994 :000,002,032,210,255,169,206
51000 :000,133,212,200,076,239,148
51006 :198,169,095,153,000,002,167
51012 :152,096,032,231,255,169,235
51018 :133,160,194,032,181,194,200
51024 :032,103,202,162,001,201,013
51030 :084,240,011,162,008,201,024
51036 :068,240,005,104,104,076,177
51042 :177,194,141,054,002,160,058
51048 :000,169,001,032,186,255,235
51054 :169,151,160,194,032,237,029
51060 :194,032,237,198,208,007,224
51066 :173,054,002,201,084,208,076
51072 :237,173,054,002,201,068,095
51078 :208,066,169,064,141,020,034
51084 :002,169,048,141,021,002,011
51090 :169,058,141,022,002,160,186
51096 :000,185,000,002,153,023,003
51102 :002,200,204,055,002,208,061
51108 :244,169,044,153,023,002,031
51114 :169,080,153,024,002,173,003
51120 :050,002,201,083,208,012,220
51126 :169,044,153,025,002,169,232
51132 :087,153,026,002,200,200,088
51138 :200,200,200,200,200,076,246
51144 :216,199,160,000,185,000,192
51150 :002,153,020,002,200,204,019
51156 :055,002,208,244,152,162,011
51162 :020,160,002,032,189,255,108
51168 :169,160,133,178,096,083,019
51174 :065,086,069,032,065,076,111
51180 :076,032,070,082,079,077,140
51186 :032,072,069,082,069,063,117
```

```
51192 :032,040,089,047,078,041,063
51198 :095,032,070,199,032,184,098
51204 :192,169,229,160,199,032,217
51210 :181,194,032,103,202,201,155
51216 :089,208,007,162,000,160,130
51222 :064,076,037,200,024,165,076
51228 :253,105,064,170,165,254,015
51234 :105,000,168,165,253,133,090
51240 :251,165,254,133,252,032,103
51246 :195,200,169,251,032,216,085
51252 :255,176,011,032,183,255,196
51258 :208,006,032,205,200,076,017
51264 :177,194,032,205,200,032,136
51270 :231,255,173,054,002,201,218
51276 :068,240,013,169,114,160,072
51282 :194,032,181,194,032,103,050
51288 :202,076,177,194,169,000,138
51294 :032,189,255,169,015,162,148
51300 :008,160,015,032,186,255,244
51306 :032,192,255,162,015,032,026
51312 :198,255,160,000,032,207,196
51318 :255,201,013,240,007,153,219
51324 :000,002,200,076,116,200,206
51330 :169,095,153,000,002,032,069
51336 :204,255,169,000,160,002,158
51342 :032,181,194,162,015,032,246
51348 :201,255,169,073,032,210,064
51354 :255,169,013,032,210,255,064
51360 :032,231,255,076,086,200,016
51366 :032,070,199,032,195,200,126
51372 :032,184,192,169,000,166,147
51378 :253,164,254,032,213,255,069
51384 :176,136,076,205,200,169,122
51390 :004,141,136,002,000,169,130
51396 :000,141,021,208,169,147,114
51402 :076,210,255,169,001,141,030
51408 :021,208,169,147,032,210,227
51414 :255,032,059,196,032,007,027
51420 :192,076,177,194,248,169,252
51426 :000,141,000,001,141,001,254
51432 :001,224,000,240,021,202,152
51438 :024,173,000,001,105,001,030
51444 :141,000,001,173,001,001,049
51450 :105,000,141,001,001,076,062
51456 :233,200,216,173,001,001,056
51462 :009,048,141,002,001,173,124
51468 :000,001,041,240,074,074,186
51474 :074,074,009,048,141,001,109
51480 :001,173,000,001,041,015,255
```

```
51486 :009,048,141,000,001,096,069
51492 :056,165,045,233,002,133,158
51498 :045,165,046,233,000,133,152
51504 :046,169,001,133,097,169,151
51510 :008,133,098,169,000,133,083
51516 :057,133,058,160,000,177,133
51522 :097,200,017,097,240,027,232
51528 :160,002,177,097,133,057,186
51534 :200,177,097,133,058,160,135
51540 :000,177,097,072,200,177,039
51546 :097,133,098,104,133,097,240
51552 :076,063,201,024,165,057,170
51558 :105,001,133,057,165,058,109
51564 :105,000,133,058,032,184,108
51570 :192,160,000,132,098,160,088
51576 :000,024,165,045,105,037,240
51582 :145,045,200,165,046,105,064
51588 :000,145;045,200,165,057,232
51594 :145,045,200,165,058,145,128
51600 :045,200,169,131,145,045,111
51606 :200,132,097,164,098,132,205
51612 :098,177,253,170,032,224,086
51618 :200,164,097,173,002,001,031
51624 :145,045,173,001,001,200,221
51630 :145,045,173,000,001,200,226
51636 :145,045,200,169,044,145,160
51642 :045,200,132,097,164,098,154
51648 :200,152,041,007,208,213,245
51654 :132,098,164,097,136,169,226
51660 :000,145,045,160,000,177,219
51666 :045,072,200,177,045,133,114
51672 :046,104,133,045,230,057,063
51678 :208,002,230,058,164,098,214
51684 :192,064,208,143,160,000,227
51690 :152,145,045,200,145,045,198
51696 :024,165,045,105,002,133,202
51702 :045,165,046,105,000,133,228
51708 :046,076,094,166,032,135,033
51714 :193,173,045,002,010,109,022
51720 :045,002,168,162,000,185,058
51726 :173,202,157,237,202,200,161
51732 :232,224,003,208,244,032,195
51738 :135,193,173,045,002,010,072
51744 :109,045,002,168,162,000,006
51750 :177,253,029,237,202,145,057
51756 :253,200,232,224,003,208,140
51762 :243,096,173,056,002,073,181
51768 :001,141,056,002,096,032,128
51774 :184,192,160,000,162,060,052
```

```
51780 :169,003,133,097,177,253,132
51786 :157,173,202,200,232,198,212
51792 :097,165,097,208,243,138,004
51798 :056,233,006,170,016,232,031
51804 :160,062,185,173,202,145,251
51810 :253,136,016,248,096,032,111
51816 :228,255,240,251,096,013,163
```

Chris Metcalf

The Graphics Package

Creating graphics with machine language routines is one of the most effective ways to use ML. Machine language is fast, just the thing for displaying complex patterns on the screen, especially when you're using the high-resolution mode on the 64. These four short routines, combined or used singly, make graphics creation quick and simple.

Working in the Commodore's high-resolution mode can be rewarding, but it can also be frustrating. If you're programming exclusively in BASIC, plotting points, drawing lines, filling in areas, or even doing simple things such as clearing the screen and selecting graphics modes can be hard work. Worst of all, the graphics are drawn on the screen with excruciating slowness.

Fortunately, machine language routines can overcome this lack of speed. Each of the four programs in "The Graphics Package" can create dazzling displays at machine language speed with a single SYS command. And because they're listed in a BASIC loader form, you won't have any trouble adding them to your own programs.

Using these four routines, you'll be able to plot a point on the screen, draw a line, fill an area, set graphics or text mode, clear the screen, or set the high-resolution display color. The point-plotting routine is the only one that can operate alone; the others are independent of each other, but each requires that the plot routine be present in memory. To call one of the routines, you don't need to POKE values into memory locations. Instead, you simply call the routine with a SYS command and follow it with the appropriate parameters. For example:

```
SYS(49152),147,83,1
```

turns on (plots) the pixel at x-position 147 and y-position 83. The parameters for the other routines are set in the same way. Each routine explains its parameters and values allowed.

In or Out

All of these SYS commands can be used in direct mode (without line numbers) or called from within your own program, as long as the routines have been loaded into your 64's memory. In a program, treat them as normal BASIC commands. In other words, use a colon to

separate the different commands. For instance,

```
FOR I = 0 TO 319:SYS(49152),I,I/319 * 199,1,2:NEXT I
```

draws a line from the upper left-hand corner of the screen to the lower right-hand corner. Although this is an example of the plot routine, you end up with a line because of the way you're turning high-resolution pixels on. Notice, too, the complexity of the second numeric expression in the above example. In fact, *any* expression that creates a number can be used in place of an integer.

One warning when you use these routines: Make sure you don't press the RUN/STOP–RESTORE keys while they are actually processing. Since the routines that normally control the computer are absent while these graphics utilities run, attempting to return control to them can be disastrous. Since the plot and line-drawing routines operate quickly, and since the fill routine includes a check for the STOP key, this has been minimized as much as possible. To halt a fill, just press the STOP key.

Applications, Not Tutorials

Since this package of utilities is meant to *use*, and not actually to show you how to create high-resolution graphics from scratch, it's best to go elsewhere if you're not familiar with programming graphics in high-resolution mode. Two excellent references are *COMPUTE!'s Reference Guide to Commodore 64 Graphics*, by John Heilborn, and *COMPUTE!'s First Book of Commodore 64 Sound and Graphics*, which includes a number of articles showing you the basics of high-resolution graphics programming. Of course, if you're already comfortable with this feature of the Commodore 64, you can use these routines immediately. And even if you aren't an expert in high-resolution graphics, you can type in the routines and the demonstration program, just to see how it all works. After examining the demonstration, you can easily change some of the parameters to see new designs.

Type In and Use

To use these routines, all you need to do is type them in. Make sure you read Appendix C, "The Automatic Proofreader," before you do this. It will insure mistake-proof entry. Each of the routines also includes a final checksum of the DATA statements entered. This checksum will tell you if you have the numbers entered correctly. Before you run any of these routines for the first time, SAVE them to tape or disk. A single error can cause the computer to crash, forcing you to turn it off, then on again, to regain control. Of course, this

will erase all of your work unless it was previously saved.

You can add these routines to your own programs easily, just by altering the first three line numbers to fit into your program. Once the data has been POKEd into the appropriate memory locations, you can access the routine. If you want, you can use this graphics package by itself, and even see a demonstration of its capabilities. Load and RUN each of the routines, one at a time; then type in and RUN Program 5, "Package Demonstration," to see the routines operate.

Most of the time, you'll be using all these routines together. To make it easier to enter the high-resolution (bitmapping) mode, use Program 4, "Utilities," whenever you use any of the first three routines. Program 1, "Point Plotting," is the only routine which can stand on its own. The others — Program 2, "Line Drawing," and Program 3, "Area Fill" — need to have the point-plotting routine in memory in order to work.

Each of the routines in Graphics Package is explained in more detail below.

The Point-Plot Routine

The first, and most crucial, routine is responsible for plotting individual points. Once this routine has been loaded, you can plot a point on the screen by SYSing location 49152, and specifying parameters in the form:

SYS(49152)*x,y,m*

The first number after the SYS location is the x-position of the point to be plotted, and the second number is the y-position. On the Commodore 64, the high-resolution (bitmap) screen is 320 pixels (picture elements, or dots on the screen) wide and 200 pixels high. The location 0,0 corresponds to the upper left-hand corner, and 319,199 specifies the lower right-hand corner. Anything outside this range will return an ?ILLEGAL QUANTITY ERROR message.

The third number (designated *m* above) sets the *mode* of the plotting routine. There are three possibilities for this setting:

0 Erase the pixel at the x,y coordinate given.
1 Turn on the pixel (plot the point).
2 Do an exclusive OR on the point. This term, which comes from logic and machine language programming, means that we blank the point if it's on, and plot it if it's not. That way, if we plot the point twice with mode 2, it will return to its original state.

A fourth number can be specified: an optional color. The number can be from 0 to 15, corresponding to the Commodore 64's

color numbers. If specified, the routine will plot the point in the given color. Otherwise, the point will appear in the pixel color already assigned to that area (see the COLOR command in Program 4, "Utilities," for a way of setting the pixel colors on the entire screen). A few examples of point plotting:

SYS(49152),12,13,1,1 plots a point in white at coordinate 12, 13.
SYS(49152),319,199,0 erases the point at 319,199 (no color).
SYS(49152),160,100,2,11 reverses (exclusive OR) the point at 160,100 and plots dark gray (color 11).
SYS(49152),0,0,1 plots a point in the local color at 0,0.

Remember that to use this point-plotting routine, you must be in the high-resolution mode. If you remain in the normal (text) mode, these commands will do nothing. The easiest way to enter the high-resolution mode is to have placed Program 4, Utilities, in memory. Then you can set the screen for high-resolution graphics with a single SYS. Read the explanation of the *H* command in the Utilities description.

There is one difficulty with the use of color in the pure high-resolution mode that I've been describing. The colors (foreground or pixels, and background) are defined not for each pixel, but for each group of 8 × 8 pixels (equivalent to one character in normal text mode). Therefore, plotting two pixels close together in different colors cannot be done, unless they are in different 8 × 8 blocks.

Multicolor Plotting. There is a solution to this problem. A special multicolor mode exists on the Commodore 64 which allows up to four different colors, including the background, to occupy one 8 × 8 area. Unfortunately, only 160 separate dots can be plotted horizontally, not 320. This decreases resolution. When in multicolor mode, therefore, another number has to be included, to specify which of the three foreground colors you wish to use. This parameter, which I call the *brush*, follows the mode number when in multicolor. (Enter the multicolor mode by typing SYS(50400),m when Program 4, Utilities, is in memory.) The brush is required, as is the mode, and uses a value from 1 to 3. Following the brush number is the color, if you wish to specify it. Otherwise, it will plot a pixel with the specified brush but in the previously defined color in that 8 × 8 square. Here are a few more examples of multicolor plotting:

SYS(49152),0,0,1,1,3 will plot a pixel in brush 1 in the color cyan at coordinate 0,0.
SYS(49152),160,100,2,3,7 reverses the pixel at 160,100 (the center of the screen) with brush 3 and sets the color to yellow).
SYS(49152),319,0,0 erases the pixel at the top right.

(Note that in multicolor mode an erase command can take only three parameters: x-position, y-position, and the zero indicating erase. Brush and color cannot apply, so they must be left out.)

The decrease in resolution has an effect on the plotting of multicolor points. The full range of 0–319 is still legitimate, but since there are in fact only 160 points horizontally, even-numbered x-positions will generate the same point as the next higher x-position. For example, SYS(49152),136,43,1,2 and SYS(49152),137,43,1,2 plot the same point.

The Line-Draw Routine

This routine is a logical extension of the point-plotting routine and obeys all the rules specified above. However, two points must be specified after the SYS, so that the line can have a starting and ending point. The location of the routine is 49600, so a SYS(49600) followed by two pairs of x,y coordinates, the mode, the brush (if in multicolor), and the color, if desired, will draw a line anywhere on the screen. As with the point-plotting routine, the line-drawing routine will draw lines both in normal high-resolution mode and in multicolor mode. Here are a few examples to show you the format:

SYS(49600),0,0,319,199,1 draws a line from corner to corner, in whatever colors lie beneath the pixels in each square.

SYS(49600),160,100,160 + COS(I)* 100,100 + SIN(I)* 100,1,4 draws a line from the center of the screen to a point on a circle with a radius of 100 at I radians, in the color purple.

SYS(49600),0,0,319,0,2,1 plots a white line across the top of the screen. Execute this command again and the line disappears.

In multicolor:

SYS(49600),0,0,319,199,1,2,3 draws a line corner to corner in brush 2, color cyan (3).

SYS(49600),33,75,108,9,2,3 exclusive ORs the line with brush 3, but does not change the color.

Note that the easiest way to enter either normal high-resolution mode or multicolor mode is to use the commands available in Program 4, Utilities.

The Area-Fill Routine

This routine will fill an area within a certain border with the specified pixel type and, if designated, a particular color. The fill routine obeys the rules outlined in the point-plotting description. The fill routine

begins at 50000, so a SYS(50000),*x,y,mode,brush,color* call begins a fill starting at coordinates x,y. The routine can fill any area that is not too complicated (and *too* complicated is a very high ceiling for this routine); typically a border of some sort is set up first, then the routine is called to fill it in. The fill routine is designed to slip through any hole between two horizontal or two vertical pixels; diagonally separated pixels (for instance, 45,89 and 44,90) act as an effective border. The edge of the screen also acts as a border if no pixel border is specified, or if the fill routine slips between two pixels. The routine can be called with mode set either to fill an empty area with pixels (mode 1) or to erase an area already covered with pixels (mode 0). Mode 2 is not allowed: It's a little vague what might constitute a border for such a fill. Two straightforward examples:

SYS(50000),160,100,1,6 fills starting at coordinates 160,100, coloring blue as the fill progresses.

SYS(50000),0,0,0 erases an area starting at the top-left corner of the screen.

Multicolor works as it does with plot and line, but the concept of a border can be changed with three separate types of plotting. Normally, this routine takes pixels of the mode it was called with as boundary pixels (off pixels with mode 0, on pixels with mode 1, for example, in normal high resolution). In multicolor the routine assumes that a boundary can only be of the same brush type that was specified in the SYS call. Any intervening pixels are simply covered over. However, it is possible to override the program's assumption about the border pixels. If the first punctuation following the SYS call is a *semicolon* rather than a comma, the routine stops at any pixel that is on, rather than stopping only at those of its own brush type. A good example of the two types of fill can be found in the last example in the accompanying program, "Package Demonstration." In multicolor, SYS(50000), 160,100,1,2 will fill any area enclosed by brush-2 pixels, disregarding what's inside. By contrast, SYS(50000); 160,100,1,2 will fill an area enclosed by any on pixels, regardless of the type, and not overwrite anything. The uses of these two options can be seen in Program 5.

As with the other routines, enter high-resolution mode, or multicolor mode, by using the SYS(50400),H or SYS(50400),M commands once Program 4 has been placed in memory.

The Utility Routines

The last module, beginning at location 50400, contains a variety of short but useful routines. To access one, call the routine using SYS(50400), followed by a comma and the first letter of the command, and then any necessary parameters. The commands, the parameters, and their descriptions follow:

High-Resolution Mode. SYS(50400),H sets the screen for high-resolution graphics. Note, however, that this operation does *not* clear the high-resolution screen, nor does it set the colors on the screen. See WIPE and COLOR, below, for those.

Multicolor Mode. SYS(50400),M sets the screen for multicolor graphics. Like the above command, however, that's all it does.

Text Mode. SYS(50400),T returns the screen to normal text mode. Notice, however, that multicolor plotting with brush 3 will actually change the character colors on the text screen.

Wipe. SYS(50400),W removes all the pixels from the screen. In normal high resolution this does *not* mean that the screen will necessarily be blank. The background colors are individually changeable in each square; at power-up, they can be anything. Use the COLOR command to restore the background colors to what you want.

Color. There are two distinct modes for this command, one for normal high resolution and one for multicolor. For high resolution, the command syntax is SYS(50400),*C,border,background,pixel color*. The border color corresponds to location 53280; the background sets location 53281 and, square by square, the bitmap color area. The pixel color is the color that any command without color specified will plot in. So if you enter SYS(50400),C,0,0,5, any commands which don't specify color will show up in green on black. The second syntax of this command, for multicolor, is SYS(50400),*C,border, background,col1,col2,col3*. Border and background (53280 and 53281) are as above; col1, col2, and col3 establish the defaults for the three brushes, brush 1, brush 2, and brush 3. However, brush-3 colors are sensitive to any PRINT statements, since PRINT puts color information in the area used for brush 3's color. Scrolling or clearing the screen also alters or destroys brush-3 color information. The COLOR command does not just set up default colors passively; it can also be used to alter the colors of things already plotted, both in multicolor and standard modes. See the examples of COLOR in the demonstration program.

Showing Off

To see the complete Graphics Package in action, first load and run all four routines. Now they're in the computer's memory. Type NEW, then load Program 5. Enter RUN and watch it all work. Pressing any key moves to the next screen display. And since the demonstration program is all in BASIC, you won't find it hard to change the SYS commands' parameters and alter the graphics.

Program 1. Point Plotting

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

```
10 FOR X=49152 TO 49601:READ A:POKE X,A:CK=CK+A:NE
   XT                                          :rem 198
20 IF CK<>53595 THEN PRINT "ERROR IN DATA" :rem 63
30 END                                          :rem 58
49152 DATA 032,020,192,032,056,192            :rem 138
49158 DATA 120,169,053,133,001,032            :rem 137
49164 DATA 132,192,169,055,133,001            :rem 146
49170 DATA 088,096,032,110,192,240            :rem 147
49176 DATA 008,192,002,176,022,224            :rem 145
49182 DATA 064,176,018,134,251,132            :rem 149
49188 DATA 252,032,110,192,208,009            :rem 147
49194 DATA 224,200,176,005,134,253            :rem 144
49200 DATA 132,254,096,162,014,076            :rem 144
49206 DATA 055,164,169,255,133,002            :rem 149
49212 DATA 173,022,208,041,016,133            :rem 132
49218 DATA 005,032,110,192,134,006            :rem 132
49224 DATA 224,000,240,019,165,005            :rem 132
49230 DATA 240,015,032,110,192,138            :rem 131
49236 DATA 041,003,208,002,169,003            :rem 133
49242 DATA 010,010,010,133,005,032            :rem 111
49248 DATA 121,000,208,001,096,032            :rem 132
49254 DATA 110,192,138,041,015,133            :rem 138
49260 DATA 002,096,032,115,000,032            :rem 125
49266 DATA 158,173,165,013,240,004            :rem 148
49272 DATA 162,248,154,096,032,247            :rem 160
49278 DATA 183,166,020,164,021,096            :rem 156
49284 DATA 032,246,192,165,006,240            :rem 150
49290 DATA 023,201,002,240,011,177            :rem 127
49296 DATA 003,061,059,193,029,027            :rem 157
49302 DATA 193,076,167,192,177,003            :rem 158
49308 DATA 093,027,193,076,167,192            :rem 167
49314 DATA 177,003,061,059,193,145            :rem 153
49320 DATA 003,036,002,016,001,096            :rem 125
49326 DATA 165,253,074,074,074,168            :rem 164
49332 DATA 165,252,074,165,251,106            :rem 150
49338 DATA 074,074,024,121,091,193            :rem 152
49344 DATA 133,003,185,116,193,105            :rem 145
49350 DATA 000,133,004,160,000,165            :rem 121
```

```
49356 DATA 005,201,016,240,017,176          :rem 140
49362 DATA 024,165,002,010,010,010          :rem 117
49368 DATA 010,081,003,041,240,081          :rem 133
49374 DATA 003,145,003,096,177,003          :rem 146
49380 DATA 041,240,005,002,145,003          :rem 125
49386 DATA 096,165,004,009,016,133          :rem 154
49392 DATA 004,165,002,145,003,096          :rem 143
49398 DATA 165,253,074,074,074,170          :rem 166
49404 DATA 165,251,069,253,041,248          :rem 155
49410 DATA 069,253,024,125,141,193          :rem 146
49416 DATA 133,003,189,166,193,101          :rem 150
49422 DATA 252,133,004,165,251,041          :rem 136
49428 DATA 007,005,005,170,160,000          :rem 129
49434 DATA 096,128,064,032,016,008          :rem 150
49440 DATA 004,002,001,064,064,016          :rem 125
49446 DATA 016,004,004,001,001,128          :rem 125
49452 DATA 128,032,032,008,008,002          :rem 133
49458 DATA 002,192,192,048,048,012          :rem 153
49464 DATA 012,003,003,127,191,223          :rem 134
49470 DATA 239,247,251,253,254,063          :rem 159
49476 DATA 063,207,207,243,243,252          :rem 154
49482 DATA 252,063,063,207,207,243          :rem 151
49488 DATA 243,252,252,063,063,207          :rem 157
49494 DATA 207,243,243,252,252,000          :rem 145
49500 DATA 040,080,120,160,200,240          :rem 118
49506 DATA 024,064,104,144,184,224          :rem 145
49512 DATA 008,048,088,128,168,208          :rem 163
49518 DATA 248,032,072,112,152,192          :rem 149
49524 DATA 200,200,200,200,200,200          :rem 106
49530 DATA 200,201,201,201,201,201          :rem 108
49536 DATA 201,202,202,202,202,202          :rem 120
49542 DATA 202,202,203,203,203,203          :rem 122
49548 DATA 203,000,064,128,192,000          :rem 138
49554 DATA 064,128,192,000,064,128          :rem 151
49560 DATA 192,000,064,128,192,000          :rem 139
49566 DATA 064,128,192,000,064,128          :rem 154
49572 DATA 192,000,224,225,226,227          :rem 147
49578 DATA 229,230,231,232,234,235          :rem 153
49584 DATA 236,237,239,240,241,242          :rem 158
49590 DATA 244,245,246,247,249,250          :rem 165
49596 DATA 251,252,254,013,013,013          :rem 143
```

Program 2. Line Drawing

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

```
10 FOR X=49600 TO 49911:READ A:POKE X,A:CK=CK+A:NE
   XT                                       :rem 200
20 IF CK<>40651 THEN PRINT"ERROR IN DATA"   :rem 52
30 END                                      :rem 58
```

```
49600 DATA 032,020,192,160,003,185      :rem 132
49606 DATA 251,000,153,193,000,136      :rem 135
49612 DATA 016,247,032,020,192,160      :rem 138
49618 DATA 003,185,251,000,153,168      :rem 147
49624 DATA 000,136,016,247,032,056      :rem 141
49630 DATA 192,120,169,053,133,001      :rem 139
49636 DATA 160,015,185,097,000,153      :rem 150
49642 DATA 248,194,136,016,247,162      :rem 162
49648 DATA 002,181,168,056,245,193      :rem 163
49654 DATA 149,038,181,169,245,194      :rem 174
49660 DATA 149,039,169,001,149,108      :rem 161
49666 DATA 169,000,149,109,181,194      :rem 165
49672 DATA 213,169,144,027,208,006      :rem 154
49678 DATA 181,168,213,193,176,019      :rem 172
49684 DATA 169,255,149,108,149,109      :rem 176
49690 DATA 181,193,056,245,168,149      :rem 172
49696 DATA 038,181,194,245,169,149      :rem 180
49702 DATA 039,202,202,016,198,169      :rem 153
49708 DATA 000,133,104,133,106,133      :rem 131
49714 DATA 105,133,107,166,038,164      :rem 151
49720 DATA 039,208,014,228,040,176      :rem 149
49726 DATA 010,166,040,032,081,194      :rem 144
49732 DATA 133,104,076,090,194,032      :rem 148
49738 DATA 081,194,133,106,076,090      :rem 160
49744 DATA 194,132,099,152,074,134      :rem 163
49750 DATA 098,138,106,096,169,000      :rem 162
49756 DATA 133,102,133,103,165,193      :rem 147
49762 DATA 133,251,165,194,133,252      :rem 155
49768 DATA 165,195,133,253,165,098      :rem 177
49774 DATA 024,105,001,133,100,165      :rem 134
49780 DATA 099,105,000,133,101,165      :rem 143
49786 DATA 005,240,014,165,159,197      :rem 164
49792 DATA 253,208,008,165,251,041      :rem 154
49798 DATA 254,197,158,240,003,032      :rem 163
49804 DATA 132,192,165,251,041,254      :rem 149
49810 DATA 133,158,165,253,133,159      :rem 157
49816 DATA 162,002,181,104,024,117      :rem 139
49822 DATA 038,149,104,181,105,117      :rem 150
49828 DATA 039,149,105,197,099,240      :rem 174
49834 DATA 004,144,033,208,006,181      :rem 143
49840 DATA 104,197,098,144,025,181      :rem 160
49846 DATA 104,229,098,149,104,181      :rem 165
49852 DATA 105,229,099,149,105,181      :rem 165
49858 DATA 251,024,117,108,149,251      :rem 158
49864 DATA 181,252,117,109,149,252      :rem 162
49870 DATA 202,202,016,200,230,102      :rem 123
49876 DATA 208,002,230,103,165,103      :rem 141
49882 DATA 197,101,144,006,165,102      :rem 150
49888 DATA 197,100,176,003,076,121      :rem 159
```

```
49894 DATA 194,160,015,185,248,194          :rem 173
49900 DATA 153,097,000,136,016,247          :rem 147
49906 DATA 169,055,133,001,088,096          :rem 163
```

Program 3. Area Fill

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

```
10 FOR X=50000 TO 50353:READ A:POKE X,A:CK=CK+A:NE
   XT                                        :rem 178
20 IF CK<>44135 THEN PRINT "ERROR IN DATA" :rem 53
30 END                                        :rem 58
50000 DATA 169,001,133,165,032,121          :rem 120
50006 DATA 000,201,044,240,002,198          :rem 118
50012 DATA 165,032,020,192,032,056          :rem 125
50018 DATA 192,165,006,201,002,208          :rem 129
50024 DATA 005,162,014,076,055,164          :rem 134
50030 DATA 165,052,056,229,050,201          :rem 129
50036 DATA 004,176,003,076,053,164          :rem 137
50042 DATA 165,052,056,233,003,133          :rem 129
50048 DATA 033,169,000,133,031,133          :rem 127
50054 DATA 029,120,169,053,133,001          :rem 130
50060 DATA 169,007,133,164,169,001          :rem 139
50066 DATA 133,163,165,006,133,034          :rem 136
50072 DATA 173,022,208,041,016,240          :rem 127
50078 DATA 021,165,251,041,254,133          :rem 136
50084 DATA 251,198,164,230,163,165          :rem 151
50090 DATA 006,240,007,165,005,074          :rem 131
50096 DATA 074,074,133,034,169,000          :rem 142
50102 DATA 133,027,133,028,165,252          :rem 132
50108 DATA 208,004,165,251,240,029          :rem 135
50114 DATA 165,251,056,229,163,133          :rem 142
50120 DATA 251,165,252,233,000,133          :rem 122
50126 DATA 252,032,106,196,208,230          :rem 136
50132 DATA 165,251,024,101,163,133          :rem 126
50138 DATA 251,144,002,230,252,230          :rem 125
50144 DATA 253,032,106,196,240,013          :rem 132
50150 DATA 165,027,208,013,032,151          :rem 128
50156 DATA 196,169,001,133,027,208          :rem 146
50162 DATA 004,169,000,133,027,198          :rem 138
50168 DATA 253,198,253,032,106,196          :rem 156
50174 DATA 240,013,165,028,208,013          :rem 133
50180 DATA 032,151,196,169,001,133          :rem 136
50186 DATA 028,208,004,169,000,133          :rem 137
50192 DATA 028,230,253,032,132,192          :rem 135
50198 DATA 165,251,024,101,163,133          :rem 138
50204 DATA 251,144,002,230,252,165          :rem 126
50210 DATA 252,240,006,165,251,201          :rem 122
50216 DATA 064,176,005,032,106,196          :rem 141
50222 DATA 208,175,164,029,240,018          :rem 141
50228 DATA 169,055,133,001,032,234          :rem 135
```

```
50234 DATA 255,032,234,255,008,169          :rem 146
50240 DATA 053,133,001,040,208,006          :rem 117
50246 DATA 169,055,133,001,088,096          :rem 152
50252 DATA 165,033,133,032,136,177          :rem 139
50258 DATA 031,133,251,230,032,177          :rem 134
50264 DATA 031,133,252,230,032,177          :rem 132
50270 DATA 031,133,253,132,029,201          :rem 125
50276 DATA 200,176,201,076,180,195          :rem 146
50282 DATA 032,246,192,189,059,193          :rem 161
50288 DATA 073,255,049,003,072,138          :rem 152
50294 DATA 041,007,170,104,228,164          :rem 138
50300 DATA 176,006,074,232,228,164          :rem 139
50306 DATA 144,250,166,165,240,005          :rem 136
50312 DATA 197,034,076,150,196,201          :rem 143
50318 DATA 000,240,003,162,000,096          :rem 120
50324 DATA 162,001,096,164,029,165          :rem 143
50330 DATA 033,133,032,165,251,145          :rem 129
50336 DATA 031,230,032,165,252,145          :rem 132
50342 DATA 031,230,032,165,253,145          :rem 130
50348 DATA 031,230,029,096,013,013          :rem 133
```

Program 4. Utilities

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

```
10 FOR X=50400 TO 50621:READ A:POKE X,A:CK=CK+1:NE
   XT                                       :rem 164
20 IF CK<>222 THEN PRINT"ERROR IN DATA"     :rem 202
30 END                                       :rem 58
50400 DATA 032,115,000,240,029,160          :rem 115
50406 DATA 008,217,007,197,208,018          :rem 146
50412 DATA 185,016,197,141,252,196          :rem 151
50418 DATA 185,017,197,141,253,196          :rem 159
50424 DATA 032,115,000,076,255,255          :rem 134
50430 DATA 136,136,016,229,162,011          :rem 133
50436 DATA 076,055,164,072,044,077          :rem 153
50442 DATA 044,084,044,087,044,067          :rem 149
50448 DATA 026,197,056,197,065,197          :rem 172
50454 DATA 089,197,109,197,173,000          :rem 160
50460 DATA 221,041,252,141,000,221          :rem 115
50466 DATA 169,040,141,024,208,173          :rem 144
50472 DATA 017,208,009,032,141,017          :rem 134
50478 DATA 208,173,022,208,041,239          :rem 148
50484 DATA 141,022,208,096,032,026          :rem 139
50490 DATA 197,009,016,141,022,208          :rem 141
50496 DATA 096,173,000,221,009,003          :rem 137
50502 DATA 141,000,221,169,021,141          :rem 118
50508 DATA 024,208,173,017,208,041          :rem 138
50514 DATA 223,141,017,208,076,047          :rem 140
50520 DATA 197,169,000,168,132,003          :rem 139
```

```
50526 DATA 162,224,134,004,145,003          :rem 130
50532 DATA 200,208,251,232,224,000          :rem 120
50538 DATA 208,244,096,032,110,192          :rem 145
50544 DATA 142,032,208,032,110,192          :rem 129
50550 DATA 138,041,015,141,033,208          :rem 130
50556 DATA 133,252,032,110,192,138          :rem 138
50562 DATA 041,015,133,251,173,022          :rem 129
50568 DATA 208,041,016,240,019,032          :rem 137
50574 DATA 110,192,138,041,015,133          :rem 135
50580 DATA 252,032,110,192,169,216          :rem 141
50586 DATA 133,004,138,032,172,197          :rem 149
50592 DATA 169,200,133,004,165,251          :rem 140
50598 DATA 010,010,010,010,005,252          :rem 115
50604 DATA 160,000,132,003,162,004          :rem 114
50610 DATA 145,003,200,208,251,230          :rem 120
50616 DATA 004,202,208,246,096,013          :rem 137
```

Program 5. Package Demonstration

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

```
100 :                                       :rem 203
110 REM DEMOS FOR GRAPHICS SUBROUTINES      :rem 137
120 REM                                     :rem 119
130 :                                       :rem 206
140 SYS50400,H:SYS50400,C,0,0,1:GOSUB700      :rem 8
150 DATA "{WHT}SIMPLE FIGURE NUMBER 1"      :rem 127
160 DATA "HIT ANY KEY AFTER THIS DESIGN, AND ALL"
                                            :rem 231
170 DATA "FOLLOWING DESIGNS, ARE COMPLETE" :rem 17
180 DATA "TO GO ON TO THE NEXT ONE.",       :rem 204
190 FORI=0TO270STEP5:SYS49600,I,100+SIN(I/50)*100,
    319-I,100+COS(I/25)*50,1:NEXT           :rem 241
200 GETA$:IFA$=""THEN200                     :rem 71
210 GOSUB700                                :rem 170
220 DATA "THIS FIGURE IS DRAWN IN HIRES THEN"
                                             :rem 69
230 DATA "REDISPLAYED IN MULTICOLOR FOR AN":rem 64
240 DATA "INTERESTING EFFECT",               :rem 25
250 FORI=0TO309STEP2:SYS49600,I,100+SIN(I/50)*100,
    I+10,100+SIN(I/50)*50,1:NEXT            :rem 179
260 GOSUB640:GOSUB700                         :rem 3
270 DATA "HIRES/MULTICOLOR FIGURE NUMBER 2",
                                            :rem 148
280 FORI=0TO309STEP2:SYS49600,I,100+COS(I/50)*100,
    I+10,100+SIN(I/50)*50,1:NEXT            :rem 177
290 GOSUB640:GOSUB700                         :rem 6
300 DATA "SIMPLE FIGURE NUMBER 2",          :rem 164
310 FORI=0TO319STEP2:SYS49600,I,100+SIN(I/50)*100,
    319-I,100+COS(I/50)*50,1                :rem 113
```

```
320 NEXT                                          :rem 212
330 GETA$:IFA$=""THEN330                           :rem 79
340 GOSUB700                                      :rem 174
350 DATA "SIMPLE FIGURE NUMBER 3"                 :rem 126
360 DATA "TWO FILLS AT 0,0 AND 319,199 ARE SHOWN"
                                                  :rem 50
370 DATA "AFTER YOU HIT ANY KEY; THESE FILLS"
                                                  :rem 75
380 DATA "GIVE AN IDEA OF THE FILL MECHANISM",
                                                  :rem 36
390 FORI=0TO310STEP5:SYS49600,I,100+SIN(I/50)*100,
    319-I,100+SIN(I/50)*50,2:NEXT                 :rem 242
400 GETA$:IFA$=""THEN400                           :rem 75
410 SYS50000,0,0,1:SYS50000,319,199,1             :rem 193
420 GETA$:IFA$=""THEN420                           :rem 79
430 GOSUB700:SYS50400,C,11,15,11                  :rem 253
440 DATA "THE NEXT IMAGE IS A CIRCLE WITH THE"
                                                  :rem 37
450 DATA "RADII PLOTTED AS 'EXCLUSIVE-OR'",:rem 16
460 FORI=0TO2*↑-↑/100STEP↑/100:SYS49600,160,100,16
    0+COS(I)*100,100-SIN(I)*80,2                   :rem 94
470 NEXT                                          :rem 218
480 GETA$:IFA$=""THEN480                           :rem 91
490 SYS50400,C,0,0,1:GOSUB700                     :rem 106
500 DATA "THIS IS A MULTICOLOR IMAGE"             :rem 117
510 DATA "CREATED WITH LINE AND FILL ROUTINES",
                                                  :rem 239
520 SYS50400,M:N=32:FORI=0TO2*↑STEP↑/N            :rem 191
530 SYS49600,160,100,160+COS(I)*80,100-SIN(I)*64,1
    ,1,15:NEXT                                    :rem 224
540 N=16:FORI=0TO2*↑STEP↑/N:X=160+COS(I)*100:Y=100
    -SIN(I)*80                                    :rem 135
550 SYS49600,X,Y,160+COS(I+↑/N)*100,100-SIN(I+↑/N)
    *80,1,2,14:NEXT                               :rem 224
560 SYS50000;160,21,1,3,6:SYS50000;0,0,1,1,11
                                                  :rem 75
570 SYS49600,0,0,319,0,1,2,2:SYS49600,319,0,319,19
    9,1,2,2                                       :rem 166
580 SYS49600,319,199,0,199,1,2,2:SYS49600,0,199,0,
    0,1,2,2                                       :rem 179
590 GETA$:IFA$=""THEN590                           :rem 95
600 SYS50000,160,21,1,2,14                         :rem 40
610 GETA$:IFA$=""THEN610                           :rem 81
620 SYS50400,W:PRINT"{CLR}";:SYS50400,T:END
                                                  :rem 175
630 :                                             :rem 211
640 GETA$:IFA$=""THEN640                           :rem 87
650 SYS50400,M:SYS50400,C,0,0,2,5,1               :rem 129
660 GETA$:IFA$=""THEN660                           :rem 91
```

```
670 SYS50400,H:SYS50400,C,0,0,1                    :rem 191
680 RETURN                                         :rem 126
690 :                                              :rem 217
700 SYS50400,W:PRINT"{CLR}{DOWN}":SYS50400,T:K=0
                                                   :rem 101
710 READN$:IFN$=""THEN730                          :rem 171
720 PRINTTAB(20-LEN(N$)/2)N$"{DOWN}":K=K+1:GOTO710
                                                   :rem 27
730 PRINTTAB(17)"[6 @]":PRINTTAB(17)"{RVS} WAIT
    {UP}"                                          :rem 70
740 FORI=1TO350*K:GETA$:IFA$=""THENNEXT            :rem 133
750 SYS50400,H:RETURN                              :rem 34
```

Chapter 4

Game Programming

Gregg Peele

Two-Sprite Joystick

Machine language joystick routines are fairly common game programming aids. You've probably seen several. This routine, however, is a bit different, for it allows you to use two joysticks on the Commodore 64. Not only that, but it lets you move sprites smoothly and quickly across the screen.

One of the greatest advantages of machine language is its speed of execution. What once seemed slow can instead seem fast. Machine language is especially handy when you're designing and writing arcade-style games on your Commodore 64. BASIC is just too slow and too cumbersome for many of the things you want your game to do. Moving figures using the joystick is one aspect of game play that suffers when you have only BASIC to work with. Sprites seem to move even slower, and when you try to use the right-hand side of the screen, getting across the invisible "seam" becomes a nightmare of POKEs and PEEKs. Some programmers use only the left side of the screen because of this.

"Two-Sprite Joystick" gives you the speed of machine language, as well as easy use of sprites and joysticks. Once you've included it as part of your own game program, you'll be able to use both joystick ports and the entire screen. It moves two sprites, each controlled by a separate joystick, quickly and smoothly, even across the dreaded seam.

All the sprite movements are calculated by the routine, including checking for the seam and keeping the sprites on the screen. You can use your own sprite data, creating your own sprite design, with only minor modification. You can even change the color of the sprites with a single POKE. Writing games, especially two-player games, becomes much easier when you have this routine in hand.

Smooth and Fast

Type in and save Two-Sprite Joystick. It's in the form of a BASIC loader which POKEs the machine language routine into an area of memory safe from any BASIC program. It's a good idea to save the program before you try to run it. Otherwise, if you've mistyped any part of the program, and cause the computer to crash, you'd have to turn it off, then on again, to regain control. That would erase all of your

typing. Once you have it saved, type LOAD, then RUN. Wait a few moments for the data to be POKEd into memory, then respond to the two prompts. Enter a number between 0 and 15 to select the colors for the two sprites. The blocks then appear on the screen.

Sprite 0 is controlled by the joystick plugged into port 1, while sprite 1 is controlled by the joystick inserted into port 2. Maneuver the sprites around the screen, noticing how quickly and smoothly each moves, even when it crosses the seam. The sprites stay on the screen at all times. When you're moving sprite 0, controlled by the joystick in port 1, you'll see odd characters such as 2, P, or the ← (back arrow). This is unavoidable in the direct mode of the demonstration, but will not occur when you use the routine in your own program.

To use this routine in your own game, all you have to do is create two sprites, and POKE the machine language data into memory with a BASIC loader such as:

```
10 I=49152
20 READ A:IF A=256 THEN40
30 POKE I,A:I=I+1:CK=CK+A:GOTO20
40 IFCK<>71433THENPRINT"{CLR}ERROR IN DATA STATEME
   NTS":END
```

When you run your complete program (which includes this BASIC loader and the DATA statements), just SYS 49152 to access the two-sprite joystick routine. You can even do that from within your program if you want.

If you want to change the sprites' colors in the middle of the game, just insert the statement:

POKE 49228,x to change the color of sprite 0, or
POKE 49233,x to change the color of sprite 1, where x in both statements is the color value, from 0 to 15. The sprite color will change instantly. For instance, POKEing 49228,14 turns sprite 0 light blue.

Two-Sprite Joystick

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

```
40000 FORT=12288TO12288+128:POKET,255:NEXT:REM ADD
      SPRITE DATA HERE                      :rem 217
40010 I=49152                               :rem 128
40020 READ A:IF A=256 THEN40040              :rem 94
40030 POKE I,A:I=I+1:CK=CK+A:GOTO40020      :rem 169
40040 IFCK<>71433THENPRINT"{CLR}ERROR IN DATA STAT
      EMENTS":END                           :rem 120
40050 INPUT "COLOR FOR SPRITE0";C0:POKE49228,C0
                                            :rem 249
```

```
40060 INPUT "COLOR FOR SPRITE1";C1:POKE49233,C1
                                                :rem 249
40070 SYS49152                                  :rem 255
49152 DATA 169,125,141,224,207,169,0            :rem 246
49160 DATA 141,225,207,169,125,141,226           :rem 86
49168 DATA 207,169,0,141,227,207,169              :rem 1
49176 DATA 125,141,0,208,141,1,208              :rem 134
49184 DATA 141,2,208,141,3,208,169              :rem 145
49192 DATA 0,141,16,208,120,169,52              :rem 140
49200 DATA 141,20,3,169,192,141,21              :rem 129
49208 DATA 3,88,96,162,0,32,65                  :rem 212
49216 DATA 192,162,1,32,65,192,76               :rem 103
49224 DATA 49,234,169,3,141,21,208              :rem 147
49232 DATA 169,192,141,248,7,169,1              :rem 158
49240 DATA 141,39,208,169,2,141,40              :rem 141
49248 DATA 208,169,193,141,249,7,189             :rem 18
49256 DATA 0,220,41,15,157,228,207              :rem 141
49264 DATA 56,169,15,253,228,207,157               :rem 8
49272 DATA 232,207,160,0,200,152,221            :rem 224
49280 DATA 232,207,208,249,224,1,208            :rem 245
49288 DATA 2,162,2,152,10,168,185               :rem 100
49296 DATA 135,192,72,185,134,192,72               :rem 9
49304 DATA 96,1,194,213,193,217,193             :rem 206
49312 DATA 1,194,225,193,229,193,236            :rem 255
49320 DATA 193,1,194,221,193,250,193            :rem 246
49328 DATA 243,193,1,194,169,50,221             :rem 203
49336 DATA 1,208,176,12,189,1,208               :rem 100
49344 DATA 56,189,1,208,233,1,157               :rem 104
49352 DATA 1,208,96,169,229,221,1               :rem 102
49360 DATA 208,144,12,189,1,208,24              :rem 145
49368 DATA 189,1,208,105,1,157,1                 :rem 50
49376 DATA 208,96,56,189,224,207,233             :rem 14
49384 DATA 65,157,228,207,189,225,207            :rem 63
49392 DATA 233,1,29,228,207,144,13              :rem 147
49400 DATA 169,65,157,224,207,169,1             :rem 205
49408 DATA 157,225,207,76,247,192,24               :rem 6
49416 DATA 189,224,207,105,1,157,224            :rem 249
49424 DATA 207,189,225,207,105,0,157            :rem 249
49432 DATA 225,207,56,189,224,207,233            :rem 48
49440 DATA 0,157,228,207,189,225,207            :rem 253
49448 DATA 233,1,29,228,207,144,19              :rem 155
49456 DATA 224,2,240,34,173,16,208              :rem 145
49464 DATA 9,1,141,16,208,189,224               :rem 104
49472 DATA 207,157,0,208,96,224,2               :rem 101
49480 DATA 240,30,173,16,208,41,254             :rem 192
49488 DATA 141,16,208,189,224,207,157            :rem 58
49496 DATA 0,208,96,173,16,208,9                 :rem 64
49504 DATA 2,141,16,208,189,224,207             :rem 196
49512 DATA 157,0,208,96,173,16,208              :rem 153
```

```
49520 DATA 41,253,141,16,208,189,224      :rem 246
49528 DATA 207,157,0,208,96,56,189        :rem 170
49536 DATA 224,207,233,25,157,228,207      :rem 47
49544 DATA 189,225,207,233,0,29,228       :rem 207
49552 DATA 207,176,13,169,24,157,224        :rem 1
49560 DATA 207,169,0,157,225,207,76       :rem 207
49568 DATA 127,193,56,189,224,207,233      :rem 63
49576 DATA 1,157,224,207,189,225,207        :rem 4
49584 DATA 233,0,157,225,207,56,189       :rem 212
49592 DATA 224,207,233,0,157,228,207      :rem 250
49600 DATA 189,225,207,233,1,29,228       :rem 201
49608 DATA 207,144,19,224,2,240,34        :rem 144
49616 DATA 173,16,208,9,1,141,16           :rem 47
49624 DATA 208,189,224,207,157,0,208      :rem 255
49632 DATA 96,224,2,240,30,173,16          :rem 94
49640 DATA 208,41,254,141,16,208,189      :rem 252
49648 DATA 224,207,157,0,208,96,173       :rem 211
49656 DATA 16,208,9,2,141,16,208           :rem 51
49664 DATA 189,224,207,157,0,208,96       :rem 216
49672 DATA 173,16,208,41,253,141,16       :rem 198
49680 DATA 208,189,224,207,157,0,208        :rem 1
49688 DATA 96,32,158,192,96,32,178        :rem 183
49696 DATA 192,96,32,198,192,96,32        :rem 182
49704 DATA 78,193,96,32,158,192,32        :rem 169
49712 DATA 78,193,96,32,178,192,32        :rem 170
49720 DATA 78,193,96,32,178,192,32        :rem 169
49728 DATA 198,192,96,32,158,192,32       :rem 225
49736 DATA 198,192,96,96,256              :rem 144
```

Peter Marcotty

Scroll 64

A window can make a static screen more dynamic. This short machine language routine gives you control over screen scrolling from within BASIC programs.

Someone spots a tornado and reports it to the local weather bureau. Your television beeps and a warning moves across the bottom of the screen.

How would you create that effect on your 64? How do you make words scroll sideways?

Scroll Control and Windows

When you LIST a program, the screen fills quickly. As new lines appear, the screen scrolls from bottom to top (everything moves up a notch).

But there may be times when you want movement from top to bottom, or right to left. Or perhaps you want some information to stay in one section of the screen while everything else moves.

You need a screen window. Things in the window move, while everything else stays put. Some new computers, such as the Apple Macintosh, have built-in windowing.

"Scroll 64" won't turn your 64 into a Macintosh, but it can make your screen displays more dynamic.

Asteroid Belts and Invoices

There are many ways to creatively use screen windows and scrolling. For example, scrolling is common in certain types of videogames. You drive a car on a road that moves toward you. Or your spaceship at the bottom of the screen has to shoot at descending asteroids. In addition to the action window, there is usually a section with information about your current score, remaining fuel, velocity, and so on. It would be confusing if your score moved with the asteroids, so the action of the game is put in a window. Your score goes somewhere outside the window.

Business programs can benefit from windows, as well. You might want a command line in an invoicing program, to remind the user of the various options. The window would cover all of the screen except the last line, which says "F1 = Help F3 = New F5 = Help F7 = Continue." Everything scrolls on the screen except the line at the bottom. Another possibility is a product list window in the corner of the screen. When the user of the invoice program wants to look up a product number, the window opens up and the list scrolls by.

Customizing Your Programs

Scroll 64 is a machine language program which goes into memory locations 49152–49528 ($C000–$C172). It does not use any BASIC RAM. The BASIC loader program reads the DATA statements and POKEs the numbers into memory. When the ML program is safe in memory, type NEW to get rid of the loader and clear RAM.

To use Scroll 64, type LOAD and RUN 60, type NEW, and then LOAD your own program. To activate it, simply SYS 49152. It scrolls once and returns to BASIC.

Or if you prefer, you could build the BASIC loader into your program. Renumber the lines (starting at 60000, for instance), add a RETURN, and call it with a GOSUB at the beginning of your program.

Scroll 64 moves a certain section of the screen in a certain direction, along with the corresponding color memory. These memory addresses contain the pertinent information:

Location	Function
49522	**Direction**
49523	**Left Boundary**
49524	**Right Boundary**
49525	**Top Boundary**
49526	**Bottom Boundary**
49527	**Horizontal Wrap**
49528	**Vertical Wrap**

Direction is the way in which the screen scrolls. To change it, POKE 49522 with one, two, three, or four (for left, right, up, or down respectively). The boundary values define the size of the window. Left and right boundaries can range from 0 to 39. Top and bottom must be between 0 and 24. When the program is first run, a five by five window goes in the top-left corner.

The wrap values determine what happens to characters when they reach the edge of the window. You can make them disappear or wrap around to the beginning. POKE 49527 and 49528 as follows:

Number	Effect
0	**Don't wrap around, leave a trail**
1	**Wrap around**
2	**Don't wrap around, erase trail**

To activate the scroll window, SYS 49152. You can SYS over and over, changing the direction, boundaries, and wrap values as you wish. Note that when the ML routine is activated, whatever is in the window scrolls, but at all other times, the screen acts as it normally does.

Special Loading Instructions

Enter the program and SAVE it before you test anything. To put the ML into memory, type

RUN 60 (not just RUN)

The computer will take a few moments to complete the POKEs. As added insurance there is a checksum routine built into the program. Type RUN, and the values in memory are checked. If an error message appears, check the DATA statements. Block 1 includes lines 5010–5050, block 2 includes lines 5060–5100, and so on. If you find a mistake, fix it and type RUN 60 followed by RUN. Remember to save the final, debugged version.

There is one thing to watch out for. If you decide to use a single line for your window, you can scroll left or right, but don't try to move up or down. For example, if you set the top boundary to five and the bottom to five, you can scroll line five to the left or to the right. But try to scroll up and the computer crashes. And you cannot escape the crash with RUN/STOP–RESTORE. You have to turn your computer off and then on again (and lose whatever you have in memory).

Smoother Scrolling

Regular scrolls move whole characters. It's like picking up a letter and dropping it down one line.

The 64 can do smoother scrolls, moving characters a pixel at a time. The key is memory locations 53270 (horizontal) and 53265 (vertical). To do smooth scrolls, use these formulas:

```
POKE 53270, (PEEK(53270)AND248) + X
POKE 53265, (PEEK(53265)AND248) + Y
```

X and Y can be any numbers from 0 to 7. Once you've gone to 7 or 0, you'll have to do a regular scroll and reset the smooth scroll to the other limit. Smooth scrolling can make an action game look more realistic — the characters don't jump around, they slide.

A minor annoyance in this method is that while the screen is doing a smooth scroll, you may see small gaps at the edges. You can get around this by turning off bit 3 of these two registers; in the POKEs above, AND with 240 instead of 248. In effect, you pull the border in a notch, resulting in a 38 column by 24 row display (instead of 40 × 25).

Because smooth scrolling affects the whole screen, it is not compatible with Scroll 64 windows. If you combined the two, you would see smooth scrolling inside the window and jittery, vibrating

characters outside the window. To fix this would require a high-res screen, customized word sprites, or a raster interrupt wedge. All of these are beyond the scope of this simple program.

Scroll 64

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

```
10 DATA11507,12573,12522,11001                    :rem 238
20 A=49152:B=84:C=A+B:FORI=1TO4:D=0:READX:FORJ=ATO
   C:D=D+PEEK(J):NEXT                             :rem 236
30 IFD<>XTHENPRINTTAB(19)"ERROR IN BLOCK #"I:GOTO5
   0                                              :rem 106
40 PRINT"BLOCK #"I"IS CORRECT"                    :rem 254
50 A=C+1:C=A+B:NEXT:END                           :rem 116
60 FORI=1TO4:READA:NEXT:READLO,HI:FORI=LOTOHI:READ
   X:POKEI,X:NEXT:END                             :rem 125
5000 DATA 49152, 49528                             :rem 28
5010 DATA 174,114,193,224,3,144,3,76,117,192,188,1
     14,193,140,121,193,174                       :rem 222
5020 DATA 118,193,232,202,32,30,193,172,121,193,17
     3,119,193,201,2,208,10                       :rem 198
5030 DATA 169,32,72,173,33,208,72,76,50,192,177,90
     ,72,177,92,72,204                            :rem 252
5040 DATA 116,193,240,20,200,177,90,72,177,92,136,
     145,92,104,145,90,200                        :rem 166
5050 DATA 204,116,193,208,238,240,18,136,177,90,72
     ,177,92,200,145,92,104                       :rem 230
5060 DATA 145,90,136,204,115,193,208,238,173,119,1
     93,201,0,208,5,104,104                       :rem 210
5070 DATA 76,111,192,104,145,92,104,145,90,236,117
     ,193,208,160,96,172,116                       :rem 23
5080 DATA 193,200,189,114,193,170,32,30,193,173,12
     0,193,201,2,208,19,136                       :rem 214
5090 DATA 169,32,153,122,193,173,33,208,153,162,19
     3,204,115,193,208,239,240                    :rem 122
5100 DATA 16,136,177,90,153,122,193,177,92,153,162
     ,193,204,115,193,208,240                      :rem 71
5110 DATA 236,117,193,240,37,202,32,30,193,172,116
     ,193,200,136,177,90,72                       :rem 215
5120 DATA 177,92,32,48,193,145,92,104,145,90,32,56
     ,193,204,115,193,208                         :rem 136
5130 DATA 234,236,117,193,208,221,240,46,202,206,1
     18,193,232,32,30,193,172                      :rem 53
5140 DATA 116,193,200,136,32,48,193,177,90,72,177,
     92,32,56,193,145,92                          :rem 96
5150 DATA 104,145,90,204,115,193,208,234,236,118,1
     93,208,221,238,118,193,232                   :rem 166
5160 DATA 32,30,193,173,120,193,201,0,240,20,172,1
     15,193,136,200,185,162                       :rem 194
```

```
5170 DATA 193,145,92,185,122,193,145,90,204,116,19
     3,208,240,96,189,89,193                    :rem 54
5180 DATA 133,91,24,105,212,133,93,189,64,193,133,
     90,133,92,96,72,152                        :rem 88
5190 DATA 24,105,40,168,104,96,72,152,56,233,40,16
     8,104,96,0,40,80                           :rem 179
5200 DATA 120,160,200,240,24,64,104,144,184,224,8,
     48,88,128,168,208,248                      :rem 172
5210 DATA 32,72,112,152,192,4,4,4,4,4,4,4,5,5,5,5,
     5                                          :rem 173
5220 DATA 5,6,6,6,6,6,6,6,7,7,7,7,7,3,0,4,0:rem 44
5230 DATA 4,1,1                                 :rem 210
```

Dan Carmichael and Tom R. Halfhill

64 Paddle Reader

An enhancement of a program which first appeared in the July 1983 issue of COMPUTE!'s Gazette, *this game utility reads two paddles and reduces the "jitters" commonly experienced with these game controllers.*

One of the articles in the premier issue of *COMPUTE!'s Gazette* was a paddle reader routine for the Commodore 64. The idea was to reduce the "jitter" in screen objects controlled by the game paddles. This jittering is caused by minor fluctuations in the paddle's readings. To calm down the jitter, Bobby Williams wrote a short machine language routine which read the paddle 256 times in a split second, averaged the readings, and used the average for a final paddle value.

The routine worked fine, but some readers wanted more. The original routine was for one paddle only, ignoring the second paddle. The result of these readers' requests is this new and improved routine. It is still a machine language program, still POKEd into memory by a BASIC loader so that you don't need to know anything about ML to use it, and it still reduces the paddle jitters. But now it works with two controllers instead of only one.

A BASIC Loader

As before, you don't have to know anything about machine language to use this routine. It's in the form of a BASIC loader — a short BASIC subprogram which you add to your own BASIC programs. Using the POKE statement, it loads decimal numbers into memory which correspond to the proper machine language commands.

The program is stored in a normally safe area of memory, the 88 bytes from address 679 to 710 (decimal). This is not the same area where the previous paddle reader routine was stored. The earlier routine was stored in an often-used block of memory that we've decided to preserve for other purposes.

Be sure to type in the program correctly, and as always when dealing with machine language, SAVE the program before the first RUN. This allows you to recover your work in case of a typing error that crashes the computer.

Reading the Paddles

Once the routine is added to your BASIC program, it must be activated with a SYS statement each time you want to read the paddles. To start the routine, use SYS 679.

You then read the paddles with a simple PEEK statement. To get the averaged reading of paddle 1, use PEEK(251); for paddle 2, use PEEK(252). Here's an example:

```
10 SYS679:P1=PEEK(251):P2=PEEK(252)
20 PRINTP1;:PRINTP2
30 GOTO10
```

Note that these locations are different from the usual paddle locations. That's because the routine stores the averaged readings at 251 and 252, not at the customary locations (that is, 54297 on the Commodore 64).

The short program above displays the values found at the paddle locations. Plug the paddles into port 1. As you move the paddle knobs, the values will change. Note that each time you read the paddles, you must use the SYS 679 command.

64 Paddle Reader

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

```
10 FORA=679 TO 710:READB:POKEA,B:NEXT      :rem 208
679 DATA 169,0,170,168,24,109,25            :rem 54
686 DATA 212,144,1,200,202,208,247          :rem 130
693 DATA 132,251,138,168,24,109,26          :rem 149
700 DATA 212,144,1,200,202,208,247          :rem 117
707 DATA 132,252,96,0                       :rem 10
```

Charles Bond
Translated to machine language by Gary E. Marsa and for the 64 by Gregg Peele.

Maze Generator

This program, although quite short, creates a totally random maze display every time it's run. Use it as the basis for any maze game you'd like to design.

"Maze Generator" is a remarkably short algorithm which produces random mazes on your television set or monitor. There are two versions of the program included here. The first, in BASIC, may seem shorter and easier to type in, but if you compare it to the second program, you'll see how slow it is. Program 2 is a machine language program. In the form of a BASIC loader which POKEs the data into your computer's memory, it creates mazes almost instantly. This is a good example of how fast machine language is, compared to BASIC.

The BASIC version of the program has also been included to better explain how the maze generator works. It's difficult, if you aren't familiar with machine language, to understand what each value POKEd into memory does. For those of us who are more comfortable with BASIC, it's much easier to see how that version operates.

Refer to Program 1 and the flowchart figure as the program's details are explained. You can use either Program 1 or Program 2 to see how the mazes are created. You can even use either version as part of your own game program. However, remember that the machine language version executes much faster. If you use the ML version (Program 2), a player would have to wait only a moment. Since Program 2 is in the form of a BASIC loader, SAVE it before you try to run it. If you enter the data incorrectly, the computer could crash, and you'd have to retype the entire listing unless it had been previously saved on tape or disk.

The Background Field

The algorithm operates on a background field which must be generated on the screen prior to line number 210 in Program 1. The field must consist of an odd number of horizontal rows, each containing an odd number of cells — in other words, a rectangular array. It's convenient to think of the field as a two-dimensional array with the upper-left corner having coordinates X = 0 and Y = 0, where X is the horizontal direction and Y is vertical. No coordinates are used to identify absolute locations by the program, but the concept is useful in configuring the field.

Given that the upper-left cell of the field has coordinates 0,0, then the terminal coordinates both horizontally and vertically must be even numbers. (Remember that there is an *odd* number of rows and columns.) In addition, the background field must be surrounded on all sides by memory cells whose contents are different from the number used to identify the field. That is, if the field consists of reverse video spaces, the number corresponding to that character must not be visually adjacent to the field.

This could happen inadvertently if the screen RAM and system ROM have contiguous addresses. A sufficient precaution is to avoid covering the entire screen with field. Leave at least one space at the beginning or end of each line and, in general, leave the uppermost and lowermost lines on the screen blank.

The Maze Generator

The creation of the maze begins by placing a special marker in a suitable starting square. The program here always begins at the square just inside the upper-left cell of the previously drawn field. (Note that with our coordinate scheme this would be cell 1,1.) Any cell with odd-numbered coordinates would work, however, as long as it is internal to the field.

Next, a random direction is chosen by invoking the random number generator in the computer and producing an integer from 0 to 3. This integer, with the aid of a short table, determines a direction and a corresponding cell just two steps away from the current cell. This new cell is examined (PEEKed) to see if it is part of the field. If it is, the direction integer is put there as a marker, and the barrier between it and the current cell is erased.

In addition, the pointer to the current cell is moved to point to the new one. This process is repeated until the new cell fails the test; that is, it is not a field cell. When this happens, the direction vector is rotated 90 degrees and the test is repeated. Thus, the path carved out of the field will continue until a dead end is reached.

A dead end, incidentally, could occur in as few as five steps. When it does occur, we can make use of the markers which were dropped along the way Hansel-and-Gretel style. These can be checked to determine which direction we came from, so that we can back up and look for untrodden paths. So long as none can be found, the program will back up, one step at a time, erasing the markers as it goes. When a new direction can be taken, the pointer is set off in that direction, and the process continues as before.

Ultimately, the pointer will return to the start, a condition which is detected by the recovery of the special starting (now "ending") marker. This cell is then blanked and the program is done, leaving the pointer as it was at the start.

The Program

The direction table set up in lines 100 and 110 of Program 1 converts an integer to an address offset. In this case (40-column screen), we wish to step two cells to the right, up, left, or down.

Line 120 contains the variable SC, which is the memory address of the start of screen RAM. Lines 130–160 establish the background field on the screen.

The rest of the program draws the maze, as previously explained. Line 310 is simply a convenient stopping point which prevents the screen from scrolling.

It may not be immediately obvious that this algorithm always produces a maze with only one nontrivial path between any two points, or that the maze will always be completely filled, but this can be proved. While the proofs will not be provided here, math buffs may find it interesting that for a maze of any size there will be exactly:

$$\frac{(H-1)(V-1)}{2} - 1$$ **empty cells in the completed maze**

where H is the number of cells in each field row and V is the number of rows.

An interesting feature of this algorithm is that it works equally well in certain types of nonrectangular fields. U-shaped fields or fields with holes in them are quite suitable — as long as certain restrictions are observed. Just make sure that the coordinates of the upper-left and lower-right cells of any cut-out area are pairs of odd numbers. Also, if there is a single row of field cells between any cut-out areas and the outside of the original field, it may be removed.

Machine Language Mazes

Program 2 is a machine language translation of Program 1. It is in the form of a BASIC loader. It can be inserted into any BASIC program just as Program 1.

Program 3 is the disassembly listing of the machine language routine found in Program 2.

The Mouse

The subroutine on lines 1000 to 1020 of Program 1 produces an artificial mouse which roams the maze endlessly. The mouse adheres

to a "left-hand rule" when a choice of directions is possible. That is, when it is confronted with a branch-point, it will move off to the left, if possible. Otherwise, it will go forward. When no choice is available, it will turn around. These lines are unnecessary for the creation of the maze and may be deleted. Program 2 does not contain the mouse.

Program 1. BASIC Maze Generator

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

```
100 DIMA(3)                                          :rem 48
110 A(0)=2:A(1)=-80:A(2)=-2:A(3)=80                  :rem 208
120 WL=160:HL=32:SC=1024:A=SC+81                     :rem 48
130 PRINT"{CLR}"                                     :rem 248
140 FORI=1TO23                                       :rem 59
150 PRINT"{RVS}{WHT}{39 SPACES}"                     :rem 126
160 NEXTI                                            :rem 31
210 POKEA,4                                          :rem 99
220 J=INT(RND(1)*4):X=J                              :rem 52
230 B=A+A(J):IF PEEK(B)=WLTHENPOKEB,J:POKEA+A(J)/2
    ,HL:A=B:GOTO220                                  :rem 166
240 J=(J+1)*-(J<3):IFJ<>XTHEN230                     :rem 30
250 J=PEEK(A):POKEA,HL:IFJ<4THENA=A-A(J):GOTO220
                                                     :rem 192
310 GETC$:IFC$=""THEN310                             :rem 79
1000 POKEA,81:J=2                                    :rem 185
1010 B=A+A(J)/2:IFPEEK(B)=HLTHENPOKEB,81:POKEA,HL:
     A=B:J=(J+2)+4*(J>1)                             :rem 33
1020 J=(J-1)-4*(J=0):GOTO1010                        :rem 11
```

Program 2. Machine Language Maze Generator

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

```
10 I=49152:IF PEEK(I+2)=216THENSYS49160:END
                                                     :rem 234
20 READ A:IF A=256 THENSYS49160:END                  :rem 230
30 POKE I,A:I=I+1:GOTO 20                            :rem 130
49152 DATA 1,0,216,255,255,255,40                     :rem 89
49160 DATA 0,169,81,133,251,169,40                   :rem 146
49168 DATA 133,253,169,4,133,252,133                 :rem 250
49176 DATA 254,169,147,32,210,255,162                 :rem 49
49184 DATA 0,160,0,169,160,145,253                   :rem 142
49192 DATA 200,192,39,208,249,24,165                    :rem 0
49200 DATA 253,105,40,133,253,144,2                  :rem 177
49208 DATA 230,254,232,224,23,208,229                 :rem 36
49216 DATA 160,0,169,4,145,251,169                   :rem 149
49224 DATA 255,141,15,212,169,128,141                 :rem 37
49232 DATA 18,212,173,27,212,41,3                     :rem 85
49240 DATA 133,173,170,10,168,24,185                 :rem 243
49248 DATA 0,192,101,251,133,170,185                 :rem 240
49256 DATA 1,192,101,252,133,171,24                  :rem 186
```

```
49264 DATA 185,0,192,101,170,133,253      :rem 240
49272 DATA 185,1,192,101,171,133,254      :rem 242
49280 DATA 160,0,177,253,201,160,208      :rem 237
49288 DATA 18,138,145,253,169,32,145        :rem 9
49296 DATA 170,165,253,133,251,165,254    :rem 100
49304 DATA 133,252,76,62,192,232,138      :rem 250
49312 DATA 41,3,197,173,208,189,177       :rem 212
49320 DATA 251,170,169,32,145,251,224      :rem 35
49328 DATA 4,240,26,138,10,168,162        :rem 147
49336 DATA 2,56,165,251,249,0,192         :rem 103
49344 DATA 133,251,165,252,249,1,192      :rem 250
49352 DATA 133,252,202,208,238,76,62      :rem 249
49360 DATA 192,169,1,160,0,153,0           :rem 37
49368 DATA 216,153,0,217,153,0,218        :rem 144
49376 DATA 153,0,219,200,208,241,96,256   :rem 143
```

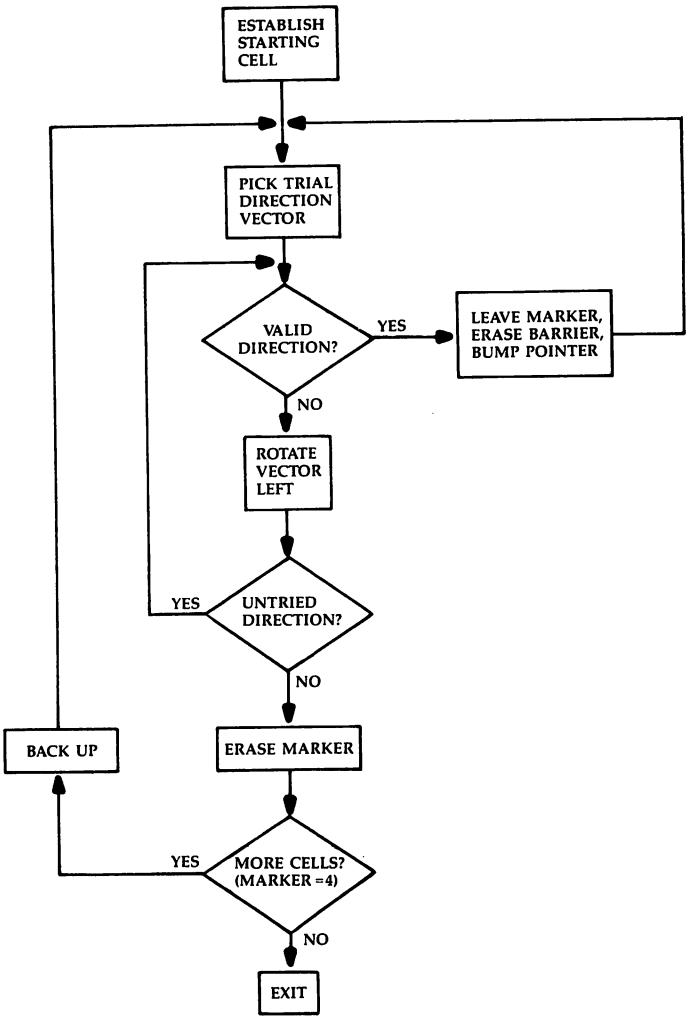
Program 3. Source Listing

```
C000 01 00
C002 D8
C003 FF
C004 FF
C005 FF
C006 28
C007 00
C008 A9 51      LDA #$51
C00A 85 FB      STA $FB
C00C A9 28      LDA #$28
C00E 85 FD      STA $FD
C010 A9 04      LDA #$04
C012 85 FC      STA $FC
C014 85 FE      STA $FE
C016 A9 93      LDA #$93
C018 20 D2 FF   JSR $FFD2
C01B A2 00      LDX #$00
C01D A0 00      LDY #$00
C01F A9 A0      LDA #$A0
C021 91 FD      STA ($FD),Y
C023 C8         INY
C024 C0 27      CPY #$27
C026 D0 F9      BNE $C021
C028 18         CLC
C029 A5 FD      LDA $FD
C02B 69 28      ADC #$28
C02D 85 FD      STA $FD
C02F 90 02      BCC $C033
C031 E6 FE      INC $FE
C033 E8         INX
C034 E0 17      CPX #$17
C036 D0 E5      BNE $C01D
```

```
C038 A0 00      LDY #$00
C03A A9 04      LDA #$04
C03C 91 FB      STA ($FB),Y
C03E A9 FF      LDA #$FF
C040 8D 0F D4   STA $D40F
C043 A9 80      LDA #$80
C045 8D 12 D4   STA $D412
C048 AD 1B D4   LDA $D41B
C04B 29 03      AND #$03
C04D 85 AD      STA $AD
C04F AA         TAX
C050 0A         ASL
C051 A8         TAY
C052 18         CLC
C053 B9 00 C0   LDA $C000,Y
C056 65 FB      ADC $FB
C058 85 AA      STA $AA
C05A B9 01 C0   LDA $C001,Y
C05D 65 FC      ADC $FC
C05F 85 AB      STA $AB
C061 18         CLC
C062 B9 00 C0   LDA $C000,Y
C065 65 AA      ADC $AA
C067 85 FD      STA $FD
C069 B9 01 C0   LDA $C001,Y
C06C 65 AB      ADC $AB
C06E 85 FE      STA $FE
C070 A0 00      LDY #$00
C072 B1 FD      LDA ($FD),Y
C074 C9 A0      CMP #$A0
C076 D0 12      BNE $C08A
C078 8A         TXA
C079 91 FD      STA ($FD),Y
C07B A9 20      LDA #$20
C07D 91 AA      STA ($AA),Y
C07F A5 FD      LDA $FD
C081 85 FB      STA $FB
C083 A5 FE      LDA $FE
C085 85 FC      STA $FC
C087 4C 3E C0   JMP $C03E
C08A E8         INX
C08B 8A         TXA
C08C 29 03      AND #$03
C08E C5 AD      CMP $AD
C090 D0 BD      BNE $C04F
C092 B1 FB      LDA ($FB),Y
C094 AA         TAX
C095 A9 20      LDA #$20
C097 91 FB      STA ($FB),Y
```

```
C099 E0 04      CPX #$04
C09B F0 1A      BEQ $C0B7
C09D 8A         TXA
C09E 0A         ASL
C09F A8         TAY
C0A0 A2 02      LDX #$02
C0A2 38         SEC
C0A3 A5 FB      LDA $FB
C0A5 F9 00 C0   SBC $C000,Y
C0A8 85 FB      STA $FB
C0AA A5 FC      LDA $FC
C0AC F9 01 C0   SBC $C001,Y
C0AF 85 FC      STA $FC
C0B1 CA         DEX
C0B2 D0 EE      BNE $C0A2
C0B4 4C 3E C0   JMP $C03E
C0B7 A9 01      LDA #$01
C0B9 A0 00      LDY #$00
C0BB 99 00 D8   STA $D800,Y
C0BE 99 00 D9   STA $D900,Y
C0C1 99 00 DA   STA $DA00,Y
C0C4 99 00 DB   STA $DB00,Y
C0C7 C8         INY
C0C8 D0 F1      BNE $C0BB
C0CA 60         RTS
```

Maze Generator Flow Chart



Chris Metcalf

Multiple-Key Scanner

Writing two-player games can be difficult. One of the problems is figuring out how to allow input by both players. Although there's a two-joystick routine included in this book, you may not have (or want to use) joysticks. Here's a short routine which allows both players to use the keyboard at the same time.

This routine, located at address 828, allows a program to input several keys simultaneously from the keyboard. In a two-player keyboard action game, this can be crucial, since both moves should be evaluated simultaneously. This subroutine can serve other uses, too; it was originally developed to allow chords to be struck on the keyboard and be interpreted as chords, rather than single notes.

To use the routine, enter a SYS 828. The result of this SYS is to place the ASCII values of all the keys that are being pressed at *that* moment in the keyboard buffer, where they can be read with GET statements. For example, if two keys were being held down, *s* and *l*, and a SYS 828 is executed, *s* and *l* are placed in the keyboard buffer. Two GET statements would retrieve them; any further GETs would return "", the null string. Up to ten characters can be interpreted in this fashion.

Not Perfect

However, the routine is not perfect. It really can't be. Commodore did not design the keyboard to be used in this fashion, so multiple-key reading does not always return the correct keys. As far as I know, any two keys are reported accurately; most (but not all) three-key combinations seem to work. As you get to four and above, unfortunately, extraneous characters begin to be reported. Another problem is that BASIC itself is always watching for keys being pressed. Because of this, two copies of one of the keys are sometimes reported when it is first pressed.

One feature of this routine is that it will continue to return the keys pressed, even if the routine is called several times. Thus you cannot only tell when a key is pressed, but also when it is released. Each SYS 828 loads the buffer with the keys currently pressed, regardless how many times the keys have been reported.

Special Codes

Another feature of this routine is that it returns not only ASCII values for all the keys (including function keys, cursor keys, and so forth), but it also returns special codes to indicate when left-SHIFT, right-SHIFT, Commodore, or CTRL keys are depressed. CHR$(1) for the left-hand SHIFT key and SHIFT LOCK, CHR$(2) for the right-hand SHIFT, CHR$(3) for the Commodore key, and CHR$(4) for the CTRL key are returned. This routine also removes the effects of SHIFT or Commodore on other keys. For instance, if SHIFT-3 is pressed, SYS 828 returns a CHR$(1) or CHR$(2) along with the code for *3, not* simply the code for *#*.

Enter and Demonstrate

Program 1, "Keyscan," is a machine language program. Its data is POKEd into the cassette buffer (starting at location 828) by the BASIC loader (lines 10–30). Save the program before you first run it. That will prevent the loss of your typing time if you've entered it incorrectly and the computer locks up.

You can add this program to your own game, simply by changing the line numbers (to 60000 and up, for example), adding a RETURN to its end, and then calling the subroutine with a GOSUB. Once the data has been placed in memory, you access the routine by SYSing 828. Of course, you'll also need to place a GET A$ (or something similar) in your program to retrieve the characters that the routine puts in the keyboard buffer. (See Program 2 for an example of this.)

Program 2, "Demo/Scan," is a BASIC program which shows you what Keyscan can do. Make sure you've loaded and run Keyscan, then load and run Program 2. It prints out the characters received from SYS 828. You can get a good idea of the power and limitations of the Keyscan routine by testing various combinations of keys.

Since Keyscan occupies the cassette buffer (locations 828–1019), it will be erased when you load from or save to your Datassette. If that happens, you'll have to reload and rerun the routine to place the machine language data back in memory.

Program 1. Keyscan

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

```
10 FORI=828 TO 959:READA:POKEI,A:NEXT      :rem 233
20 CK=CK+1:IFCK<>1 THEN PRINT"ERROR IN DATA"
                                            :rem 84
30 END                                      :rem 58
828 DATA 120,169,000,133,198,170            :rem 44
834 DATA 169,254,141,000,220,172            :rem 36
```

```
840 DATA 001,220,148,088,232,056          :rem 38
846 DATA 042,176,243,162,007,160          :rem 44
852 DATA 007,181,088,106,176,027          :rem 52
858 DATA 072,132,002,138,010,010          :rem 26
864 DATA 010,005,002,168,185,125          :rem 37
870 DATA 003,164,198,153,119,002          :rem 43
876 DATA 192,010,240,011,230,198          :rem 39
882 DATA 104,164,002,136,016,223          :rem 34
888 DATA 202,016,216,088,096,017          :rem 57
894 DATA 135,134,133,136,029,013          :rem 44
900 DATA 020,001,069,083,090,052          :rem 28
906 DATA 065,087,051,088,084,070          :rem 56
912 DATA 067,054,068,082,053,086          :rem 54
918 DATA 085,072,066,056,071,089          :rem 62
924 DATA 055,078,079,075,077,048          :rem 68
930 DATA 074,073,057,044,064,058          :rem 50
936 DATA 046,045,076,080,043,047          :rem 50
942 DATA 094,061,002,019,059,042          :rem 41
948 DATA 092,005,081,003,032,050          :rem 33
954 DATA 004,095,049,013,013,013          :rem 35
```

Program 2. Demo/Scan

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

```
10 PRINT"HOLD DOWN KEYS IN VARIOUS COMBINATIONS"
                                          :rem 19
20 PRINT"TO GET A FEEL FOR THE ROUTINE."  :rem 47
30 PRINT"NOTE THAT SHIFT AND COMMODORE DO NOT"
                                          :rem 21
40 PRINT"AFFECT OTHER KEYS BUT GENERATE CODES OF"
                                         :rem 213
50 PRINT"THEIR OWN INSTEAD.{DOWN}"       :rem 237
100 SYS(828):REM STUFF GET-BUFFER        :rem 240
110 GETA$:IFA$=""THEN100                  :rem 70
120 POKE216,1:PRINTA$;                    :rem 31
130 GETA$:IFA$<>""THEN120                :rem 135
140 PRINT:GOTO100                         :rem 38
```

Chapter 5

Applications and Utilities

Glen Colbert

String Search

Data management programs often use string arrays to store information. Retrieving that information can be time-consuming, however, if you have to rely on BASIC. This machine language routine searches an entire array for your designated string, and even returns a flag to mark its place.

Have you ever had to search through a string array to find a certain occurrence of a string? If you have, I'm sure that you can testify to the slowness of BASIC. I have a data-managing program that keeps track of addresses and bills. The program used to take forever to find a string match.

The solution was to use a short machine language program that would do the search and return a flag to the BASIC program. I found Ronald A. Blattel's "PET Searcher" in the April 1983 issue of *COMPUTE!* magazine and began converting the addresses so that it would run on my computer. But after running the converted version a few times, I decided that it was too clumsy. Each time a match was made, control returned to BASIC and the search routine had to be restarted with the USR function. I wanted to be able to scan forward and backward through the matched strings, so I had to build an array and set a flag in a matching element each time the program returned from the routine.

I was using memory space for an array already and I had to check against that array in BASIC to see if there was a match. A more efficient way of doing the search would be to have the ML routine set the flags in the other array while it was working on the search. Thus, when the program returned to BASIC, the checking array would be ready.

Zero Page Swap

This routine utilizes one string variable for matching against the array, a string array which contains the information to be checked, and an integer array to keep track of which elements in the string array match the string variable. It was written without internal JMPs or JSRs so that it can be relocated anywhere in memory. In general, the routine is very dependable. However, there are two things that you must take into consideration when using it: First, it swaps out a section of the zero page into the cassette buffer, and second, the pointers to the variables must be where the routine expects them.

Anyone who has done any machine language programming on

the 64 knows just how limited free space in zero page is. To get around this, the routine moves a part of the zero page into the cassette buffer to make room. Once the routine is finished with the zero page work space, it moves the data from the buffer back where it belongs. Normally this is not a problem, but if there are other ML routines or unprocessed data in the cassette buffer, the routine will write over them.

Picky Variables

The variables must be initialized in the correct sequence to place them in memory locations where the routine can look for them. The *first* and *second* variables defined in the program must be strings (for example, A$ = "" and B$ = ""). The string variable that you want to check for *must* be the *second* variable. The string array to be searched *must* be the first array DIMensioned. The integer array *must* be the second array DIMensioned. All of this has to be done before the routine is called. (Look at Program 2, lines 20–40, for an example of the proper way to initialize these variables.)

Pointers and Counters

The search method used is quite simple. When it's called, the first operation is to swap out a portion of the zero page locations $D9 to $E9, in hexadecimal. The length of the string to be checked for is put into location $D9 and the address of the string is set into locations $DA–$DB. Next, addresses $DC–$DD are set to point to the 0 element of the integer array. Addresses $E0–$E1 are set to point to the three bytes of string array information (length, low byte of address, and high byte of address) for the 0 element of the string array. Things are now ready for the processing loop.

The first step in the processing loop is to increment the pointers for the arrays to the next element. For this reason, the 0 element is not searched. The information for the string array element being worked is moved to locations $E5–$E7. Address $E5 is checked for a 0 (string = "") and if so, swaps the zero page information back in and returns to BASIC. A counter for the search string ($E2) and one for the searched string ($E3) are set to zero and the search begins.

If the search string counter is equal to the length of the search string, there has been a match. If the searched string counter is equal to the length of the searched string, there was no match. On either event, the routine sets the value in the integer array and returns to the main loop to try the next element of the array.

If the counters do not match, the accumulator is loaded with the

first character of the search string. This is compared against each element of the searched string until a match is found. Then the second character of the search string is compared against the next character in the searched string and so on until the counter equals the length of the search string. If a match is not found, the search string counter is reset (but not the searched string counter) and the program loops back.

Machine Language Speed

Using "String Searcher" is not as complicated as you might think. Type in and save Program 1. This is a BASIC loader which POKEs in the machine language routine. Once it's in memory, you can access it by this command:

```
SYS(PEEK(55)+256*PEEK(56))
```

Inserting that command in your program allows you to use the search function. Remember, however, the restrictions on variable placement that were explained earlier. The first two variables defined in your program must be string variables, with the second being the string you want to check for. The string array must be the first array DIMensioned, and the integer array must be the second array DIMed. Running your program, making sure the above SYS command is included, will search through the entire string array and flag any occurrence of the string you selected to check for. The integer variable will contain a 1 if the string was found, a 0 if it was not found. All you have to do, then, is PRINT all the strings that do not equal 0. That's your list of the strings which contain the item(s) you were looking for.

Program 2 is a good example of all this. Notice that the variables are set in the correct form and order in lines 20–40. The string to be checked, Q$, is set in line 30, while the string and integer arrays are DIMed in line 40. Three hundred strings are built and put into the A$(L) array, and each string is searched for Q$, which has been set equal to "GOOD" in line 110. Note that five strings in the array (lines 170–195) have been set to include that word. This, of course, is for demonstration purposes. If you were using String Searcher in your own program, you would already have strings set that you would want to search through.

Make sure you have Program 1 loaded and run. Then load and run Program 2. The string array is built, and two searches are done. The first is with a BASIC program. Once it's finished, it will tell you how long it took and display the strings that included the word you checked for. The machine language routine then searches through the

same string array, again displaying the time used and the strings found. Notice the difference in time. The speed of machine language is clearly demonstrated.

Lines 390 and 450 decide which strings to display. If the integer variable Q% for a particular string does not equal 0 (in other words, if it is a 1), the word searched for was included in that string, and it's printed on the screen. You can use the same process in your own programs to see which strings include the item(s) you checked for.

Program 1. String Searcher BASIC Loader

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

```
100 PRINT"{CLR}{4 DOWN}{12 SPACES}STRING SEARCHER"
                                                :rem 93
110 PRINT"{2 DOWN}ONE MOMENT PLEASE"          :rem 241
120 TP=PEEK(55)+256*PEEK(56)                   :rem 39
130 TP=TP-186:H=INT((TP)/256):L=TP-H*256:POKE55,L:
    POKE56,H                                  :rem 215
140 IN=PEEK(55)+256*PEEK(56):FORC=IN TO IN+185:REA
    DI:POKEC,I:CK=CK+I:NEXT:                  :rem 209
150 IFCK<>26449 THEN PRINT"ERROR IN DATA":END
                                              :rem 130
160 REM****************** STRING SEARCH DATA******
    ****************                          :rem 178
180 DATA 160,17,185,216,0,153,60,3,136,208 :rem 11
190 DATA 247,160,9,177,45,133,217,200,177,45
                                              :rem 128
200 DATA 133,218,200,177,45,133,219,24,160,2
                                              :rem 102
210 DATA 177,47,101,47,105,7,133,220,200,177
                                              :rem 105
220 DATA 47,101,48,133,221,160,0,24,165,47  :rem 3
230 DATA 105,7,133,224,165,48,105,0,133,225:rem 53
240 DATA 169,0,240,12,160,17,185,60,3,153 :rem 212
250 DATA 216,0,136,208,247,96,24,165,224,105
                                              :rem 118
260 DATA 3,133,224,165,225,105,0,133,225,160
                                               :rem 98
270 DATA 0,177,224,153,229,0,200,192,3,208  :rem 7
280 DATA 246,24,165,220,105,2,133,220,165,221
                                              :rem 150
290 DATA 105,0,133,221,165,229,240,202,208,2
                                               :rem 98
300 DATA 240,210,162,0,134,227,134,226,24,165
                                              :rem 146
310 DATA 217,197,226,240,31,24,165,229,197,227
                                              :rem 228
320 DATA 144,37,164,226,177,218,164,227,209,230
                                              :rem 19
```

```
330 DATA 208,6,230,227,230,226,208,226,230,227
                                                :rem 207
340 DATA 169,0,133,226,240,218,160,0,169,1   :rem 7
350 DATA 145,220,200,169,0,145,220,240,197,160
                                                :rem 202
360 DATA 0,152,145,220,240,242                  :rem 179
999 PRINT"DONE":NEW                             :rem 198
```

Program 2. Timed Search

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

```
20 A$="DUMMY DATA":REM{2 SPACES}MUST BE A STRING
                                                :rem 244
30 Q$="":REM THIS IS TO BE USED AS THE SEARCH STRI
   NG                                           :rem 168
40 DIMA$(300),Q%(300):REM SEARCHED STRING AND FLAG
    ARRAY                                       :rem 173
45 ML=PEEK(55)+256*PEEK(56):REM START ADDRESS
                                                :rem 164
100 PRINT"BUILDING ARRAY"                        :rem 47
110 Q$="GOOD"                                   :rem 177
120 FORL=1TO299                                 :rem 123
130 :                                           :rem 206
140 :A$(L)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"          :rem 49
150 :                                           :rem 208
160 NEXTL                                        :rem 34
170 A$(1)="GARBAGE GOOD MORE GARBAGE"            :rem 46
180 A$(10)="GARB GOOD MORE GARB"                :rem 197
185 A$(70)="GOOD GARBAGE"                        :rem 78
190 A$(100)="GARBAGE GOOD"                      :rem 116
195 A$(250)="GARBAGE GOOD MORE GARBAGE"         :rem 155
200 PRINT"ARRAY FINISHED"                        :rem 44
300 REM{2 SPACES}BASIC SEARCH                   :rem 143
310 PRINT"BASIC SEARCH":TI$="000000"             :rem 25
320 FORL=1TO299                                 :rem 125
330 :FORJ=1TOLEN(A$(L))-LEN(Q$)+1               :rem 114
340 ::IFMID$(A$(L),J,LEN(Q$))=Q$THENQ%(L)=1:NEXTL
                                                :rem 89
350 :NEXTJ                                       :rem 91
360 NEXTL                                        :rem 36
370 PRINTTI;"JIFFIES"                            :rem 67
380 FORL=1TO299                                 :rem 131
390 :IFQ%(L)<>0THENPRINTA$(L)                   :rem 224
395 NEXTL                                        :rem 44
400 REM{2 SPACES}ML SEARCH                      :rem 199
410 PRINT"ML SEARCH":TI$="000000"                :rem 81
420 SYS(ML)                                     :rem 127
430 PRINTTI;"JIFFIES"                            :rem 64
440 FORL=1TO299                                 :rem 128
450 :IFQ%(L)<>0THENPRINTA$(L)                   :rem 221
460 NEXTL                                        :rem 37
```

John W. Ross

Ultrasort

This is probably the fastest sorting program ever published for a home computer. It can alphabetize 1000 items in less than eight seconds. Included is the BASIC loader to place the program in memory, as well as a demonstration that lets you see how "Ultrasort" works.

"Ultrasort" is a sequel. Sort II, I could have called it. It's an improved, faster version of a program first published in the February 1983 issue of *COMPUTE!* magazine. In that article, entitled "Super Shell Sort for PET/CBM," I described a shell sort for the CBM 8032 written entirely in machine language. It worked as expected and was, overall, quite fast. But it had a couple of shortcomings. First of all, it had a clumsy interface with BASIC; that is, the calling sequence was not very efficient. Second, the sorting was performed by a shell short algorithm. This method of sorting, although faster than some other types of sorts, is not the best available.

C.A.R. Hoare's Quicksort algorithm is possibly the fastest yet developed for most applications. So I rewrote my machine language sort program based on the Quicksort algorithm.

Speed Improvements

How much better is it? In order to test the program, I wrote a small sort test program (Program 2), similar to the one in my original article. This program generates a character array containing N items (line 110).

Different items are generated, depending on the value of the random number seed, SD in line 140; SD must be a negative number.

I generated six 1000-element arrays and sorted them using both the shell sort and Ultrasort. Super Shell Sort required an average of 29.60 seconds to sort all 1000 elements, while Ultrasort required an average of only 8.32 seconds. The sorting time has decreased over 300 percent. I don't believe you will find a faster sort for an eight-bit machine anywhere.

The way you start the sort (see Program 2) has also been refined. To run the sort, you simply type:

```
SYS 49152,N,AA$(K)
```

Running the Program

Ultrasort can be used either from within a program or in immediate mode. Running Ultrasort causes N elements from array AA$, starting with element K, to be sorted into ascending order. The sort occurs

in place; there is no additional memory overhead. N and K can be constants or variables, and any character array name can be substituted for AA$.

Before running the sort, though, it must be loaded by BASIC. The appropriate loader is supplied in Program 1. The tradeoff for the increased speed of Ultrasort is increased complexity, especially in machine language. The sort program starts at location 49152 ($C000) on the 64. The increased size, of course, creates a greater possibility of errors when you enter the numbers. In order to minimize this, make sure you read and use "The Automatic Proofreader," Appendix C. This program makes it simple to enter Ultrasort correctly the first time. It's important that you save a copy of Ultrasort before you try to run it. One error can make the computer lock up, and if you haven't saved it first, you might lose all of your typing.

You can add Ultrasort to the end of your own program if you like, simply by changing the first three line numbers (perhaps to 49100, 49110, and 49120), then typing it in as you enter your own program. Or you can use Ultrasort in immediate mode.

Program 2 is a short demonstration that shows how to use Ultrasort, as well as how fast the sorting process really is. To see this demonstration, first load and run Ultrasort. Type NEW, then load and run Program 2, "Sort Test." One hundred random strings will be created for you, and after you press any key, they'll be sorted. You can see them displayed in alphabetical order by pressing any other key. The time taken to sort the strings is displayed at the end of the listing. You can change the number of strings created and sorted by altering the value of N in line 110 of Program 2.

Ultrasort is fast. It will sort 100 strings in about half a second. One thousand strings can be sorted in less than eight seconds. You'll find this program the fastest sort available for the 64, and one that you'll use again and again to sift through lists.

Program 1. Ultrasort

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

```
10 I=49152                                        :rem 236
20 READA:IFA=256THENEND                           :rem 169
30 POKEI,A:I=I+1:GOTO20                           :rem 130
49152 DATA76,100,192,170,170,170,170               :rem 33
49159 DATA170,170,170,170,170,170,170              :rem 86
49166 DATA170,170,170,170,170,170,170              :rem 84
49173 DATA170,170,170,170,170,170,170              :rem 82
49180 DATA170,170,170,170,170,170,170              :rem 80
49187 DATA170,170,170,170,170,170,170              :rem 87
49194 DATA170,170,170,170,170,170,170              :rem 85
```

```
49201 DATA170,170,170,170,170,170,170      :rem 74
49208 DATA170,170,170,170,170,170,170      :rem 81
49215 DATA170,170,170,170,170,170,170      :rem 79
49222 DATA170,170,170,170,170,170,170      :rem 77
49229 DATA170,170,170,170,170,170,170      :rem 84
49236 DATA170,170,170,170,170,170,170      :rem 82
49243 DATA170,170,170,170,170,170,170      :rem 80
49250 DATA170,170,32,253,174,32,158       :rem 244
49257 DATA173,32,247,183,165,20,141       :rem 250
49264 DATA12,192,165,21,141,13,192        :rem 191
49271 DATA32,253,174,32,158,173,56        :rem 205
49278 DATA165,71,233,3,133,75,165         :rem 158
49285 DATA72,233,0,133,76,162,1            :rem 45
49292 DATA173,12,192,157,20,192,173       :rem 252
49299 DATA13,192,157,40,192,169,1         :rem 161
49306 DATA157,60,192,169,0,157,80         :rem 156
49313 DATA192,189,60,192,141,16,192       :rem 255
49320 DATA189,80,192,141,17,192,189         :rem 6
49327 DATA20,192,141,18,192,189,40        :rem 202
49334 DATA192,141,19,192,32,47,195        :rem 208
49341 DATA173,11,192,48,4,202,208         :rem 142
49348 DATA221,96,189,60,192,141,16        :rem 211
49355 DATA192,189,80,192,141,17,192          :rem 8
49362 DATA169,1,141,18,192,169,0          :rem 102
49369 DATA141,19,192,32,101,195,189          :rem 5
49376 DATA20,192,141,18,192,141,14        :rem 195
49383 DATA192,189,40,192,141,19,192          :rem 7
49390 DATA141,15,192,32,47,195,173        :rem 205
49397 DATA11,192,48,3,76,167,193          :rem 119
49404 DATA32,131,195,173,16,192,141       :rem 244
49411 DATA3,192,173,17,192,141,4           :rem 93
49418 DATA192,173,14,192,141,5,192        :rem 203
49425 DATA173,15,192,141,6,192,32         :rem 148
49432 DATA132,194,32,180,194,173,11       :rem 245
49439 DATA192,48,218,173,16,192,141          :rem 6
49446 DATA3,192,173,17,192,141,4          :rem 101
49453 DATA192,173,18,192,141,16,192           :rem 0
49460 DATA173,19,192,141,17,192,169           :rem 4
49467 DATA1,141,18,192,169,0,141           :rem 98
49474 DATA19,192,32,101,195,173,16        :rem 204
49481 DATA192,141,18,192,173,17,192           :rem 2
49488 DATA141,19,192,173,3,192,141        :rem 207
49495 DATA16,192,173,4,192,141,17         :rem 157
49502 DATA192,32,47,195,173,11,192        :rem 202
49509 DATA16,35,173,14,192,141,3           :rem 97
49516 DATA192,173,15,192,141,4,192        :rem 202
49523 DATA173,18,192,141,5,192,173        :rem 203
49530 DATA19,192,141,6,192,32,132         :rem 144
49537 DATA194,32,180,194,173,11,192           :rem 1
```

```
49544 DATA48,152,32,47,195,173,11          :rem 156
49551 DATA192,16,18,173,16,192,141         :rem 202
49558 DATA3,192,173,17,192,141,4           :rem 105
49565 DATA192,32,132,194,32,31,195         :rem 204
49572 DATA76,241,192,234,189,20,192          :rem 6
49579 DATA141,3,192,189,40,192,141         :rem 209
49586 DATA4,192,173,16,192,141,5           :rem 107
49593 DATA192,173,17,192,141,6,192         :rem 211
49600 DATA32,132,194,32,31,195,173         :rem 193
49607 DATA16,192,141,18,192,141,3          :rem 147
49614 DATA192,173,17,192,141,19,192          :rem 1
49621 DATA141,4,192,32,81,195,189          :rem 157
49628 DATA20,192,141,18,192,189,40         :rem 206
49635 DATA192,141,19,192,32,101,195        :rem 251
49642 DATA173,11,192,48,15,189,60          :rem 158
49649 DATA192,141,18,192,189,80,192         :rem 15
49656 DATA141,19,192,32,101,195,169          :rem 2
49663 DATA1,141,18,192,169,0,141            :rem 96
49670 DATA19,192,173,3,192,141,16          :rem 153
49677 DATA192,173,4,192,141,17,192         :rem 212
49684 DATA173,11,192,16,52,189,60          :rem 160
49691 DATA192,232,157,60,192,202,189        :rem 55
49698 DATA80,192,232,157,80,192,32         :rem 215
49705 DATA101,195,173,16,192,157,20        :rem 249
49712 DATA192,173,17,192,157,40,192          :rem 1
49719 DATA32,131,195,32,131,195,202        :rem 246
49726 DATA173,16,192,157,60,192,173          :rem 6
49733 DATA17,192,157,80,192,76,128         :rem 217
49740 DATA194,32,131,195,232,173,16        :rem 250
49747 DATA192,157,60,192,173,17,192         :rem 11
49754 DATA157,80,192,202,189,20,192          :rem 4
49761 DATA232,157,20,192,202,189,40        :rem 249
49768 DATA192,232,157,40,192,202,32        :rem 253
49775 DATA101,195,32,101,195,173,16        :rem 251
49782 DATA192,157,20,192,173,17,192          :rem 6
49789 DATA157,40,192,232,76,162,192         :rem 13
49796 DATA160,3,165,75,133,79,133          :rem 165
49803 DATA81,165,76,133,80,133,82          :rem 156
49810 DATA24,165,79,109,3,192,133          :rem 154
49817 DATA79,165,80,109,4,192,133          :rem 164
49824 DATA80,24,165,81,109,5,192           :rem 107
49831 DATA133,81,165,82,109,6,192          :rem 157
49838 DATA133,82,136,208,223,96,160          :rem 4
49845 DATA0,140,11,192,177,79,141          :rem 152
49852 DATA7,192,177,81,141,8,192           :rem 115
49859 DATA200,152,205,7,192,240,2          :rem 145
49866 DATA176,13,205,8,192,240,21          :rem 153
49873 DATA144,19,238,11,192,76,30          :rem 159
49880 DATA195,205,8,192,240,2,176          :rem 159
```

```
49887 DATA62,206,11,192,76,30,195          :rem 163
49894 DATA140,9,192,160,1,177,79           :rem 117
49901 DATA133,77,200,177,79,133,78         :rem 213
49908 DATA172,9,192,136,177,77,141         :rem 220
49915 DATA10,192,140,9,192,160,1            :rem 93
49922 DATA177,81,133,77,200,177,81         :rem 211
49929 DATA133,78,172,9,192,177,77          :rem 181
49936 DATA200,205,10,192,208,3,76          :rem 145
49943 DATA195,194,144,184,76,224,194        :rem 69
49950 DATA96,160,2,177,79,72,177           :rem 124
49957 DATA81,145,79,104,145,81,136         :rem 217
49964 DATA16,243,96,169,0,141,11           :rem 105
49971 DATA192,173,17,192,205,19,192           :rem 8
49978 DATA144,6,240,8,238,11,192           :rem 111
49985 DATA96,206,11,192,96,173,16          :rem 171
49992 DATA192,205,18,192,144,244,208        :rem 56
49999 DATA238,96,173,16,192,24,109         :rem 228
50006 DATA18,192,141,16,192,173,17         :rem 190
50013 DATA192,109,19,192,141,17,192        :rem 241
50020 DATA96,169,0,141,11,192,56            :rem 87
50027 DATA173,16,192,237,18,192,141        :rem 245
50034 DATA16,192,173,17,192,237,19         :rem 198
50041 DATA192,141,17,192,176,3,206         :rem 187
50048 DATA11,192,96,238,16,192,208         :rem 202
50055 DATA3,238,17,192,96,170,170          :rem 148
50062 DATA170,170,170,170,170,170,170       :rem 71
50069 DATA170,170,170,170,170,170,170       :rem 78
50076 DATA170,170,170,170,170,170,170       :rem 76
50083 DATA170,170,170,170,170,170,170       :rem 74
50090 DATA170,170,170,170,170,170,170       :rem 72
50097 DATA170,170,170,170,170,170,170       :rem 79
50104 DATA170,170,170,170,170,170,170       :rem 68
50111 DATA170,170,170,170,170,81,85        :rem 232
50118 DATA73,67,75,83,79,82,84              :rem 21
50125 DATA32,76,79,65,42,32,32             :rem 252
50132 DATA3,255,50,48,44,82,69             :rem 254
50139 DATA65,68,32,69,82,82,79              :rem 21
50146 DATA82,44,49,56,44,48,48              :rem 12
50153 DATA0,170,170,170,170,81,85          :rem 134
50160 DATA73,67,75,83,79,82,84              :rem 18
50167 DATA32,76,79,65,68,69,82              :rem 25
50174 DATA16,255,256                        :rem 19
```

Program 2. Sort Test

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

```
100 PRINT"{CLR}"                                    :rem 245
110 N=1000                                          :rem 174
120 DIM AA$(N)                                      :rem 178
130 PRINT"CREATING"N" RANDOM STRINGS"                :rem 47
140 SD=-TI:A=RND(SD)                                :rem 183
150 FOR I=1 TO N                                     :rem 37
160 PRINT I"{UP}"                                    :rem 66
170 N1=INT(RND(1)*10+1)                             :rem 221
180 A$=""                                           :rem 127
190 FOR J=1 TO N1                                    :rem 91
200 B$=CHR$(INT(RND(1)*26+65))                       :rem 81
210 A$=A$+B$                                         :rem 43
220 NEXT J                                           :rem 29
230 AA$(I)=A$                                       :rem 119
240 NEXTI                                            :rem 30
250 PRINT "HIT ANY KEY TO START SORT"               :rem 151
260 GET A$:IF A$="" THEN 260                         :rem 83
270 PRINT "SORTING..."                               :rem 26
280 T1=TI                                           :rem 249
290 SYS 49152,N,AA$(1)                              :rem 109
300 T2=TI                                           :rem 243
310 PRINT "DONE"                                    :rem 139
320 PRINT "HIT ANY KEY TO PRINT SORTED STRINGS"
                                                     :rem 71
330 GET A$:IF A$="" THEN 330                         :rem 79
340 FOR I=1 TO N:PRINT I,AA$(I):NEXT                 :rem 27
350 PRINT:PRINT N" ELEMENTS SORTED IN"(T2-T1)/60"S
    ECONDS"                                         :rem 180
```

Dan Carmichael

64 Freeze

Freezing a BASIC program, stopping it in midframe, is a handy feature, especially in game programs. Players get exhausted, want to answer the telephone, or make a sandwich, but don't want to give up that high score. "64 Freeze" lets you stop and restart programs with single keypresses.

It's happened. You're playing a fast-action arcade game, and your hand is cramped from being wrapped too tightly around the joystick. Or your back is giving you spasms again. Or the phone rings and you just have to answer it. But you've got the highest score ever, and if you get up, the game will continue. Unfortunately, the joystick can't run itself, and you'll lose the game.

If you've placed "64 Freeze" in memory, however, you can stop the program at any time by pressing one key. Nothing will be lost; the program simply freezes. Anything on the screen still shows; it just doesn't move. Hitting another key unfreezes the program, restarting it. You can continue with the program from where you left off.

Freeze Keys

Type 64 Freeze in and SAVE it to tape or disk. "The Automatic Proofreader," Appendix C, will make it simple to enter the program correctly the first time.

After loading and running the program, you'll see a display list. You can customize 64 Freeze by selecting your own key combination for freeze and unfreeze. If you want to use the default keys, just hit RETURN twice. The f1 key will then freeze the action, and the f3 key restarts the program. To choose your own keys, enter the appropriate number before hitting RETURN.

The SYS command to access the routine also shows on the screen. Whenever you want to use 64 Freeze, just enter SYS679 in either direct mode or as a program line within your game. If you use the last method, make sure that 64 Freeze has been loaded into memory before you try to call it.

Once you've selected the two control keys, try the freeze function. Type NEW, then load and run a BASIC program. Let it run for a bit, then hit the freeze key (f1 if you chose the default setting). The program immediately pauses. Press the unfreeze key (f3 if default used) to restart the program. That's it.

Interrupting Danger

64 Freeze uses a machine language interrupt by calling the IRQ interrupt vectors at $315–$314. Because of this, if your program also uses interrupts, 64 Freeze may not work. Programs which use machine language in other ways still should be able to access 64 Freeze; it's only interrupts that interfere. Any completely BASIC program can call this routine. We've used this program at COMPUTE! for several months, freezing programs so that we can take photographs of the monitor screen, and we've had difficulties with only a few. All of them used machine language interrupts.

64 Freeze

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

```
10 FORA=679TO714:READB:POKEA,B:NEXT          :rem 212
20 PRINT"{CLR}{WHT}{DOWN}{15 RIGHT}64 FREEZE"
                                             :rem 186
31 PRINT"{YEL}{DOWN}KEY ASSIGNMENTS:":PRINT"{CYN}
   {DOWN}F1= 4{4 SPACES}F3= 5{4 SPACES}F5= 6
   {3 SPACES}F7= 3{6 SPACES}"                :rem 188
32 PRINT"{DOWN}£ = 48{3 SPACES}= = 53{3 SPACES}<
   {SPACE}= 47{3 SPACES}> = 44"              :rem 245
33 PRINT"{DOWN}← = 57{3 SPACES}↑ = 54{3 SPACES}+ =
    40{3 SPACES}- = 43"                      :rem 241
34 PRINT"{DOWN}? = 55{3 SPACES}CRSR{5 SPACES}CRSR"
                                             :rem 163
35 PRINT"{9 SPACES}UP = 7{3 SPACES}RIGHT = 2"
                                              :rem 63
36 PRINT"{DOWN}ENTER THE KEY YOU WISH TO FREEZE TH
   E C64":PRINT"{UP}WITH (SEE TABLE)"         :rem 43
40 INPUT"{3 RIGHT}4{3 LEFT}";K1:POKE715,K1:rem 255
45 PRINT"{DOWN}ENTER THE KEY YOU WISH TO UNFREEZE
   {SPACE}THE":PRINT"C64 WITH (SEE TABLE)" :rem 61
50 INPUT"{3 RIGHT}5{3 LEFT}";K2:POKE716,K2  :rem 4
60 PRINT"{DOWN} TO START PROGRAM{2 SPACES}* SYS679
    *[7]"                                    :rem 36
100 DATA120,169,180,141, 20, 3, 169, 2     :rem 168
110 DATA141,21,3,88, 96, 165, 197, 205     :rem 191
120 DATA203, 2, 240, 3, 76, 49, 234, 32     :rem 73
130 DATA159,255,165,197,205,204, 2, 240     :rem 79
140 DATA243,76,190,2,234, 234, 234, 234     :rem 25
```

Harold D. Vanderpool

64 Merge

DATA statements, subroutines, and even entire files can be merged with other programs using this machine language utility. Not only will it save you time, since you won't have to retype the lines merged, but it allows you to write long programs in pieces, or modules, and then later link them together. "64 Merge" is for disk merges, although an explanation of how to process tape merges is included.

Up to now, if you wanted to put two BASIC programs together, or add DATA statements created by a sound, sprite, or character editor to your own program, you had to retype everything. It was twice the work. Many programmers use a modular approach to program design, and like to use the same subroutine over and over in different programs. Why retype that joystick routine when it's already on a disk?

"64"Merge" lets you avoid all that time at the keyboard. It splices programs together as if you'd typed them in. New lines are placed in the proper order, and if a new line number matches an old line number, the latter is replaced.

To use this program, you need a disk drive. There is a way to merge programs using tape, which is also explained in this article, but the method is more cumbersome. You can't use the 64 Merge program with tape.

Type in and save 64 Merge. Use Appendix C, "The Automatic Proofreader," to make sure it's entered correctly. As you type in the DATA statements in lines 200–240, you'll notice that there are strange-looking number and letter combinations. You probably haven't typed in a listing like this before. The two-character combinations do mean something; they are values converted into hexadecimal notation. You don't have to know how to convert from decimal to hexadecimal, or even anything about machine language, to enter this program. Simply type it in exactly as it appears. As soon as you've done that, you're ready to begin merging programs.

ASCII Files

In order to put together two programs using 64 Merge, you first have to make an ASCII file of one of them. It's probably easier if you choose the shorter program for this.

Load the program or routine you wish to merge and then type this line in direct mode (without a line number):

OPEN 2,8,3, "*filename*,SEQ,W":CMD2:LIST[n1-n2]

(The n1–n2 range in brackets after the LIST command is optional. If you want, you can include a number range here, and only that portion of the program will be merged. Make sure you do include the command LIST, however, even if you don't specify a range. Omitting the range option means the entire program will be merged later.)

As soon as you press RETURN, an ASCII file is created on the disk in the drive under the filename you specified. You can put the ASCII file on any disk that has room. The cursor will return to the screen, and you should type:

```
PRINT#2:CLOSE2
```

Press RETURN, and the red indicator light on the drive will go off. The file is now CLOSEd, and it's ready to use in the merge operation.

Prompted SYSing

Type NEW, then load and run 64 Merge. The program is completely relocatable. In other words, it can be placed anywhere in memory, as long as there's room. By default it occupies part of the cassette buffer, an area you don't normally use when you have a disk drive. Other suitable locations are in the range from 49152 to 53247. If you don't want it placed in the cassette buffer, the next best place would be at 49152.

The first prompt you'll see when the program has been run is the starting address of the routine. If you want to relocate it, enter the appropriate memory location and press RETURN. Hitting the RETURN key without typing anything in simply tells the program to use the default area, the cassette buffer.

At the next prompt, enter the filename you used earlier when you created your ASCII file. Hitting the RETURN key then runs 64 Merge. The machine language data is POKEd into memory, and several SYS commands are displayed on the screen. You'll be using these SYSs in a moment. If you relocate the program to another area of memory, you should jot these SYS addresses down, for they'll vary from what shows below. Relocating the program to different areas will display different locations to SYS. It's simpler if you just leave the program in the cassette buffer.

Merging

Load the program you want merged to (not the one you created the ASCII file from). Make sure the disk with the ASCII file is in the disk drive and then type:

```
SYS 882
SYS 904
```

hitting RETURN after each. (Remember that if you relocated the program, all the SYSs will be different from what shows here.)

Press the SHIFT and CLR/HOME keys together. As soon as the screen clears, type:

SYS 828

to actually merge the two programs. The drive will make some noise and the red light will remain on until you CLOSE the operation by entering:

SYS 882:SYS 910

What was once two BASIC programs is now one. LIST it to check. You've successfully completed a disk merge. You can run or modify the merged program, or save it normally under a new filename.

Tape Merge

The method used to merge two programs using the disk drive was first explored by Raeto West in his book *Programming the PET/CBM*. My program is a Commodore 64 version of that process. Merging programs on tape, however, was best described by Jim Butterfield in "BASIC Program Merges: PET and VIC," which appeared in the June 1982 issue of *COMPUTE!* magazine. If you don't have a copy of that issue, here's a review of the process.

Create an ASCII version of the program to be merged by loading it into memory, making sure a blank tape is in the Datassette, and typing the following in direct mode:

OPEN2,1,1,"*filename*":CMD2:LIST[n1-n2]

The LIST command is necessary, but the number range is optional. You can specify a range of lines if you want to merge only part of a program, such as a subroutine, into another program.

Press the RETURN key and obey the PRESS RECORD & PLAY ON TAPE prompt. The tape drive will start and stop, eventually stopping completely. Again in direct mode, enter:

PRINT#2:CLOSE2

When you press RETURN, the tape will move again for a bit, then stop. The screen should show the READY message. You've just created an ASCII file on tape. Set this tape aside.

Load the program to be merged to, place the tape with the ASCII file back in the Datassette, and type:

POKE19,1:OPEN2

Press RETURN, then the tape PLAY button. The tape will move, eventually displaying the ASCII filename you earlier specified. Do not release the PLAY button on the tape player. The next few key entries must be followed *exactly*. Press the SHIFT and CLR/HOME keys together, wait for the screen to clear, and then press the cursor down key *three* times. You'll see the cursor blinking on line four. In direct mode, enter:

PRINT"{home}":POKE198,1:POKE631,13:POKE153,1

After the tape has moved and stopped a few times, you see either the ?SYNTAX ERROR or ?OUT OF DATA message on the screen. Ignore it; it's not a real error. Close the file by typing:

CLOSE2

When RETURN is pressed, the two programs have been merged. You can save, run, or modify the merged version.

That's all there is to it. By using this utility, you'll find it easy to splice two BASIC programs together. You'll never have to retype your favorite subroutines or DATA statements again.

64 Merge

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

```
5 PRINT"{CLR}{DOWN}"TAB(10)"COMMODORE 64 MERGE
  {DOWN}"                                    :rem 7
10 INPUT"START ADDRESS";AD                  :rem 137
20 IFAD<820THENAD=828:REM DEFAULTS TO CASSETTE BUF
   FER                                       :rem 14
30 INPUT"{DOWN}FILENAME";N$:L=LEN(N$):IFN$=""THEN3
   0                                         :rem 233
40 PRINT"{DOWN}WAIT ONE MINUTE !"            :rem 80
50 FORI=0TO89:READA$:A=ASC(A$)-48:IFA>9THENA=A-7
                                             :rem 130
60 B=ASC(RIGHT$(A$,1))-48:IFB>9THENB=B-7     :rem 49
70 N=A*16+B:POKEAD+I,N:NEXT:BI=I+AD          :rem 129
80 FORI=1TOL:POKEBI+I-1,ASC(MID$(N$,I,1)):NEXT:BI=
   I+BI                                      :rem 28
90 FORI=0TO5:READC:POKEBI+I-1,C:NEXT         :rem 37
100 POKEAD+55,L+6:AA=AD+88:A1%=AA/256:A2=AA-A1%*25
    6                                        :rem 31
110 POKEAD+57,A2:POKEAD+59,A1%               :rem 198
120 PRINT"{DOWN}OPEN":PRINT"{2 SPACES}SYS"AD+54:PR
    INT"{2 SPACES}SYS"AD+76                  :rem 241
130 PRINT"{DOWN}MERGE":PRINT"{2 SPACES}PRESS [CLEA
    R]":PRINT"{2 SPACES}SYS"AD               :rem 44
140 PRINT"{DOWN}CLOSE":PRINT"{2 SPACES}SYS"AD+54":
    SYS"AD+82                                :rem 167
```

```
200 DATA A9,08,20,B4,FF,A9,03,20,96,FF,A2,00,20,A5
    ,FF,C9,0A,F0,F9,C9,0D,F0,0A              :rem 228
210 DATA 9D,00,02,E8,E0,51,F0,14,D0,EB,8D,77,02,20
    ,CA,AA,A9,13,20,D2,FF,A9,01              :rem 179
220 DATA 85,C6,4C,86,A4,4C,48,B2,A9,20,A2,20,A0,20
    ,20,BD,FF,A9,02,A2,08,A0,03              :rem 144
230 DATA 20,BA,FF,20,C0,FF,60,A2,02,20,C6,FF,60,A9
    ,02,20,C3,FF,60,30,3A                    :rem 118
240 DATA 44,83,44,81,20,20,20                :rem 130
```

Richard Mansfield

RAMtest

How can you test your computer's RAM chips to make sure they're working? For all you know, they may be unreliable. This short machine language program tests every RAM memory cell, from location 2048 to address 40960, and tells you if each is working properly.

Don't let anyone tell you that there's something impossibly complex about machine language. It *can* be harder to debug (locate and fix errors) than BASIC is, but it's not inherently more difficult to learn or to write.

You just need to memorize some new commands; obtain and practice with some new tools (assemblers, disassemblers, monitors); and pick up a few new programming techniques. Discovering this for yourself, that machine language can be an easy — and fascinating — way to communicate with your computer, may spur you on to write your own ML programs.

"RAMtest," the program below, is an example of one of the most common ways that machine language routines are printed in books and magazines. This kind of program is called a *BASIC loader*. The value of loaders is that the user need not understand anything about the machine language program. It's the easy way to use machine language. Just type in the BASIC program as it appears, type RUN, and the machine language (the numbers in the DATA statements) is POKEd into memory for you.

Strange Strings

RAMtest is a useful program: It tests your Random Access Memory (RAM) to be sure that every cell is operating correctly. RAM chips are generally quite reliable, but you might have one fail on you. There are various odd things that can happen during a program RUN as the result of a faulty RAM chip. One sign would be the sudden appearance of strange strings. For example, you might type A$ = "ABCDEFG" and when you asked to see A$ (by typing ?A$), you would get ABC)EFG or something.

Here's how to use RAMtest:

1. Type it in.

2. Type SAVE (to keep a copy on tape or disk).

3. Type RUN. The DATA will be loaded into a safe area of your computer which is not part of BASIC RAM. We're loading the machine language program into decimal address 828–1019. This is the cassette buffer RAM, and it's unused by BASIC except during Datassette operations. We can't store the machine language program in normal

BASIC RAM because we're going to fill each memory cell with all 256 possible numbers as our test. That would cause the program to test — and thereby obliterate — itself. The cassette buffer is a popular, safe place to put machine language since it is out of BASIC's way.

4. After you see READY on the screen, your machine language is sitting down in lower RAM memory (decimal 864–995), waiting for you to activate it. You send control of the computer to a machine language program by using BASIC's SYS command. However, machine language programs do not necessarily start with the first number in their sequence. The *entry point* could be anywhere within the routine. Unlike BASIC, which always begins with the lowest line number, machine language might store text messages or other information below the entry point. That's the case in the RAMtest program. To start it going, type SYS 884.

A Vibrating Square

If all your DATA numbers were correctly typed in, you should now see two things happening onscreen. Up in the left corner you'll see a vibrating square. This is a visual demonstration of what's happening to each of your RAM memory cells in turn. As each number from 0 to 255 is POKEd into each cell of the computer, it's also being POKEd into the first screen memory cell so you can see it happening. (Machine language POKEs are called STA, meaning STore the Accumulator.)

The other thing you'll notice is that the decimal address range currently being tested appears onscreen. This program tests cells from 2048 up to address 40960. At the conclusion, the words TEST OVER will signify that every memory cell tested has correctly stored every possible number.

Now, type LIST. You can see the effect of our mass POKEs. For a line number you get 65535. (However, for technical reasons, you can't actually use line numbers larger than 63999 in BASIC.) Line numbers are always stored in two-byte units, and this is the biggest number that the computer can hold within two memory bytes. Following that are more than 200 pi symbols. This is the symbol you get by typing ?CHR$(255). We're not seeing screen RAM memory when we ask for a LIST. Instead, we see a translation of a BASIC program. The series of 255's appears, after this translation, as pi symbols. It means that each of these cells — you're looking at the bottom of BASIC RAM where BASIC programs start — is now holding a number 255 after having held everything from 0 up to 255 during the test.

If you want to regain control and return to normal BASIC conditions after this test, you'll need to POKE 2048,0. The very first cell in BASIC RAM must contain a zero for things to work correctly.

Most likely, your RAM memory passed the test. Just to see what would happen if there were a bad cell, we can make the test try to POKE into Read Only Memory (ROM). It will try, but ROM is protected against being written over, so the attempted POKE will fail and it will appear to the RAMtest program that there is a bad cell. To try this, LOAD RAMtest from your disk or tape. Then RUN. Then type POKE 885,245. This will set the testing to start at memory cell 62720, and you'll see the results when you start the test with SYS 884.

Tortoise and Hare

You *could* write a program in BASIC to perform this same test, but you'd need to start the test higher up in memory and you'd need to leave it RUNning overnight. The great speed of machine language execution makes it ideal for large tasks like RAM testing. Machine language does have advantages, even if it may seem more difficult to write.

RAMtest

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

```
800 FOR ADRES=864 TO 995:READ DATTA:POKE ADRES,DAT
    TA:NEXT ADRES                              :rem 53
864 DATA 84, 69, 83, 84, 32, 79                :rem 31
870 DATA 86, 69, 82, 32, 32, 66                :rem 18
876 DATA 65, 68, 32, 66, 89, 84                :rem 34
882 DATA 69, 32, 169, 8, 133, 58               :rem 72
888 DATA 169, 0, 133, 57, 160, 0               :rem 56
894 DATA 24, 141, 0, 4, 145, 57                 :rem 1
900 DATA 209, 57, 240, 21, 152, 72             :rem 144
906 DATA 165, 58, 72, 32, 179, 3                :rem 64
912 DATA 104, 133, 58, 104, 168, 169            :rem 255
918 DATA 0, 230, 57, 208, 7, 230                :rem 47
924 DATA 58, 24, 105, 1, 208, 221               :rem 94
930 DATA 200, 208, 218, 32, 193, 3              :rem 142
936 DATA 230, 58, 165, 58, 201, 160             :rem 205
942 DATA 144, 207, 76, 208, 3, 162              :rem 154
948 DATA 10, 160, 0, 185, 106, 3                :rem 43
954 DATA 32, 210, 255, 200, 202, 208            :rem 236
960 DATA 246, 72, 152, 72, 169, 32              :rem 160
966 DATA 32, 210, 255, 32, 201, 189             :rem 201
972 DATA 104, 168, 104, 96, 169, 13             :rem 212
978 DATA 32, 210, 255, 160, 0, 185              :rem 151
984 DATA 96, 3, 32, 210, 255, 200                :rem 99
990 DATA 192, 10, 208, 245, 96, 0               :rem 105
```

Appendices

A Beginner's Guide to Typing In Programs

What Is a Program?

A computer cannot perform any task by itself. Like a car without gas, a computer has potential, but without a program, it isn't going anywhere. Most of the programs published in this book are written in a computer language called BASIC. BASIC is easy to learn and is built into all Commodore 64s.

BASIC Programs

Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, or number is significant. A common mistake is substituting a letter such as O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, you must enter all punctuation such as colons and commas just as they appear in the book. Spacing can be important. To be safe, type in the listings *exactly* as they appear.

Braces and Special Characters

The exception to this typing rule is when you see the braces, such as {DOWN}. Anything within a set of braces is a special character or characters that cannot easily be listed on a printer. When you come across such a special statement, refer to Appendix B, "How to Type In Programs."

About DATA Statements

Some programs contain a section or sections of DATA statements. These lines provide information needed by the program. Some DATA statements contain actual programs (called machine language); others contain graphics codes. These lines are especially sensitive to errors.

If a single number in any one DATA statement is mistyped, your machine could lock up, or crash. The keyboard and STOP key may seem dead, and the screen may go blank. Don't panic — no damage is done. To regain control, you have to turn off your computer, then turn it back on. This will erase whatever program was in memory, *so always save a copy of your program before you run it*. If your computer crashes, you can load the program and look for your mistake.

Sometimes a mistyped DATA statement will cause an error message when the program is run. The error message may refer to the program line that reads the data. *The error is still in the DATA statements, though.*

Get to Know Your Machine

You should familiarize yourself with your computer before attempting to type in a program. Learn the statements you use to store and retrieve programs from tape or disk. You'll want to save a copy of your program, so that you won't have to type it in every time you want to use it. Learn to use your machine's editing functions. How do you change a line if you made a mistake? You can always retype the line, but you at least need to know how to backspace. Do you know how to enter reverse video, lowercase, and control characters? It's all explained in your computer's manuals.

A Quick Review

1. Type in the program a line at a time, in order. Press RETURN at the end of each line. Use the DELete key to correct mistakes.
2. Check the line you've typed against the line in the book. You can check the entire program again if you get an error when you run the program.

How to Type In Programs

To make it easy to know exactly what to type when entering one of these programs into your computer, we have established the following listing conventions.

Generally, Commodore 64 program listings will contain words within braces which spell out any special characters: {DOWN} would mean to press the cursor down key. {5 SPACES} would mean to press the space bar five times.

To indicate that a key should be shifted (hold down the SHIFT key while pressing the other key), the key would be underlined in our listings. For example, S would mean to type the S key while holding the SHIFT key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (e.g., {10 N}), you should type the key as many times as indicated (in our example, 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 necessary.

Rarely, you'll see a solitary letter of the alphabet enclosed in braces. These characters can be entered by holding down the CTRL key while typing the letter in the braces. For example, {A} would indicate that you should press CTRL-A.

About the quote mode: You know that you can move the cursor around the screen with the CRSR keys. Sometimes a programmer will want to move the cursor under program control. That's why you see all the {LEFT}'s, {HOME}'s, and {BLU}'s in our programs. The only way the computer can tell the difference between direct and programmed cursor control is the quote mode.

Once you press the quote (the double quote, SHIFT-2), you are in the quote mode. If you type something and then try to change it by moving the cursor left, you'll only get a bunch of reverse video lines. These are the symbols for cursor left. The only editing key that isn't programmable is the DEL key; you can still use DEL to back up and edit the line. Once you type another quote, you are out of quote mode.

You also go into quote mode when you INSerT spaces into a line. In any case, the easiest way to get out of quote mode is to just press RETURN. You'll then be out of quote mode and you can cursor up to the mistyped line and fix it.

Use the following table when entering cursor and color control keys:

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 + 3	
{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	

Charles Brannon

The Automatic Proofreader

"The Automatic Proofreader" will help you type in program listings without typing mistakes. It is a short error-checking program that hides itself in memory. When activated, it lets you know immediately after typing a line from a program listing if you've made a mistake. Please read these instructions carefully before typing any programs in this book.

Preparing the Proofreader

1. Using the listing below, type in the Proofreader. Be very careful when entering the DATA statements — don't type an l instead of a 1, an O instead of a 0, extra commas, etc.

2. Save the Proofreader on tape or disk at least twice *before running it for the first time*. This is very important because the Proofreader erases part of itself when you first type RUN.

3. After the Proofreader is saved, type RUN. It will check itself for typing errors in the DATA statements and warn you if there's a mistake. Correct any errors and save the corrected version. Keep a copy in a safe place — you'll need it again and again, every time you enter a program from this book, *COMPUTE!'s Gazette*, or *COMPUTE!* magazine.

4. When a correct version of the Proofreader is run, it activates itself. You are now ready to enter a program listing. If you press RUN/STOP–RESTORE, the Proofreader is disabled. To reactivate it, just type the command SYS 886 and press RETURN.

Using the Proofreader

All listings in this book have a *checksum number* appended to the end of each line, for example, :rem 123. *Don't enter this statement when typing in a program*. It is just for your information. The rem makes the number harmless if someone does type it in. It will, however, use up memory if you enter it, and it will confuse the Proofreader, even if you entered the rest of the line correctly.

When you type in a line from a program listing and press RETURN, the Proofreader displays a number at the top of your screen. *This checksum number must match the checksum number in the printed listing*. If it doesn't, it means you typed the line differently than the way it is listed. Immediately recheck your typing. Remember, don't type the rem statement with the checksum number; it is

published only so you can check it against the number which appears on your screen.

The Proofreader is not picky with spaces. It will not notice extra spaces or missing ones. This is for your convenience, since spacing is generally not important. But occasionally proper spacing is important, so be extra careful with spaces, since the Proofreader will catch practically everything else that can go wrong.

There's another thing to watch out for: If you enter the line by using abbreviations for commands, the checksum will not match up. But there is a way to make the Proofreader check it. After entering the line, LIST it. This eliminates the abbreviations. Then move the cursor up to the line and press RETURN. It should now match the checksum. You can check whole groups of lines this way.

Special Tape SAVE Instructions

When you're done typing a listing, you must disable the Proofreader before saving the program on tape. Disable the Proofreader by pressing RUN/STOP–RESTORE (hold down the RUN/STOP key and sharply hit the RESTORE key). This procedure is not necessary for disk, but you must disable the Proofreader this way before a tape SAVE.

SAVE to tape erases the Proofreader from memory, so you'll have to load and run it again if you want to type another listing. SAVE to disk does not erase the Proofreader.

Hidden Perils

The Proofreader's home in the 64 is not a very safe haven. Since the cassette buffer is wiped out during tape operations, you need to disable the Proofreader with RUN/STOP–RESTORE before you save your program. This applies only to tape use. Disk users have nothing to worry about.

Not so for 64 owners with tape drives. What if you type in a program in several sittings? The next day, you come to your computer, load and run the Proofreader, then try to load the partially completed program so you can add to it. But since the Proofreader is trying to hide in the cassette buffer, it is wiped out!

What you need is a way to load the Proofreader after you've loaded the partial program. The problem is, a tape load to the buffer destroys what it's supposed to load.

After you've typed in and run the Proofreader, enter the following lines in direct mode (without line numbers) exactly as shown:

```
A$ = "PROOFREADER.T": B$ = "{ 10 SPACES }": FOR X = 1 TO 4:
A$ = A$ + B$: NEXTX
```

```
FOR X = 886 TO 1018: A$ = A$ + CHR$ (PEEK(X)): NEXTX
OPEN 1,1,1,A$:CLOSE1
```

After you enter the last line, you will be asked to press record and play on your cassette recorder. Put this program at the beginning of a new tape. This gives you a new way to load the Proofreader. Anytime you want to bring the Proofreader into memory without disturbing anything else, put the cassette in the tape drive, rewind, and enter:

```
OPEN1:CLOSE1
```

You can now start the Proofreader by typing SYS 886. To test this, PRINT PEEK (886) should return the number 173. If it does not, repeat the steps above, making sure that A$ ("PROOFREADER.T") contains 13 characters and that B$ contains 10 spaces.

You can now reload the Proofreader into memory whenever LOAD or SAVE destroys it, restoring your personal typing helper.

Replace Original Proofreader

If you typed in the original version of the Proofreader from the October 1983 issue of *COMPUTE!'s Gazette*, you should replace it with the improved version below.

Automatic Proofreader

```
100 PRINT"{CLR}PLEASE WAIT...":FORI=886TO1018:READ
    A:CK=CK+A:POKEI,A:NEXT
110 IF CK<>17539 THEN PRINT"{DOWN}YOU MADE AN ERRO
    R":PRINT"IN DATA STATEMENTS.":END
120 SYS886:PRINT"{CLR}{2 DOWN}PROOFREADER ACTIVATE
    D.":NEW
886 DATA 173,036,003,201,150,208
892 DATA 001,096,141,151,003,173
898 DATA 037,003,141,152,003,169
904 DATA 150,141,036,003,169,003
910 DATA 141,037,003,169,000,133
916 DATA 254,096,032,087,241,133
922 DATA 251,134,252,132,253,008
928 DATA 201,013,240,017,201,032
934 DATA 240,005,024,101,254,133
940 DATA 254,165,251,166,252,164
946 DATA 253,040,096,169,013,032
952 DATA 210,255,165,214,141,251
958 DATA 003,206,251,003,169,000
964 DATA 133,216,169,019,032,210
970 DATA 255,169,018,032,210,255
976 DATA 169,058,032,210,255,166
```

```
982 DATA 254,169,000,133,254,172
988 DATA 151,003,192,087,208,006
994 DATA 032,205,189,076,235,003
1000 DATA 032,205,221,169,032,032
1006 DATA 210,255,032,210,255,173
1012 DATA 251,003,133,214,076,173
1018 DATA 003
```

Charles Brannon

Using the Machine Language Editor: MLX

Remember the last time you typed in the BASIC loader for a long machine language program? You typed in hundreds of numbers and commas. Even then, you couldn't be sure if you typed it in right. So you went back, proofread, tried to run the program, crashed, went back and proofread again, corrected a few typing errors, ran again, crashed again, rechecked your typing . . . . Frustrating, wasn't it?

Until now, though, that has been the best way to get machine language into your computer. Unless you happen to have an assembler and are willing to tangle with machine language on the assembly level, it is much easier to enter a BASIC program that reads DATA statements and POKEs the numbers into memory.

Some of these "BASIC loaders" use a checksum to see if you've typed the numbers correctly. The simplest checksum is just the sum of all the numbers in the DATA statements. If you make an error, your checksum does not match up with the total. Some programmers make your task easier by including checksums every few lines, so you can locate your errors more easily.

Now, MLX comes to the rescue. MLX is a great way to enter all those long machine language programs with a minimum of fuss. MLX lets you enter the numbers from a special list that looks similar to DATA statements. It checks your typing on a line-by-line basis. It won't let you enter illegal characters when you should be typing numbers. It won't let you enter numbers greater than 255 (forbidden in ML). It will prevent you from entering the numbers on the wrong line. In short, MLX makes proofreading obsolete.

Tape or Disk Copies

In addition, MLX generates a ready-to-use copy of your machine language program on tape or disk. You can then use the LOAD command to read the program into the computer, as with any other program. Specifically, you enter:

LOAD "program name",1,1(for tape)
or
LOAD "program name",8,1(for disk)

To start the program, you need to enter a SYS command that transfers control from BASIC to your machine language program.

The starting SYS is always listed in the article which presents the machine language program in MLX format.

Using MLX

Type in and save MLX (you'll want to use it in the future). When you're ready to type in the machine language program, run MLX. MLX asks you for two numbers: the starting address and the ending address. These numbers are given in the article accompanying the ML program you're typing. For example, the addresses for "BASIC Aid" should be 49152 and 52997 respectively.

You'll see a prompt. The prompt is the current line you are entering from the MLX-format listing. It increases by six each time you enter a line. That's because each line has seven numbers — six actual data numbers plus a checksum number. The checksum verifies that you typed the previous six numbers correctly. If you enter any of the six numbers wrong, or enter the checksum wrong, the 64 sounds a buzzer and prompts you to reenter the line. If you enter the line correctly, a bell tone sounds and you continue to the next line.

A Special Editor

You are not using the normal 64 BASIC editor with MLX. For example, it will accept only numbers as input. If you make a typing error, press the INST/DEL key; the entire number is deleted. You can press it as many times as necessary, back to the start of the line. If you enter three-digit numbers as listed, the computer automatically prints the comma and goes on to accept the next number. If you enter less than three digits, you can press either the space bar or the RETURN key to advance to the next number. The checksum automatically appears in reverse video for emphasis.

To make it even easier to enter these numbers, MLX redefines part of the keyboard as a numeric keypad (lines 581–584).

```
   U  I  O                    7  8  9
H  J  K  L     becomes     0  4  5  6
   M  ,  .                    1  2  3
```

When testing it, I've found MLX to be an extremely easy way to enter long listings. With the audio cues provided, you don't even have to look at the screen if you're a touch-typist.

Done at Last!

When you get through typing, assuming you type your machine language program all in one session, you can then save the completed

and bug-free program to tape or disk. Follow the instructions displayed on the screen. If you get any error messages while saving, you probably have a bad disk, or the disk is full, or you made a typo when entering the MLX program. (Sorry, MLX can't check itself!)

Command Control

You don't have to enter the whole ML program in one sitting. MLX lets you enter as much as you want, save it, and then reload the file from tape or disk later. MLX recognizes these commands:

SHIFT-S:Save
SHIFT-L:Load
SHIFT-N:New Address
SHIFT-D:Display

Hold down SHIFT while you press the appropriate key. MLX jumps out of the line you've been typing, so I recommend you do it at a prompt. Use the Save command to store what you've been working on. It will save on tape or disk as if you've finished, but the tape or disk won't work, of course, until you finish typing. Remember what address you stopped on. The next time you run MLX, answer all the prompts as you did before, then insert the disk or tape containing the stored file. When you get the entry prompt, press SHIFT-L to reload the partly completed file into memory. Then use the New Address command (SHIFT-N) to resume typing.

New Address and Display

After you press SHIFT-N, enter the address where you previously stopped. The prompt will change, and you can then continue typing. Always enter a New Address that matches up with one of the line numbers in the special listing, or else the checksums won't match up. You can use the Display command to display a section of your typing. After you press SHIFT-D, enter two addresses within the line number range of the listing. You can abort the listing by pressing any key.

Tricky Stuff

The special commands may seem a little confusing, but as you work with MLX, they will become valuable. For example, what if you forgot where you stopped typing? Use the Display command to scan memory from the beginning to the end of the program. When you reach the end of your typing, the lines will contain a random pattern of numbers, quite different from what should be there. When you see the end of your typing, press any key to stop the listing. Use the New Address command to continue typing from the proper location.

You can use the Save and Load commands to make copies of the complete machine language program. Use the Load command to reload the tape or disk, then insert a new tape or disk and use the Save command to create a new copy. When resaving on disk it is best to use a different filename each time you save. For example, I like to number my work and use filenames such as AID1, AID2, AID3, and so on.

One quirk about tapes made with the MLX Save command: When you load them, the message "FOUND program" may appear twice. The tape will load just fine, however.

I think you'll find MLX to be a true labor-saving program. Since it has been tested by entering actual programs, you can count on it as an aid for generating bug-free machine language. Be sure to save MLX; it will be used for future applications in COMPUTE! Books, *COMPUTE!* magazine, and *COMPUTE!'s Gazette*.

MLX

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

```
10 REM LINES CHANGED FROM MLX VERSION 2.00 ARE 750
   ,765,770 AND 860                              :rem 50
100 PRINT"{CLR}{6}";CHR$(142);CHR$(8);:POKE53281,1
    :POKE53280,1                                 :rem 67
101 POKE 788,52:REM DISABLE RUN/STOP             :rem 119
110 PRINT"{RVS}{39 SPACES}";                     :rem 176
120 PRINT"{RVS}{14 SPACES}{RIGHT}{OFF}{*}£{RVS}
    {RIGHT} {RIGHT}{2 SPACES}{*}{OFF}{*}£{RVS}£
    {RVS}{14 SPACES}";                           :rem 250
130 PRINT"{RVS}{14 SPACES}{RIGHT} {G}{RIGHT}
    {2 RIGHT} {OFF}£{RVS}£{*}{OFF}{*}{RVS}
    {14 SPACES}";                                :rem 35
140 PRINT"{RVS}{41 SPACES}"                      :rem 120
200 PRINT"{2 DOWN}{PUR}{BLK} MACHINE LANGUAGE EDIT
    OR VERSION 2.01{5 DOWN}"                     :rem 237
210 PRINT"{5}{2 UP}STARTING ADDRESS?{8 SPACES}
    {9 LEFT}";                                   :rem 143
215 INPUTS:F=1-F:C$=CHR$(31+119*F)               :rem 166
220 IFS<256OR(S>40960ANDS<49152)ORS>53247THENGOSUB
    3000:GOTO210                                 :rem 235
225 PRINT:PRINT:PRINT                            :rem 180
230 PRINT"{5}{2 UP}ENDING ADDRESS?{8 SPACES}
    {9 LEFT}";:INPUTE:F=1-F:C$=CHR$(31+119*F)
                                                 :rem 20
240 IFE<256OR(E>40960ANDE<49152)ORE>53247THENGOSUB
    3000:GOTO230                                 :rem 183
250 IFE<STHENPRINTC$;"{RVS}ENDING < START
    {2 SPACES}":GOSUB1000:GOTO 230               :rem 176
```

```
260 PRINT:PRINT:PRINT                                    :rem 179
300 PRINT"{CLR}";CHR$(14):AD=S:POKEV+21,0                :rem 225
310 A=1:PRINTRIGHT$("0000"+MID$(STR$(AD),2),5);":"
    ;                                                    :rem 33
315 FORJ=ATO6                                            :rem 33
320 GOSUB570:IFN=-1THENJ=J+N:GOTO320                     :rem 228
390 IFN=-211THEN 710                                     :rem 62
400 IFN=-204THEN 790                                     :rem 64
410 IFN=-206THENPRINT:INPUT"{DOWN}ENTER NEW ADDRES
    S";ZZ                                                :rem 44
415 IFN=-206THENIFZZ<SORZZ>ETHENPRINT"{RVS}OUT OF
    {SPACE}RANGE":GOSUB1000:GOTO410                      :rem 225
417 IFN=-206THENAD=ZZ:PRINT:GOTO310                      :rem 238
420 IF N<>-196 THEN 480                                  :rem 133
430 PRINT:INPUT"DISPLAY:FROM";F:PRINT,"TO";:INPUTT
                                                         :rem 234
440 IFF<SORF>EORT<SORT>ETHENPRINT"AT LEAST";S;"
    {LEFT}, NOT MORE THAN";E:GOTO430                     :rem 159
450 FORI=FTOTSTEP6:PRINT:PRINTRIGHT$("0000"+MID$(S
    TR$(I),2),5);":";                                    :rem 30
451 FORK=0TO5:N=PEEK(I+K):PRINTRIGHT$("00"+MID$(ST
    R$(N),2),3);",";                                     :rem 66
460 GETA$:IFA$>""THENPRINT:PRINT:GOTO310                 :rem 25
470 NEXTK:PRINTCHR$(20);:NEXTI:PRINT:PRINT:GOTO310
                                                         :rem 50
480 IFN<0 THEN PRINT:GOTO310                             :rem 168
490 A(J)=N:NEXTJ                                         :rem 199
500 CKSUM=AD-INT(AD/256)*256:FORI=1TO6:CKSUM=(CKSU
    M+A(I))AND255:NEXT                                   :rem 200
510 PRINTCHR$(18);:GOSUB570:PRINTCHR$(146);:rem 94
511 IFN=-1THENA=6:GOTO315                                :rem 254
515 PRINTCHR$(20):IFN=CKSUMTHEN530                       :rem 122
520 PRINT:PRINT"LINE ENTERED WRONG : RE-ENTER":PRI
    NT:GOSUB1000:GOTO310                                 :rem 176
530 GOSUB2000                                            :rem 218
540 FORI=1TO6:POKEAD+I-1,A(I):NEXT:POKE54272,0:POK
    E54273,0                                             :rem 227
550 AD=AD+6:IF AD<E THEN 310                             :rem 212
560 GOTO 710                                             :rem 108
570 N=0:Z=0                                               :rem 88
580 PRINT"[£]";                                           :rem 81
581 GETA$:IFA$=""THEN581                                  :rem 95
582 AV=-(A$="M")-2*(A$=",")-3*(A$=".")-4*(A$="J")-
    5*(A$="K")-6*(A$="L")                                :rem 41
583 AV=AV-7*(A$="U")-8*(A$="I")-9*(A$="O"):IFA$="H
    "THENA$="0"                                          :rem 134
584 IFAV>0THENA$=CHR$(48+AV)                             :rem 134
585 PRINTCHR$(20);:A=ASC(A$):IFA=13ORA=44ORA=32THE
    N670                                                 :rem 229
```

```
590 IFA>128THENN=-A:RETURN                          :rem 137
600 IFA<>20 THEN 630                                :rem 10
610 GOSUB690:IFI=1ANDT=44THENN=-1:PRINT"{OFF}
    {LEFT} {LEFT}";:GOTO690                         :rem 62
620 GOTO570                                         :rem 109
630 IFA<48ORA>57THEN580                             :rem 105
640 PRINTA$;:N=N*10+A-48                            :rem 106
650 IFN>255 THEN A=20:GOSUB1000:GOTO600             :rem 229
660 Z=Z+1:IFZ<3THEN580                              :rem 71
670 IFZ=0THENGOSUB1000:GOTO570                      :rem 114
680 PRINT",";:RETURN                                :rem 240
690 S%=PEEK(209)+256*PEEK(210)+PEEK(211)            :rem 149
691 FORI=1TO3:T=PEEK(S%-I)                          :rem 67
695 IFT<>44ANDT<>58THENPOKES%-I,32:NEXT             :rem 205
700 PRINTLEFT$("{3 LEFT}",I-1);:RETURN              :rem 7
710 PRINT"{CLR}{RVS}*** SAVE ***{3 DOWN}"           :rem 236
715 PRINT"{2 DOWN}(PRESS {RVS}RETURN{OFF}(ALONE TO
     CANCEL SAVE){DOWN}"                            :rem 106
720 F$="":INPUT"{DOWN} FILENAME";F$:IFF$=""THENPRI
    NT:PRINT:GOTO310                                :rem 71
>730 PRINT:PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D
    {OFF}ISK: (T/D)"                                :rem 228
740 GETA$:IFA$<>"T"ANDA$<>"D"THEN740                :rem 36
750 DV=1-7*(A$="D"):IFDV=8THENF$="0:"+F$:OPEN15,8,
    15,"S"+F$:CLOSE15                               :rem 212
760 T$=F$:ZK=PEEK(53)+256*PEEK(54)-LEN(T$):POKE782
    ,ZK/256                                         :rem 3
762 POKE781,ZK-PEEK(782)*256:POKE780,LEN(T$):SYS65
    469                                             :rem 109
763 POKE780,1:POKE781,DV:POKE782,1:SYS65466:rem 69
765 K=S:POKE254,K/256:POKE253,K-PEEK(254)*256:POKE
    780,253                                         :rem 17
766 K=E+1:POKE782,K/256:POKE781,K-PEEK(782)*256:SY
    S65496                                          :rem 235
770 IF(PEEK(783)AND1)OR(191ANDST)THEN780            :rem 111
775 PRINT"{DOWN}DONE.{DOWN}":GOTO310                :rem 113
780 PRINT"{DOWN}ERROR ON SAVE.{2 SPACES}TRY AGAIN.
    ":IFDV=1THEN720                                 :rem 171
781 OPEN15,8,15:INPUT#15,E1$,E2$:PRINTE1$;E2$:CLOS
    E15:GOTO720                                     :rem 103
790 PRINT"{CLR}{RVS}*** LOAD ***{2 DOWN}"           :rem 212
795 PRINT"{2 DOWN}(PRESS {RVS}RETURN{OFF}(ALONE TO
     CANCEL LOAD)"                                  :rem 82
800 F$="":INPUT"{2 DOWN} FILENAME";F$:IFF$=""THENP
    RINT:GOTO310                                    :rem 144
810 PRINT:PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D
    {OFF}ISK: (T/D)"                                :rem 227
820 GETA$:IFA$<>"T"ANDA$<>"D"THEN820                :rem 34
830 DV=1-7*(A$="D"):IFDV=8THENF$="0:"+F$            :rem 157
```

```
840 T$=F$:ZK=PEEK(53)+256*PEEK(54)-LEN(T$):POKE782
    ,ZK/256                                    :rem 2
841 POKE781,ZK-PEEK(782)*256:POKE780,LEN(T$):SYS65
    469                                       :rem 107
845 POKE780,1:POKE781,DV:POKE782,1:SYS65466:rem 70
850 POKE780,0:SYS65493                     :rem 11
860 IF(PEEK(783)AND1)OR(191ANDST)THEN870  :rem 111
865 PRINT"{DOWN}DONE.":GOTO310             :rem 96
870 PRINT"{DOWN}ERROR ON LOAD.{2 SPACES}TRY AGAIN.
    {DOWN}":IFDV=1THEN800                 :rem 172
880 OPEN15,8,15:INPUT#15,E1$,E2$:PRINTE1$;E2$:CLOS
    E15:GOTO800                           :rem 102
1000 REM BUZZER                           :rem 135
1001 POKE54296,15:POKE54277,45:POKE54278,165
                                          :rem 207
1002 POKE54276,33:POKE 54273,6:POKE54272,5 :rem 42
1003 FORT=1TO200:NEXT:POKE54276,32:POKE54273,0:POK
     E54272,0:RETURN                      :rem 202
2000 REM BELL SOUND                        :rem 78
2001 POKE54296,15:POKE54277,0:POKE54278,247
                                          :rem 152
2002 POKE 54276,17:POKE54273,40:POKE54272,0:rem 86
2003 FORT=1TO100:NEXT:POKE54276,16:RETURN  :rem 57
3000 PRINTC$;"{RVS}NOT ZERO PAGE OR ROM":GOTO1000
                                          :rem 89
```

The 6502 Instruction Set

ADC	Add Memory to Accumulator with Carry					
Status Flags	**N**	**Z**	**C**	**I**	**D**	**V**
	•	•	•			•

Addressing Mode	Mnemonics	Opcode	Size in Bytes
Immediate	ADC #Arg	69	2
Zero Page	ADC Arg	65	2
Zero Page, X	ADC Arg, X	75	2
Absolute	ADC Arg	6D	3
Absolute, X	ADC Arg, X	7D	3
Absolute, Y	ADC Arg, Y	79	3
(Indirect, X)	ADC (Arg, X)	61	2
(Indirect), Y	ADC (Arg), Y	71	2

AND	AND Memory with Accumulator					
Status Flags	**N**	**Z**	**C**	**I**	**D**	**V**
	•	•				

Addressing Mode	Mnemonics	Opcode	Size in Bytes
Immediate	AND #Arg	29	2
Zero Page	AND Arg	25	2
Zero Page, X	AND Arg, X	35	2
Absolute	AND Arg	2D	3
Absolute, X	AND Arg, X	3D	3
Absolute, Y	AND Arg, Y	39	3
(Indirect, X)	AND (Arg, X)	21	2
(Indirect), Y	AND (Arg), Y	31	2

ASL	Shift Left One Bit		
Status Flags	**N** •	**Z** •	**C** •	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Accumulator	ASL A	0A	1
Zero Page	ASL Arg	06	2
Zero Page, X	ASL Arg, X	16	2
Absolute	ASL Arg	0E	3
Absolute, X	ASL Arg, X	1E	3

BCC	Branch on Carry Clear		
Status Flags	**N**	**Z**	**C**	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Relative	BCC Arg	90	2

BCS	Branch on Carry Set		
Status Flags	**N**	**Z**	**C**	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Relative	BCS Arg	B0	2

BEQ	Branch on Zero		
Status Flags	**N**	**Z**	**C**	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Relative	BEQ Arg	F0	2

BIT	Test Bits in Memory Against Accumulator		
Status Flags	**N** •	**Z** •	**C** **I** **D** **V** •
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Zero Page	BIT Arg	24	2
Absolute	BIT Arg	2C	3

BMI	Branch on Minus		
Status Flags	**N**	**Z** **C** **I** **D** **V**	
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Relative	BMI Arg	30	2

BNE	Branch on Anything but Zero		
Status Flags	**N**	**Z** **C** **I** **D** **V**	
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Relative	BNE Arg	D0	2

BPL	Branch on Plus		
Status Flags	**N**	**Z** **C** **I** **D** **V**	
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Relative	BPL Arg	10	2

BRK	Break		
Status Flags	N Z C I D V •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	BRK	00	1

BVC	Branch on Overflow Clear		
Status Flags	N Z C I D V		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Relative	BVC Arg	50	2

BVS	Branch on Overflow Set		
Status Flags	N Z C I D V		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Relative	BVS Arg	70	2

CLC	Clear Carry Flag		
Status Flags	N Z C I D V •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	CLC	18	1

CLD	Clear Decimal Mode		
Status Flags	N Z C I D V •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	CLD	D8	1

CLI	Clear Interrupt Disable Bit		
Status Flags	N Z C I D V •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	CLI	58	1

CLV	Clear Overflow Flag		
Status Flags	N Z C I D V •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	CLV	B8	1

CMP	Compare Memory and Accumulator		
Status Flags	N Z C I D V • • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	CMP #Arg	C9	2
Zero Page	CMP Arg	C5	2
Zero Page, X	CMP Arg, X	D5	2
Absolute	CMP Arg	CD	3
Absolute, X	CMP Arg, X	DD	3
Absolute, Y	CMP Arg, Y	D9	3
(Indirect, X)	CMP (Arg, X)	C1	2
(Indirect), Y	CMP (Arg), Y	D1	2

CPX	Compare Memory Against X Register		
Status Flags	**N** •	**Z** •	**C** •	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	CPX #Arg	E0	2
Zero Page	CPX Arg	E4	2
Absolute	CPX Arg	EC	3

CPY	Compare Memory Against Y Register		
Status Flags	**N** •	**Z** •	**C** •	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	CPY #Arg	C0	2
Zero Page	CPY Arg	C4	2
Absolute	CPY Arg	CC	3

DEC	Decrement Memory by One		
Status Flags	**N** •	**Z** •	**C**	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Zero Page	DEC Arg	C6	2
Zero Page, X	DEC Arg, X	D6	2
Absolute	DEC Arg	CE	3
Absolute, X	DEC Arg, X	DE	3

DEX	Decrement X Register by One		
Status Flags	**N** •	**Z** •	**C**	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	DEX	CA	1

DEY	Decrement Y Register by One		
Status Flags	N Z C I D V • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	DEY	88	1

EOR	Exclusive-OR Memory with Accumulator		
Status Flags	N Z C I D V • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	EOR #Arg	49	2
Zero Page	EOR Arg	45	2
Zero Page, X	EOR Arg, X	55	2
Absolute	EOR Arg	4D	3
Absolute, X	EOR Arg, X	5D	3
Absolute, Y	EOR Arg, Y	59	3
(Indirect, X)	EOR (Arg, X)	41	2
(Indirect), Y	EOR (Arg), Y	51	2

INC	Increment Memory by One		
Status Flags	N Z C I D V • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Zero Page	INC Arg	E6	2
Zero Page, X	INC Arg, X	F6	2
Absolute	INC Arg	EE	3
Absolute, X	INC Arg, X	FE	3

INX	Increment X Register by One					
Status Flags		N	Z	C	I	D	V
		•	•				
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	INX	E8	1

INY	Increment Y Register by One					
Status Flags		N	Z	C	I	D	V
		•	•				
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	INY	C8	1

JMP	Jump					
Status Flags		N	Z	C	I	D	V
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Absolute Indirect	JMP Arg JMP (Arg)	4C 6C	3 3

JSR	Jump to New Location, but Save Return Address					
Status Flags		N	Z	C	I	D	V
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Absolute	JSR Arg	20	3

LDA	Load Accumulator with Memory		
Status Flags	**N** •	**Z** •	**C** **I** **D** **V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	LDA #Arg	A9	2
Zero Page	LDA Arg	A5	2
Zero Page, X	LDA Arg, X	B5	2
Absolute	LDA Arg	AD	3
Absolute, X	LDA Arg, X	BD	3
Absolute, Y	LDA Arg, Y	B9	3
(Indirect, X)	LDA (Arg, X)	A1	2
(Indirect), Y	LDA (Arg), Y	B1	2

LDX	Load X Register		
Status Flags	**N** •	**Z** •	**C** **I** **D** **V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	LDX #Arg	A2	2
Zero Page	LDX Arg	A6	2
Zero Page, Y	LDX Arg, Y	B6	2
Absolute	LDX Arg	AE	3
Absolute, Y	LDX Arg, Y	BE	3

LDY	Load Y Register		
Status Flags	**N** •	**Z** •	**C** **I** **D** **V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	LDY #Arg	A0	2
Zero Page	LDY Arg	A4	2
Zero Page, X	LDY Arg, X	B4	2
Absolute	LDY Arg	AC	3
Absolute, X	LDY Arg, X	BC	3

LSR	Shift Right One Bit in Either Memory or Accumulator		
Status Flags	**N** •	**Z** •	**C** •	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Accumulator	LSR A	4A	1
Zero Page	LSR Arg	46	2
Zero Page, X	LSR Arg, X	56	2
Absolute	LSR Arg	4E	3
Absolute, X	LSR Arg, X	5E	3

NOP	No Operation		
Status Flags	**N**	**Z**	**C**	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	NOP	EA	1

ORA	OR Memory with Accumulator		
Status Flags	**N** •	**Z** •	**C**	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	ORA #Arg	09	2
Zero Page	ORA Arg	05	2
Zero Page, X	ORA Arg, X	15	2
Absolute	ORA Arg	0D	3
Absolute, X	ORA Arg, X	1D	3
Absolute, Y	ORA Arg, Y	19	3
(Indirect, X)	ORA (Arg, X)	01	2
(Indirect), Y	ORA (Arg), Y	11	2

PHA	Push Accumulator onto the Stack		
Status Flags	**N Z C I D V**		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	PHA	48	1

PHP	Push Processor Status onto the Stack		
Status Flags	**N Z C I D V**		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	PHP	08	1

PLA	Pull Accumulator from the Stack		
Status Flags	**N Z C I D V** • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	PLA	68	1

PLP	Pull Processor Status from the Stack		
Status Flags	**N Z C I D V** From Stack		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	PLP	28	1

ROL	Rotate One Bit Left in Memory or the Accumulator		
Status Flags	**N** •	**Z** •	**C** •	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Accumulator	ROL A	2A	1
Zero Page	ROL Arg	26	2
Zero Page, X	ROL Arg, X	36	2
Absolute	ROL Arg	2E	3
Absolute, X	ROL Arg, X	3E	3

ROR	Rotate One Bit Right in Memory or the Accumulator		
Status Flags	**N** •	**Z** •	**C** •	**I**	**D**	**V**
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Accumulator	ROR A	6A	1
Zero Page	ROR Arg	66	2
Zero Page, X	ROR Arg, X	76	2
Absolute	ROR Arg	6E	3
Absolute, X	ROR Arg, X	7E	3

RTI	Return from Interrupt		
Status Flags	**N**	**Z**	**C**	**I**	**D**	**V**
	From Stack					
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	RTI	40	1

RTS	Return from Subroutine		
Status Flags	**N Z C I D V**		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	RTS	60	1

SBC	Subtract Memory from Accumulator, with Borrow		
Status Flags	**N Z C I D V** • • • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Immediate	SBC #Arg	E9	2
Zero Page	SBC Arg	E5	2
Zero Page, X	SBC Arg, X	F5	2
Absolute	SBC Arg	ED	3
Absolute, X	SBC Arg, X	FD	3
Absolute, Y	SBC Arg, Y	F9	3
(Indirect, X)	SBC (Arg, X)	E1	2
(Indirect), Y	SBC (Arg), Y	F1	2

SEC	Set Carry Flag		
Status Flags	**N Z C I D V** •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	SEC	38	1

SED	Set Decimal Mode		
Status Flags	**N Z C I D V**		
			•
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	SED	F8	1

SEI	Set Interrupt Disable Status		
Status Flags	**N Z C I D V**		
			•
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	SEI	78	1

STA	Store Accumulator in Memory		
Status Flags	**N Z C I D V**		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Zero Page	STA Arg	85	2
Zero Page, X	STA Arg, X	95	2
Absolute	STA Arg	8D	3
Absolute, X	STA Arg, X	9D	3
Absolute, Y	STA Arg, Y	99	3
(Indirect, X)	STA (Arg, X)	81	2
(Indirect), Y	STA (Arg), Y	91	2

STX	Store X Register in Memory		
Status Flags	**N Z C I D V**		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Zero Page	STX Arg	86	2
Zero Page, Y	STX Arg, Y	96	2
Absolute	STX Arg	8E	3

STY	Store Y Register in Memory		
Status Flags	**N Z C I D V**		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Zero Page	STY Arg	84	2
Zero Page, X	STY Arg, X	94	2
Absolute	STY Arg	8C	3

TAX	Transfer Accumulator to X Register		
Status Flags	**N Z C I D V** • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	TAX	AA	1

TAY	Transfer Accumulator to Y Register		
Status Flags	**N Z C I D V** • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	TAY	A8	1

TSX	Transfer Stack Pointer to X Register		
Status Flags	**N** **Z** **C** **I** **D** **V** • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	TSX	BA	1

TXA	Transfer X Register to Accumulator		
Status Flags	**N** **Z** **C** **I** **D** **V** • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	TXA	8A	1

TXS	Transfer X Register to Stack Pointer		
Status Flags	**N** **Z** **C** **I** **D** **V**		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	TXS	9A	1

TYA	Transfer Y Register to Accumulator		
Status Flags	**N** **Z** **C** **I** **D** **V** • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size in Bytes**
Implied	TYA	98	1

Number Tables

This lookup table should make it convenient when you need to translate hex, binary, or decimal numbers. The first column lists the decimal numbers between 1 and 255. The second column is the hexadecimal equivalent. The third column is the decimal equivalent of a hex *most significant byte*, or MSB. The fourth column is the binary.

If you need to find out the decimal equivalent of the hex number $FD15, look up $FD in the Hex column and you'll see that it's 64768. Then look up the $15 in the Hex column (it's 21 decimal) and add 21 + 64768 to get the answer: 64789.

Going the other way, from decimal to hex, you could translate 64780 into hex by looking in the MSB column for the closest number (it must be smaller, however). In this case, the closest smaller number is 64768 so jot down $FD as the hex MSB. Then subtract 64768 from 64780 to get the LSB: 12. Look up 12 in the decimal column (it is $0C hex) and put the $FD MSB together with the $0C LSB for your answer: $FD0C.

With a little practice, you can use this chart for fairly quick conversions between the number systems. Most of your translations will only involve going from hex to decimal or vice versa with the LSB of hex numbers, the first 255 numbers, which require no addition or subtraction. Just look them up in the table.

Hex	LSB	MSB	Binary
01	1	256	00000001
02	2	512	00000010
03	3	768	00000011
04	4	1024	00000100
05	5	1280	00000101
06	6	1536	00000110
07	7	1792	00000111
08	8	2048	00001000
09	9	2304	00001001
0A	10	2560	00001010

Hex	LSB	MSB	Binary
0B	11	2816	00001011
0C	12	3072	00001100
0D	13	3328	00001101
0E	14	3584	00001110
0F	15	3840	00001111
10	16	4096	00010000
11	17	4352	00010001
12	18	4608	00010010
13	19	4864	00010011
14	20	5120	00010100
15	21	5376	00010101
16	22	5632	00010110
17	23	5888	00010111
18	24	6144	00011000
19	25	6400	00011001
1A	26	6656	00011010
1B	27	6912	00011011
1C	28	7168	00011100
1D	29	7424	00011101
1E	30	7680	00011110
1F	31	7936	00011111
20	32	8192	00100000
21	33	8448	00100001
22	34	8704	00100010
23	35	8960	00100011
24	36	9216	00100100
25	37	9472	00100101
26	38	9728	00100110
27	39	9984	00100111
28	40	10240	00101000
29	41	10496	00101001
2A	42	10752	00101010
2B	43	11008	00101011
2C	44	11264	00101100
2D	45	11520	00101101
2E	46	11776	00101110
2F	47	12032	00101111
30	48	12288	00110000
31	49	12544	00110001
32	50	12800	00110010
33	51	13056	00110011
34	52	13312	00110100
35	53	13568	00110101
36	54	13824	00110110
37	55	14080	00110111
38	56	14336	00111000
39	57	14592	00111001
3A	58	14848	00111010
3B	59	15104	00111011
3C	60	15360	00111100
3D	61	15616	00111101
3E	62	15872	00111110

Hex	LSB	MSB	Binary
3F	63	16128	00111111
40	64	16384	01000000
41	65	16640	01000001
42	66	16896	01000010
43	67	17152	01000011
44	68	17408	01000100
45	69	17664	01000101
46	70	17920	01000110
47	71	18176	01000111
48	72	18432	01001000
49	73	18688	01001001
4A	74	18944	01001010
4B	75	19200	01001011
4C	76	19456	01001100
4D	77	19712	01001101
4E	78	19968	01001110
4F	79	20224	01001111
50	80	20480	01010000
51	81	20736	01010001
52	82	20992	01010010
53	83	21248	01010011
54	84	21504	01010100
55	85	21760	01010101
56	86	22016	01010110
57	87	22272	01010111
58	88	22528	01011000
59	89	22784	01011001
5A	90	23040	01011010
5B	91	23296	01011011
5C	92	23552	01011100
5D	93	23808	01011101
5E	94	24064	01011110
5F	95	24320	01011111
60	96	24576	01100000
61	97	24832	01100001
62	98	25088	01100010
63	99	25344	01100011
64	100	25600	01100100
65	101	25856	01100101
66	102	26112	01100110
67	103	26368	01100111
68	104	26624	01101000
69	105	26880	01101001
6A	106	27136	01101010
6B	107	27392	01101011
6C	108	27648	01101100
6D	109	27904	01101101
6E	110	28160	01101110
6F	111	28416	01101111
70	112	28672	01110000
71	113	28928	01110001
72	114	29184	01110010

Hex	LSB	MSB	Binary
73	115	29440	01110011
74	116	29696	01110100
75	117	29952	01110101
76	118	30208	01110110
77	119	30464	01110111
78	120	30720	01111000
79	121	30976	01111001
7A	122	31232	01111010
7B	123	31488	01111011
7C	124	31744	01111100
7D	125	32000	01111101
7E	126	32256	01111110
7F	127	32512	01111111
80	128	32768	10000000
81	129	33024	10000001
82	130	33280	10000010
83	131	33536	10000011
84	132	33792	10000100
85	133	34048	10000101
86	134	34304	10000110
87	135	34560	10000111
88	136	34816	10001000
89	137	35072	10001001
8A	138	35328	10001010
8B	139	35584	10001011
8C	140	35840	10001100
8D	141	36096	10001101
8E	142	36352	10001110
8F	143	36608	10001111
90	144	36864	10010000
91	145	37120	10010001
92	146	37376	10010010
93	147	37632	10010011
94	148	37888	10010100
95	149	38144	10010101
96	150	38400	10010110
97	151	38656	10010111
98	152	38912	10011000
99	153	39168	10011001
9A	154	39424	10011010
9B	155	39680	10011011
9C	156	39936	10011100
9D	157	40192	10011101
9E	158	40448	10011110
9F	159	40704	10011111
A0	160	40960	10100000
A1	161	41216	10100001
A2	162	41472	10100010
A3	163	41728	10100011
A4	164	41984	10100100
A5	165	42240	10100101
A6	166	42496	10100110

Hex	LSB	MSB	Binary
A7	167	42752	10100111
A8	168	43008	10101000
A9	169	43264	10101001
AA	170	43520	10101010
AB	171	43776	10101011
AC	172	44032	10101100
AD	173	44288	10101101
AE	174	44544	10101110
AF	175	44800	10101111
B0	176	45056	10110000
B1	177	45312	10110001
B2	178	45568	10110010
B3	179	45824	10110011
B4	180	46080	10110100
B5	181	46336	10110101
B6	182	46592	10110110
B7	183	46848	10110111
B8	184	47104	10111000
B9	185	47360	10111001
BA	186	47616	10111010
BB	187	47872	10111011
BC	188	48128	10111100
BD	189	48384	10111101
BE	190	48640	10111110
BF	191	48896	10111111
C0	192	49152	11000000
C1	193	49408	11000001
C2	194	49664	11000010
C3	195	49920	11000011
C4	196	50176	11000100
C5	197	50432	11000101
C6	198	50688	11000110
C7	199	50944	11000111
C8	200	51200	11001000
C9	201	51456	11001001
CA	202	51712	11001010
CB	203	51968	11001011
CC	204	52224	11001100
CD	205	52480	11001101
CE	206	52736	11001110
CF	207	52992	11001111
D0	208	53248	11010000
D1	209	53504	11010001
D2	210	53760	11010010
D3	211	54016	11010011
D4	212	54272	11010100
D5	213	54528	11010101
D6	214	54784	11010110
D7	215	55040	11010111
D8	216	55296	11011000
D9	217	55552	11011001
DA	218	55808	11011010

Hex	LSB	MSB	Binary
DB	219	56064	11011011
DC	220	56320	11011100
DD	221	56576	11011101
DE	222	56832	11011110
DF	223	57088	11011111
E0	224	57344	11100000
E1	225	57600	11100001
E2	226	57856	11100010
E3	227	58112	11100011
E4	228	58368	11100100
E5	229	58624	11100101
E6	230	58880	11100110
E7	231	59136	11100111
E8	232	59392	11101000
E9	233	59648	11101001
EA	234	59904	11101010
EB	235	60160	11101011
EC	236	60416	11101100
ED	237	60672	11101101
EE	238	60928	11101110
EF	239	61184	11101111
F0	240	61440	11110000
F1	241	61696	11110001
F2	242	61952	11110010
F3	243	62208	11110011
F4	244	62464	11110100
F5	245	62720	11110101
F6	246	62976	11110110
F7	247	63232	11110111
F8	248	63488	11111000
F9	249	63744	11111001
FA	250	64000	11111010
FB	251	64256	11111011
FC	252	64512	11111100
FD	253	64768	11111101
FE	254	65024	11111110
FF	255	65280	11111111

The following program will print copies of this number table. You might need to make some adjustments to the printout conventions and your printer itself.

Table Printer

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

```
10 OPEN4,4:REM OPEN CHANNEL TO PRINTER      :rem 55
100 HE$="0123456789ABCDEF"                  :rem 101
110 FOR X=1 TO 255:D=X:GOSUB 230            :rem 224
120 L$=RIGHT$("{4 SPACES}"+STR$(X),6)       :rem 202
130 M$=RIGHT$("{4 SPACES}"+STR$(X*256),8) :rem 149
```

```
140 PRINT#4,H$;L$;M$;"{3 SPACES}";          :rem 34
145 REM CREATE BINARY                       :rem 247
150 C=1:B=2:IF X AND 1 THEN B$(C)="1":GOTO 170
                                            :rem 61
160 B$(C)="0"                               :rem 66
170 C=C+1:IF B AND X THEN B$(C)="1":GOTO 190
                                            :rem 213
180 B$(C)="0"                               :rem 68
190 B=B*2:IFC>8 THEN 210                    :rem 251
200 GOTO 170                                :rem 99
210 FOR I=8 TO 1 STEP-1:PRINT#4,B$(I);:NEXT I
                                            :rem 237
220 PRINT#4:NEXTX:END                       :rem 90
225 REM CONVERT TO HEX                      :rem 38
230 H$="":FOR M=1 TO 0 STEP-1:N%=D/(16↑M):D=D-N%*1
    6↑M                                     :rem 102
240 H$=H$+MID$(HE$,N%+1,1):NEXT:RETURN      :rem 193
```

Index

accumulator 23
ADC (ADd memory to accumulator with Carry) instruction 230
address 8
addressing modes 8
"A Disassembler" program 14–19
AND (AND memory with accumulator) instruction 230
animation 135
arcade games 4
"Area-Fill Routine" (Graphics Package) 154–55, 160–61
arrays, how stored 102
ASCII code 91, 186
ASCII files 204–5
"ASCII/POKE Printer" program 91–95
ASL (Shift Left one bit) instruction 231
assembler 5, 6–9, 14, 98
"Assembler, The" program 6–13
AUTO command (BASIC Aid) 46
"Auto Line Numbering" program 69–70
"Automatic Proofreader" 219–22
BASIC v, 3, 14
"BASIC Aid" program v, 45–67
"BASIC Maze Generator" program 181
BCC (Branch on Carry Clear) instruction 231
BCS (Branch on Carry Set) instruction 231
BEQ (Branch if EQual) instruction 15, 231
BIT (test BITs in memory against accumulator) instruction 232
bitmapped graphics 26–30
BMI (Branch on MInus) instruction 232
BNE (Branch if Not Equal to zero) instruction 24, 232
BPL (Branch on PLus) instruction 232
BREAK command (in "BASIC Aid" program) 47
BRK (BReaK) instruction 6, 233
BVC (Branch on oVerflow Clear) instruction 233
BVS (Branch on oVerflow Set) instruction 233
byte 6
cassette buffer 76, 89, 91–92, 191, 205, 209
CHANGE command (in "BASIC Aid" program) 47–48
character sets, editing 111–18
CHRGET ROM routine 81
CHROUT ROM routine 93
CLC (CLear Carry flag) instruction 233
CLD (CLear Decimal mode) instruction 234
CLI (CLear Interrupt disable bit) instruction 234
CLV (CLear oVerflow flag) instruction 234
CMP (CoMPare memory and accumulator) instruction 234
Cochrane, F. Arthur 45
COLD command (in "BASIC Aid" program) 48
Commodore 64 Programmer's Reference Guide 72, 92
COMPUTE!'s First Book of 64 Sound and Graphics 115, 151
COMPUTE!'s Reference Guide to Commodore 64 Graphics 115, 151
COMPUTE!'s Second Book of Machine Language 5
CPX (ComPare memory against x register) instruction 235
CPY (ComPare memory against y register) instruction 235
CRT command (in "BASIC Aid" program) 48
cursor control 78–79
"DATAmaker" program 24–25
DATA statement 117, 137–38, 215
DEC (DECrement memory by one) instruction 235
DELETE command (in "BASIC Aid" program) 48–49
"Demo/scan" program 188
DEX (DEcrement x register by one) instruction 235
DEY (DEcrement y register by one) instruction 236
"Disassembler" program 23
disassembling 14–19
"Disk Defaulter" program 106–7
DOS support commands (in "BASIC Aid" program) 52–53
"Dr. Video" program 78–79
DUMP command (in "BASIC Aid" program) 49
EOR (Exclusive-OR memory with accumulator) instruction 236
FIND command (in "BASIC Aid" program) 49
FLIST command (in "BASIC Aid" program) 49
"Foolproof Input" program 83–86
"Four-Speed Brake" program 89–90

function keys 74–75, 89–90
GET statement 84
GETIN ROM routine 93
HELP command (in "BASIC Aid" program) 49–50
HEX command (in "BASIC Aid" program) 50
Hoare, C.A.R. 196
immediate addressing 8
implied addressing 8
INC (INCrement memory by one) instruction 236
INPUT statement 83–85
interrupts 78
inverse video 92
INX (INCrement X) instruction 7, 24, 237
INY (INCrement Y) instruction 237
IRQ (Interrupt ReQuest) 78–79, 98–99
JMP (JuMP) instruction 6, 191, 237
joystick 112, 132–33, 167–68
JSR (Jump to SubRoutine) instruction 6, 191, 237
Kernal ROM 29, 71
"Keyscan" program 187–88
KILL command (in "BASIC Aid" program) 50
LDA (LoaD the Accumulator) instruction 14, 23, 238
LDX (LoaD X) instruction 23, 238
LDY (LoaD Y) instruction 7, 238
"Line-Draw Routine" 154, 158–60
LIST command 14–15
listing conventions 217–18
LSR instruction 239
"Machine Language Maze Generator" program 181–84
maze generator programs 178–85
  algorithm 178–80
  flow chart 185
memory locations, safe 91
MERGE command (in "BASIC Aid" program) 50
mixing BASIC and ML 4–5
"MLX" program 45–46, 111, 132, 223–29
mnemonics 5, 6–8
"Monitor Disassembly" program 22–23
multicolor mode 115–16, 153–56
NOP (No OPeration) instruction 239
number tables 246–51
"Numeric Keypad" program 71–73
  colors and 72
OFF command (in "BASIC Aid" program) 50–51
OLD command (in "BASIC Aid" program) 50
"One-Touch Commands" program 74–77
ORA (OR memory with Accumulator) instruction 239
"Package Demonstration" program 162–64
pages, memory 22
PHA (PusH Accumulator onto stack) instruction 240
PHP (PusH Processor status onto stack) instruction 240
PLA (PuLl Accumulator from stack) instruction 240
"Plotstring" programs 26–41
PLP (PuLl Processor status from stack) instruction 240
"Point-Plot Routine" 152–54, 157–58
pointers, BASIC 103–4
PRINT command 91
"Quick Clear" routines 3–4
Quicksort algorithm 196
quote mode 97–100
"RAMtest" program 209–11
READ command (in "BASIC Aid" program) 51
RENUMBER command (in "BASIC Aid" program) 51
REPEAT command (in "BASIC Aid" program) 51
ROL (ROtate one bit Left in memory or the accumulator) instruction 241
ROM character addresses 42
ROR (ROtate one bit Right in memory or the accumulator) instruction 241
RTI (ReTurn from Interrupt) instruction 241
RTS (ReTurn from Subroutine) instruction 6, 242
SBC (SuBtract memory from accumulator with borrow) 242
SCROLL command (in "BASIC Aid" program) 51–52
"Scroll 64" program 171–75
scrolling 171–74
SEC (SEt Carry flag) instruction 242
SED (SEt Decimal mode) instruction 243
SEI (SEt Interrupt disable status) instruction 243
shell sort 196
6502 instruction set 230–45
6510 chip 3, 132
"64 Escape Key" program 97–101
"64 Freeze" program 202–3
"64 Loader" program 21–22
"64 Merge" program 204–8
"64 Paddle Reader" program 176–77
"64 Searcher" program 87–88
"Sort Test" program 201

"Sprite Magic" sprite editor 131–49
sprite page number 134
sprite seam 167
sprites 131–39
  multicolor 135–37
  BASIC and 167
STA (STore the Accumulator) instruction 24, 210, 243
START command (in "BASIC Aid" program) 52
"Step Lister" program 80–82
Strasma, James 45
"String Search" program 191–95
STX (STore x register in memory) instruction 244
STY (STore Y) instruction 7, 244
"Table Printer" program 251–52
TAX (TrAnsfer accumulator to x register) instruction 244
TAY (Transfer Accumulator to y register) instruction 244
"Timed Search" program 195
tokens, BASIC 14, 80-81
TSX (Transfer Stack pointer to x register) instruction 245
"Two-Sprite Joystick" program 167–70
TXA (Transfer x register to Accumulator) instruction 245
TXS (Transfer x register to Stack pointer) instruction 245
TYA (Transfer y register to Accumulator) instruction 245
"Ultrafont + Character Editor" program 111–30, 131
"Ultrasort" program 196–201
"Variable Lister" program 102–5
variables, where stored 102
wedges 80–81
windowing 171–72
"Window, The" program 22
word processors 4
x register 23
y register 8
zero page 191–92

COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**

800-334-0868

In NC call 919-275-9809

Quantity	Title	Price	Total
_____	Machine Language for Beginners	**$14.95***	_____
_____	Home Energy Applications	**$14.95***	_____
_____	COMPUTE!'s First Book of VIC	**$12.95***	_____
_____	COMPUTE!'s Second Book of VIC	**$12.95***	_____
_____	COMPUTE!'s First Book of VIC Games	**$12.95***	_____
_____	COMPUTE!'s First Book of 64	**$12.95***	_____
_____	COMPUTE!'s First Book of Atari	**$12.95***	_____
_____	COMPUTE!'s Second Book of Atari	**$12.95***	_____
_____	COMPUTE!'s First Book of Atari Graphics	**$12.95***	_____
_____	COMPUTE!'s First Book of Atari Games	**$12.95***	_____
_____	Mapping The Atari	**$14.95***	_____
_____	Inside Atari DOS	**$19.95***	_____
_____	The Atari BASIC Sourcebook	**$12.95***	_____
_____	Programmer's Reference Guide for TI-99/4A	**$14.95***	_____
_____	COMPUTE!'s First Book of TI Games	**$12.95***	_____
_____	Every Kid's First Book of Robots and Computers	**$ 4.95†**	_____
_____	The Beginner's Guide to Buying A Personal Computer	**$ 3.95†**	_____

* Add $2 shipping and handling. Outside US add $5 air mail; $2 surface mail.

†/Add $1 shipping and handling. Outside US add $5 air mail; $2 surface mail.

Please add shipping and handling for each book ordered. _____

Total enclosed or to be charged. _____

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

☐ Payment enclosed Please charge my: ☐ VISA ☐ MasterCard
☐ American Express Acc't. No. _____ Expires ___/___

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Allow 4-5 weeks for delivery.

If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!** Magazine. Use this form to order your subscription to **COMPUTE!.**

For Fastest Service,
Call Our **Toll-Free** US Order Line

800-334-0868

In NC call 919-275-9809

COMPUTE!
P.O. Box 5406
Greensboro, NC 27403

My Computer Is:
☐ Commodore 64 ☐ TI-99/4A ☐ Timex/Sinclair ☐ VIC-20 ☐ PET
☐ Radio Shack Color Computer ☐ Apple ☐ Atari ☐ Other _____
☐ Don't yet have one...

☐ $24 One Year US Subscription
☐ $45 Two Year US Subscription
☐ $65 Three Year US Subscription

Subscription rates outside the US:

☐ $30 Canada
☐ $42 Europe, Australia, New Zealand/Air Delivery
☐ $52 Middle East, North Africa, Central America/Air Mail
☐ $72 Elsewhere/Air Mail
☐ $30 International Surface Mail (lengthy, unreliable delivery)

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

☐ Payment Enclosed ☐ VISA
☐ MasterCard ☐ American Express
Acct. No. _____ Expires ___/___

Speed and Power

Machine language, the language that your Commodore 64 uses to calculate and process information, is fast and powerful. Much more so than BASIC, the programming language you're probably most familiar with. Until now, unless you knew how to program in ML, you could only look at machine language programs with envy. But it *is* possible to make BASIC and machine language work together.

The routines and programs in this book can be easily added to your own BASIC programs, or simply placed in your computer's memory. Once in your program or in the 64's memory, they can make it easier to program, create dazzling, high-speed graphics, speed up games, merge files, or sort thousands of items. All you have to do is type them in.

The best machine language programs from recent issues of *COMPUTE!* magazine and *COMPUTE!'s Gazette* have been revised and enhanced for this book. Other programs appear here for the first time anywhere. And all are of the high quality you expect from COMPUTE! Publications.

Here are some of the routines and programs you'll find in this book:

- "BASIC Aid," which gives you 20 tools to make BASIC programming easier.
- Routines which automatically number BASIC program lines, turn your keyboard into a numeric keypad, and let you enter BASIC commands with one key.
- High-speed graphics applications, such as "Ultrafont +," "Sprite Magic," and "The Graphics Package."
- Arcade-speed joystick, paddle, and keyboard controllers.
- Programs that let you search for specific strings, sort lists, freeze the screen, merge files, or even test your 64's RAM chip.
- A machine language assembler, a disassembler, and simple explanations of how ML is created and how it works.

You'll find these routines and their detailed explanations easy to use, right from the moment you finish typing them in. There are even programs included to insure error-free entry of every program. With this book, and your own BASIC programs, you'll soon be using the power and speed of ML.

ISBN 0-942386-48-5

COMPUTE!'s Machine Language Routines for the Commodore 64

COMPUTE! Books