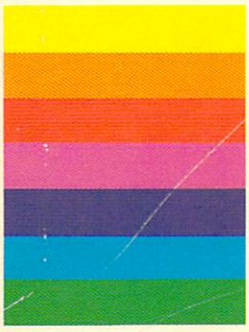
A **COMPUTE! Books** Publication

$12.95

VIC 2



COMPUTE!'s Second Book of VIC

Applications, utilities, games, and other helpful information for users of the VIC-20® home computer.

COMPUTE!'s Second Book of VIC

COMPUTE!™ Publications, Inc.
A Subsidiary Of American Broadcasting Companies, Inc.
Greensboro, North Carolina

The following articles were originally published in *COMPUTE!* Magazine, copyright 1982, Small System Services, Inc.:
"Using Atari Joysticks With Your VIC" (June)
"VIC Super Expander Memory Map" (July)
"Electric Eraser" (August)
"VIC Pause" (September)
"VIC Sticks" (September)
"Pixelator" (October)
"The VIC Keyboard Redefined" (October)
"VIC Joystick and Keyboard Routine" (October)
"UXB" (November)
"Programming VIC's Function Keys" (November)
"VIC Harmony" (November)
"VIC File Clerk" (December)
"Understanding VIC High-Res Graphics" (December)
"VIC Block SAVE and LOAD" (December)
The following articles were originally published in *COMPUTE!* Magazine, copyright 1983, Small System Services, Inc.:
"VIC Sound Generation" (January)
"A Day at the Races" (February)
"UFO Pilot-VIC Custom Characters for Game Graphics" (February)
"The Expanded/Unexpanded VIC" (February)
"Bi-directional VIC Scrolling" (February)
"Fighter Aces — Add a Second VIC Joystick" (March)
"VIC Tracing Disassembler" (March)
"VIC Editype: A Text Editing and Storage Program" (April)
"VIC Word" (April)
"VIC Automatic BASIC" (April)
"Hexedit: A BASIC Hex Editor for the VIC" (April)
"VIC Kaleidoscope" (May)
"Vertical Data Acquisition With VIC" (May)

Printed in the United States of America

ISBN 0-942386-16-7

COMPUTE! Publications, Inc. Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is a subsidiary of American Broadcasting Companies, Inc., and is not associated with any manufacturer of personal computers. VIC-20 is a trademark of Commodore Electronics Limited. Atari is a trademark of Atari, Inc.

Contents

Foreword

High-quality programs and easy-to-understand articles brought *COMPUTE!'s First Book of VIC* to the top of computer book best-seller lists all over the country. Now *COMPUTE!'s Second Book of VIC* offers even more to VIC users: programming techniques, useful software, computer utilities, and lots of useful information. And everything here is up to *COMPUTE!*'s well-known standards.

VIC users at every level of experience will find something useful.

If you use your VIC just for fun, there are six games, including the all-machine-language "Snake."

If you want to explore ways to get input for your programs, there are articles on programming with the function keys and joysticks, and a utility for redefining the VIC keyboard.

If you need sophisticated sound effects or music in your programs, or want to create dazzling graphics, the tools and techniques you need are here.

More advanced programmers will especially appreciate the VIC memory maps, which identify practically every location used by the VIC operating system.

And that's just the beginning.

Of the articles that originally appeared in *COMPUTE!* Magazine and *COMPUTE!'s Gazette for Commodore,* many have been since enhanced. Many other articles, however, are appearing here for the first time anywhere.

—Chapter 1—

Applications and Recreations

—1—

File Clerk

Dennis Surek

If you don't have a disk drive, this program will allow you to store and quickly locate up to 60 pages of information on one cassette tape.

This program is designed to save you some space around the house — space perhaps presently occupied by large filing cabinets or old cardboard storage boxes. You will be able to file and at any time read back quickly 60 pages of information stored on one side of a 60-minute cassette.

Whether it is recipes, budgets, or utility bills, the computer stores them efficiently and accurately. This program should be SAVEd at the beginning of every tape that is to be converted into a filing cabinet.

The program first displays the file numbers and names and then asks which one you wish to access and whether you wish to read or write to that file. If you are writing, the instructions will appear. Whether you are writing or reading, you will "fast find" to the proper file.

If you are writing, you can write as many pages as the file maximum allows. If you are reading, you can switch to writing subsequent pages, or you can continue reading through following pages and files.

Line 10 sets the number of files (NF) at 15 and the number of pages per file (NP) at 4. Changing either or both of these to lower values is easily done and requires no further changes to the program. The product NF x NP should be kept to 60 or less. With this in mind, it is just as easy to decrease NF and increase NP. But note that the program only fast finds to each file, and that increasing the number of pages per file defeats this fast find feature.

Increasing NF to more than 15 creates some minor problems. You will have to put additional DATA statements for file names between lines 100 and 240. Second, to keep the menu from scrolling up when the program is run, insert the following four lines:

```
81 IF I<>INT(NF/2) THEN 9Ø
82 PRINT"PRESS ANY KEY TO":PRINT"CONTINUE"
83 GET B$:IF B$="" THEN 83
84 PRINT"{CLR}"
```

These lines allow you to see half of the file names first and then to call for the rest when you are ready.

Three Naming Choices

Lines 100 to 240 are reserved for file names. There are three methods for dealing with file names. If you know all of the file names ahead of time, you could enter them when you type in this program. Conversely, you might not bother with file names at all, but use only the file numbers, writing descriptions of the files on the cassette box.

The system that I use is to save the program at the exact beginning of the magnetic portion of each tape. I then simply edit any of these lines to the title I want and reSAVE the program starting at the same position on the tape. The new program has not changed in length and therefore will still fast find to the proper file headers.

Lines 250 to 290 determine which file you want and whether you wish to read or write. If you are reading file #1, then line 300 branches to the read file routine beginning on line 660. This is possible because the PLAY key is already down from loading the program, and no fast forward is required. In all other cases, some cassette key instructions will be needed. Line 310 determines if any keys are down and instructs you to press STOP in order to bring all keys up. Line 320 temporarily halts the program until this is done. If you are writing file #1, then line 330 branches to the write routine on line 420. Again, no fast forward is required for this file.

For all other files the cassette must be put into fast forward. Line 340 gives this instruction, and line 350 halts the program until the fast forward key is depressed. Line 360 begins the timer, and line 370 halts the program until an elapsed time of 90 jiffies per page per file is reached. At that instant, line 380 stops the cassette motor. Lines 390 and 400 get all keys up in a manner described previously. Line 410 branches to the read routine, and lines 420 to 500 are the instructions for writing a file.

Line 510 opens the file for writing and increments the page count. In the command OPEN 1,1,1 the first "1" is the logical file number or reference number for our data file. The second specifies cassette drive #1, and the third indicates that the file is being opened for writing with no end-of-tape marker. It is the absence of this marker that allows the reading of consecutive pages later. For convenience, all files are assigned logical file #1. The program

keeps track of the actual file number with the variable F.

Lines 520 to 590 input from the keyboard up to 20 message lines that make up one page. If a message line containing more than 22 characters is entered, it is edited to that length by line 540. Line 550 displays the last five characters of the message line as accepted so that you know how to begin your next message line.

If you are writing fewer than 20 message lines and have signaled this with the input message STOP, then line 580 will fill the rest of the page with blank message lines. This keeps all the pages the same length and therefore at a specific location on the tape. This enables you to later change any page simply by writing over the old one without having to rewrite the following pages in that file. Lines 600 to 650 determine if you wish to write the next page. If the answer is no, the program terminates.

Lines 660 to 740 are the read file routine. The zero in the command OPEN 1,1,0 indicates a read operation. Line 720 moves the cursor up one line if the message line is 22 characters so that no blank lines will be displayed between message lines.

Lines 750 to 780 are for inputting and branching on commands to read or write subsequent pages. Lines 790 to 810 are the usual instructions to get all cassette keys up when changing from reading one page to writing the next page.

This program has been kept reasonably short so that load time is at a minimum. For that reason, there is no programming of special color or sound commands.

File Clerk

```
10 NF=15:NP=4:DIMA$(NF),O$(20)
20 PRINT"{CLR} ***VIC FILE CLERK***"
30 REM
40 REM
50 REM
60 PRINT"THIS PROGRAM WILL"
70 PRINT"READ OR WRITE TO FILE:"
80 FORI=1TONF
90 READ A$(I):PRINTI;TAB(5);A$(I):NEXTI
100 DATA UNNAMED
110 DATA UNNAMED
120 DATA UNNAMED
130 DATA UNNAMED
140 DATA UNNAMED
150 DATA UNNAMED
160 DATA UNNAMED
170 DATA UNNAMED
```

```
180 DATA UNNAMED
190 DATA UNNAMED
200 DATA UNNAMED
210 DATA UNNAMED
220 DATA UNNAMED
230 DATA UNNAMED
240 DATA UNNAMED
250 INPUT"FILE SELECTED";F
260 IFF<1ORF>NFTHEN250
270 INPUT"R-READ/W-WRITE";C$
280 IFC$="W"ORC$="R"THEN300
290 GOTO270
300 IFF=1ANDC$="R"THEN660
310 PRINT"{CLR}";:IF(PEEK(37151)AND64)=0T
    HENPRINT"PRESS STOP ON TAPE"
320 IF(PEEK(37151)AND64)=0THEN320
330 IFF=1THEN420
340 PRINT"PRESS FAST FORWARD"
350 IF(PEEK(37151)AND64)=64THEN350
360 PRINT"OK":A=TI
370 IFABS(TI-A)<(F-1)*NP*90THEN370
380 POKE37148,PEEK(37148)AND247
390 PRINT"PRESS STOP ON TAPE"
400 IF(PEEK(37151)AND64)=0THEN400
410 IFC$="R"THEN660
420 PRINT"{CLR}";
430 PRINT"INSTRUCTIONS TO"
440 PRINT"{RVS}WRITE FILE"
450 PRINT"{2 DOWN}MAXIMUMS:"
460 PRINT"========"
470 PRINT"{DOWN}-20 LINES PER PAGE"
480 PRINT" (TYPE STOP IF LESS)"
490 PRINT"-";NP;"PAGES PER FILE"
500 PRINT"{2 DOWN}{RVS}WAIT{OFF} FOR PROM
    PT.FIRST"
510 OPEN1,1,1:PC=PC+1
520 PRINT"{CLR}{RVS}WRITE FILE";F;"PAGE";
    PC
530 FORK=1TO20:INPUTO$(K):IFLEN(O$(K))<=2
    2THEN560
540 O$(K)=LEFT$(O$(K),22)
550 PRINT"*LINE EDITED TO*";RIGHT$(O$(K),
    5)
560 IFO$(K)="STOP"THEN580
570 PRINT#1,O$(K):NEXTK
580 FORI=KTO20:PRINT#1," ":NEXTI
590 CLOSE1
600 PRINT"WRITE NEXT PAGE?":INPUT"Y/N";W$
610 IFW$="N"THEN820
```

```
620 IFW$="Y"ANDR$="N"THEN790
630 IFW$="Y"ANDPC<NPTHEN510
640 IFPC>=NPTHENPRINT"MAX";NP;"PAGES REAC
    HED":GOTO820
650 GOTO600
660 OPEN1,1,0:PC=PC+1
670 IFPC>NPTHENPC=1:F=F+1
680 PRINT"{CLR}";
690 PRINT"{RVS}READ FILE";F;"PAGE";PC
700 FORK=1TO20
710 INPUT#1,O$(K)
720 PRINTO$(K):IFLEN(O$(K))=22THENPRINT"
    {UP}";
730 NEXTK
740 CLOSE1
750 PRINT"READ NEXT PAGE?":INPUT"Y/N";R$
760 IFR$="Y"THEN660
770 IFR$="N"THEN600
780 GOTO750
790 PRINT"PRESS STOP ON TAPE":R$="Y"
800 IF(PEEK(37151)AND64)=0THEN800
810 GOTO510
820 END
```

EDITYPE

Paul Bishop

This mini word processor lets you enter, edit, and save text to tape. It works with the VIC printer and any memory expansion.

If you are at all like me, the minute you saw the VIC-20 sitting there on the showroom table flashing its upper-lowercase mode, you smiled to yourself and said what a wonderful text storage and manipulation device it would make. *Wonderful* in this context means inexpensive, and Commodore promised us no less in its literature.

This program is a miniature word processor. It will allow the user to input text, edit it (with certain limitations), and save it to tape. The text may be printed on any line length specified, though it will not right justify. The program uses a word-wrapping scheme to minimize the limited display size. The program requires extra memory, but will work with any expander package (3K, 8K, 16K, etc.). The PRINT routines were written for the VIC 1515 or 1525 printer.

Entering Text

The program is menu driven, and we will discuss the options in detail. New mode is used for entering text. It is also the mode in which the formatting features are selected. Centering is done by pressing the up-arrow (↑) (next to the RESTORE key) at the beginning of the line that is to be centered. Remember to use the carriage return at the end of the line, and note that the line may not exceed the line length you intend to print.

The second function is an inset line length. This is selected by pressing the first bracket ([) (shifted colon) at the start of the text to be inset. All text before the next return character will be printed on the alternate line length, which will be specified during printing. Line numbering is something that I use frequently. It is selected with the second bracket (]), and the line will be printed with a number (numbered sequentially by the computer) before and after the line. Examples of all the formatting options are represented in Figure 1.

Figure 1. Formatting Options

```
                        Sample Text

This is a page of demo text for "Editype."  This is the normal
line length.  Note that there is no hyphenation of words in the
print routine, so the edges may be somewhat ragged.  Resetting
the line length may help.

        This is an inset line.  Insets may be set to
        any length and may be longer than the normal
        line length if necessary.

                    This line is autocentered.

1.      This is an example of a numbered line.                  1.

Note that the computer keeps track of line numbers.  The line
above could have been given any number as a starting point and
subsequent numbered lines would be renumbered from there.
```

Backspacing in the New mode may be done only with the DEL key and may continue only to the first character of the line on which the cursor rests. Any further DELeting will result in an Illegal Quantity error. If an error is in an earlier line, it must be corrected in the Edit mode. All keys repeat, and the British pound symbol (£) (next to CLR HOME) is used to return to the menu. Once the menu is chosen, no further text may be entered in the New mode. (This is something the user could change.)

A final note: Text entry becomes progressively slower as memory fills, and subsequent printing is also adversely affected by large quantities (relatively speaking) of text. So, although the low memory warning should keep you from overtyping the machine's capacity, it is best to save the text and then continue when the word-wrap starts to slow down.

Text entered in the New mode can be reviewed and modified in the Edit mode. The mode has three options: Forward, Correct, and Return to Main Menu. The Forward option scrolls through the text one screen line at a time. To make changes in entered text, use the Correct option. You will be given the prompt "error:", at which point you enter the characters you wish to change *as they appear in the text.* End your entry with the up-arrow (↑) key, *not* the RETURN key. The next prompt is "correction:". Enter the text as you wish it to appear in the corrected version. Again follow your input with the up-arrow key rather than RETURN. The computer will then search the text for the "error" and replace it with the "correction." If the search characters are not found in the text, the program will provide an error message.

Saving and Printing

The Save mode is straightforward in operation: simply press the S key and RETURN and the text will be stored under the title you entered in the New mode. Load is just like it. If you include a file name, the cassette drive will search for that file; otherwise it will load the first file it comes to. The Load and Print mode is for files too long to be contained in memory and is fairly automatic. You simply set the formatting in the Print mode, and let the computer do the rest.

The Print mode is also straightforward. First it asks for the normal line length. This may be any value up to 80, but values between 40 and 70 are recommended. Next you are asked for the inset line length. Again, this should be between 40 and 70. Next you are asked for s for single or d for double spacing. Finally, the computer asks for the number at which it will begin the sequential line numbering. This may be set at any value, but usually will be one.

Obviously, this program will not meet everyone's writing needs. I am looking forward to further memory expansion which will allow me to implement further editing functions, as well as longer text entry. And you may wish to delete functions which you will not use and add others. That is the beauty of a word processor written in BASIC.

Before we consider the program in detail, a few comments about operation will be in order here. First, the cursor does not function as well as it should. I am searching for a cure. In the meantime, if you find it more distracting than helpful, you may get rid of it by deleting POKE 204,0 from line 120. Also, from time to time, errors will happen which will cause the machine to default to BASIC. This is no cause for alarm. A few moments studying the program listing and a GOTO in the immediate mode will get you out of all but the worst spots. If in doubt, GOTO 51 (the menu).

Figure 2. New Mode Commands

↑	Center Text
[	Inset
]	Number Line
DEL key	Backspace
£	Return to Menu

Program Structure

Since I have included no documentation in the body of the program, I will list the various parts of it here. You will want to keep this handy for reference, since every REM you add will cost you valuable memory space.

Line 42 is initial housekeeping, setting variables and DIMing the text string array.

Lines 51-67 are the menu.

Lines 100-280 are the text entry and word-wrapping routine, including the delete routine in line 200.

Lines 3010-3350 are the string search and replace, the "Edit Mode."

Lines 3800-4710 are the print routine. Lines 4060-4095 are for getting a string of printing length. Lines 4200-4240 are used in the centering function. 4300-4710 are for tidying up the print strings and sending them to the printer.

Lines 5000-5080 are the load routine.

Lines 6000-6080 are for saving text.

Lines 7000-7009 are for the page numbering function.

Variable List

A$ is the actual text string.
C$ is the get character string in the New mode.
C4$ is the error string in the Edit mode.
C5$ is the correction string in the Edit mode.
C6$ is the right remainder of the string being searched for the error in the Edit mode.
DE$ is the string of the variable SL.
J$ is the get character string for the correction string in Edit mode.
M$ is the string for the mode selection in the menu.
P$ is the print string.
**T1$** is the leftover from P$ after searching for a space at the end of the line.
**T2$** is the working string of A$ in the Print mode.
W$ is the get string in the Edit mode.
**X$** is the working character in getting an 80-character line for P$.
Z$ is the get string for the Load mode.
LA is the normal line length.
LB is the inset line length.
LC is the line count.
PC is the page count.
SL is the line numbering counter.

Editype

```
42 PC=1:LC=1:FL=Ø:PRINTCHR$(14):DIMA$(20
   Ø):PRINT"{CLR}":POKE65Ø,128
51 M$=""
53 PRINT"{CLR}{3 SPACES}MODE
   {SHIFT-SPACE}SELECTION":PRINT:PRINT:P
   RINT"LP=LOAD{SHIFT-SPACE}AND
   {SHIFT-SPACE}PRINT":PRINT
55 PRINT"N=NEW":PRINT:PRINT"E=EDIT":PRIN
   T:PRINT"P=PRINT
58 PRINT:PRINT"S=SAVE":PRINT:PRINT"L=LOA
   D":PRINT:PRINT"C=CONTINUE"
6Ø PRINT:INPUT"SELECT{SHIFT-SPACE}MODE:
   ";M$
61 IFM$="E"THEN3Ø1Ø
62 IFM$="P"THEN38ØØ
63 IFM$="N"THEN1ØØ
64 IFM$="L"THEN5ØØØ
65 IFM$="S"THEN6ØØØ
66 IFM$="LP"THEN38ØØ
67 IFM$="C"THENFORB=1TOK-1:PRINTA$(B):NE
   XTB:PRINTA$(K);:GOTO12Ø
68 GOTO51
1ØØ FORA=1TO2ØØ:A$(A)="":NEXTA
1Ø3 INPUT"TYPE{SHIFT-SPACE}FILE
    {SHIFT-SPACE}NAME";V$
1Ø5 PRINT"{CLR}{7 SPACES}NEW MODE":K=1
12Ø POKE2Ø4,Ø:POKE2Ø7,Ø:GETC$:IFC$=""THE
    N12Ø
13Ø IFC$="{DOWN}"THEN12Ø
14Ø IFC$="{UP}"THEN12Ø
15Ø IFC$="{RIGHT}"THEN12Ø
16Ø IFC$="{LEFT}"THEN12Ø
17Ø IFC$="ß"THEN51
171 IFC$="{HOME}"THEN12Ø
172 IFC$="{CLR}"THEN12Ø
175 IFC$=CHR$(2Ø)AND LEN(A$(K))=ØTHEN12Ø
18Ø PRINTC$;
19Ø IFC$=CHR$(13)THENK=K+1:A$(K)=A$(K)+C
    $:GOTO12Ø
2ØØ IFC$=CHR$(2Ø)THENA$(K)=LEFT$(A$(K),L
    EN(A$(K))-1):GOTO12Ø
21Ø A$(K)=A$(K)+C$:C$="":IFLEN(A$(K))<22
    THEN12Ø
22Ø IFRIGHT$(A$(K),1)=CHR$(32)THEN24Ø
221 IFRIGHT$(A$(K),1)=CHR$(16Ø)THEN24Ø
23Ø A$(K+1)=RIGHT$(A$(K),1)+A$(K+1):A$(K
    )=LEFT$(A$(K),LEN(A$(K))-1):GOTO22Ø
```

```
240 FORU=1TO22-LEN(A$(K)):PRINTCHR$(20);
    :NEXTU
250 IFLEN(A$(K))<11THENPRINT,,
260 IFLEN(A$(K))>10THENPRINT,
264 IFA$(K)=""THENA$(K)=" "
265 IFFRE(0)<600THENPRINT"{RVS}MEMORY
    {SHIFT-SPACE}LOW{OFF}":PRINT
266 IFFRE(0)<500THEN51
270 K=K+1:PRINTA$(K);:GOTO120
280 GOTO51
3010 C4$="":C5$=""
3015 PRINT"{CLR}{5 SPACES}EDIT
     {SHIFT-SPACE}MODE":Q=1
3025 PRINT:PRINT"F=FORWARD":PRINT"←=RET
     URN TO MENU":PRINT"C=CORRECT"
3026 PRINT"SELECTION? "
3030 GETW$:IFW$=""THEN3030
3040 IFW$="F"THENPRINTA$(Q):Q=Q+1:IFQ>19
     9THEN51:GOTO3030
3055 IFW$="←"THEN51
3060 IFW$="C"THEN3200
3061 GOTO3030
3200 PRINT"ERROR:"
3210 FORA=1TO80
3220 GETJ$:IFJ$=""THEN3220
3225 IFJ$="↑"THEN3250
3226 IFJ$=CHR$(20)THENC4$=LEFT$(C4$,LEN(
     C4$)-1):GOTO3235
3230 C4$=C4$+J$
3235 PRINTJ$;
3240 NEXTA
3250 PRINT:PRINT"CORRECTION: "
3260 FORA=1TO80
3270 GETJ$:IFJ$=""THEN3270
3280 IFJ$="↑"THEN3310
3281 IFJ$=CHR$(20)THENC5$=LEFT$(C5$,LEN(
     C5$)-1):GOTO3290
3285 C5$=C5$+J$
3290 PRINTJ$;
3300 NEXTA
3310 PRINT"{CLR}{3 SPACES}{RVS}CORRECTIN
     G{OFF}"
3320 FORA=1TO200
3325 FORB=1TOLEN(A$(A))
3327 O=LEN(C4$)
3329 IFMID$(A$(A),B,O)=C4$THENOO=LEN(A$(
     A))-B+1-LEN(C4$)
3330 IFMID$(A$(A),B,O)=C4$THENC6$=RIGHT$
     (A$(A),OO)
```

```
3340 IFMID$(A$(A),B,O)=C4$THENA$(A)=LEFT
     $(A$(A),B-1):GOTO3344
3341 GOTO3346
3344 A$(A)=A$(A)+C5$+C6$:C4$="":C5$=""
3345 PRINT"{CLR}":FORH=1TOA:PRINTA$(H):N
     EXTH:Q=H::GOTO3025
3346 NEXTB
3347 NEXTA
3348 PRINT"{CLR}{RED}{RVS}ERROR
     {SHIFT-SPACE}NOT{SHIFT-SPACE}FOUND
     {BLU}{OFF}":PRINT:GOTO3025
3350 GOTO3010
3800 PRINT:INPUT"NORMAL{SHIFT-SPACE}LINE
     {SHIFT-SPACE}LENGTH";LA
3810 PRINT:INPUT "INSET{SHIFT-SPACE}LINE
     {SHIFT-SPACE}LENGTH";LB
3903 PRINT"SINGLE{SHIFT-SPACE}OR
     {SHIFT-SPACE}DOUBLE{6 SHIFT-SPACE}S
     PACE? S/D
3904 INPUTSD$
3905 INPUT"LINE{SHIFT-SPACE}NUMBERING
     {SHIFT-SPACE}#";SL
4000 T1$="":N=1:LL=LA
4002 OPEN4,4
4003 T2$="":P$="":LC=1
4010 PRINT#4:PRINT#4:PRINT#4
4016 LC=3
4040 CLOSE4,4
4050 IFA$(N)=""ANDM$="LP"THEN5002
4051 IFA$(N)=""THEN4660
4059 T2$=A$(N)
4060 FORA=1TOLL-LEN(P$)
4061 IFT2$=""THEN4094
4065 X$=LEFT$(T2$,1):T2$=RIGHT$(T2$,LEN(
     T2$)-1)
4075 IFX$="["THENLL=LB:GOTO4060
4076 IFX$="]"THENFL=1:GOTO4060
4080 IFX$="↑"THEN4200
4085 IFX$=CHR$(13)THEN4660
4090 P$=P$+X$
4094 IFLEN(T2$)=0THENN=N+1:GOTO4050
4095 NEXTA
4100 GOTO4610
4200 FORA=1TOLA
4210 X$=LEFT$(T2$,1):T2$=RIGHT$(T2$,LEN(
     T2$)-1)
4211 IFLEN(T2$)=0THENN=N+1:T2$=A$(N)
4214 IFA$(N)=""ANDLEN(T2$)=0THENP$=P$+X$
     :GOTO4660
```

```
4220 IFX$=CHR$(13)THEN4300
4230 P$=P$+X$
4240 NEXTA
4300 IN=(80-LEN(P$))/2:GOTO4670
4610 FORA=1TOLEN(P$)
4620 IFRIGHT$(P$,1)=CHR$(32)THEN4660
4622 IFRIGHT$(P$,1)=CHR$(160)THEN4660
4630 T1$=RIGHT$(P$,1)+T1$:P$=LEFT$(P$,LE
     N(P$)-1)
4640 NEXTA
4660 IFLEFT$(P$,1)=CHR$(32)THENP$=RIGHT$
     (P$,LEN(P$)-1)
4661 IFLEFT$(P$,1)=CHR$(160)THENP$=RIGHT
     $(P$,LEN(P$)-1)
4662 PRINTP$
4665 IN=(80-LL)/2
4666 DE$=STR$(SL):IFFL=1THENOPEN4,4
4667 IFFL=1THENPRINT#4,CHR$(17)DE$."SPC
     (IN-LEN(DE$)-1)P$SPC(76-LEN(P$)-IN)
     DE$."
4668 IFFL=1THENCLOSE4:LC=LC+1:SL=SL+1:P$
     ="":FL=0:P$=T1$:T1$="":GOTO4680
4670 OPEN4,4:PRINT#4,CHR$(17)SPC(IN)P$:C
     LOSE4,4:P$="":P$=T1$:T1$="":LC=LC+1
4680 IFSD$="D"THENOPEN4,4:PRINT#4:CLOSE4
     :LC=LC+1
4690 IFLC>60THEN7000
4700 IFX$=CHR$(13)THENLL=LA
4701 IFA$(N)=""ANDM$="LP"THENP$=P$+X$:GO
     TO5002
4705 IFA$(N)=""THEN51
4710 GOTO4060
5000 INPUT"TYPE{SHIFT-SPACE}FILE
     {SHIFT-SPACE}NAME";V$
5002 FORA=1TO200:A$(A)="":NEXTA
5005 PRINT"{CLR}{6 SPACES}LOAD
     {SHIFT-SPACE}MODE"
5010 OPEN1,1,0,V$
5015 PRINT"FILE{SHIFT-SPACE}OPEN, LOADIN
     G."
5020 FORA=1TO200
5025 FORB=1TO22
5030 GET#1,Z$
5031 A$(A)=A$(A)+Z$
5040 IFZ$=""THEN5065
5042 NEXTB
5050 NEXTA
5065 CLOSE1:N=1
5077 IFM$="LP"THENN=1:GOTO4050
```

```
5080 GOTO51
6000 PRINT"{CLR}SAVE{SHIFT-SPACE}MODE"
6010 OPEN1,1,1,V$
6030 FORA=1TO200
6040 PRINT#1,A$(A);
6050 IFA$(A)=""THEN6075
6060 NEXTA
6075 CLOSE1
6080 GOTO51
7000 OPEN4,4
7001 FORM=1TO66-LC
7002 PRINT#4
7003 NEXTM
7004 PRINT#4:PC=PC+1
7005 PRINT#4,CHR$(17)SPC(70)"PAGE "PC
7006 PRINT#4
7007 CLOSE4
7008 LC=3
7009 GOTO4060
```

UXB

Roger Hagerty

"UXB" is a game which tests both dexterity and nerve. Unexploded bombs litter a minefield. Your job is to quickly defuse the bombs. This game incorporates a technique called "chaining" by putting the instructions in a separate program. It is important to type and SAVE the first program and then type and SAVE the second program immediately after the first program on the same tape.

World War II. London is battered and scorched. And although there is a pause in the fighting, a peril remains among the rubble: UXBs, Unexploded Bombs. These are shells that failed to detonate, but remain a danger, their unstable nature making them literally time bombs.

Your Mission

You are an explosives expert, charged with the vital duty of defusing or harmlessly detonating the UXBs. Use the keys I, J, K, and M to move. I is up, M is down, J is left, and K is right. Touch your marker to a UXB to render it harmless.

A Few Complications

Your job is not as easy as it may sound. First, you have only 30 seconds to perform your task. Second, the field you're working in is also a minefield. Littered about the playfield are numerous colored bombs that you must avoid, lest you meet an untidy fate.

Using the keyboard for movement makes the game quite challenging, since it takes a while to get used to such movement. Hold a key down to continue movement in the selected direction, but let go before you hit a mine!

Chaining Programs

"UXB," like many programs for the unexpanded VIC, uses a technique called "chaining." Often, as in UXB, the instructions and character definitions are given in the first program and the main program is contained in the second program. The user can read the instructions while the second program is being loaded from tape.

Line 610 in UXB's first program starts the loading of the sec-

ond program. The VIC interprets line 610 as if you had pressed the STOP/RUN key while holding down the SHIFT key (LOAD and RUN a program from tape).

POKE 198,1 tells the computer there is one character in the keyboard buffer. POKE 631,131 places the ASCII code for the SHIFTed RUN/STOP key (131) into the buffer.

This technique can easily be used in your programs. It's a simple way to expand your VIC without an expander. Remember, though, variables will not pass from one program to another, and chained programs must be SAVEd one right after the other on the same tape.

Program 1. UXB — Instructions

```
300 PRINT"{CLR}"
310 POKE56,28
320 CH=32776
330 FORX=7184TO7600STEP2
340 POKEX,PEEK(CH):POKEX+1,PEEK(CH)
350 CH=CH+1:NEXTX
360 POKE36879,25
370 POKE36869,255
371 POKE36867,47
375 POKE36878,10
376 FORL=240TO180STEP-1
377 POKE36876,L
378 FORM=1TO20:NEXTM:NEXTL
379 POKE36876,0:POKE36877,200
380 FORL=5TO0STEP-2
381 POKE36878,L:NEXTL
382 POKE36877,0
390 PRINT"{5 RIGHT}{2 DOWN}{3 RIGHT}DANG
    ER":PRINT"{9 RIGHT}{2 DOWN}UXB"
400 FORI=1TO100
420 POKE36869,240
430 POKE36869,255
435 POKE36879,47
440 NEXTI
441 POKE36867,46
442 POKE36879,154:GOTO800
445 POKE36869,242:POKE36879,154
450 PRINT"{CLR}YOU HAVE BEEN SOMEHOW TRA
    NSPORTED INTO THE{2 SPACES}MIDDLE OF
     A FIELD{5 SPACES}WHICH";
460 PRINT" CONTAINS BOTH{3 SPACES}ANTIQU
    ATED BOMBS AND{2 SPACES}WWII UXB'S(U
    NEXPLODED GERMAN ROCKET";
```

```
470 PRINT" BOMBS). YOU MUST DE-FUSE THE
    {2 SPACES}UXB'S BY SIMPLY RUN-
    {2 SPACES}NING INTO THEIR TAILS.";
480 PRINT"IF YOU HIT AN OLD BOMBYOU WILL
     BE BLASTED!! IF YOU GET ALL THE "
490 PRINT"UXB'S YOU WILL GET TWOMORE ON
    THE NEXT ROUND-IF YOU DON'T MAKE IT"
    ;
500 PRINT" YOU GET TWO LESS (DOWNTO ZERO
    ).{5 LEFT}{3 DOWN}PRESS ANY KEY"
510 GET A$:IF A$=""THEN510
511 PRINT"{CLR}"
520 PRINT" THERE IS SCREEN WRAP-AROUND F
    ROM SIDE TO{3 SPACES}SIDE, BUT IF YO
    U RUN{2 SPACES}OVER THE ";
530 PRINT"TOP OR BOTTOMYOU WILL BE RETUR
    NED{2 SPACES}TO THE UPPER LEFTHAND C
    ORNER."
540 PRINT"{DOWN}{2 RIGHT}MOVEMENT KEYS A
    RE:{DOWN}
550 PRINTTAB(10)"{RVS}I{OFF}(UP)
560 PRINT"{DOWN}{5 RIGHT}(L){RVS}J{OFF}
    {4 RIGHT}{RVS}L{OFF}(RT)
570 PRINTTAB(10)"{DOWN}{RVS}M{OFF}(DOWN)
580 PRINT"{4 DOWN}{4 RIGHT}PRESS ANY KEY
590 GETA$:IFA$=""THEN590
595 PRINT"{CLR}":POKE36879,27
596 POKE51,0:POKE58,28:POKE55,0:POKE56,2
    8:CLR:CB=7168
597 READ A:IF A=-1 THEN 600
598 FOR N=0TO7:READ B:POKE(CB+A*8+N),B:N
    EXT
599 GOTO 597
600 PRINT"{2 DOWN}{3 RIGHT}PLEASE WAIT F
    OR{7 SPACES}TAPE TO LOAD"
610 POKE198,1:POKE631,131:END
800 POKE36869,240:PRINT"{3 DOWN}
    {5 RIGHT}INSTRUCTIONS?"
810 GETA$:IFA$=""THEN810
815 IFA$="N"THEN 595
820 GOTO445
900 DATA 1,153,219,189,153,129,66,36,36
910 DATA 17,126,255,199,203,211,227,255,
    126
920 DATA 24,36,36,36,36,60,36,66,129
930 DATA 26,4,24,24,60,126,126,126,60
940 DATA 32,0,0,0,0,0,0,0,0
950 DATA -1
```

Program 2. UXB — Game Program

```
1 POKE45,121:POKE46,21:POKE51,0:POKE55,0
  :CLR
3 POKE36869,255:QQ=10
4 A=30720:C=0:K=0:TI$="000000":CH=7954:Q
  =20
5 PRINT"{CLR}"
15 FORL=1TOQQ
16 M={2 SPACES}7680+INT(RND(1)*506)
17 POKEM,1:POKEM+A,C:POKEM+22,24:POKEM+2
   2+A,C
18 NEXT L
19 GOSUB1000
25 IFCH+D>8186 THENCH=7680:D=0
26 IFCH+D<7680 THENCH=7680:D=0
27 IFPEEK(CH+D)=1THENPOKECH+D,32:POKECH+
   D+22,32:GOTO200
28 IFPEEK(CH+D)=26THEN2000
29 IFTI>=2000THEN299
30 POKECH+D,17
31 POKE36878,15:POKE36876,220
32 FORP=1TO5:NEXTP
33 POKE36878,0:POKE36876,0
40 POKECH+D+A,C
41 FOR R=1TOQ :NEXTR
45 POKECH+D,32
70 IFPEEK(197)=12THEND=D-22{3 SPACES}
75 IFPEEK(197)=36THEND=D+22
80 IFPEEK(197)=21THEND=D+1
85 IFPEEK(197)=20THEND=D-1
90 IFTI<=500THENQ=10
100 IFTI=>1000THENQ= 8
110 IFTI=>1500THENQ= 5
120 IFTI=>1700THENQ=2
121 GOTO25
200 K=K+1
210 POKE36877,220
215 FORL=14 TO 5STEP-1
220 POKE36878,L
230 FORM=1TO50
240 NEXTM
250 NEXT L
260 POKE36877,0
270 POKE36878,0
275 IFK=QQTHEN300
280 GOTO25
299 POKE36869,240:PRINT"{CLR}{DOWN}
    {RIGHT}{DOWN} YOUR TIME IS UP":FORT=
    1TO1500:NEXTT
```

```
300 POKE36869,240: PRINT"{CLR}{4 DOWN}
    {4 RIGHT}YOUR SCORE=";K
301 PRINT"{2 DOWN}NUMBER OF UXB'S WAS";Q
    Q
302 IFK>HSC THEN 340
335 PRINT"{2 DOWN}{4 RIGHT}HIGH SCORE=";
    HSC:GOTO342
340 PRINT"{2 DOWN}{4 RIGHT}HIGH SCORE=";
    K"{4 DOWN}{9 RIGHT}{RVS}A NEW HIGH
    {OFF}"
341 HSC=K
342 FORDR=1TO3000:NEXT
344 IFK=QQTHENQQ=QQ+2:GOTO346
345 IFK<QQTHENQQ=QQ-2:GOTO346
346 IFQQ=0THEN3
350 D=0:POKE36869,255:GOTO4
1000 FORL=1TO85
1010 R=7680+INT(RND(1)*506)
1015 IFPEEK(R)=1THENPOKER,1:POKER+A,C:GO
     TO1030
1020 POKER,26
1025 POKER+A,INT(RND(1)*6)+2
1030 NEXTL
1036 POKE7954,32
1040 RETURN
2000 POKE36869,240:PRINT"{CLR}{RVS}
     {RIGHT}{9 DOWN}YOU'VE BEEN BLASTED!
     {OFF}"
2010 POKE36878,15
2020 FORI=225TO128STEP-2
2030 POKE36877,I
2040 FORD=1TO50:NEXTD
2050 NEXTI
2055 FORX=14TO0STEP-.1
2060 POKE36878,X
2065 NEXTX
2066 POKE36878,0:POKE36877,0
2080 GOTO300
```

1

A Day at the Races

Robert B. Ferree

This simulation of a racetrack, complete with animation and color, can serve as an effective model for beginners interested in programming their own games.

An occasional complaint heard about game playing on personal computers is the lack of the high-resolution graphics of arcade machines. In the direct or program mode, the basic VIC with 5K has a resolution of 22 x 23. This makes the mechanics of arcade games possible, but the movement is rather jerky. The VIC can be improved to a resolution of 176 x 184 through BASIC with programmable characters.

VIC Game Techniques

First, the programmer needs to know about programmable characters. An in-depth explanation is found in the *VIC-20 Programmer's Reference Guide.* Briefly, the unexpanded VIC has memory locations from 7168 to 7679 for programmable characters. Each programmable character is made up of eight bytes. By POKEing numbers from 0 to 255 into these locations, a character is programmed. To shift into the programmed character mode, you POKE 36869,255. POKEing 36869,240 will return you to the direct, or program, mode. To find the memory location of a character, use:

```
10 INPUT"CHARACTER";A$
20 A=ASC(A$)
30 IF A>=64 THEN ML=(A-64)*8+7168:PRINTML
    ;"-";ML+7GOTO 50
40 ML=A*8+7168:PRINT ML;"-";ML+7
50 GOTO 10
```

INPUTting A into the above program should give a reading of 7176-7183, which is the location of the character A.

The eight bytes of memory for a character each have eight digits in binary. If you place these eight bytes in binary, each

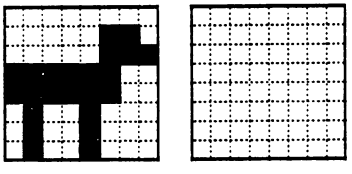
under the previous one, and imagine the 1's are pixel dots and 0's are spaces, you can decide what eight numbers should go into these locations. For example, type:

```
100 FOR C=7432 TO 7439:READ A:POKE C,A:NE
    XT C
110 DATA 0,6,7,252,252,72,72,72
```

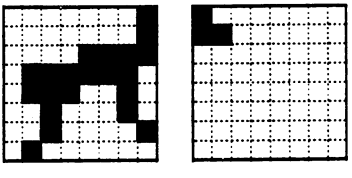
Now RUN. Nothing happens! Now type POKE 36869,255 and everything will turn to garbage. Type a few !'s and you should see a horse. The character ! has been reprogrammed to be a horse. Try your own, remembering to figure from the top to the bottom, or the character will appear upside down.

The next trick is to move these programmable characters. Most programs for personal computers in BASIC move their graphics by drawing a character and then erasing it while drawing it again in the next space. This can cause a rather jerky motion. By programming a series of characters, each just one pixel dot farther in the direction you wish to go, and then erasing the previous character, you can improve your resolution to 176 x 184. For example, your first two characters might be:

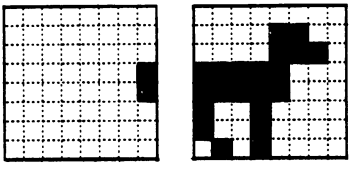
The space character is the area that you are heading for.



The next two characters might be:

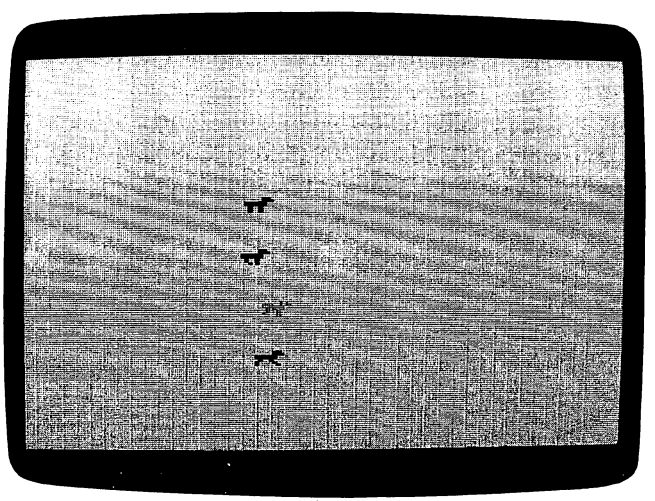


This would continue until:



Now you are ready to do the series over again in the next two character spaces.

For a demonstration of how this can work, try Program 1. Before RUNning, take out line 330. We will use it later in preparation for the game. RUN the program, and if the horses look funny, check your DATA lines (50-210). If it all works right, add line 330 and SAVE. This information will provide the programmable characters for the game. All programmable character information will remain in memory until the machine is turned off or the memory location information is changed. The latter can be intentional, or it can happen accidentally if these locations are not protected. To protect all programmable character locations, you will need to POKE 52,28 and POKE 56,28. For this game, only the upper half of the space for programmable characters is protected, leaving more program space.



It looks like a close race at "A Day at the Races."

After debugging the game (Program 2), be sure to SAVE it right after the already SAVEd Program 1. RUNning Program 1 will automatically LOAD/RUN Program 2 (the game).

The Rules of the Game

"A Day At The Races" is a game for one to six players. It consists of five races on random track conditions. Each horse is given odds for a particular track in the initialization. These odds are kept throughout the five races. Try to avoid long names for people or

horses; they may cause an OUT OF MEMORY error. Five letters work nicely. Each win pays three-to-one while each loss costs you the amount you bet.

A Major Hint

Remember each horse's performance on the different track conditions. They may run the same way the next time that track condition comes up.

The game sections are marked with REM statements. Changing the denominators in lines 110-130 will change the difference between each horse's odds. You can change the number of races in line 680, the payoff in line 620, and the losses in line 630.

You will notice that one horse moves nicely when it is alone, but things slow down considerably when four horses are involved. Still, I think the programmed characters enhance the movement of the game (it was originally written with the character π as the horses and they were moving one space at a time).

Program 1. A Day at the Races — Programmable Characters

```
10 PRINT"{CLR}":POKE36879,8:S=7856:Z=33
20 PRINT"{4 DOWN}{RIGHT}WELCOME TO VIC DO
   WNS"
30 FORX=1TO2500:NEXT
40 FORF=7424TO7559:READA:POKEF,A:NEXT
50 DATA0,0,0,0,0,0,0,0:REM READ DATA FOR
   HORSE
60 DATA0,6,7,252,252,72,72,72
70 DATA0,0,0,0,0,0,0,0
80 DATA3,3,14,126,116,36,34,32
90 DATA0,128,0,0,0,0,0,0
100 DATA1,1,7,63,58,17,16,32
110 DATA128,192,0,0,0,0,0,0
120 DATA0,0,31,31,9,16,16,0
130 DATA192,224,128,128,0,128,64,0
140 DATA0,0,15,15,4,8,8,0
150 DATA96,112,192,192,128,64,64,0
160 DATA0,0,7,7,2,2,4,0
170 DATA0,48,56,224,224,64,64,64
180 DATA0,0,3,3,1,1,1,0
190 DATA0,24,156,240,112,32,64,64
200 DATA0,0,0,1,1,0,0,0
210 DATA0,12,14,248,248,144,144,80
220 PRINT"{CLR}"
230 POKE36869,255:REM SWITCH TO PROGRAMMA
    BLE CHARACTERS
```

```
240 POKES+C-2,32:POKES+C-1,32:REM ERASE O
    LD HORSE
250 POKES+C,Z:POKES+C+1,Z+1:REM DRAW NEW
    HORSE
260 Z=Z+2:REM COUNT HORSES IN THIS SERIES
270 IFZ=49THENC=C+1:Z=33:REM IF SERIES IS
     FINISHED MOVE TO NEXT SERIES
280 H=H+1:REM]COUNT HORSES
290 IFH<169THENGOTO240:IF NOT THE END OF
    THE LINE, CONTINUE
300 PRINT"{CLR}":POKE36869,240:PRINT"
    {4 DOWN}EACH PLAYER STARTS{6 SPACES}
    WITH $500.":REM SWITCH BACK
310 PRINT"{2 DOWN}A WINNING BET PAYS
    {6 SPACES}3 TO 1."
320 PRINT"{2 DOWN}PRESS PLAY AND WAIT."
330 PRINT"{BLK}":POKE631,131:POKE198,1:RE
    M LOAD AND RUN NEXT PROGRAM
```

Program 2. A Day at the Races — Game Program

```
40 REM INITIALIZATION
50 POKE52,29:POKE56,29:REM PROTECT MEMOR
   Y LOCATIONS ABOVE 7424
60 PRINT"{CLR}{WHT}":POKE36879,8
70 Z=33:Z1=Z:Z2=Z:Z3=Z:Z4=Z
80 POKE36878,15:SO=36877
90 S1=7856:S2=7922:S3=7988:S4=8054
100 REM GIVE ODDS
110 D1=RND(1)/12:D2=RND(1)/12:D3=RND(1)/
    12:D4=RND(1)/12
120 U1=RND(1)/12:U2=RND(1)/12:U3=RND(1)/
    12:U4=RND(1)/12
130 T1=RND(1)/12:T2=RND(1)/12:T3=RND(1)/
    12:T4=RND(1)/12
140 REM NAME PLAYERS AND HORSES
150 INPUT"{CLR}{2 DOWN}HOW MANY PLAYERS"
    ;PL
160 IFPL>6ORPL<1THENGOTO150
170 FOR X=1TOPL:W(X)=500
180 INPUT"{2 DOWN}NAME OF PLAYER";N$(X):
    NEXT
190 PRINT"{CLR}{DOWN}NAME THE FOUR HORSE
    S:":FORX=1TO4:INPUTA$(X):NEXT
200 REM SETS TRACK CONDITIONS
210 TR=RND(1)*10:PRINT"{BLK}{CLR}"
220 IFTR<3THENCO$="DRY":O1=D1:O2=D2:O3=D
    3:O4=D4:POKE36879,248:GOTO260
```

```
230 IFTR<6THENCO$="TURF":O1=T1:O2=T2:O3=
    T3:O4=T4:POKE36879,216:GOTO260
240 CO$="MUDDY":POKE36879,200
250 O1=U1:O2=U2:O3=U3:O4=U4
260 R=R+1:PRINT"{CLR}{DOWN}RACE #"R:PRIN
    T"THE TRACK IS ";CO$
270 FORY=1TO4:PRINTTAB(5)A$(Y):NEXT
280 FORX=1TOPL:PRINTN$(X);W(X):NEXT:PRIN
    T
290 FOR Q=1 TO PL:IF W(Q)=0 THEN B(Q)=0:
    GOTO 298
292 PRINTN$(Q);:INPUT" BETS";B(Q):IF B(Q
    )<=W(Q) THEN 296
294 PRINT"CAN'T BET THAT MUCH!":GOTO 292
296 INPUT"{2 SPACES}ON";B$(Q)
298 NEXT
300 PRINT"{CLR}"
310 PRINT"{2 DOWN} ALL BETS ARE DOWN!!!"
320 REM SETS COLOR OF TRACK (PATHS)
330 FORX=S1TOS1+22:POKEX+30720,0:NEXT
340 FORX=S2TOS2+22:POKEX+30720,4:NEXT
350 FORX=S3TOS3+22:POKEX+30720,5:NEXT
360 FORX=S4TOS4+22:POKEX+30720,6:NEXT
370 READP:IFP=-1THENPRINT"{UP}
    {21 SPACES}":GOTO390
380 READD:POKE36876,P:FORX=1TOD:NEXT:POK
    E36876,0:FORX=1TO50:NEXT:GOTO370
390 POKE36869,255:REM PROGRAMMABLE CHARC
    TER MODE
400 REM MOVE HORSES AND HORSES SOUND
410 M1=RND(1)+O1:IFM1>.9THEN440
420 POKES1+C1-1,32:POKES1+C1-2,32:POKES1
    +C1,Z1:POKES1+1+C1,Z1+1:Z1=Z1+2:H1=H
    1+1
430 POKESO,200:POKESO,0:IFZ1=49THENC1=C1
    +1:Z1=Z
440 M2=RND(1)+O2:IFM2>.9THEN470
450 POKES2+C2-1,32:POKES2+C2-2,32:POKES2
    +C2,Z2:POKES2+1+C2,Z2+1:Z2=Z2+2:H2=H
    2+1
460 IFZ2=49THENC2=C2+1:Z2=Z
470 M3=RND(1)+O3:IFM3>.9THEN500
480 POKES3+C3-1,32:POKES3+C3-2,32:POKES3
    +C3,Z3:POKES3+1+C3,Z3+1:Z3=Z3+2:H3=H
    3+1
490 POKESO,130:POKESO,0:IFZ3=49THENC3=C3
    +1:Z3=Z
500 M4=RND(1)+O4:IFM4>.9THEN530
```

```
510 POKES4+C4-1,32:POKES4+C4-2,32:POKES4
    +C4,Z4:POKES4+1+C4,Z4+1:Z4=Z4+2:H4=H
    4+1
520 IFZ4=49THENC4=C4+1:Z4=Z
530 REM FIND WINNER
540 IFH1>168THENJ$=A$(1):GOTO590
550 IFH2>168THENJ$=A$(2):GOTO590
560 IFH3>168THENJ$=A$(3):GOTO590
570 IFH4>168THENJ$=A$(4):GOTO590
580 GOTO400
590 FORC=1TO10:FORX=150TO250STEP7:POKE36
    876,X:NEXT:NEXT:POKE36876,0
600 POKE36869,240:PRINT"{CLR}"J$" WINS"
610 FORX=1TOPL
620 IFB$(X)=J$THENW(X)=W(X)+B(X)*3:GOTO6
    40
630 W(X)=W(X)-B(X):IFB$(X)<>J$THENPRINT"
    {DOWN}"N$(X)" LOSES $";B(X):GOTO650
640 PRINT"{DOWN}"N$(X)" WINS $";B(X)*3
650 NEXT
660 REM READY FOR NEXT RACE
670 H1=0:H2=0:H3=0:H4=0:Z1=Z:Z2=Z:Z3=Z:Z
    4=Z:C1=0:C2=0:C3=0:C4=0
680 IFR=5THENGOTO710
690 FORX=1TO5500:NEXT
700 RESTORE:GOTO200
710 REM ENDING
720 FORX=1TO2500:NEXT
730 PRINT"{4 DOWN}{3 SPACES}HAVE A GOOD
    DAY!{2 DOWN}":FORX=1TOPL:PRINTN$(X);
    "$";W(X):NEXT
740 REM SONG DATA
750 DATA195,50,209,50,219,50,225,50,225,
    50,225,50
760 DATA219,50,219,50,219,50,209,50,219,
    50,209,50,195,300
770 DATA195,50,209,50,219,50,225,50,225,
    50,225,50
780 DATA195,50,195,50,195,50,209,300,-1
```

1

Catch

Ronnie Koffler

Here is a simple game which can be enjoyed by both children and adults. There are three options. The simplest option, no motion, is ideal for preschoolers. Other options offer more of a challenge.

You are a sky diver floating in the air. You have only 50 seconds to catch as many balloons as you can before opening your chute. There are three options in this game of "Catch." You can select balloons which do not move at all, balloons which move every so often, or balloons which move constantly.

Playing Catch

When the first screen appears, you will be given a choice of which of the three options you wish to play. No motion is by far the easiest, and constant motion the most difficult. After you make your pick, begin play by pressing the S key. Use the K key to move left, the L key to move right, the F1 key to go up, and the F7 key to go down.

The high score is kept from game to game along with the name of the person who got the high score. This simple added feature makes competition a bit easier, especially for young children.

Good luck and good diving!

Catch

```
3 POKE36879,110:CLR
4 HI=0
5 K$="":L$="":PRINT"{CLR}{WHT}":PRINTCHR$
   (142)
6 PRINTCHR$(147)"{WHT}"
9 PRINT"{DOWN}{4 RIGHT}";
10 PRINT"{3 DOWN}{RVS}{YEL}{3 RIGHT}CATCH
    !{OFF}{WHT}":PRINT"{2 RIGHT}{2 DOWN}'
    {RVS}K{OFF}'-LEFT-'{RVS}L{OFF}'-RIGHT
    {5 RIGHT}{DOWN}{WHT}F1--UP//F7--DOWN"
12 PRINT"{DOWN}{RIGHT}YOU HAVE 50 SECONDS
    "
13 IFI$<>""THEN19
```

```
14 PRINT"{HOME}":FORK=1TO13:PRINT:NEXTK:P
    RINT"{3 RIGHT}1-CONSTANT MOTION "
15 PRINT"{DOWN}{3 RIGHT}2-{3 RIGHT} MOTIO
    N"
16 PRINT"{DOWN}{2 RIGHT}ANY KEY-NO MOTION
    "
17 GOSUB9600
18 GETI$:IFI$=""THEN18
19 GOTO998
20 TI$="000000"
21 SC=0
30 S=8165
35 POKEA,42
36 POKE36879,110
40 POKES,160
41 PRINTCHR$(142)"{WHT}{HOME}{2 RIGHT}TIM
    E :";MID$(TI$,5,4);
44 IFHI=0THENPRINT"{2 RIGHT}HIGH --":GOTO
    46
45 PRINT"{3 RIGHT}HIGH";HI
46 PRINT"{HOME}{DOWN}{YEL}{2 RIGHT}{I}
    {SHIFT-SPACE}{I}{SHIFT-SPACE}{I}
    {SHIFT-SPACE}{I}{SHIFT-SPACE}{I}
    {SHIFT-SPACE}{I}{SHIFT-SPACE}{I}
    {SHIFT-SPACE}{I}{SHIFT-SPACE}{I}
    {WHT}";
47 IFI$<>"1"THEN55
48 B=INT(RND(10)*7)-2
49 IFB=-2THENB=-22
50 IFB>2THENB=0
51 IFB=2THENB=22
53 POKEA,32
54 A=A+B:POKEA,42
55 IFI$<>"2"THEN67
56 CC=INT(RND(1)*25)+1
57 HJ=INT(RND(1)*99)+7724
58 IFCC=6THENPOKEA,32:POKEHJ,42:A=HJ
67 GETQ$:IFQ$="K"THENPOKES,160:S=S-1:POKE
    S+1,32
68 IFQ$="L"THENPOKES,160:S=S+1:POKES-1,32
70 IFQ$="{F1}"THENPOKES,160:S=S-22:POKES+
    22,32
78 IFQ$="{F7}"THENPOKES,160:S=S+22:POKES-
    22,32
96 IFS<7724THENS=S+22
97 IFS>8185THENS=S-22
98 IFA>8185THENA=A-22
99 IFA<7724THENPOKEA,32:A=A+22
100 IFS=ATHENPOKEA,32:GOTO9050
```

```
110 IFTI$<>"000050"THEN35
120 IFTI$="000050"THEN1900
950 PRINTCHR$(5):A=INT(RND(1)*445+7724):G
    OTO30
998 IFHI>0THEN1004
999 A=INT(RND(1)*445+7702):PRINT"{5 UP}
    {2 RIGHT}{2 SPACES}HIT S TO START
    {4 SPACES}";
1000 PRINT"{75 SPACES}";
1001 PRINT"{13 SPACES}";
1002 GETWW$:IFWW$<>"S"THEN1002
1003 PRINTCHR$(147):GOTO20
1004 A=INT(RND(1)*445+7724):PRINT"{HOME}"
     :FORG=1TO14:PRINT:NEXTG:PRINT"
     {2 RIGHT}{2 SPACES}HIT S TO START"
1005 GETY$:IFY$<>"S"THEN1005
1006 PRINTCHR$(147):GOTO20
1101 POKE36879,110
1102 PRINTCHR$(147)"{DOWN}SORRY YOU'RE TI
     ME HAS{2 SPACES}RUN OUT---NUMBER OF
     {4 SPACES}CATCHES IS--";SC
1103 IFSC=0THEN1106
1104 IFSC=HITHENPRINT"{DOWN}{2 RIGHT}YOU
     HAVE TIED THE{DOWN}";SPC(8);"HIGH S
     CORE":GOTO1111
1105 IFSC>HITHENHI=SC:L$=" ":PRINT"{DOWN}
     {3 RIGHT} YOU HAVE THE {DOWN} ";SPC
     (9);"HIGH SCORE"
1106 IFSC=<5THENPRINT"{RIGHT}{2 DOWN}RATI
     NG-NOT TOO GOOD"
1107 IFSC=>13THENPRINT"{2 RIGHT}{2 DOWN}R
     ATING-EXCELLENT"
1108 IFSC=0ORSC<HITHEN1113
1109 FORFF=1TO2:PRINT:NEXTFF:PRINT"PLEASE
       INPUT YOUR NAME"
1110 INPUT K$:GOTO1113
1111 FORJ=1TO2:PRINT:NEXTJ:PRINT"PLEASE I
     NPUT YOUR NAME"
1112 INPUTL$
1113 FORJJ=1TO2:PRINT:NEXTJJ:PRINT"
     {RIGHT}DO YOU WANT TO PLAY
     {6 RIGHT}AGAIN(Y=YES)":INPUTCC$
1115 IFCC$="Y"THENSC=0:GOTO5999
1120 POKE36879,27:PRINT"{CLR}":END
1900 PRINTCHR$(142)"{CLR}":POKE36879,108
1901 O=7680:OO=8164
1902 FORT=1TO20
```

```
1903 O=O+1:POKEO,120
1904 OO=OO+1:POKEOO,119
1905 FORR=1TO39:NEXTR
1907 NEXTT
1917 X=7680:XX=7701
1918 FORT=1TO21.3
1919 POKEX,118
1923 X=X+22:POKEX,118
1926 POKEXX,117
1927 XX=XX+22:POKEXX,117
1928 FORE=1TO39:NEXTE
1929 NEXTT
1953 S=7716:A=7708
1954 FORT=1TO21
1955 POKES,81
1957 S=S+22:POKES,81:POKES-22,32
1967 POKEA,81
1968 A=A+22:POKEA,81:POK?EA-22,32
1969 IFA=7928ANDS=7936THENGOSUB5000
1970 FORZ=1TO60:NEXTZ
1971 NEXTT
2000 POKE36878,15
2001 FORL=148TO220STEP.7
2002 POKE36876,L
2003 NEXTL
2004 FORL=128TO200
2005 POKE36876,L
2006 NEXTL
2007 FORL=200TO128STEP-1
2008 POKE36876,L
2009 NEXTL
2010 POKE36878,0
2011 POKE36876,0
2118 POKES,32:POKEA,32
2120 PRINT"{UP}{RIGHT}{15 SPACES}"
2122 FORR=1TO1500:NEXTR
2123 GOTO1101
3000 FORT=110TO170STEP2:POKE36879,T:NEXTT
      :PRINT"{WHT}"
3001 RETURN
5000 PRINTCHR$(142);"{HOME}":FORT=1TO10:P
      RINT:NEXTT:PRINTSPC(7);"{RVS}{YEL}T
      IME!!!{WHT}"
5010 RETURN
5020 END
5999 IFHI=0THEN3
6000 IFK$=""THENK$="??":IFL$=""THENL$="??
      "
```

```
6001 PRINTCHR$(147):FORC=1TO6:PRINT:NEXTC
      :PRINT"{2 RIGHT}HIGH SCORE--"HI"BY
      {DOWN}{2 RIGHT}{2 SPACES}{RVS}
      {RIGHT}1{OFF}--{RIGHT}";K$
6002 IFK$=L$THEN6010
6003 IFL$=" "THEN6010
6005 IFL$<>" "THENPRINT"{DOWN} {2 RIGHT}
      {RVS}2{OFF}--{RIGHT}";L$
6010 FORT=1TO2000:NEXTT:GOTO6
7999 END
8000 GETI$:IFI$=""THEN8000
8001 RETURN
9000 FORDD=2TO20
9002 POKE7680+(DD*22),170
9004 POKE7680+21+(DD*22),170
9005 NEXTDD
9010 RETURN
9050 FORJ=1TO8:PRINT:NEXTJ:PRINTSPC(8);"
      {RED}CATCH";
9051 FORS=1TO2:GOSUB3000:NEXTS:PRINTCHR$(
      147):SC=SC+1:GOTO950
9052 END
9500 FORP=7681TO7701STEP1.5
9501 POKEP,170
9502 NEXTP
9503 FORW=8165TO8185STEP1.5
9504 POKEW,170
9505 NEXTW:GOSUB9000
9506 RETURN
9600 FORXX=135TO25STEP-1:POKE36865,XX:FOR
      LO=1TO30:NEXTLO:NEXTXX
9601 POKE36879,111
9650 GOTO9500
```

1

Financial Advisor

Steve Hamilton

"Financial Advisor" is a useful tool for persons who need to analyze their loans, and supplies valuable information to the would-be borrower. Most options need only the unexpanded VIC.

Before I had a personal computer, I found myself, when trying to analyze a loan, calling the bank to get the particulars such as current interest rates, monthly payments, and amortization schedules. Now, with the aid of my VIC-20, I do this all at home without a single telephone call.

The following program was designed to provide answers to most of the questions I had when analyzing any loan based on a fixed interest rate and declining monthly balance, such as home mortgages and automobile loans. The program will LOAD and RUN on the unexpanded VIC; however, the full potential of the program will not be realized without 8K or more of additional memory.

Using the Financial Advisor

There are four sections to the program which are named *Range, Amortize, Balance Due,* and *Monthly Payment.* After you LOAD the program and type RUN, the menu is displayed showing the four possible sections ready to be used. By pressing one of the keys corresponding to the particular section desired, the program jumps to that section and awaits entry of the particulars of the loan. After using any section, the program will return to the menu.

Monthly Payment

The *Monthly Payment* section first asks for the amount of the loan. The next entry is for the interest rate. Be sure to enter the interest rate in the form specified by the input prompt; otherwise the monthly payment amount will be incorrect. The final entry is the term of the loan. At this point the VIC will calculate and display the monthly payment schedule for the life of the loan. You will then be given the option of changing your entries. If you do not want to change any of the figures, type N followed by RETURN. If you do wish to change one or all of the figures, press Y followed

by RETURN. Then either press RETURN to retain the above displayed figure or enter the appropriate change.

Range

The *Range* section is enabled by pressing F1. The term of the loan is the first required entry. You will then be asked what monthly payment you believe you can afford to pay. The next input required is a midpoint interest rate. The program will then calculate a range of eleven points, five points below and five points above the midpoint interest rate. The display will tell you how much you can borrow at each interest rate based on your affordable monthly payments. I have found this part of the program very valuable. As a reminder, be sure to enter the interest rate in the form specified by the input prompt.

Amortize

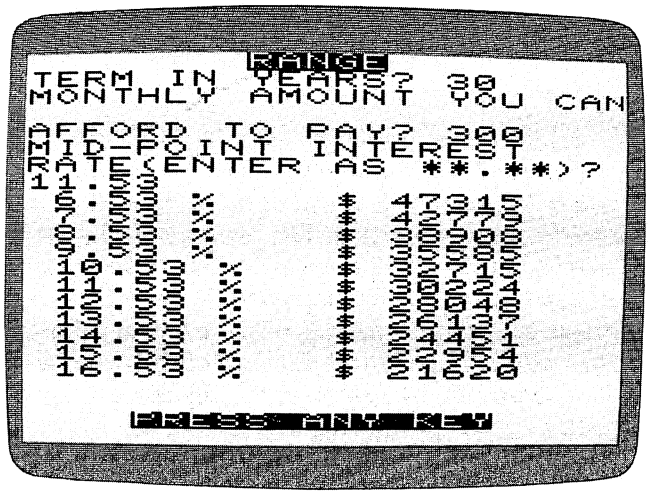
Pressing F3 will bring you to the *Amortize* section of the program. After receiving the required entries, three columns of figures will be displayed. Reading across they represent: the number of the month for which payment is due; the amount of the payment which is for that month's principal; and the balance due on the loan after this payment. These three figures will be displayed for the entire term of the loan. To slow down the screen scrolling, press and hold the CTRL key, or press RUN/STOP to halt processing. Type CONT to resume processing.

Balance Due

All of the above sections can be executed on the unexpanded VIC. The *Balance Due* section can be run on the unexpanded VIC if you are analyzing a declining balance loan with a term of 48 months or less, such as an automobile loan. If you happen to have 8K or more of memory expansion, you will be able to process a normal home mortgage of 30 years. Processing a 30-year loan requires just over 9K bytes of RAM.

After making the required entries for this part of the program, do not despair if nothing happens on the screen immediately. The processor is calculating and assigning values to 720 variables on a 30-year loan. I have seen this take as long as 28 seconds.

At this point you will be given the opportunity to calculate how much interest will be saved if you pay extra towards the principal.



The "Financial Advisor" will help you decide how much you can afford to borrow.

Something Extra

This concludes the basic functions of the program. There is one other interesting calculation that can be performed after having run the *Balance Due* section. Since, for each month, the principal payment and the interest payment have been assigned a specific value, we can pull out this information upon demand. Suppose you wanted to know how much of your payments are for interest during the second year of the loan, which will be for months 13 through 24. After having RUN the Balance Due section, press the RUN/STOP key. The principal payments are DIMensioned as PP(T) and the interest payments are DIMensioned as IP(T) where T represents the term of the loan in months. Now, type the following command without a line number:

```
Q=0:FOR T=13 TO 24:Q=Q+IP(T):NEXT:PRINT Q
```

The figure which will be displayed is the total amount of interest that will be paid during the period from month 13 through month 24.

If you take the time to enter the program and experiment a little, I think that you too will find this to be an invaluable tool. If you do not feel like typing in the entire program, you should be able to type in the section that interests you, since I tried to make each section independent of the others.

Financial Advisor

```
5 REM FINANCIAL ADVISOR
10 PRINT"{CLR}{2 RIGHT}{RVS}FINANCIAL ADV
    ISOR":PRINTSPC(5);"{3 DOWN}SELECTIONS
    :"
20 PRINT"{2 DOWN}{4 RIGHT}{RVS}F1{OFF}=RA
    NGE":PRINT"{4 RIGHT}{RVS}F3{OFF}=AMOR
    TIZE"
30 PRINT"{4 RIGHT}{RVS}F5{OFF}=BALANCE DU
    E":PRINT"{RVS}RETURN{OFF}=MONTHLY PAY
    MENT"
50 GETA$:IFA$=""THEN50RUN
60 IFA$=CHR$(133)THEN340
70 IFA$=CHR$(134)THEN490
80 IFA$=CHR$(135)THEN620
90 IFA$<>CHR$(133)ANDA$<>CHR$(134)ANDA$<>
    CHR$(135)ANDA$<>CHR$(13)THEN50
100 PRINT"{CLR}{RVS}{3 RIGHT}MONTHLY PAYM
     ENTS{OFF}":INPUT"AMOUNT OF LOAN";P
110 PRINT"INTEREST RATE(ENTER"
120 INPUT"AS **.**)";I
130 INPUT"TERM IN YRS.";T
140 Z=P*(I/1200)
150 X={2 SPACES}(1+(I/1200))↑(-12*T)
160 Y=1-X
170 M=Z/Y
190 PRINT"{CLR}LOAN AMOUNT=$";P
200 PRINT"TERM=";T;"YEARS"
210 PRINT"INTEREST RATE=";I;"%"
220 PRINT"YOUR MONTHLY PAYMENT{2 SPACES}W
     OULD BE";" $";INT(M*100+.5)/100
230 PRINT:PRINT"DO YOU WISH TO CHANGE"
240 PRINT"ANY FIGURES";:INPUTQ$
260 IF Q$="N"THEN10
270 PRINT"IF YOU DO NOT WANT"
280 PRINT"TO CHANGE THE EXISTING"
290 PRINT"FIGURE PRESS {RVS}RETURN{OFF}."
300 INPUT"LOAN AMOUNT";P
310 INPUT"TERM IN YEARS";T
320 INPUT"INTEREST RATE";I
330 GOTO140
340 PRINT"{CLR}";SPC(8);"{RVS}RANGE":INPU
     T"TERM IN YEARS";T
350 PRINT"MONTHLY AMOUNT YOU CAN"
360 INPUT"AFFORD TO PAY";M
370 PRINT"MID-POINT INTEREST":PRINT"RATE(
     ENTER AS **.**)";:INPUTI
390 I1=I
```

```
400 FOR I=I-5 TO I1+5
410 X=(1-(1+(I/1200))↑(-12*T))*M
420 Y=I/1200
430 P=X/Y
440 PRINT I;"%","$"INT(P)
450 NEXTI
460 PRINT"{2 DOWN}{4 RIGHT}{RVS}PRESS ANY
    KEY"
470 GET A$: IF A$="" THEN 470
480 GOTO10
490 PRINT"{CLR}";SPC(7);"{RVS}AMORTIZE":I
    NPUT"AMOUNT OF LOAN";M
500 PRINT"INTEREST RATE,(ENTER"
510 PRINT"AS .****)";:INPUT I
520 INPUT"MONTHLY PAYMENT";P
530 INPUT"TERM IN MONTHS";T:Q=0
540 X=INT(M*I/12*100+.5)/100:Y=INT((P-X)*
    100+.5)/100
550 Q=Q+1
560 M=INT((M-Y)*100+.5)/100
570 PRINTQ;Y;M
580 IFQ<T THEN 540
590 PRINT"{RVS}{4 RIGHT}PRESS ANY KEY"
600 GETA$:IFA$=""THEN600
610 GOTO10
620 PRINT"{CLR}{RVS}TO FIGURE BALANCE DUE
    ."
630 INPUT"TERM IN MONTHS";T
640 DIMM(T),PP(T),IP(T),B(T)
650 INPUT"AMOUNT BORROWED";B(0)
660 PRINT"INTEREST RATE(ENTER"
670 INPUT"AS .****)";I
680 INPUT"MO.PAYMENT";P
690 FORQ=1TOT
700 IP(Q)=INT(B(Y)*I/12*100+.5)/100
710 PP(Q)=INT((P-IP(Q))*100+.5)/100
720 B(Y+1)=INT((B(Y)-PP(Q))*100+.5)/100
730 Y=Y+1
740 NEXTQ
750 INPUT"EXTRA PAID";EP
760 INPUT"NO.OF MONTHS PAID";ZZ
770 FORF=1TOZZ
780 G=PP(F)+H
790 H=G
800 NEXTF
810 HH=H+EP
820 K=1
830 L=PP(K)+J
840 J=L
```

```
850 IFJ>=HHTHEN880
860 K=K+1
870 GOTO830
880 FORN=ZZ+1TOK
890 O=IP(N)+R
900 R=O
910 NEXTN
920 IFEP=0THENR=0:IFZZ=0THENHH=0
930 PRINT:PRINT"INT.SAVED=$"R
940 PRINT:PRINT"EXTRA PAID=$"EP
950 PRINT:PRINT"TOTAL PRINC.PAID="
960 PRINT"$"HH
970 PRINT:PRINT"BALANCE DUE=$"INT((B(0)-H
    H)*100+.5)/100
980 PRINT"{RVS}{4 RIGHT}PRESS ANY KEY"
990 GETA$:IFA$=""THEN990
1000 CLR
1010 GOTO10
```

Banner

Michael Habeck and Michael Tyborski

Is your printer tired of listings or sitting idle? If it is, you may find "Banner" to be a refreshing change of pace.

"Banner" produces display messages on the VIC 1515 or 1525 or an RS-232 printer. Unlike simpler programs, it prints upper- and lowercase text and graphic characters. It also has a half-size character mode. And more important, it allows format intermixing within one messsage.

Your friends will love to create their own banners. Just watch your paper supply and have fun. How about a five-foot "HAPPY BIRTHDAY" or "WELCOME HOME"?

Program Operation

When you RUN the program, you will see an introduction and a printer type request. You should press RETURN if you are using the VIC 1515 or 1525 printer.

You may now enter a message. To print a half-size character, you should precede a letter with SHIFT S. Similarly, use SHIFT L to print a lowercase letter. Both modes may be used together. These print formats are shown in the figure.

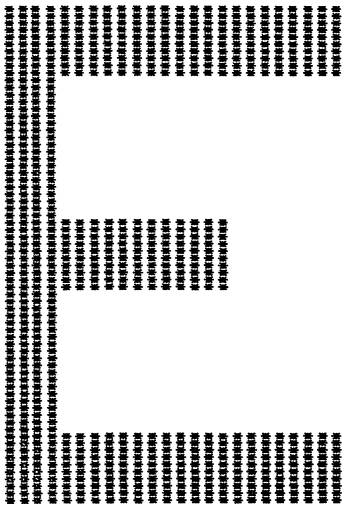
OPENing to the Printer

Before studying the program listing, you should know how to open and access a file. This is necessary because Commodore computers use channelized input/output. To the programmer, this means a general program can access different input/output devices.

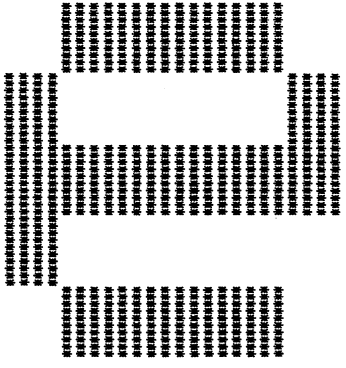
You OPEN a channel using the OPEN file#, device# statement. The file number specifies a channel for transferring data to a device, and can be an integer from 1 to 255. It is the same number that is used with the CLOSE, GET#, INPUT#, and PRINT# statements for a device. The device number specifies which device will be used and is set within that device.

Printers will use device numbers 2, 4, or 5. Of these, device 2 specifies an RS-232 printer. The VIC 1515 or 1525 printer can be either device 4 or 5. This is switch selectable, but usually the printer will be device 4.

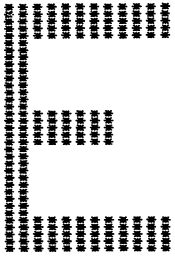
Four Possible Print Modes



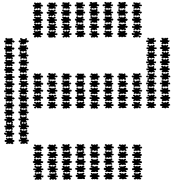
Full-Size Capital



Full-Size Lowercase



Half-Size Capital



Half-Size Lowercase

Note: These examples have been reduced to 45 percent of the original.

RS-232 printers must be set up in the OPEN statement. This requires complete specification of all parameters: baud rate, word length, number of stop bits, and parity. Fortunately, RS-232 interface board manuals explain how this is done.

After the file is OPENed, you may send data to the printer using the PRINT or PRINT# statement. The PRINT statement may be used only in the CMD mode. This mode sends all output to the device specified in the OPEN statement. OPEN 1,4:CMD 1, for example, sends all printed material to the VIC printer.

A PRINT#1 statement, however, allows data to be selectively sent to the printer. This is useful for summary printouts or printer control.

We found that multiple PRINT# statements produced DEVICE NOT PRESENT errors when used with the VIC printer. This occurred after only a few consecutive prints. Because of this problem, we used the CMD mode in Banner.

Program Description

After initializing and printing an introduction, Banner asks for a message. If a RETURN is entered, it repeats the request.

Banner then enters the main loop. It reads each letter in the message, obtains character matrix data, changes the data format, and prints it. It also checks for special function requests.

Lines 340 and 350 test for these functions. If a SHIFT S is read, line 340 reduces the character size and gets another character. Line 320 prevents an error if no letter is found.

Line 350 similarly checks for SHIFT L. If found, it sets the character matrix pointer to the start of the lowercase character set.

Lines 360 to 430 convert the ASCII code for a character into a character position in the character generator ROM. This was necessary because of duplicate codes and different order. It also allows graphic characters to be printed.

The character matrix data is then read in lines 440 to 520. This data is also rotated ninety degrees for proper printing. In effect, this converts the character definition from row to column format. The new data is stored in variable P$.

You should note the special use of the character generator. Since it can be read with the PEEK statement, it eliminates many character definition DATA statements. This, in turn, reduces the program size.

After data conversion, lines 530 to 620 print a character. They read each character in variable P$ and print either a series of

spaces or asterisks. Line 570 compensates for half-size characters. If it was not included, they would not print inline with full-size characters.

Finally, the printer output buffer is emptied, and the file is CLOSEd in line 630. The computer will then ask if another run is desired. Any response starting with Y will continue the run.

Banner

```
100 POKE36879,27
110 PRINT"{CLR}{7 DOWN}";TAB(6);"VIC BANN
    ER"
150 FORI=1TO3500:NEXT
160 PRINTCHR$(14)
170 PRINT"{CLR}{6 DOWN}{4 SPACES}BANNER P
    RODUCES":PRINT"DISPLAY MESSAGES ON A
    "
180 PRINT"VIC OR RS-232 PRINTER."
190 PRINT"{UP}IN ADDITION TO UPPER":PRINT
    "AND LOWER CASE, IT "
200 PRINT"ALSO PRINTS HALF-SIZE":PRINT"CH
    ARACTERS."
210 INPUT"{3 DOWN}RS-232 PRINTER
    {2 SPACES}N{3 LEFT}";R$
220 IFLEFT$(R$,1)="Y"THEN OPEN128,2,3,CHR
    $(6)+CHR$(0):LF=128:GOTO240
230 OPEN4,4:LF=4
240 PRINTCHR$(142)
250 PRINT"{CLR}{DOWN}{2 SPACES}{RVS}SPECI
    AL FUNCTIONS{OFF}"
260 PRINT"{2 DOWN}SHIFT L (L)=LOWER CASE"
270 PRINT"SHIFT S (S)=HALF SIZE"
280 M$="":INPUT "{3 DOWN}MESSAGE";M$
290 L=LEN(M$):IFL=0THEN250
300 FOR I=1 TO L
310 H=10:W=4:M=0
320 IFI>L THEN 630
330 C=ASC(MID$(M$,I,1))
340 IFC=211THENH=5:W=2:I=I+1:GOTO 320
350 IFC=204THENM=2048:I=I+1:GOTO 320
360 REM TRANSLATE ASC CODE
370 REM TO CHAR MATRIX CODE
380 IFC>63 AND C<96 THEN C=C-64:GOTO460
390 IFC>95 AND C<128 THEN C=C-32:GOTO460
400 IFC>159 AND C<192 THEN C=C-64:GOTO460
410 IFC>191 AND C<224 THEN C=C-128:GOTO46
    0
420 IFC>223 AND C<255 THEN C=C-128:GOTO46
    0
```

```
430 IF C=255 THEN C=94
440 REM OBTAIN CHAR.
450 REM MATRIX DATA
460 M=M+32768+C*8:P$=""
470 FOR J=7 TO 0 STEP -1
480 FOR B=7 TO 0 STEP -1
490 P=PEEK(M+J)AND2↑B
500 A$=" ":IF P<>0 THEN A$="*"
510 P$=P$+A$
520 NEXT:NEXT
530 REM PRINT LETTER
540 CMD LF
550 FOR J=1 TO 8
560 FORK=1TOW
570 IFH=5THENPRINT"{5 SPACES}";
580 FOR B=0 TO 7
590 P=J+B*8:A$=MID$(P$,P,1)
600 FORX=1TOH:PRINTA$;:NEXT
610 NEXT:PRINT
620 NEXT:NEXT
630 NEXT
640 PRINT#LF:CLOSE LF
650 INPUT "{6 DOWN}AGAIN{2 SPACES}Y
    {3 LEFT}";M$
660 IF LEFT$(M$,1)="Y" THEN 220
670 END
```

—Chapter 2—

Graphics

—2—

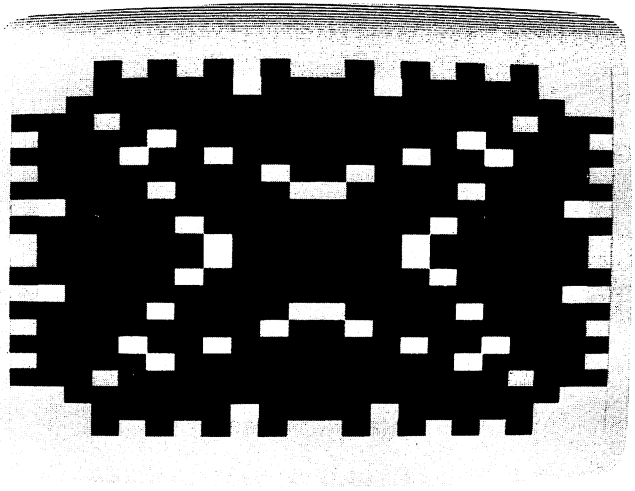
Kaleidoscope

Alan W. Poole

This program, for any size VIC, illustrates the amazing color displays available.

Amaze your friends with a continuous display of random patterns on your TV. The sound and music will continue endlessly as you sit mesmerized by the beauty of it all. When you see a display that really pleases you, simply press the space bar to freeze the action.

If you want to talk but the sound of "Kaleidoscope" is too loud, simply press the S key to shut off the sound. Press S again to turn the sound back on.



"Kaleidoscope" will amaze you with its ever-changing screen.

For anyone interested in how this works, here is a list of the variables used:

Variables

A: Used in the MOD function and used as the address to plot a square
B: Used in the MOD function

C: Color number
CC: Color number for border
I,J: Loop counters
K$: Key pressed
N: Number of function being used to calculate coordinates of points
R: Random number
S: Kaleidoscope stopped flag. 1 =kaleidoscope going, 0 =kaleidoscope stopped
S1: Speaker address
SA: Screen memory starting address
SD: Sound flag. 1 =sound on, 0 =sound off
X,Y: Position to plot a square

Kaleidoscope

```
20 GOSUB5000
97 REM
98 REM *** MAIN LOOP ***
99 REM
100 FORI=0TO999999
110 FORJ=0TO10
120 ONNGOSUB500,550,600,650,700,750
129 REM PLOT POINTS
130 A=SA+22*Y+X:POKEA,160:POKEA+30720,C
140 A=SA+22*(21-Y)+X:POKEA,160:POKEA+307
    20,C
150 A=SA+22*Y+21-X:POKEA,160:POKEA+30720
    ,C
160 A=SA+22*(21-Y)+21-X:POKEA,160:POKEA+
    30720,C
170 A=SA+22*X+Y:POKEA,160:POKEA+30720,C
180 A=SA+22*X+21-Y:POKEA,160:POKEA+30720
    ,C
190 A=SA+22*(21-X)+Y:POKEA,160:POKEA+307
    20,C
200 A=SA+22*(21-X)+21-Y:POKEA,160:POKEA+
    30720,C
205 GETK$:IFK$="S"THENSD=1-SD:IFSD=0THEN
    POKEV,0
210 IFSD=0THEN230
220 POKES1,128+(X+Y)*2.8:POKEV,15
230 IFK$=" "THENS=1-S
235 IFS=0THENPOKEV,0:GETK$:GOTO230
239 REM RANDOMLY CHANGE COLOR, FUNCTION,
     AND BORDER
240 IFRND(1)<.1THENC=INT(RND(1)*16)
270 IFRND(1)<.07THENN=INT(RND(1)*6+1)
275 IFRND(1)<.065THENGOSUB1000
```

```
280 NEXT:NEXT:END
497 REM
498 REM *** FUNCTIONS TO CALCULATE POINT
    S ***
499 REM
500 B=15:X=FNMOD(ABS(I-SGN(J-6)*(J+2)))
510 B=21:Y=FNMOD(J*J+2*J+7)
520 RETURN
550 B=18:X=FNMOD(I*J)
560 B=12:Y=FNMOD(ABS(ABS(I-ABS(2*I-2*J))
    ))
570 RETURN
600 B=20:X=FNMOD(I)
610 B=20:Y=FNMOD(J)
620 RETURN
650 B=12:X=FNMOD(ABS(Y-J))
660 B=20:Y=FNMOD(ABS(2*J-ABS(I-ABS(2*I-J
    )))+RND(1)*3)
670 RETURN
700 B=16:X=FNMOD(ABS(I-SGN(J-10)*J))
710 B=21:Y=FNMOD(I*J)
720 RETURN
750 B=22:X=FNMOD(ABS(3*J-ABS(2*I-ABS(2*I
    -J))))
760 B=22:Y=FNMOD(ABS(2*J-ABS(2*X-ABS(2*X
    -J))))
770 RETURN
997 REM
998 REM *** CHANGE BORDER COLOR ***
999 REM
1000 CC=INT(RND(1)*7)
1010 POKE36879,PEEK(36879)AND248ORCC
1020 POKE646,CC
1029 REM CHANGE 23RD ROW TO MATCH BORDER
1030 PRINT"{HOME}{22 DOWN}";
1040 PRINT"{RVS}{21 SPACES}";
1045 POKESA+505,160:POKESA+31225,CC
1050 RETURN
4997 REM
4998 REM *** INITIALIZATION ***
4999 REM
5000 PRINT"{HOME}{CLR}":POKE36879,8
5010 PRINTTAB(5)"{RED}K{CYN}A{PUR}L{GRN}
     E{BLU}I{YEL}D{WHT}O{RED}S{CYN}C
     {PUR}O{GRN}P{BLU}E"
5020 PRINT:PRINT:PRINT"{GRN}PRESS SPACE
     BAR TO{4 SPACES}FREEZE KALEIDOSCOPE
     "
5025 PRINT:PRINT"PRESS SPACE BAR AGAIN T
     O CONTINUE"
```

```
5030 PRINT:PRINT"PRESS S TO TURN OFF
     {3 SPACES}SOUND"
5035 PRINT:PRINT"PRESS S AGAIN TO TURN S
     OUND BACK ON"
5040 PRINT"{4 DOWN}"
5050 PRINT"{WHT}PRESS RETURN TO BEGIN";
5060 GETK$:R=RND(1):IFK$<>CHR$(13)THEN50
     60
5070 R=RND(R*1000)
5080 SD=1:S=1:N=INT(RND(1)*5+1):C=INT(RN
     D(1)*7+1)
5090 PRINT"{CLR}"
5100 SA=4*(PEEK(36866)AND128)+64*(PEEK(3
     6869)AND112)
5110 S1=36876:V=36878
5120 DEFFNMOD(A)=INT((A/B-INT(A/B))*B+.0
     5)*SGN(A/B)
5130 RETURN
```

—2—

Understanding High-Resolution Graphics

Roger N. Trendowski

This article explores high-resolution graphics on both the unexpanded and extended 8K VIC.

The VIC performs high-resolution (hi-res) graphics through bitmapping the screen. Bitmapping is a method where each dot of resolution on the screen (called a *pixel*) is assigned its own bit in memory. If the bit is one, then the pixel is on; if zero, the pixel is off.

Your screen displays 506 alpha/numeric/graphic characters, 22 horizontal and 23 vertical. Since each character is made of 8x8 pixels, your screen consists of 32,384 pixels. With hi-res graphics, you can selectively turn off or on each of these 32,384 pixels — if you have enough memory (more about memory requirements later). With enough memory, the X or horizontal coordinate may range from 0 to 176, and Y from 0 to 184.

VIC Technique

Bitmapping is done on the VIC using the "programmable character" technique — when you POKE a screen location with a number from within that location. Try this on an unexpanded VIC: press the RUN-STOP/RESTORE keys, then type in:

```
POKE36879,62
POKE7690,0
```

This places a character display code of zero in the top middle of your screen (location 7690). An @ character should appear. The first POKE turns the screen blue so that you can see the character. To display this character, VIC takes the display code and looks up the corresponding eight lines in ROM (Read Only Memory) starting with location 32768.

In the case of display code 0, the first eight bytes (memory locations) of ROM are used — 32768 through 32775. Each eight-bit byte in ROM defines a row of pixels which make up part of the @ character. Now, if the display code 1 was POKEd instead of 0, an A would be displayed — it is stored in eight bytes of ROM starting at 32776.

The next step in understanding the bitmapping technique is to see how programmable characters are changed. Since the ROM area where the alpha/numeric/graphic characters are stored cannot be changed by a POKE command, we must change the VIC pointer from ROM to unused locations in RAM (Random Access Memory). To change this pointer, type in:

```
POKE36869,253
```

This memory location, which contains both the character memory pointer and a screen memory pointer, now points to RAM location 5120. The graphic garbage on your screen represents random data stored in the new eight-byte character RAM locations. Hit the RUN-STOP/RESTORE keys to clear the screen.

Try this short program which will show some of the fundamentals of hi-res graphics and bitmapping.

```
10 POKE36879,62
20 FORI =5120TO6143:POKEI,0:NEXT
30 POKE7680,0
40 POKE36869,253
50 POKE5120,1
60 GOTO60
```

Look at what has happened at the top left of the screen. A pixel has been turned on in the first row. Line 20 of the program clears random data out of the RAM memory locations 5120-6143. Line 30 puts a display character code of zero in 7680 (normally an @character equals display code zero). Line 40 changes the character pointer from ROM to RAM location 5120. Line 50 creates a new character in the first of eight bytes that define display character zero. The remaining seven bytes of display character zero (locations 5121 through 5127) remain clear, meaning their bits are equal to zeros. Line 50 causes bit position 0 (right-most bit in the byte) to equal one. Line 60 causes VIC to remain in a loop so that the screen does not display "READY" and interrupt our demonstration. A conclusion from this exercise is that setting a bit to one in programmable character memory (for example, 5120, bit #0) turns on a corresponding pixel.

Try using binary word encoding with different values (0-255) in line 50 of the above program.

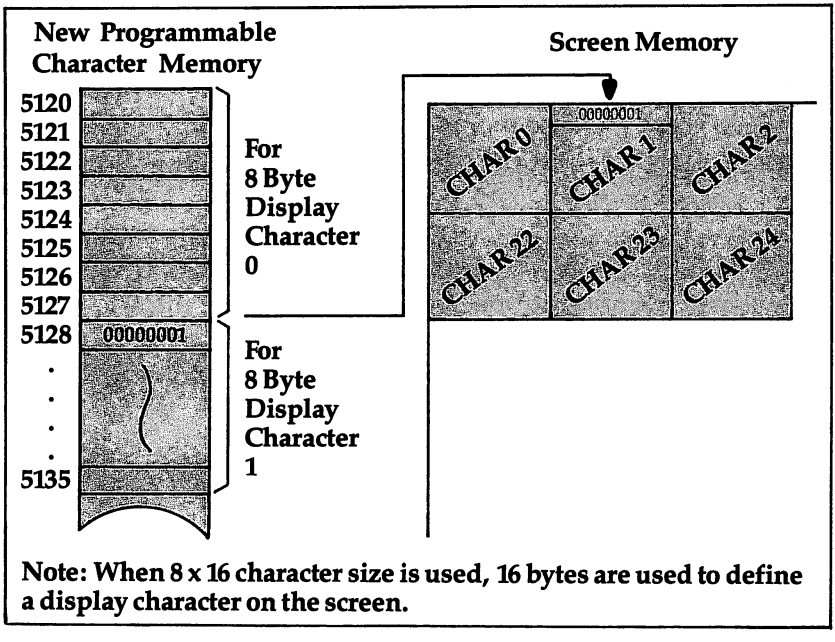
```
Bit #  7 6 5 4 3 2 1 0
Byte 5120 ^ ^ ^ ^ ^ ^ ^ ^
          0 0 0 0 0 0 0 1 = 1
          0 0 0 0 0 0 1 0 = 2
          1 0 0 0 0 0 0 0 = 128
```

To expand your understanding, type the following change to the above program and RUN it:

```
30 U=0:FORJ=7680TO7701:POKEJ,U:U=U+1:NEXT
50 POKE5128,1
```

The screen should show a pixel set in the 16th position from the left. Line 30 POKEd display codes of 0,1,2...21 into VIC's screen memory 7680 through 7701. Corresponding eight-byte blocks of RAM, starting with 5120, are cleared except for the bit 0 in byte 5128 — the top row of character number 1. Therefore, VIC turns on the corresponding screen pixel.

Transferring Character Data to Screen Memory



Note: When 8 x 16 character size is used, 16 bytes are used to define a display character on the screen.

—2—

Display Characters

If there are 506 character positions on the screen and only 256 possible display characters, then the question is: how do you fill up the rest of the screen? Use an obscure memory location — 36867, bit 0.

Type NEW and then type the following lines without line numbers:

```
POKE36879,62
POKE36867,(PEEK(36867)OR1)
POKE7690,0
```

Among graphic garbage, two characters should have appeared at the top center of the screen: an @ over an A. The second line changed the VIC to a character matrix size of 8x16 (when bit 0 of this location equals 1). The VIC now uses the first 16 bytes to define display character 0. The third line POKEs display code zero into location 7690. In this way, by POKEing from 0 through 253 display codes on the screen, we can display all 506 character positions.

Memory Requirements

As mentioned earlier, bitmapping the entire screen would require 32,384 pixels or 4048 bytes of RAM (32,384 divided by eight bits per byte). With the original VIC-20, you have only 3583 bytes of BASIC RAM to work with for both the program and bitmapping. Therefore, you will have to limit the area of the screen you map. With a 3K or 8K memory expander cartridge, you can map a larger portion of the screen. It takes both the 3K and 8K expansions to bitmap the entire screen.

When using an 8K expander, you must also perform some extra operations. A critical step will be to locate your hi-res program above screen memory and programmable character memory. I suggest location 8192, which is the first location in the 8K expander. The following 8K hi-res demonstration program will explain this technique.

X and Y Coordinate Calculations

Given that we now know how to turn a pixel off or on by changing a bit in programmable character memory (5120 +), we still must have the program take an X or Y coordinate and translate it to the corresponding byte number and bit location. The section "High Resolution Graphics" (p. 88) in the *VIC-20 Programmer's Reference*

Guide provides this information. The following calculations must be made by the program:

CHAR = INT(X/8) +INT(Y/16)*22

This gives the display code of the character you want to change. Next, calculate the proper row in the character by using:

ROW = (Y/16-INT(Y/16))*16

From the CHAR# and ROW#, you can calculate the byte where X and Y lie.

BYTE = 5120 +16*CHAR +ROW

The last calculation to be made identifies which bit must be changed.

BIT = 7-(X-(INT(X/8)*8))

To turn on any bit with the coordinates X,Y, use this formula:

POKE BYTE,PEEK (BYTE) OR (2↑ BIT)

Example

Program 1, for the unexpanded 5K VIC, bitmaps approximately two-thirds of the screen and allows you to control pixel plotting with a joystick. The portion of the screen used for hi-res graphics is limited by your BASIC RAM area. Only 1022 bytes are left available for a BASIC program (locations 4096 to 5019). By changing the programmable character pointer from location 5120 to 6144 or 7168 (see Table 1), you make more bytes available for your BASIC program; therefore, there is less bitmap area of the screen.

Table 1. Important Memory Locations for Hi-Res Graphics (5K VIC)

7680	Start of screen memory
5120 or 6144 or 7168	Start of special RAM for programmable characters
36869	Pointer to character set RAM memory 253 for location 5120 254 for location 6144 255 for location 7168
36867	Sets 8x16 dot character size (Bit 0 = 1)

In Program 1, line 50 sets up parameters for joystick control and starting X and Y coordinates. Line 60 colors the screen so that pixels will show. Line 70 clears all programmable character locations. Line 80 changes the VIC screen to an 8x16 character matrix size. Line 90 POKEs display codes zero through 153 in screen memory locations 7680 through 7832. If you insert an END statement between lines 90 and 100, you can see the display characters as taken from ROM. Line 100 changes the character pointer from ROM to RAM (location 5120). The screen clears to black because there are no programmable characters defined in 5120 to 7679.

The main program loop starts at line 110. This line points to the subroutine for reading the X and Y coordinates from the joystick. Lines 120 through 160 perform the necessary character (CH), row (RO), byte (BY), and bit (BI) calculations and operations to turn on a pixel. Warning: When you are playing with the demo program, don't go out of bounds or else you will invade other important memory locations. Strange things will appear!

Example Program for 8K Expanded VIC-20

This demonstration program will bitmap approximately 75 percent of the screen, leaving 8192 bytes free for your application program. By the way, these 8192 bytes are all located in the 8K expander. The 75 percent limitation results from the VIC requirement that all screen memory and programmable character memory be resident in the VIC and not in the 8K RAM expander.

Type in Program 2 and SAVE it. Next type in the following three POKE commands and then LOAD Program 2.

```
POKE44,32
POKE642,32
POKE8192,0
```

These three POKEs are critical! The first and second commands place the new page number of where your BASIC program will be loaded into RAM. The page number is derived by dividing the intended starting address by 256 since there are 256 bytes per page in the VIC (8192/256 =32). The third command zeros the first word of your BASIC program area — a must if you expect this program to run. Table 2 indicates the important memory locations for a VIC with +8K expander.

The explanation of this 8K program is the same as for the 5K demo program, except for three lines. Line 90 now contains the

Table 2. Important Memory Locations for Hi-Res Graphics (VIC with +8K Expander)

43,44	Pointer to start of BASIC Program (Normally, 1,18; change to 1,32 for location 8193)
641,642	Pointer to start of memory (Normally, 0,18; change to 0,32 for location 8192)
5120 or 6144 or 7168	Start of special RAM for programmable characters
8192	First memory location of BASIC program area. Must be set to zero.
36869	Pointer to character set RAM memory, normally 192; must be set to: 205 for 5120 206 for 6144 207 for 7168
36867	Sets 8x16 dot character size (Bit 0 = 1)

starting screen address of 4096 and character display codes up to 190. Line 100 POKEs a 205 into the character pointer to point to location 5120. This difference (253 vs. 205) is due to the dual function that 36869 performs. Only the lower four bits of this location contain the character memory pointer. Line 295 is also changed. The Y represents the maximum Y coordinate you can turn on with the joystick.

Program 1. 5K VIC Hi-Res Graphics

```
10 REM ORIGINAL 5K VIC{3 SPACES}EXAMPLE
   OF HIGH RES{3 SPACES}GRAPHICS
40 REM
50 DD=37154:P1=37151:P2=37152:X=10:Y=10
60 POKE36879,8:PRINT"{CLR}"
70 FORI=5120TO8185:POKEI,0:NEXT
80 POKE36867,PEEK(36867)OR1
90 FORI=0TO153:POKE7680+I,I:NEXTI
100 POKE36869,253
110 GOSUB200
120 CH=INT(X/8)+INT(Y/16)*22
130 RO=(Y/16-INT(Y/16))*16
140 BY=5120+16*CH+RO
```

```
150 BI=7-(X-(INT(X/8)*8))
160 POKEBY,PEEK(BY)OR(2↑BI)
170 GOTO110
180 REM
200 POKEDD,127:P=PEEK(P2)AND128
210 J0=-(P=0)
220 POKEDD,255:P=PEEK(P1)
230 J1=-((PAND8)=0)
240 J2=-((PAND16)=0)
250 J3=-((PAND4)=0)
260 IFJ0=1THENX=X+1
270 IFJ2=1THENX=X-1
280 IFJ1=1THENY=Y+1
290 IFJ3=1THENY=Y-1
295 IFY>104THENY=104
300 RETURN
```

Program 2. 8K VIC Hi-Res Graphics

```
10 REM ORIGINAL 8K VIC{2 SPACES}EXAMPLE O
   F HIGH RES GRAPHICS
40 REM
50 DD=37154:P1=37151:P2=37152:X=10:Y=10
60 POKE36879,8:PRINT"{CLR}"
70 FORI=5120TO8185:POKEI,0:NEXT
80 POKE36867,PEEK(36867)OR1
90 FORI=0TO190:POKE4096+I,I:NEXTI
100 POKE36869,205
110 GOSUB200
120 CH=INT(X/8)+INT(Y/16)*22
130 RO=(Y/16-INT(Y/16))*16
140 BY=5120+16*CH+RO
150 BI=7-(X-(INT(X/8)*8))
160 POKEBY,PEEK(BY)OR(2↑BI)
170 GOTO110
180 REM
200 POKEDD,127:P=PEEK(P2)AND128
210 J0=-(P=0)
220 POKEDD,255:P=PEEK(P1)
230 J1=-((PAND8)=0)
240 J2=-((PAND16)=0)
250 J3=-((PAND4)=0)
260 IFJ0=1THENX=X+1
270 IFJ2=1THENX=X-1
280 IFJ1=1THENY=Y+1
290 IFJ3=1THENY=Y-1
295 IFY>143THENY=143
300 RETURN
```

—2—

PIXELATOR

James Calloway

"Pixelator" is an easier way to design custom characters. Three accompanying programs let you save and load the character data from cassette and convert it into DATA statements — ready to use in a program.

The first time you design your own characters on the VIC-20, the process can be downright thrilling. Marking off graph paper in eight-by-eight squares and drawing in a figure. Converting each line into a number as if the dark squares were binary one and the light squares were binary zero. Storing the numbers in memory.

Then you POKE the magic address, 36869. The screen fills with gobbledygook. But wait! Isn't that a spaceship there where the A of READY is supposed to be? And that three-legged alien must be the D.

Once the thrill wears off, the work can turn to drudgery. Converting your design into numbers is bad enough, but the job of typing all those numbers into DATA statements is not only boring but also subject to typographical error. A slip of the finger and your beautiful rocket cruiser looks as if it had been shot full of laser holes.

Designing Characters with Pixelator

A program called "Pixelator" restores some of the thrill of designing screen characters. Pixelator gives you four large eight-by-eight work areas on the screen for creating, editing, and comparing characters. Pixelator then stores those characters in RAM. On standard VICs with 3.5K memory, Pixelator will store up to 64 characters. With additional memory, the program will store up to 128 characters; it also can retrieve from memory any character you have already stored. You can even copy from the VIC's own ROM character set and change those characters to suit your needs.

Like most small computers, the VIC stores mosaics or maps of its characters in ROM (addresses 32768 to 36863). Unlike some other computers, whose characters may be five pixels wide by seven pixels tall, the VIC's characters are eight by eight. (A pixel is simply the smallest portion of the video image that a particular computer can control.) That makes the VIC's characters look a bit

squat, but it's a tidy use of memory. Eight bytes are needed to describe a single character, with each byte corresponding to a horizontal line of the character. The vertical information comes from breaking the bytes into binary ones and zeros, corresponding to dark and light areas.

Just Enough Memory

By POKEing different numbers into address 36869, you can change where the Video Interface Chip looks for its character maps. You do this automatically when you change the keyboard from graphics to text mode. Graphics is a value of 240 at 36869, and text is 242. The value in between, 241, represents reversed graphics characters, but using the reversed characters doesn't normally change the value at 36869.

A value of 252 moves the map location to 4096, the start of standard 3.5K memory. Above 252 the corresponding address increases by increments of 1024, up to 7168 for a value of 255. Because of the length of the Pixelator program, it uses the highest value. (For a fuller explanation of what happens at address 36869, consult Jim Butterfield's "Browsing the VIC Chip" in *COMPUTE!'s First Book of VIC.*)

The Pixelator program, once it is up and running, consumes almost 3K of memory (see Program 1). On VICs that haven't been expanded, that leaves just room enough to store 64 characters. That limit coincides with the fact that the second half of the map memory starting at 7168 corresponds to screen memory in most machines. We'll discuss a way of getting around this 64-character limit later.

Of course, with expanded memory, all you have to do is select a memory location that doesn't interfere with screen memory. Sometimes the problem is solved automatically because the screen memory moves (as do the screen color addresses). The three variables in line 20 allow you to change the program to compensate. XX is map memory and should always be a multiple of 1024. SC is screen memory. CL is color memory.

When you run Pixelator, you are first offered a choice of creating a new character or retrieving an old one from memory. The choices are color-coded green and cyan, respectively. If you select "new character" by pressing the programmable key F1, the border changes from white to green, and you are asked to select one of the four work frames by keying F1, F3, F5, or F7. Next you are asked to select the character at the address where you intend to design a new shape.

—2—

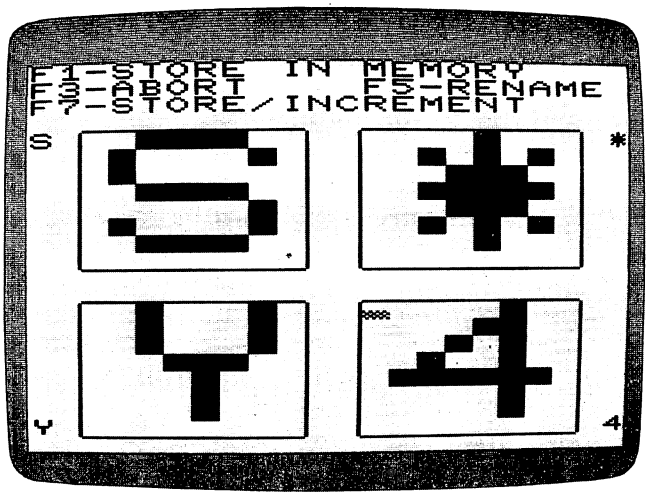
Four Options Following Design

Once you've selected a character, you'll see a half-height dot screen figure pop up in the top left corner of the frame. That's your cursor, and you can move it anywhere within the frame by using the cursor controls. To design a character, use the space bar. SHIFT/SPACE leaves a trail of red spaces in its wake. These red spaces correspond to the pixels which will be darkened in the completed character. Without shifting, the space bar returns the spaces to white. You can clear a cluttered frame simply by holding the space bar down until all the red is gone.

After you have worked on the character to your satisfaction, you have four options. F1 stores your creation in the appropriate eight bytes of memory and then returns you to the opening format. F3 aborts the frame, returning you to the opening format without storing the character. F5 renames the character, enabling you to reassign it to a memory location different from the one for which it originally was named. This is of more use when retrieving characters from memory than when creating new ones, but it works in both modes. F7 allows you to work on a series of characters without having to go through the "select frame — select character" process every time. The command stores the current character, jumps to the next frame, and increments the character name. You can keep doing this until you have stored the question mark, at which point you are returned to the opening format.

If at the opening format you opt to retrieve a character from memory, the border changes to cyan, and you are given five choices. F1 retrieves from RAM; that is, it accesses either characters you have already stored or whatever garbage happens to be in memory at the time. F2 accesses the VIC's ROM characters from the graphics mode, and F4 calls up the reverse of those characters. F6 and F8 are for text mode, the latter key again applying to reversed characters. You can freely mix characters from all modes and modify them to suit your needs. (If you need a full alphabet to go along with your custom characters, there is a short cut, provided you store your characters at 7168. After POKEing 255 into 36869, you can use RVS ON to get any normal character from @ to ?. RVS OFF gives you your custom characters. This works only at 255.)

From there you are asked to select frame and select character again, but if you call up a graphics character (or, in text mode, an uppercase character) from ROM, you will be asked to rename it to something with a screen value less than 64. You now have the



Four characters as seen by "Pixelator."

same options as before: to store, to abort, to rename, or to store and increment. If you have renamed a character, both the original character and its new name will be incremented.

Saving Your Custom Characters

More than likely, you will want to use Pixelator to create characters for use in some other program, such as a videogame. Three shorter programs allow you to save the information the Pixelator has created. To save the characters directly on cassette as a data file, interrupt the Pixelator with the STOP key and type NEW to get rid of the program. Then load "Pixaver" into the VIC. Pixaver (Program 2) allows you to save a block of characters of any size, up to 64, on tape as a single data file. The first number in the file represents the screen value of the first character; the second number is the last character. This allows you to record as many different blocks as you like. Each file will contain the information necessary to store the data in the right place. Also, for convenience, each file will be tagged with the name of its first character. Now you can turn your VIC off.

Loading Your Custom Characters

The "Pixeloader" program (Program 3) will read the data off the cassette and enter it back into memory. Notice line 10, which sets

the value of XX, the start of map memory. By changing that value, you can load character data into many different memory locations, thus bypassing the 64-character limit. Be sure that XX is a multiple of 1024, or else the characters won't properly correspond to the keyboard.

A third accessory program, called "Pixdata," will convert a block of RAM character memory into DATA statements, one for each character (see Program 4). The line numbers of the DATA statements will correspond to the screen value of the characters, plus 5000. DATA statements are highly inefficient, memory-wise, for storing that information, but they are much more convenient than cassette data files because they can be included within a program, which saves you the trouble of loading the characters separately.

Pixdata is not as user-oriented as the other programs because it has been stripped down to bare essentials. You probably will have to modify some lines of Pixdata each time you run it. The values SR and LS initialized in line 30, for example, represent the first and last characters, respectively. If you have only 3.5K of free RAM, don't do more than 30 characters at a time, because you'll run out of memory.

What makes Pixdata interesting is that it self-destructs, saving you the chore of deleting it line by line to make room for your own program. (When you type in Pixdata, be sure to save it on tape or disk before trying it.)

The secret of Pixdata lies in the way the VIC-20 stores BASIC lines. The first two bytes of a line represent the address of the *next* line. The third and fourth bytes are the line number. After that, the line consists of numbers that represent either tokens for BASIC commands (the token for DATA is 131) or the ASCII values of string characters. All numerals are treated as strings, so a DATA statement may need as many as three bytes to represent a single numerical value. The number 128, for example, becomes 49, 50, and 56. Throw in a 44 for each comma, and you see why a DATA statement can use up more than four times the memory needed to store the numbers it represents.

Pixdata starts creating DATA statements at location 5120, which is represented by the variable ZZ in line 40. Line 10 also sets location 5120 as the end of BASIC memory, thereby protecting the DATA statements from the program itself (0 + (256*20) = 5120). When Pixdata finishes creating DATA statements, it POKEs the low-high values of ZZ into the first and second bytes of line 1,

the line that says "REM DELETE THIS LINE AFTER RUNNING." This causes BASIC to skip from line 1 to the first DATA statement, ignoring the rest of Pixdata in between. When you delete line 1 (simply type a 1 on a blank line and hit RETURN), the line editor compacts the DATA statements to the beginning of memory, destroying Pixdata in the process. If by adding RAM you have changed the start of BASIC memory, be sure to adjust the two addresses in line 170 (4097 and 4098) accordingly before running Pixdata.

To use the DATA statements in a program, you will need a line like the following:

```
FOR L=SR TO LS:FOR M=Ø TO 7:READ C:POKE X
   X+L*8+M,C:NEXT M:NEXT L
```

The values of XX (map memory), SR (first character screen value), and LS (last character) should be the same as they were in Pixdata.

Memory expansion of 8K or more on the VIC usually moves the screen memory so that there is not enough room between the end of the screen and the beginning of the last available character map area in RAM for the Pixelator to operate. Before loading the Pixelator, 8K users should enter the following as a single line and then hit RETURN:

```
POKE 43,Ø:POKE 44,24:POKE 6143,Ø:NEW
```

Now LOAD the Pixelator, delete line 30, and make this change:

```
2Ø XX=512Ø:SC=4Ø96:CL=37888
3570 IF S2>1 THEN POKE 36869,PEEK(36869)
      AND NOT 15 OR 2:GOTO 16Ø
3580 POKE 36869,PEEK(36869) AND NOT 15:GO
      TO 16Ø
```

SAVE the program before using it. Make the following change in both Pixaver and Pixeloader:

```
1Ø XX=512Ø
```

Make these changes in Pixdata:

```
1Ø C=PEEK(56):POKE 51,Ø:POKE 52,32:POKE 5
    5,Ø:POKE 56,32
2Ø XX=512Ø
4Ø ZZ=8192:AA=ZZ
```

8K users can access the RAM character set by typing:

—2—

```
POKE 36869,PEEK(36869) AND NOT 15 OR 13
```

The following articles from *COMPUTE!'s First Book of VIC* provided valuable information and inspiration for Pixelator: Jim Butterfield's "Memory Map Above Page Zero" and "Browsing The VIC Chip"; Doug Ferguson's "Large Alphabet"; and Charles H. Gould's "Renumber BASIC Lines The Easy Way."

Program 1. Pixelator

```
10 REM "PIXELATOR"
20 XX=7168:SC=7680:CL=38400
30 POKE51,240:POKE52,XX/256-1:POKE55,240
   :POKE56,XX/256-1
40 FORLX=16TO1STEP-1:READXZ:POKEXX-LX,XZ
   :NEXTLX
50 POKEXX-10,SC/256:POKEXX-1,XX/256-1
60 PRINT"{CLR}{2 DOWN}";
70 FORY=1TO2:PRINT"{DOWN}{BLU}{2 RIGHT}
   {8 P}{2 RIGHT}{8 P}"
80 FORZ=1TO8:PRINT"{RIGHT}{M}{RED}
   {8 SPACES}{BLU}{G}{M}{RED}
   {8 SPACES}{BLU}{G}":NEXTZ
90 PRINT"{2 RIGHT}{8 Y}{2 RIGHT}
   {8 Y}{UP}":NEXTY
100 POKE36879,25:F=0:J=0:SYSXX-16:PRINT"
    {HOME}{GRN}{RVS}F1{OFF}{BLU}-CREATE
    NEW CHAR."
110 PRINT"{CYN}{RVS}F3{OFF}{BLU}-RETRIEV
    E MEMORY"
120 GETS1$:IFS1$=""THEN120
130 IFS1$="{F1}"THENK=0:POKE36879,29:GOT
    O160
140 IFS1$="{F3}"THENPOKE36879,27:GOTO350
    0
150 GOTO120
160 IFJ=1THEN190
170 SYSXX-16:PRINT"{HOME}SELECT"SPC(4)"F
    1 F3":PRINT"FRAME:"SPC(4)"F5 F7";
180 GETS$:IFS$=""THEN180
190 IFASC(S$)>132THENONASC(S$)-132GOTO21
    0,220,230,240
200 GOTO180
210 VV=3:HH=1:F=88:GOTO250
220 VV=3:HH=11:F=109:GOTO250
230 VV=13:HH=1:F=462:GOTO250
240 VV=13:HH=11:F=483
250 POKEF+SC,160:IFK>0THENPOKEF+CL,3:GOT
    O270
```

—2—

```
260 IFJ=0THENPOKEF+CL,5:GOTO280
270 IFJ>0THENC=CJ:C0=CG:GOTO320
280 SYSXX-16:PRINT"{HOME}SELECT CHARACTE
    R";
290 GETC$:IFC$=""THEN290
300 GOSUB5000
310 IFCE=2ANDS2$="{F1}"THEN290
320 IFK=1ANDI=0ANDCE<>1THEN4000
330 IFCE>0THEN290
340 POKEF+SC,C:POKEF+CL,0:V=1:H=1:P=SC+2
    3+VV*22+HH:PA=P:PQ=PEEK(P)+72:PP=PQ
350 I=0:J=0:SYSXX-16:PRINT"{HOME}F1-STOR
    E IN MEMORY"
360 PRINT"F3-ABORT"SPC(4)"F5-RENAME F7-S
    TORE/INCREMENT";
370 GETG$:POKEP,PQ:POKEPA,PP:IFG$=""THEN
    370
380 IFASC(G$)=32ORASC(G$)=160THENPOKEP,A
    SC(G$):H=H+1:GOTO440
390 IFG$="{DOWN}"THENV=V+1:GOTO440
400 IFG$="{UP}"THENV=V-1:GOTO440
410 IFG$="{RIGHT}"THENH=H+1:GOTO440
420 IFG$="{LEFT}"THENH=H-1:GOTO440
430 IFASC(G$)<133ORASC(G$)>136THEN370
440 IFH>8THENH=1:V=V+1
450 IFH<1THENH=8:V=V-1
460 IFV>8THENV=1
470 IFV<1THENV=8
480 PP=PEEK(P):PA=P:IFPP=104ORPP=232THEN
    PP=PP-72
490 IFG$="{F1}"THENK=0:POKEPA,PP:GOTO100
    0
500 IFG$="{F3}"THENK=0:POKEPA,PP:GOTO100
510 IFG$="{F5}"THENI=1:POKEPA,PP:POKEF+C
    L,PEEK(36879)-24:POKEF+SC,160:GOTO41
    20
520 IFG$="{F7}"THENJ=1:POKEPA,PP:GOTO100
    0
530 P=SC+(VV+V)*22+HH+H:PQ=PEEK(P)+72
540 GOTO370
1000 SYSXX-16:PRINT"{HOME}STORING ";:POK
     ESC+8,C
1010 FORVE=1TO8:ZZ=0
1020 FORHY=1TO8:PO=SC+(VV+VE)*22+HH+HY
1030 IFPEEK(PO)=160THENZZ=ZZ+2↑(8-HY)
1040 NEXTHY
1050 POKEXX+C*8+VE-1,ZZ:NEXTVE:IFJ=0THEN
     100
1060 GOTO2000
```

—2—

```
2000 CJ=C+1:CG=C0+1:S$=CHR$(ASC(S$)+1):I
     FASC(S$)>136THENS$="{F1}"
2010 IFCJ=64ANDXX=7168ANDSC=7680THENCE=2
2020 IFK=2THENK=1
2030 IFCG>127THENCG=0
2040 IFS2$="{F1}"ANDCE=2THENJ=0:GOTO100
2050 IFK=0ANDCE=2THEN100
2060 GOTO190
3500 K=1:IFJ=1THEN3540
3510 SYSXX-16:PRINT"{HOME}F1-RETRIEVE FR
     OM RAM"
3520 PRINT"F2-ROM GFX {RVS}F4-REVERSE
     {OFF}F6-ROM TEXT {RVS}F8-REVERSE
     {OFF}";
3530 GETS2$:IFS2$=""THEN3530
3540 IFS2$="{F1}"THENXR=XX:GOTO3580
3550 S2=ASC(S2$)-137:IFS2>-1ANDS2<4THENX
     R=32768+1024*S2:GOTO3570
3560 GOTO3530
3570 IFS2>1THENPOKE36869,PEEK(36869)ANDN
     OT15OR2:GOTO160
3580 POKE36869,PEEK(36869)ANDNOT15:GOTO1
     60
4000 IFJ=0THENC0=C
4010 SYSXX-16:PRINT"{HOME}":PRINT"LOOKIN
     G AT{2 SPACES}";S5$:POKESC+33,C0
4020 FORD=1TO8:DA=PEEK(XR+C0*8+D-1):DI=0
4030 FORDD=1TO8:DI=INT(DA/2↑(8-DD)):DA=D
     A-DI*2↑(8-DD)
4040 IFDI>0THENDO=160:GOTO4060
4050 DO=32
4060 IFDD=8ANDD<8THENZD=15:GOTO4090
4070 IFD=8ANDDD=8THENPOKEZF+1,DO:GOTO410
     0
4080 ZD=1
4090 ZF=SC+(VV+D)*22+HH+DD:POKEZF,DO:POK
     EZF+ZD,PEEK(ZF+ZD)+72:NEXTDD:NEXTD
4100 IFCE>0THENK=2:GOTO4120
4110 GOTO340
4120 SYSXX-16:PRINT"{HOME}RENAME":GOTO29
     0
5000 C=ASC(C$):CE=0
5010 ONINT(C/32)GOTO5060,5040,5050,5020,
     5040,5030
5020 CE=1:RETURN
5030 C=C-64
5040 C=C-32
5050 C=C-32
5060 IFJ=1THENC0=CG
```

```
5070 IFXX=7168ANDC>63ANDSC=7680THENCE=2:
     RETURN
5080 RETURN
6000 DATA162,0,169,32,157,0,30,232,224,6
     8,208,1,96,76,244,27
```

Program 2. Pixaver

```
10 XX=(PEEK(56)+1)*256
3000 SYSXX-16:PRINT"{CLR}FIRST CHARACTER
     ?";
3010 GETSR$:IFSR$=""THEN3010
3020 C$=SR$:GOSUB5000:SR=C:IFCE>0THEN301
     0
3030 PRINT"{HOME}"SPC(15)" "SR$;SPC(5)"L
     AST CHARACTER? ";
3040 GETLS$:IFLS$=""THEN3040
3050 C$=LS$:GOSUB5000:LS=C:IFCE=1THEN304
     0
3060 IFSR>LSTHENSS=SR:SR=LS:LS=SS:SS$=SR
     $:SR$=LS$:LS$=SS$
3070 SYSXX-16:PRINT"{HOME}SAVING "SR$" T
     O "LS$;
3080 PRINT"{HOME}";:OPEN1,1,1,SR$
3090 SYSXX-16:PRINT"{HOME}SAVING "SR$" T
     O "LS$
3100 PRINT#1,SR
3110 PRINT#1,LS
3120 FORCZ=SRTOLS
3130 FORLL=0TO7
3140 PRINT#1,PEEK(XX+CZ*8+LL)
3150 NEXTLL
3160 NEXTCZ
3170 CLOSE1
3180 END
5000 C=ASC(C$):CE=0
5010 ONINT(C/32)GOTO5060,5030,5040,5020,
     5030,5050
5020 CE=1:RETURN
5030 C=C-64:GOTO5060
5040 C=C-32:GOTO5060
5050 C=C-128:GOTO5060
5060 IFXX=7168ANDPEEK(648)*256=7680ANDC>
     63THENCE=2:RETURN
5070 RETURN
```

Program 3. Pixeloader

```
10 XX=7168
20 OPEN1,1,0
30 INPUT#1,SR
40 INPUT#1,LS
50 FORS=SRTOLS
60 FORR=0TO7
70 INPUT#1,C:POKEXX+S*8+R,C:NEXTR:NEXTS
80 CLOSE1
```

Program 4. Pixdata

```
1 REM DELETE THIS LINE AFTER RUNNING
10 C=PEEK(56):POKE51,0:POKE52,20:POKE55,
   0:POKE56,20:REM MUST MATCH ZZ
20 XX=7168
30 SR=0:LS=26:REM FIRST AND LAST CHARACT
   ERS
40 ZZ=5120:AA=ZZ
50 POKEZZ-1,0
60 FORL=SRTOLS
70 L2=INT((L*10+5000)/256):L1=(L*10+5000
   )-L2*256:POKEZZ+2,L1:POKEZZ+3,L2
80 POKEZZ+4,131:X=4
90 FORLL=0TO7
100 S$=STR$(PEEK(XX+L*8+LL)):S=LEN(S$)
110 FORLZ=2TOS:X=X+1:POKEZZ+X,ASC(MID$(S
    $,LZ,1)):NEXTLZ
120 IFLL=7THEN140
130 X=X+1:POKEZZ+X,44:NEXTLL
140 X=X+1:POKEZZ+X,0
150 X=X+1:Z2=INT((ZZ+X)/256):Z1=ZZ+X-Z2*
    256:POKEZZ,Z1:POKEZZ+1,Z2:ZZ=ZZ+X:NE
    XTL
160 POKEZZ,0:POKEZZ+1,0
170 A2=INT(AA/256):A1=AA-A2*256:B=PEEK(4
    3)+256*PEEK(44):POKEB,A1:POKEB+1,A2:
    POKE56,C
180 ZZ=ZZ+257-AA+B:Z2=INT(ZZ/256):Z1=ZZ-
    Z2*256
190 POKE251,Z1:POKE174,0:POKE175,0:POKE4
    6,Z2:POKE45,PEEK(251)
```

—2—

Custom Characters for Game Graphics

Bud Banis

Using the "multicolor mode" can add a great deal to your programs. Presented here is an explanation of how to use it, plus a demonstration game, "UFO Pilot."

Your VIC-20 has outstanding color graphics capabilities. However, the unexpanded machine has limited memory to take advantage of these capabilities, and the average computerist who is trying to justify "buying more than a videogame" has to provide his family with a reasonable amount of entertainment without buying a lot of expensive memory expansion.

Two options have been offered for designing game graphics characters:

1. The Commodore graphics keys can be used to build multiple space characters. These take up a lot of space and are cumbersome to move around.

2. Custom characters can be drawn if you're willing to give up valuable RAM instead of taking existing characters from ROM. Basically, whole sets of characters are moved from ROM to RAM, and then some of the characters can be redefined by a series of POKEs to RAM. Because the pointer indicating the start of character memory has to be reset (36869), this is an all-or-nothing process. Any standard characters you want to use must also be relocated from ROM to RAM.

As an alternative, some perfectly acceptable single space characters can be created from standard characters in ROM just by POKEing their screen locations into multicolor mode. This approach uses no memory and gives a wide variety of "new" characters (about four million) to choose from.

This article describes the use of multicolor mode in detail, includes a program to find interesting characters, and concludes with a game demonstrating the technique.

—2—

How Characters Are Stored

In order to explain multicolor mode, it's important to first describe how characters are formed on the screen in the first place. The *VIC-20 Programmer's Reference Guide* (pp. 82-94) has several errors in its description of this process.

Characters are stored in memory as an 8x8 grid of dots. Each dot (bit) is turned either "on" or "off." Each eight-bit line (byte) can be represented by a number which uniquely turns some bits "on" and others "off." Each bit is represented by a number which is a power of two if "on" or by zero if "off." The value assigned to the byte is the sum of the values of its eight bits.

bit number	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
value of 2^N	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**

Thus, if only bit zero is "on," the value of the byte is $2^0 = 1$. If only bit four is "on," the value of the byte is $2^4 = 16$. If bits zero and four are both "on" and all the others are "off," then the value of the byte is 2^0 and $2^4 = 1 + 16 = 17$. If all eight bits are "on," then the value of the byte is $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$. A whole character takes eight lines or eight bytes of memory. For example, the letter A is:

bit no.	**7 6 5 4 3 2 1 0**	**value of byte**
byte 1	**0 0 0 1 1 0 0 0**	**$2^4 + 2^3 = 24$**
2	**0 0 1 0 0 1 0 0**	**$2^5 + 2^2 = 36$**
3	**0 1 0 0 0 0 1 0**	**$2^6 + 2^1 = 66$**
4	**0 1 1 1 1 1 1 0**	**$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 126$**
5	**0 1 0 0 0 0 1 0**	**$2^6 + 2^1 = 66$**
6	**0 1 0 0 0 0 1 0**	**$2^6 + 2^1 = 66$**
7	**0 1 0 0 0 0 1 0**	**$2^6 + 2^1 = 66$**
8	**0 0 0 0 0 0 0 0**	**0**

Custom characters can be stored in RAM locations by POKEing the desired values into the individual memory locations (bytes).

The unexpanded VIC-20 has room for 5120 bytes in RAM or about 3.6 thousand (K) bytes user-available RAM after buffers and screen memories, etc., are allocated. Since each character takes up eight bytes, moving 64 characters from ROM to RAM, available for use or modification in the custom character mode, uses $64 * 8 = 512$ bytes of RAM and makes it unavailable for other uses.

Multicolor Mode

In multicolor mode, characters are stored in the same way, but bits are read two at a time to specify one of four colors in a two-dot

space. Taking two bits at a time allows four possibilities, as opposed to the two ("on" or "off") when bits are taken one at a time.

bit pair	colors selected	memory location (POKE)
00	16 background colors	36879, bits 4-7
10	8 character colors	38400-38911, bits 0-2
01	8 border colors	36879, bits 0-2
11	16 auxiliary colors	36878, bits 4-7

Thus, if you were custom designing a flag with alternating background color and border color stripes, a character color square in the upper left-hand corner, and an auxiliary color pole, the stored data might look something like this:

	bit pairs	value of byte (POKE)
byte 1	(10) (10) (01) (01)	128 +32 +4 +1 =165
2	(10) (10) (00) (00)	128 +32 =160
3	(10) (10) (01) (01)	128 +32 +4 +1 =165
4	(00) (00) (00) (00)	0
5	(01) (01) (01) (01)	64 +16 +4 +2 =86
6	(11) (00) (00) (00)	128 +64 =192
7	(11) (00) (00) (00)	128 +64 =192
8	(11) (00) (00) (00)	128 +64 =192

This character wouldn't be very interpretable in ordinary, single color, mode.

Once a character is stored in memory in this way, in order to print it on screen in its full multicolor glory, we need to first specify multicolor mode in that screen location, then choose the appropriate colors for border, background, character, and auxiliary use. By POKEing these other reference locations, we can make substantial changes in the character. For example, if the auxiliary color is the same as the background color, the flagpole disappears.

Selecting Colors

Specifying colors is a little more complicated than just POKEing a number into a memory location. The reason is that the color codes use only specific bits, and the rest of the bits in the byte are used for something else. For example, the auxiliary color code uses only bits 4-7 in memory location 36878. The other four bits (0-3) are used for setting volume on the sound. Selection of multicolor mode for a given screen location involves turning on a single bit in the memory for that space on the screen. The other bits hold other information.

Choosing Border and Background Colors

By now, you should be pretty well versed in this operation, and you have probably tried some of the combinations listed in Appendix E of *Personal Computing on the VIC-20.* It seems simple enough — POKEing a number out of the table into memory location 36879 gives you the indicated combination of screen and border colors. Actually, byte 36879 specifies three things which could be referenced independently.

Border colors are specified by bits 0-2. The decimal translation is values 0-7, to give eight possible choices (0 is all "off," 7 is all "on"): 0 is black, 1 is white, 2 is red, etc., in the same sequence as the color keys. Bits 4-7 specify background, or screen, colors. The values associated with these bits are multiples of 16. For example, if bit four is turned "on," its decimal value is $2^4 = 16$; if all four bits 4-7 are turned "on," the combined decimal value of these bits is $2^4 + 2^5 + 2^6 + 2^7 = 16 + 32 + 64 + 128 = 240$.

A little fooling with the numbers should convince you that these four bits can give you any multiple of 16 from 0* 16 to 15* 16, or 16 possibilities. This corresponds to the 16 choices of screen color in the order listed in Appendix E of the book *Personal Computing on the VIC-20.* Casual inspection of this table reveals that some possible values are not listed — for example, 0-7, 16-23, etc. The lowest value listed is 8. What this means is that bit number three, decimal value $2^3 = 8$, is always "on" when one of the values in the table is used. If you POKE 36879, X, where X is a value not in the table, bit three is turned "off," and the screen is put in the *inverted* mode, which makes all the printing appear in the reverse.

Thus, byte 36879 contains three separate memory references: bits 0-2 for border color (eight colors); bit 3 for inverted mode (when "off"); and bits 4-7 for screen color (16 colors from 0* 16 to 15* 16).

Setting Character Color and Selecting Multicolor Mode

Character color is specified separately for each location on the screen (see pages 143-44 in *Personal Computing on the VIC-20*) or can be specified before printing a series of characters by using the control color keys. Character color is specified separately for each screen location by POKEing locations between 38400 and 38905 with values from 0-7 to give the familiar sequence of black to yellow character colors (eight choices). Values from 0-7 represent bits 0-2.

If bit three is turned "on" — that is, values from 8 to 15 are used instead of 0-7 — the screen location is put into multicolor mode and the bits are evaluated two at a time to give the results described above under "Multicolor Mode." In multicolor mode, the character color code is (value-8). For example, POKE 38400, 8 puts the first space into multicolor mode with character color black (0). POKE 38422, 15 puts the twenty-second space (first space, second row) into multicolor mode with character color yellow (7).

Bits 4-7 are used for something else which is not clear from the manuals. Randomly POKEing these bits eventually gives peculiar results such as "out of memory" errors. This can be avoided by ANDing POKEs with 15.

Boolean Operators

There is a way to read and write to specific bits within a byte without disturbing other bits which might carry other information. Unless you've been exposed to set theory before, the action of Boolean operators OR and AND may seem strange. These operators are used to combine information from two sets.

When AND is used, the result includes only that information which is included in *both* sets. For example, if all eight bits in a byte were turned "on," the decimal value of that byte would be 255. If another byte had only the first four bits turned "on," its decimal value would be 15. The result from ANDing bytes one and two would have only "on" bits that were "on" in *both* sets. This gives the peculiar result that 255 AND 15 =15.

If you wanted to know the status of only a single bit, you could screen out extraneous information by ANDing with the decimal value for that bit: PRINT PEEK(38400) AND 8 would return 8 if the third bit is "on" or 0 if the third bit is "off." The status of other bits doesn't matter.

The OR operator combines sets so that the result includes all bits "on" which were "on" in *either* set. Thus, 255 OR 15 =255; 248 OR 15 =255. These operators can be used to POKE a given bit "on" or "off" without disturbing other information in the byte. For example, suppose we wanted to POKE bit three (decimal value 8) in 38400 "on." We could do this by POKE 38400, 8 OR PEEK (38400). To turn bit three "off," POKE 38400, 247 AND PEEK (38400). 247 is the decimal value for a byte with all bits "on" except for bit three.

Setting Auxiliary Color

The fourth color available in multicolor mode is called auxiliary color and is set by POKEing values into the upper four bits of memory location 36878. The lower four bits are used to set volume on the sound. There are 16 colors available, in the same order as the 16 screen colors. As with the screen colors, values POKEd into the upper four bits are multiples of 16.

For example, POKE 36878, 1*16 sets auxiliary color white; POKE 36878, 15*16 sets auxiliary color light yellow. These POKEs would also set sound volume to 0. If you wanted to set auxiliary color red at the same time as keeping volume at the maximum, 15, you could POKE 36878, 15 +2*16, or, to leave the sound volume alone, use the Boolean operators: POKE 36878, 2*16 OR (PEEK (36878) AND 15).

Sampler — A Program to Find Interesting Characters

Given the above detail on multicolor mode, the first program should be self-explanatory (see Program 1). Ten characters are displayed, with the middle eight in multicolor mode to show the range of character colors. The cursor key can be used to look at the next or previous characters. Cursor down and cursor up act as "fast forward" and "fast reverse," respectively. Cursor right and cursor left can also be used to give a time delay (lines 70 and 90) in the display before changing characters.

After finding an interesting character, press F1 to explore the effects of the 128 different combinations of screen and border colors. The space bar allows a rapid perusal. F3 gives another dimension, again using the space bar (or "any key") to run through the 16 available auxiliary colors. To look at character set 2 (*Personal Computing on the VIC-20,* Appendix H, pp. 139-42), press the SHIFT and COMMODORE keys simultaneously.

Including reverse mode and both character sets, there are about 255 characters which can be modified through use of multicolor mode. With 8 border colors, 16 screen colors, 8 character colors and 16 auxiliary colors, the number of combinations for your selection is roughly 255 * 8 * 16 * 8 * 16 or about four *million* "new" characters to choose from!

UFO Pilot — A Game Demonstrating Multicolor Mode Graphics

Having progressed through the theory to empirical selection, it seems logical to come to the point of this article. "UFO Pilot" is a

game demonstrating the use of multicolor mode to make "new" game graphics characters. The program uses about 2K RAM and the only expansion required is a joystick.

Character 88 (the club) is transformed to a multicolor UFO which you pilot using the joystick. The objective is to achieve the longest flight without running into your own trail of white dots or the warplane (character 62) that's in constant pursuit. A collision results in an explosion (character 42 taken through a series of character color changes in lines 9500-9510) and a return to the demonstration mode at the beginning of the program.

If you don't have anything better to do, you can watch this display run through all the possible color combinations. The pause in midscreen in which the UFO "flashes its lights" is a demonstration of changing auxiliary colors (line 10). Otherwise, auxiliary color 0 (black) is used throughout the game — specified by POKE 36878, 15 (high volume). If the demonstration mode begins to wear on you and you want to concentrate on the game, change line 9530 to GOTO19.

Fortunately, the warplane erases dots to keep the screen less cluttered and to make higher scores possible. The high score so far is 3411.

Program 1. Sample Characters in Multicolor Mode

```
2 PRINT"{CLR}SAMPLER,SHOWS SOME{5 SPACES}
  STANDARD CHARACTERS{4 SPACES}IN MULTIC
  OLOR MODE.
4 PRINT:PRINT"USE THE CURSOR KEYS
  {3 SPACES}TO CHANGE CHARACTERS, F1,F3
  TO CHANGE COLORS
6 PRINT:PRINT"HIT A KEY"
8 GETC$:IFC$=""THEN8
10 N=0:GOTO130
20 GETC$:IFC$=CHR$(17)THEN80
30 IFC$=CHR$(29)THEN70
40 IFC$=CHR$(145)THEN100
50 IFC$=CHR$(157)THEN90
55 IFC$=CHR$(133)THEN400
57 IFC$=CHR$(134)THENGOSUB600
60 GOTO20
70 FORTT=1TO300:NEXT
80 N=N+1:IFN=256THEN10
85 GOTO130
90 FORTT=1TO300:NEXT
100 N=N-1:IFN=-1THEN10
```

```
110 GOTO130
130 PRINT"{CLR}":PRINT
140 FORI=2TO20STEP2
150 POKE7680+22+I,N
160 POKE38400+22+I,((I/2+6)AND15)
170 NEXT
180 PRINT:PRINT"CHARACTER NO.";N
190 PRINT:GOTO20
400 PS=8+16*INT(CC/8)+CS
410 POKE36879,PS:PRINT"{HOME}{5 DOWN}SCRE
    EN COLOR= {LEFT}";PS:PRINT"AUX COLOR
    =0{3 SPACES}"
420 GETC$:IFC$=""THEN420
430 IFC$=CHR$(134)THENGOSUB600
450 CC=CC+1:CS=CCAND7:IFPS=255THENPOKE368
    79,27:CC=0:CS=0:GOTO20
460 GOTO400
600 FORAN=0TO15
610 POKE36878,16*AN
650 PRINT"{HOME}{5 DOWN}SCREEN COLOR=
    {LEFT}";PS:PRINT"AUX COLOR=
    {2 SPACES}{2 LEFT}";AN
660 GETC$:IFC$=""THEN660
670 NEXT:POKE36878,0
680 PRINT"{HOME}{6 DOWN}AUX COLOR=
    {3 SPACES}{2 LEFT}0"
690 GETC$:IFC$=""THEN690
700 RETURN
```

Program 2. UFO Pilot

```
1 SS=24:POKE36879,63:POKE36878,15:DIMJS(
  2,2)
2 PRINT"{CLR}":PRINTSPC(5):PRINT"*******
  ******":PRINTSPC(5)
3 PRINT"**{RVS}UFO PILOT{OFF}**":PRINTSP
  C(5)"**********":PRINT"{5 SPACES}*
  {2 SPACES}7-28-82{2 SPACES}*"
4 PRINTSPC(5):PRINT"**********"
5 PRINT"{3 DOWN}{2 SPACES}SET DIRECTION
  OF ":PRINT"{4 SPACES}SHIP WITH THE"
6 POKE37139,0:DD=37154:PA=37137:PB=37152
  :PRINT"{5 SPACES}JOYSTICK"
7 PRINT"{DOWN}{2 SPACES}DON'T RUN INTO Y
  OUR":PRINT"{4 SPACES}OWN TRAIL OR HIT"
8 PRINT"{5 SPACES}THE WARPLANE.":PRINT"
  {2 DOWN}{4 SPACES}HIT FIRE TO START
9 FORAA=0TO21:POKE7812+AA,88:POKE38532+A
  A,9:GOSUB9000:IFFRTHEN19
```

```
10 IFAA=10THENFORTY=0TO15:POKE36878,15OR
   16*TY:POKE36874,244:FORM=1TO50:NEXT:N
   EXT
11 POKE36878,15
12 POKE36874,234+AA:POKE36874,0:POKE7812
   +AA,32:NEXT:CS=SSAND7
13 FORAA=0TO21:POKE7701-AA,60:POKE38421-
   AA,9:POKE7878+AA,62
14 POKE38598+AA,9:GOSUB9000:IFFRTHEN19
15 POKE36874,215:FORTT=1TO40:NEXT:POKE36
   874,0:POKE36875,255-5*AA
16 FORTT=1TO10:NEXT:POKE36875,0:POKE7878
   +AA,32:POKE7701-AA,32:NEXT
17 PS=8+16*INT(SS/8)+CS:POKE36879,PS:SS=
   SS+1:IFPS=255THENSS=0
18 GOTO9
19 FORI=0TO2:FORJ=0TO2:READJS(J,I):NEXTJ
   ,I
20 FF=505:PRINT"{CLR}{RVS}{44 SPACES}"
22 XX=0:AD=0:GOSUB10000:IFSC>PHTHENPH=SC
24 POKE7680+FF,88:POKE38400+FF,9:GOSUB90
   00:IFJS(X+1,Y+1)=0THEN24
29 SC=0:YY=22:GOSUB10000
30 GOSUB9000:GOSUB8000:QQ=FF:XZ=ZX:ZX=XX
   +22*YY
31 PRINT"{HOME}{RVS}{15 SPACES}":PRINT"
   {HOME}{RVS}{2 SPACES}SCORE=";SC;"
   {2 SPACES}"
32 IFJS(X+1,Y+1)THENAD=JS(X+1,Y+1):POKE3
   6876,220
33 POKE36876,0
35 POKE7680+FF,46:POKE38400+FF,1
40 FF=FF+AD:IFFF<44THENFF=QQ:GOTO9500
42 IFPEEK(7680+FF)=62THEN9500
45 IFPEEK(7680+FF)=46THEN9500
46 POKE7680+FF,88:POKE38400+FF,9
47 IFFF=XZTHEN9500
50 IFFF>505THENFF=QQ:GOTO9500
55 BL=(255-INT(ABS(XX+22*YY-FF)/2)OR128)
56 POKE7680+XZ,32:IFPEEK(7680+ZX)=88THEN
   9500
58 POKE7680+ZX,62:POKE38400+ZX,9
59 POKE36874,BL:POKE36874,0
70 GOTO30
100 DATA-23,-22,-21,-1,0,1,21,22,23
8000 SC=SC+1:XX=XX+1:IFXX=22THENXX=0:YY=
     INT(FF/22)
8020 RETURN
9000 POKEDD,127:S3=-((PEEK(PB)AND128)=0)
     :POKEDD,255
```

```
9010 P=PEEK(PA):S1=-((PAND8)=0):S2=((PAN
     D16)=0):SO=((PAND4)=0)
9020 FR=-((PAND32)=0):X=S2+S3:Y=SO+S1:RE
     TURN
9500 POKE36879,138:POKE36877,220:POKE768
     0+FF,42:FORZZ=1TO100
9510 POKE38400+FF,ZZAND15:POKE36878,INT(
     15-ZZ/7):NEXT:POKE36877,0
9520 XX=0:RESTORE:POKE36879,57:POKE36878
     ,15
9530 GOTO2
10000 PRINT"{HOME}{RVS}{15 SPACES}":PRIN
      T"{HOME}{RVS}{2 SPACES}SCORE=";SC;
      "{2 SPACES}"
10010 PRINT"{HOME}{DOWN}{RVS}PREVIOUS HI
      GH=";PH:RETURN
```

—Chapter 3—

Sound

—3—

Harmony

Henry Forson

The "Harmonizer" was designed to be friendly and easy to modify. If you like music, you'll like this program.

Believe it or not, it's the DATA statements that make this program so friendly. In fact, they were given *prime* consideration in the design. The DATA statements tell the "Harmonizer" how to play your song. These statements change with each song, although the rest of the program stays pretty much the same. The DATA statements contain three kinds of information: voice commands, notes, and separators. These are described separately below.

Voice Commands: The VIC has three voices — Soprano, Alto, and Tenor. The voice command tells which voice we want to play the following notes. A voice command consists of the letter V followed by an S, A, or T for Soprano, Alto, or Tenor, respectively. The VS on line 10 is a voice command meaning soprano.

Notes: In a DATA statement, a note consists of an A, B, C, D, E, F, G, or R, followed by a number from 1 to 9. The letters A to G are the standard music names for notes. The R (for rest) means silence. The number following the letter tells how many counts the note or rest lasts. A count is *not* always the same as a musical beat; the shortest note in a song has a value of one count. This eliminates the need for a notation involving fractions.

Separators: Separators are just commas and spaces. You can put them in the data strings wherever you want. You might find them useful to keep track of musical groupings, to make your data more readable.

Other Data Features: An X indicates the end of your data, to save you the trouble of counting notes. It makes no difference what order you put the voices in, and you can change voices whenever you want. So you could build up a complete tune a short phrase at a time using one or all voices, and check it as you go by *listening*, instead of *listing*.

Operation

When the Harmonizer is started, it seems to pause at first because it is reading the input data, sorting the notes by voices, and deter-

mining the internal note codes. Suddenly, it prints out how many notes were found for each voice and plays the music. When it finishes, it prints out how much memory was free and silences all the voices.

The first time you try it, remember to turn up the TV volume. For a quick test, you may want to leave out the DATA statements 12 to 20, 24 to 32, and 36 to 44. Also, you can leave out some of the REM statements to save space.

The key to understanding how it works inside is to study the two-dimensional array, N%. The N stands for *note* and the % means *integer.* The N% array is like a table containing three rows and 81 columns of integers. The rows are numbered 0, 1, and 2, one row for each voice. Each row has 81 columns, numbered 0 to 80. Columns 1 to 80 store each voice's notes in an internal form in sequential order. See line 480. Both the pitch and duration are packed into a single integer. So, you have a maximum of 80 notes per voice. If you get more memory, you can have a larger array just by changing the 80 in line 130. Column 0 keeps track of how many notes each voice actually uses in a particular piece.

When playing begins (around line 500), two other one-dimensional arrays are also used to keep track of where the Harmonizer is. The SP% (for *stack pointer*) array keeps track of the column of the current note for a given voice. Likewise, the TM% (for *timer*) array keeps track of how long, in counts, the current note for a given voice has been playing.

All input comes from the subroutine at line 800, which gets a single character from the DATA statements and returns it in the variable C$. This routine lets you use arbitrary length data strings and also takes care of the separators.

Enhancements

Once you have the standard program working, you will probably want to make changes. One of the first might be to add sharps and flats. These may be added using lines 350 to 410 as a guide. I've used the graphics on the front of the keys for this purpose; the one on the right means sharp, and the one on the left means flat. I've left this feature out of the article listing mainly so I could type it. Look at your VIC keyboard and imagine trying to figure out the difference between my hand-drawn C-sharp and D-sharp!

Other minor changes I might suggest would be to vary the tempo (line 680) or make the tune repeat (change line 740 to

GOTO 510). A finishing touch would be to paint a picture on the screen to match the tune.

The Harmonizer

```
3 REM THE TUNE IS
4 REM "SILENT NIGHT"
5 REM
10 DATA "VS G3A1G2E6,{2 SPACES}G3A1G2E6"
12 DATA "VS D3R1D2B6,{2 SPACES}C3R1C2G6"
14 DATA "VS A3R1A2C3B1A2, G3A1G2E4R2"
16 DATA "VS A3R1A2C3B1A2, G3A1G2E5R1"
18 DATA "VS D3R1D2F3D1B2, C6E4R2"
20 DATA "VS C3G1E2G3F1D2, C6C4R2"
22 DATA "VT C3R1C1R1C5R1, C3R1C1R1C6"
24 DATA "VT G3R1G1R1G6, C3R1C1R1C6"
26 DATA "VT F3R1F1R1F6, C3R1C1R1C5R1"
28 DATA "VT F3R1F1R1F6, C3R1C1R1C6"
30 DATA "VT G3R1G1R1G6, C6C5R1"
32 DATA "VT G3R1G1R1G6, C6C4R2"
34 DATA "VA E3F1E2C6, E3F1E2C6"
36 DATA "VA F6D6, E6E6"
38 DATA "VA F4C2A3G1F2, E3F1E2C6"
40 DATA "VA F6A3G1F2, E3F1E2C6"
42 DATA "VA F6D3F1D2, E6G5R1"
44 DATA "VA E4C2E3D1B2, E6E4R2"
46 DATA "X":{2 SPACES}REM END OF DATA SE
   CTION
100 REM START OF PROGRAM
110 NS%=2: REM NUMBER OF VOICES - 1
120 VT = 36874: REM TENOR VOICE LOCATION
130 DIM N%( NS%, 80 ): REM NOTE ARRAY
140 DIM TM%( NS% ): REM TIMER ARRAY
150 DIM SP%( NS% ): REM STACK POINTERS
160 FOR I = 0 TO NS%
170 N%( I, 0 ) = 0
180 NEXT I
190 SH% = 16: REM SHIFT CONSTANT
200 CV = 2: REM CURRENT VOICE
210 IN$ = "": REM INPUT STRING
220 C$ = "": REM INPUT CHARACTER
230 GOSUB 800
240 IF C$ = "X" THEN GOTO 510
250 IF C$ <> "V" THEN GOTO 340
260 GOSUB 800
270 REM SET THE CURRENT VOICE
280 IF C$ = "S" THEN CV = 2
290 IF C$ = "A" THEN CV = 1
300 IF C$ = "T" THEN CV = 0
```

```
310 GOTO 230
320 REM TRANSLATE NOTE TO CODE
330 REM FOR THE FREQUENCY
340 FR = -1
350 IF C$ = "C" THEN FR = 225
360 IF C$ = "D" THEN FR = 228
370 IF C$ = "E" THEN FR = 231
380 IF C$ = "F" THEN FR = 232
390 IF C$ = "G" THEN FR = 235
400 IF C$ = "A" THEN FR = 237
410 IF C$ = "B" THEN FR = 239
420 IF C$ = "R" THEN FR = 0
430 IF FR = -1 THEN PRINT "?";C$; "IN";
    IN$
440 GOSUB 800: REM GET THE COUNT IN C$
450 I% = N%( CV, 0 ) + 1
460 N%( CV, 0 ) = I%
470 REM STORE THE COUNT AND NOTE
480 N%( CV, I% ) = VAL( C$ ) + SH% * FR
490 GOTO 230
500 REM START PLAYING TUNE
510 FOR I = 0 TO NS%
520 SP%( I ) = 1
530 TM%( I ) = 0
540 PRINT "VOICE"; I; "HAS"; N%( I, 0 );
     "NOTES"
550 NEXT I
560 REM SET INITIAL VOLUMES
570 FOR I = 0 TO 4: POKE VT + I, 8: NEXT
     I
580 FOR D = 0 TO 1: REM UNTIL DONE
590 FOR I = 0 TO NS%
600 J = SP%( I )
610 IF J > N%( I, 0 ) THEN GOTO 670
620 D = 0
630 NT% = ( N%( I, J ) / SH% - INT( N%(
    I, J ) / SH% ) ) * SH%
640 IF TM%( I ) >= NT% THEN GOTO 750
650 TM%( I ) = TM%( I ) + 1
660 POKE VT + I, INT( N%( I, J ) / SH% )
670 NEXT I
680 FOR J = 0 TO 70: NEXT J: REM TEMPO C
    ONTROL
690 NEXT D
700 FOR I = 0 TO 4
710 POKE VT + I, 0: REM ALL QUIET
720 NEXT I
730 PRINT FRE( X ); "BYTES LEFT"
740 END: REM GOTO 510 FOR REPEAT
```

```
750 SP%( I ) = J + 1: REM NEXT NOTE
760 TM%( I ) = 0
770 GOTO 600
780 REM INPUT A CHARACTER IN C$
790 REM "X" STOPS INPUT
800 IF C$ = "X" THEN RETURN
810 IF LEN( IN$ ) = 0 THEN READ IN$
820 C$ = LEFT$( IN$, 1)
830 IN$ = RIGHT$( IN$, LEN( IN$ ) - 1 )
840 REM IGNORE SPACES AND COMMAS
850 IF C$ = " " OR C$ = "," THEN GOTO 80
    0
860 RETURN
```

—3—

Sound Generator

Robert Lee

With the "Sound Generator," you can add sound in BASIC without slowing down your program.

Among the novel features of the VIC-20 are its sound capabilities. These give it an advantage over the PET, bringing a new dimension to game programs. However, one of the problems I and undoubtedly other VIC owners have encountered is that, while manipulating the sound generators in a BASIC program, it is not possible to do anything else.

This is especially a problem in game programs written in BASIC and using extensive graphics. Either you have to write such programs without complex sound effects, or you have to settle for slow motion.

Faster Sound

Faced with this problem, I decided to write a machine language (ML) program for the VIC which adds speed to its sound generation capabilities. Most of the sound effects we use in game programs are sounds with increasing or decreasing tones. For example, a simple way to simulate the sound of a laser with the VIC is:

```
FOR K=250 TO 240 STEP-1:POKE36876,K:NEXT
```

The ML program works along these lines, except that it is necessary to use only one POKE command. It generates sounds with increasing or decreasing frequency to make almost any kind of sound effect possible.

The program "VIC Sound" places a machine language program in the cassette buffer of the VIC. This means, of course, that you cannot transfer data using the cassette player while you are running the program. By changing the contents of memory locations 788-789 (decimal), the interrupt system of the computer is used to run the ML program.

The VIC has four "speakers" to make music and noise. The first and second speakers, activated by POKEing memory locations 36874 and 36875, are used for sounds with increasing tones. The third speaker (36876) is used for sounds with decreasing

tones. The fourth speaker, activated by memory location 36877, is used mainly for explosions.

The ML program stores a starting number into the appropriate location and increases or decreases it for the period specified by the user. The interrupt of the computer will run through the program 60 times a second, which means that the starting number or tone will increase or decrease 60 times in one second.

Sound Duration

To make this a little clearer, let me explain that four memory locations have been assigned in the ML program to activate the four speakers, and four others to control the duration of the sounds.

Speaker	To Activate	Duration
1st	846	858
2nd	847	888
3rd	848	918
4th	849	948

The number POKEd into locations 846-849 is the starting number, which is stored in location 853 (dec); the initial value is 222, but this may be changed for the kind of sound you require. Locations 858, 888, 918, and 948 control the duration of the sounds. The program will generate the sounds for the number of jiffies (the 1/60 second interval used to measure time in Commodore machines) specified in these locations.

For a demonstration, RUN the program and then type SYS828; this will trap the interrupt. It will also set the volume control (location 36878) to maximum. Now POKE 846,222.

Location 858 contains 10 (dec), so the sound you heard was for ten jiffies. The program has stored 222 in location 36874 (first speaker), incremented it by one every sixtieth of a second until ten jiffies elapsed, then stored 0 into the memory location to switch off the speaker. To change the duration of the sound to, say, 20 jiffies, POKE 858,20. Now POKE 846,222.

The same method can be used for the other speakers. POKE 858,10. To change the starting number (that is, to get a tone which starts higher or lower), simply POKE into memory location 853. For example, POKE 853,240. Now POKE 846,240.

Explosion Simulation

It is necessary to POKE the starting number into locations 846-849; any other number will give only silence. Try POKE

847,240 (second speaker); it gives a sound of increasing frequency like the first. Now POKE 853,222:POKE 848,222. You notice this gives a sound that decreases in frequency. POKE 849,222 will simulate an explosion. By manipulating the durations and starting number, you can get almost any kind of sound from the first three speakers and explosions from the fourth. However, when you are changing the duration of the sounds, make sure it is not too long; for example, if you POKE 853,50:POKE 846,222 the program will store 222 in location 36874 and increment by one every jiffy for 50 jiffies. But in this case the contents of 36874 would increase to 255 and then cycle back to zero. You would hear a note for only 33 jiffies, since a number less than 128 in the sound generators of the VIC produces silence.

When using this program, you cannot generate sounds the normal way. To do so, you must first reset the interrupt vector by SYS996. This will also set the volume control to zero. To use the ML program, add the subroutine starting at line 8900 to your own BASIC program; and you can create sound effects using just one POKE, which would otherwise require a series of POKEs.

In a BASIC program with lines 8900-9240 added, you would first have a line like this in the main program to enter the ML into memory:

10 GOSUB 8900: REM SOUND GENERATOR

VIC Sound Generator

```
10 PRINT"{CLR}"
20 PRINT"{3 DOWN}{8 RIGHT}{RED}{RVS}VIC2
   0{OFF}"
30 PRINT"{2 DOWN}{6 RIGHT}VIC SOUND"
800 GOSUB8900
900 END
8900 FORJ=828TO1019:READF:POKEJ,F:NEXT
9000 DATA169,15,141,14,144,120,169,82
9010 DATA141,20,3,169,3,141,21,3
9020 DATA88,96,10,15,16,64,160,0
9030 DATA162,222,173,78,3,201,10,176
9040 DATA9,238,78,3,238,10,144,76
9050 DATA116,3,140,10,144,236,78,3
9060 DATA208,6,140,78,3,142,10,144
9070 DATA173,79,3,201,25,176,9,238
9080 DATA79,3,238,11,144,76,146,3
9090 DATA140,11,144,236,79,3,208,6
9100 DATA140,79,3,142,11,144,173,80
9110 DATA3,201,16,176,9,238,80,3
```

```
9120 DATA206,12,144,76,176,3,140,12
9130 DATA144,236,80,3,208,6,140,80
9140 DATA3,142,12,144,173,81,3,201
9150 DATA64,176,28,238,81,3,173,81
9160 DATA3,201,22,208,7,169,176,141
9170 DATA13,144,240,25,201,43,208,21
9180 DATA169,160,141,13,144,240,14,140
9190 DATA13,144,236,81,3,208,6,140
9200 DATA81,3,142,13,144,76,191,234
9210 DATA169,0,141,14,144,120,169,191
9224 DATA141,20,3,169,234,141,21,3
9230 DATA88,96,0,0,0,0,0,0
9240 RETURN
```

—3—

Making Sound with Blips

John Heilborn

This tutorial on using the VIC sound registers presents many fine examples. Also offered here is a simple sound editor that can be used to develop your own sounds.

A blip is the simplest kind of sound you can make with the VIC-20. You can make the most basic form of a blip by turning a tone on and then off again quickly. To get an idea of what this sounds like, enter and RUN the following program:

```
10 POKE 36878, 15: REM TURN ON THE VOLUME
20 POKE 36876, 200: REM TURN ON VOICE THR
    EE
30 POKE 36876, 0:REM TURN OFF VOICE THREE
```

By changing the value you POKE into 36876 (on line 20), you can produce blips that sound very different. For example, try POKEing 252 into 36876 instead of 200 to make a sound like a hammer hitting a chisel. Change the value to 128, and you'll get a tennis racket hitting a ball.

If you use different sound registers, you can get still greater variations with the blip. Try using the noise register (36877) instead of the third voice (36876) in lines 20 and 30:

```
10 POKE 36878,15
20 POKE 36877, 200
30 POKE 36877, 0
```

and you're chopping down a tree.

Another way you can change blips is to make them longer by adding a delay loop. Here's a line that will put a delay between lines 20 and 30. With it, the blip will sound for a longer time.

```
25 FOR R = 0 TO 5: NEXT
```

Try using different values in the delay loop, and notice the different sounds you can get from changing just the length of the blip.

Here are some sounds you can make using simple blips:

Alarm Clock

```
10 POKE 36878, 15
20 FOR T = 0 TO 10
30 POKE 36876, 240
40 POKE 36876, 0
50 FOR R = 0 TO 500: NEXT
60 POKE 36876, 237
70 POKE 36876, 0
80 FOR R = 0 TO 500: NEXT
90 NEXT
100 POKE 36876, 237
110 POKE 36875, 232
120 FOR A = 0 TO 5
130 FOR R = 0 TO 30
140 POKE 36878, 5
150 POKE 36878, 15
160 NEXT: NEXT
170 POKE 36876, 0
180 POKE 36875, 0
190 POKE 36878, 0
```

Dial Telephone

```
10 FOR I = 0 TO 6
20 POKE 36878, 3
30 G = RND(0) * 9 + 1
40 FOR R = 0 TO G * 1.5
50 POKE 36877, 228
60 POKE 36877, 0
70 NEXT
80 FOR K = 0 TO 200: NEXT
90 POKE 36878, 10
100 FOR R = 0 TO G
110 POKE 36874, 128
120 FOR D = 0 TO 1: NEXT
130 POKE 36874, 0
140 FOR O = 0 TO 50: NEXT
150 NEXT
160 FORJ=0TO350: NEXT: NEXT
```

Touch-Tone® Phone

```
10 A(0)=249:B(0)=238
20 A(1)=248:B(1)=232
30 A(2)=249:B(2)=232
40 A(3)=250:B(3)=232
50 A(4)=248:B(4)=234
60 A(5)=249:B(5)=234
70 A(6)=250:B(6)=234
```

```
80 A(7)=248:B(7)=236
90 A(8)=249:B(8)=236
100 A(9)=250:B(9)=236
110 FOR I = 0 TO 6
120 R = RND(0) * 9
130 POKE 36875, A(R)
140 POKE 36876, B(R)
150 POKE 36878, 10
160 FOR K = 0 TO 100: NEXT
170 POKE 36878, 0
180 FOR M = 0 TO 50: NEXT
190 NEXT
```

Motor Boat

```
10 POKE 36878, 4
20 FOR G = 0 TO 100
30 FOR E = 0 TO 2
40 POKE 36874, 230
50 POKE 36874, 0
60 NEXT
70 FOR H = 0 TO 40
80 NEXT
90 NEXT
```

Geiger Counter

```
10 POKE 36877, 128
20 POKE 36878, 15
30 POKE 36878, 0
40 FOR R = 0 TO RND(0) * 300: NEXT
50 GOTO 20
```

A Blip Editor

As you can see, blips can be very versatile sounds. Of course, one of the disadvantages of having many different ways of changing a sound is that it can take a long time to make the changes — especially if you have to make them all by POKEing the values into each register by hand. So to make your life a little easier, here's a little blip editor. With it you'll be able to change the tones and listen to the dynamics of each register. That way, when you find a sound you like, you can just write down the values indicated by the program and incorporate the sounds into your own programs.

Basic Blip Editor

```
10 A=200: C=5
15 PRINT "{CLR}"; A; C
20 POKE 36878, 15
```

```
30 GET A$: IF A$ = "" THEN 30
40 IF A$ = CHR$(133) THEN B = 36874: GOTO
   120
50 IF A$ = CHR$(134) THEN B = 36875: GOTO
   120
60 IF A$ = CHR$(135) THEN B = 36876: GOTO
   120
70 IF A$ = CHR$(136) THEN B = 36877: GOTO
   120
80 IF A$ = CHR$(145) AND A < 255 THEN A =
   A + 1: GOTO 15
90 IF A$ = CHR$(17) AND A > 128 THEN A =
   A - 1: GOTO 15
100 IF A$ = CHR$(29) AND C > 0 THEN C = C
   - 1: GOTO 15
110 IF A$ = CHR$(157) AND C < 255 THEN C
   = C + 1: GOTO 15
120 POKE B,A
130 FOR R= 0 TO C:NEXT
140 POKE B, 0
150 GOTO 30
```

This editor will allow you to manipulate the various tone registers and length of the blips you produce. When you RUN this program, the screen will clear and two numbers will appear. The left-hand number is the value that will be POKEd into the tone register you choose. The right-hand number is the length of the blip.

The function keys (F1, F3, F5, F7) select the tone registers. Press each of the function keys and listen to the difference between the sounds produced.

The cursor UP/DOWN key controls the tone by changing the value POKEd into the selected register. Unshifted, this key will lower the tone and shifted, it will raise the tone. Note that the lowest value allowed by this program is 128, and the highest value is 255. Press the cursor UP/DOWN key, and you will be able to see the value change. If you hold the key down, it will continue to change until it reaches its limit.

The cursor LEFT/RIGHT key controls the length of the blip. The larger the number, the longer the blip. Unshifted, this key will lower the number (shorten the blip). Shifted, it will raise the number, which will lengthen the blip.

Once you have some values chosen, hitting almost any key on the keyboard will replay the same sound. The only ones that won't work are: the COMMODORE key, RESTORE, CTRL, the

SHIFT keys, and RUN/STOP.

The last feature of this editor is the repeat function. Pressing the space bar or INST/DEL will repeat the selected sound continuously.

SAVE a copy of the blip editor; we will be adding features to it later.

Fading Blips

One of the most under-used features of the VIC sound system is the volume control, location 36878. Usually, we just set it for maximum volume and forget it until we want to turn the sound off.

But changing the setting of the volume control during a blip can create a totally new kind of blip. Here is a routine that uses a loop to fade the blip:

```
10 POKE 36876, 200
20 FOR R = 15 TO 0 STEP -1
30 POKE 36878, R
40 NEXT
```

By changing the STEP value, the fade rate can be altered. The value -.5, for example, will produce a tone twice as long, while using the value -2 will make the tone half as long.

Here are some examples of sounds made using the fade function.

Drums

```
10 POKE 36877, 223
20 FOR I = 0 TO 3
30 FOR R = 7 TO 0 STEP -7/5
40 POKE 36878, R: NEXT
50 NEXT
60 POKE 36874, 128
70 POKE 36877, 0
80 FOR R = 15 TO 0 STEP -3
90 POKE 36878, R: NEXT
100 POKE 36874, 0
110 FOR U = 0 TO 200: NEXT
120 POKE 36877, 223
130 FOR B = 0 TO 1
140 FOR R = 7 TO 0 STEP -7/5
150 POKE 36878, R: NEXT
160 FOR U = 0 TO 200: NEXT
170 NEXT
180 GOTO 20
```

Clumsy Tap-dancer

```
10 POKE 36877, RND (0) * 127 + 128
20 FOR M = 15 TO 0 STEP -(RND(0) * 15)
30 POKE 36878, M: NEXT
40 GOTO 10
```

Champagne Glasses Clinking

```
10 POKE 36876, 250
20 FOR R = 15 TO 0 STEP -3
30 POKE 36878, R: NEXT
40 FOR R = 15 TO 0 STEP -1
50 POKE 36878, R
60 NEXT
```

Firing Depth Charges

```
10 POKE 36878, 15
20 FOR R = 128 TO RND(0) * 127 + 128
30 POKE 36876, R: NEXT
40 POKE 36878, 0
50 GOTO 10
```

Clapping Hands

```
10 POKE 36877, 241
20 FOR R = 15 TO 0 STEP -(15/8)
30 POKE 36878, R: NEXT
40 FOR I = 0 TO 150: NEXT
50 GOTO 20
```

We can add a fade feature to the blip editor with just a few additional program lines:

```
72 IF A$ = CHR$(92) THEN F = F + 1: GOTO
     15
75 IF A$ = CHR$(169) AND F > 1 THEN F = F
     - 1: GOTO 15
112 IF A$ = "F" AND F$ <> "F" THEN F$ = "
     F": GOTO 15
115 IF A$ = "F" AND F$ = "F" THEN F$ = "
     ": GOTO 15
125 IF F$ = "F" THEN 160
130 POKE 36878, 15: FOR R = 0 TO C: NEXT
140 POKE B, 0
160 FOR R = 15 TO 0 STEP -(15/F)
170 POKE 36878, R
180 NEXT
190 GOTO 140
```

You'll also need to change line 15 so it will display the fade function and its value:

```
15 PRINT "{CLR}"; A; C; F$; F
```

And you will need to initialize F to 1 so line 160 will not give you a DIVISION BY ZERO ERROR. Change line 10 to:

```
10 A = 200: C = 5: F = 1
```

When you run the editor now, you will see three numbers at the top of the screen. The first two are still the tone value and the length of the blip. The third is the fade length.

If you hit the F key, an F will appear between the second and third numbers. This indicates that the fade function is on. Hitting any of the function keys now will produce a sound with the fade time shown. To increase the fade time, hit SHIFT-INST/DEL. Try increasing the fade time to ten and use the function keys again. Hitting the INST/DEL key unshifted will decrease the fade time.

Fading In

In addition to fading out a blip, we can also fade in a blip. This means starting the sound softly and gradually increasing the volume. Fading in is exactly the opposite of fading out. So instead of counting down the loop value, we count up like this:

```
10 POKE 36876, 200
20 FOR R = 0 TO 15
30 POKE 36878, R: NEXT
40 POKE 36878, 0
```

Here are some examples of sounds made using fade-in and fade-out.

Sawing Wood

```
10 POKE 36877, 200
20 FOR I = 0 TO 15 STEP (15/20)
30 POKE 36878, I: NEXT
40 FOR F = 15 TO 0 STEP-(15/20)
50 POKE 36878, F: NEXT
60 POKE 36878, 0
70 FOR G = 0 TO 20: NEXT
80 GOTO 20
```

Helicopter Taking Off

```
10 POKE 36877, 220
20 POKE 36874, 200
30 FOR Y = 15 TO 0 STEP -.05
40 FOR R = 0 TO Y
50 POKE 36878, R
60 POKE 36878, 0
```

```
70 NEXT
80 NEXT
```

Frog

```
10 POKE 36875, 240
20 FOR K = 0 TO 10
30 FOR I = 0 TO 1
40 FOR R = 0 TO 15 STEP 5
50 POKE 36878, R
60 POKE 36878, 0: NEXT
70 FOR M = 0 TO 30: NEXT
90 NEXT
100 FOR D = 0 TO 300: NEXT
110 FOR W = 0 TO 1000: NEXT
120 GOTO 20
```

Adding this feature to the blip editor is pretty much the same as adding the fade-out feature above. Start by adding these lines:

```
73 IF A$ = CHR$(47) THEN I = I + 1: GOTO
   15
74 IF A$ = CHR$(63) AND I > 1 THEN I = I -
   1: GOTO 15
113 IF A$ = "I" AND I$ <> "I" THEN I$ = "
   I": GOTO 15
114 IF A$ = "I" AND I$ = "I" THEN I$ = "
   ": GOTO 15
123 IF I$ = "I" THEN 200
200 FOR R = 0 TO 15 STEP (15/I)
210 POKE 36878, R: NEXT
220 IF F$ = "F" THEN 160
230 GOTO 140
```

Once again, you'll need to change line 15, this time to display the fade-in function and its value:

```
15 PRINT"{CLR}"; A; C; F$; F; I$; I
```

And you will need to initialize I to 1 so you won't get a DIVISION BY ZERO ERROR in line 230. Change line 10 to:

```
10 A = 200: C = 5: F = 1: I = 1
```

Finally, you'll need to change line 120 to:

```
120 POKE 36878, 0: POKE B, A
```

and line 130 to:

```
130 POKE 36878, 15: FOR R = 0 TO C: NEXT
```

With these modifications in the editor, you will see four num-

bers at the top of the screen. The first three are the same as they were before. The fourth is the fade-in function.

If you hit the I key, an I will appear between the third and fourth numbers, indicating that the fade-in function is on. Hitting any of the function keys now will produce a sound with the fade-in time shown. To increase the fade-in time, hit CTRL (left arrow). Hitting the left arrow key unshifted will decrease the fade-in time.

Some Final Tips

Look closely at the program examples. Many of them use some clever tricks that are not obvious with just a casual glance. The frog, for example, does not produce a smooth fade-in. It's broken into raspy pulses by turning the sound on and then off again right in the middle of the fade.

With the examples given and the blip editor, you should be able to produce some very interesting sounds. The whole point is not to be afraid of experimenting. That's what the blip editor is all about. And any time you get something interesting, write down the values so you'll be able to turn the numbers into a separate program.

—Chapter 4—

Programming Techniques

—4—

Programming Function Keys

Jim Wilcox

It would be nice if you could just touch one key and then a BASIC program would immediately be LISTed. Well, it can be done. Here's how.

The function keys are often used simply as an extension of the keyboard. Wouldn't it be nice to really be able to program the function keys? Well, that is exactly what is presented here.

The routine is in machine language, but no knowledge of machine language is needed to use this program. Once the program is typed in, double-check the DATA statements, since one error can cause the program to crash. SAVE the program before RUNning it. After you've typed RUN and pressed RETURN, the following should appear: F1 =? . Type in the BASIC command or statement you would like the function-one key to equal. For every carriage return you would like, type in the back arrow located on the upper left-hand corner of the VIC. Once you are sure the function key has been defined properly, press the RETURN key. The program will then ask for the rest of the function keys' definitions. After you have defined the eighth function key, the computer will print READY. The function keys are now ready to be used. Just press the appropriate function key, and the characters for which it was programmed will be printed.

```
RUN
F1 =? LIST←   LIST
F2 =? POKE    ,
F3 =? RUN←    RETURN
F4 =? PEEK(
F5 =? GOTO
F6 =? GOSUB
F7 =? PRINT PEEK(7680) ← SAVE
F8 =? LOAD←LIST ← VERIFY
```

What If It Doesn't Work?

If the VIC just locks up or if you don't get the READY message,

turn the VIC off and reLOAD the program. Recheck the program with the listing provided, from the beginning to line 65, especially the DATA statements.

When the READY message occurs after all eight keys have been defined and the VIC doesn't print the characters corresponding to the function key, check the program from lines 70 to 95.

If it still doesn't work, check the subroutine in lines 100 through 115.

How the Program Works

The BASIC program will POKE two machine language programs into your VIC. One goes into the cassette buffer, the other in the uppermost memory position. The program in the cassette buffer asks for the definition of each function key. Once the RETURN key is pressed, the program will store the ASCII value of the characters pressed in the uppermost portion of memory. After all eight keys have been programmed, the program will tell the computer to go to the other program in the top of memory every sixtieth of a second. The original program is not needed once the above operations have been performed and will be erased after any command for the cassette recorder is given. This is done to save 147 bytes of VIC's memory.

The second program will constantly check for a function key pressed. If one is pressed, the program will print the characters for which the function key was defined.

How to Save Memory

The longer each command for a function key, the more memory will be used up. If the commands are short, only about 200 bytes will be used up. The maximum amount of memory that can be used by this routine is about 800 bytes. To use the least number of bytes, the commands can be typed in the shorthand method shown on pages 133-34 in *Personal Computing on the VIC-20,* which came with your computer.

Having programmable keys can be a great aid to a computer operator. The VIC is equipped with eight keys which you can use for whatever purpose you want. Time can be saved in writing and debugging programs.

—4—

Programming Function Keys

```
5 F=Ø:C=PEEK(55)-12Ø:IFC<ØTHENC=C+256:F=
  -1
1Ø D=PEEK(56)+F:POKE55,C:POKE56,D:CLR
15 S=828:I=146:GOSUB1ØØ
2Ø DATA32,198,3,165,55,133,251,133,253,1
   65,56,133,252,133,254,169,49,133,Ø,16
   9
25 DATA133,133,1,169,13,32,21Ø,255,169,7
   Ø,32,21Ø,255,165,Ø,32,21Ø,255,169,61
3Ø DATA32,21Ø,255,169,63,32,21Ø,255,169,
   32,32,21Ø,255,32,2Ø7,255,72,16Ø,Ø,165
35 DATAl,145,55,1Ø4,32,198,3,2Ø1,13,24Ø,
   14,2Ø1,95,2Ø8,2,169,13,145,55,32
4Ø DATA2Ø7,255,76,124,3,23Ø,Ø,165,Ø,41,1
   ,2Ø8,1Ø,24,165,1,1Ø5,4,133,1
45 DATA76,17Ø,3,56,165,1,233,3,133,1,165
   ,Ø,2Ø1,57,144,163,12Ø,169,LØ,141
5Ø DATA2Ø,3,169,HØ,141,21,3,88,169,Ø,133
   ,Ø,32,68,198,76,116,196,166,55
55 DATA2Ø8,2,198,56,198,55,96
6Ø S=PEEK(55)+256*PEEK(56):I=119:GOSUB1Ø
   Ø
65 SYS(828)
7Ø DATA165,Ø,24Ø,59,16Ø,Ø,177,251,32,L99
   ,HØ,176,12,165,55,197,251,2Ø8,21,165
75 DATA56,197,252,2Ø8,15,169,Ø,133,Ø,165
   ,253,133,251,165,254,133,252,76,191,2
   34
8Ø DATA166,198,177,251,157,119,2,23Ø,198
   ,32,Ll11,HØ,165,198,2Ø1,1Ø,2Ø8,2Ø4,23
   Ø,Ø
85 DATA76,191,234,165,215,32,L99,HØ,176,
   3,76,191,234,165,8,41,1,2Ø8,247,16Ø
9Ø DATAØ,177,251,197,215,2Ø8,6,32,Ll11,H
   Ø,76,L6,HØ,32,Ll11,HØ,76,L81,HØ,2Ø1
95 DATA133,144,6,2Ø1,141,176,2,56,96,24,
   96,166,251,2Ø8,2,198,252,198,251,96
1ØØ F=Ø:FORD=STOS+I:READA$:IFASC(A$)<58T
   HENA=VAL(A$):GOTOll5
1Ø5 IFASC(A$)=76THENA=VAL(RIGHT$(A$,LEN(
   A$)-1))+PEEK(55):IFA>255THENA=A-256:
   F=1
11Ø IFASC(A$)=72THENA=VAL(RIGHT$(A$,LEN(
   A$)-1))+PEEK(56)+F:F=Ø
115 POKED,A:NEXT:RETURN
```

—4—

The Expanded/ Unexpanded VIC

Gary L. Engstrom

As more and more VIC owners add expansion memory to their computers, there is an increasing need for programs which run on all VICs, of any memory size. Here's how to write them.

The "where's my memory located now" problem can be overcome by careful programming. With or without RAM expansion in place, you should be able to run any of your own programs that require 3.5K or less of RAM. Of course, you will have to put up with removing and installing the expansion cartridge when using programs written by others, but you can have the convenience of universal VIC programs you write yourself.

For programs to be universal, they need to fulfill three requirements:

1. The program must not need more than 3.5K of memory. You just cannot squeeze more than that into the unexpanded VIC-20.

2. The program must contain memory location information for both the expanded and unexpanded VIC-20.

3. The program must be able to determine if expansion is in place and be able to choose between the two sets of memory locations.

To understand how a program can conform to these last two requirements, you need to understand that when the VIC-20 is turned on, its operating system goes through an initialization procedure. During initialization, one of the tasks that the operating system does is check to see if memory expansion is in place.

If so, the operating system sets certain pointers to one set of memory locations; if there is no memory expansion, these pointers are set to a different set of memory locations. If you have 8K or more RAM memory expansion for your VIC-20, you should be familiar with three of these memory locations (see Table 1). The computer uses the correct locations because, during initiali-

zation, pointers are set to the correct locations. It is the knowledge of the alternate memory locations and the existence of these pointers that make universal programs possible.

Table 1. Memory Locations

Memory Locations	Unexpanded	3K Expansion	8K + Expansion
User BASIC Area	4096-7679	1024-7679	4608-*
Screen Memory	7680-8191	7680-8191	4096-4607
Color Memory	38400-38911	38400-38911	37888-38399
*The end of user BASIC area with 8K or more expansion memory depends on the exact amount of memory added.			

Establish Alternate Values

Memory locations used as pointers can be used by a BASIC program to run on either an expanded or an unexpanded VIC-20. I chose memory location 43-44 ($002B-$002C), the pointer to the start of the BASIC program in memory. When the VIC-20 is not expanded, the decimal value of the high bit (location 44) is 16; when the VIC-20 is expanded by 8K or more, the decimal value of the high bit is 18.

This gives us enough information (using a PEEK statement) to create two paths for alternate memory values in a BASIC program. Thus we can assign the values for the beginning of screen memory and of color RAM for the expanded and unexpanded VIC-20 (see Program 1).

Program 1. Alternate Values

```
10 PRINT"{CLR}":REM *SET ALTERNATE VALUES
   *
20 IF PEEK (44)=18 GOTO 70: IF MEMORY IS
    IN PLACE
30 SM=7680 : REM SCREEN MEMORY FOR UNEXP
   ANDED VIC
40 CM=38400 : REM COLOR MEMORY FOR THE U
   NEXPANDED VIC
50 CS2=242 : REM CHARACTER SET 2 POINTER
    FOR THE UNEXPANDED VIC
60 GOTO 110 : REM SKIP
70 SM=4096 : REM SCREEN MEMORY FOR THE E
   XPANDED VIC
```

```
80 CM=37888 : REM COLOR MEMORY FOR THE E
   XPANDED VIC
90 CS2=194 : REM CHARACTER SET 2 POINTER
    FOR THE EXPANDED VIC.
```

You might have noticed the extra value. If you want to POKE characters from Character Set 2 to the screen, you have to POKE the character set pointer to the alternate set. The character set pointer is at memory location 36869. I have included the character set pointer value to demonstrate that you might want to use other alternate values in some of your programs.

Enter Common Values

After the alternate values have been set, you can set the values that are common to both the expanded and unexpanded VIC-20 (see Program 2). Of course, if you are not going to use a particular value, it can be left out.

Program 2. Common Values

```
100 REM *SET COMMON VALUES*
110 SB=36879:REM SCREEN/BORDER COLOR
120 V=36878 : REM VOLUME
130 S1=36874 : REM SPEAKER 1
140 S2=36875 : REM SPEAKER 2
150 S3=36876 : REM SPEAKER 3
160 S4=36877 : REM SPEAKER 4
```

Another benefit of using this method is that you don't have to constantly look up these memory locations or reenter these numbers each time you are going to use them. Every time you can avoid reentering a number, you are avoiding the possibility of an entry error.

Crunch and Save

Program 3 is a "crunched" version of Programs 1 and 2. Enter Program 3, then SAVE and VERIFY it on tape. Every time you start a new program, LOAD these four lines before you start to enter your own listing. When you write your program, start with line 100. Lines 50-90 can be used for defining variables and constants for your program.

Program 3. Lines 10 to 160 "Crunched"

```
10 PRINT"{CLR}":IF PEEK(44)=18 GOTO 30
20 SM=7680:CM=38400:CS2=242:GOTO40
```

```
30 SM=4096:CM=37888:CS2=194
40 SB=36879:V=36878:S1=36874:S2=36875:S3
   =36876:S4=36877
```

Try It Out

When all the values have been set, you can start to create your program. Program 4 is a short program that you can enter to demonstrate the flexibility of Program 3.

Program 4. Demonstration Program

```
100 REM *DEMONSTRATION PROGRAM*
110 POKE SB,120 : REM SET YELLOW SCREEN
    AND BLACK BORDER
120 POKE 36869,CS2 : REM POINT TO CHARAC
    TER SET 2
130 SS=INT(RND(1)*128)+128 : REM RANDOM
    VALUE FOR SPEAKER
140 CV=INT(RND(1)*8) : REM RANDOM COLOR
    VALUE
150 VS=INT(RND(1)*15)+1 : REM RANDOM VAL
    UE FOR VOLUME
160 X= INT(RND(1)*22) : REM RANDOM VALUE
     FOR X COORDINATE
170 Y=INT(RND(1)*23) : REM RANDOM VALUE
    FOR Y COORDINATE
180 POKE SM+X+22*Y,95 : REM POKE CHARACT
    ER TO SCREEN
190 POKE CM+X+22*Y,CV : REM POKE COLOR T
    O SCREEN
200 POKE V,VS : POKE S1,SS : POKE S2,SS
    : POKE S3,SS : POKE S4,SS : REM SOUN
    D
210 FOR T=1 TO 10 : NEXT T : REM PAUSE
220 GOTO 130 : REM REPEAT
```

Add the lines of Program 3 to Program 4, then SAVE and VERIFY the resulting program. Then, try it on both your expanded and unexpanded VIC-20. (Don't forget to turn the computer off before installing and removing the memory expander.) The program will adjust to the correct alternate set of values and work correctly with any configuration.

Practice POKEing

Using labels in place of actual numbers for POKEing might be confusing at first. However, once you get used to the labels, programming will be quicker and more accurate. To help you make

the transition, I will explain two ways that labels can be used to POKE color and characters to the screen.

Method 1: X/Y Coordinates

The X/Y coordinate method for POKEing characters to the screen takes advantage of the 22 columns and 23 rows of the VIC-20 screen. Refer to Table 2. The 22 columns are labeled X and are numbered 0 to 21; the 23 rows are labeled Y and numbered 0 to 22. All of the screen locations can be identified by column (X) and row (Y). For example, the center of the screen is at X =11 and Y =11; the lower left-hand corner is at X =0 and Y =22. To POKE characters to the screen, you must use the following formula: POKE SM +X +22 * Y,N where SM =7680 for the unexpanded and 3K expanded VIC-20, SM =4096 for the 8K or more expanded VIC-20, and N is the character code.

You can POKE color to the screen in the same way: POKE CM +X +22 * Y,N where CM =38400 for the unexpanded VIC-20, CM =37888 for the expanded VIC-20, and N is the color code.

LOAD Program 3 and then add the following POKE statements (Program 5).

Program 5. X/Y Coordinate Practice

```
100 X=0 : Y=0 : REM SET VALUES FOR X AND
    Y
110 POKE SM+X+22*Y,81 : POKE CM+X+22*Y,6
    : REM BLUE BALL--UPPER LEFT
120 X=21 : Y=0 : REM SET VALUES FOR X AN
    D Y
130 POKE SM+X+22*Y,83 : POKE CM+X+22*Y,2
    : REM RED HEART--UPPER RIGHT
140 X=11 : Y=11 : REM SET VALUES FOR X A
    ND Y
150 POKE SM+X+22*Y,90 : POKE CM+X+22*Y,0
    : REM BLACK DIAMOND--CENTER
160 X=0 : Y=22 : REM SET VALUES FOR X AN
    D Y
170 POKE SM+X+22*Y,65 : POKE CM+X+22*Y,4
    : REM PURPLE SPADE--LOWER LEFT
180 X=21 : Y=22 : REM SET VALUES FOR X A
    ND Y
190 POKE SM+X+22*Y,88 : POKE CM+X+22*Y,5
    : REM GREEN CLOVER--LOWER RIGHT
```

Table 2. Screen Memory Map

X=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Y=0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
2	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65
3	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87
4	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109
5	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131
6	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
7	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
8	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197
9	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219
10	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241
11	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263
12	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285
13	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307
14	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329
15	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351
16	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373
17	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395
18	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417
19	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439
20	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461
21	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483
22	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505

To make a character move on the screen, add a +1 to the value of X for right movement, add a -1 to the value of X for left movement, add a +1 to the value of Y for down movement, and add a -1 to the value of Y for upward movement. The limits of the screen are defined by X =0 to 21 and Y =0 to 22. Experiment by changing the values for X and Y in Program 5.

Method 2: Direct Method

There are 506 screen locations for both color and characters. The first location is SM (for Screen Memory) and CM (for Color Memory). The first location is the upper left-hand corner of the screen. The second location is to the right of the first location and has a value of SM +1 (for character placement) or CM +1 (for color placement).

We can continue to add values to the labels until we are at the bottom right-hand corner of the screen, where the values are SM +505 and CM +505. Therefore, any position on the screen can be addressed by adding the values of 0 through 505 to the labels SM or CM (see the memory map). LOAD the Alternate Values Listing (Program 3) and then add the following practice POKE statements (Program 6).

Program 6. Memory Location Practice

```
100 POKE SM+0,81 : POKE CM,6 : REM BLUE
    BALL--UPPER LEFT-HAND CORNER
110 POKE SM+21,83 : POKE CM+21,2 : REM R
    ED HEART--UPPER RIGHT-HAND CORNER
120 POKE SM+253,90 : POKE CM+253,0 : REM
     BLACK DIAMOND--CENTER
130 POKE SM+484,65 : POKE CM+484,4 : REM
     PURPLE SPACE--LOWER LEFT-HAND CORNE
    R
140 POKE SM+505,88 : POKE CM+505,5 : REM
     GREEN CLOVER--LOWER RIGHT-HAND CORN
    ER
```

To make a character move on the screen, add a +1 for right movement, add a -1 for left movement, add a +22 for down movement, and add a -22 for upward movement. The limits of the screen are defined by SM +505 and SM(for character placement) and CM +505 and CM (for color placement). Experiment by changing the values added to SM and CM in Program 6.

Which Method Is Best?

At this point you may be wondering which method for POKEing should be used. Each method has its place, depending on the requirements of your program. Generally, the direct method requires fewer commands for some applications and runs faster than the X/Y coordinate method. However, it is much easier to define complex screen boundaries using the X/Y coordinate method.

For example, let's place a five-character by five-character square on the screen. We'll use the X/Y coordinate method to place a square in the center of the screen, and the direct method to place a square in the lower left-hand corner. LOAD Program 3 and then add the following lines (Program 7).

Program 7. X/Y Coordinate Versus Direct Method

```
100 REM X/Y COORDINATE METHOD
110 FOR X=9 TO 13 : FOR Y=9 TO 13 : REM
    SET VALUES OF X AND Y
120 POKE SM+X+22*Y,160 : POKE CM+X+22*Y,
    8 : REM POKE CHARACTER AND COLOR
130 NEXT Y : NEXT X : REPEAT
140 REM DIRECT METHOD
150 L=396 : REM BEGINNING VALUE OF M
160 FOR M=L TO L+4 : REM RANGE OF M FOR
    ONE LINE
170 POKE SM+M,160 : POKE CM+M,8 REM POKE
     CHARACTER AND COLOR FOR ONE LINE
180 NEXT M : REM REPEAT TO END OF LINE
190 L=L+22 : IF L>488 THEN END : IF AT E
    ND OF LAST LINE END
200 GOTO160: REPEAT
```

When RUNning this program, you might have noticed that the second square was printed a little faster than the first one. In applications where speed is important, it is useful to know that the direct method does run quite a bit faster than the X/Y coordinate method.

This can be best illustrated by Program 8. In this program, the entire screen is filled with characters by using both methods. An added feature is that each segment of the program is timed by the VIC-20 built-in timer. LOAD Program 3 and then enter the following lines:

Program 8. Fill Screen Test

```
100 REM *FILL SCREEN TEST*
110 REM FILL SCREEN USING SCREEN MEMORY
    LOCATIONS
120 PRINT"{CLR}":REM CLEAR SCREEN
130 TI$="000000" : REM ZERO TIMER
140 FOR J=CM TO CM+505 : REM SET VALUES
    FOR COLOR MEMORY
150 POKE J,8 : REM POKE COLOR
160 NEXT J : REM REPEAT
170 FOR I=SM TO SM+505 : REM SET VALUES
    FOR SCREEN MEMORY
180 POKE 1,160 : REM POKE CHARACTER
190 NEXT I : REM REPEAT
200 T1$ = TI$ : RECORD TIME
210 REM FILL SCREEN USING X/Y COORDINATE
    S
220 PRINT"{CLR}":REM CLEAR SCREEN
230 TI$="000000" : REM ZERO TIMER
240 FOR Y=0 TO 22 : FOR X=0 TO 21 : SET
    VALUES FOR X AND Y
250 POKE CM+X+22*Y,8 : REM POKE COLOR
260 POKE SM+X+22*Y,160 : REM POKE CHARAC
    TER
270 NEXT X : NEXT Y : REPEAT
280 T2$=TI$ : REM RECORD TIME
290 PRINT"{CLR}":REM CLEAR SCREEN AND PRI
     NT RESULTS
300 POKE SB,157 : REM CHANGE SCREEN AND
    BORDER COLOR
310 PRINT "DIRECT METHOD{3 SPACES}"T1$ :
     REM PRINT TIME
320 PRINT "X/Y COORDINATES "T2$ : REM PR
    INT TIME
330 END
```

As you can see, the direct method RUNs about twice as fast as the X/Y coordinate method. If you are writing a program using a lot of POKEs, you might consider using the direct method wherever possible. This will help to speed up your program. However, the X/Y coordinate method remains the most useful when defining complex screen boundaries.

By using alternate values for screen memory and color memory, you are not only able to POKE characters and colors to the screen easily and accurately, but you will also be able to run your programs (3.5K or less) with or without your expansion cartridge.

—4—

Versatile Data Acquisition

Doug Horner and Stan Klein

This simple method of adjusting the VIC's internal jiffy clock can slow it down to match your timing needs, making possible "variable speed" machine language subroutines.

Home computers are finding their "homes" in labs, more and more frequently. Their flexibility and low cost make them excellent substitutes for more expensive special equipment. One common use is as a data acquisition device. Data acquisition systems monitor and record information on experiments in progress. For example, a chemist may use a special electrode to measure the concentration of a particular component in a chemical solution. As the concentration changes, the electrode sends a varying voltage to an analog-to-digital converter. The converter changes the voltage signal to binary data which can be recorded and stored for later analysis.

To log the data, the chemist could use a special-purpose data acquisition system perhaps costing thousands of dollars and useful only for a particular type of experiment. On the other hand, a microcomputer could be programmed to perform the same function. Moreover, to perform another type of experiment, the chemist need only modify the program instead of buying new equipment. When the data is stored, the computer might also be useful in analyzing it.

Surprisingly Simple

There is a surprisingly simple method for converting the VIC into a data acquisition system. A good acquisition system is based on a clock which uses interrupts to sample the user port at adjustable, fixed intervals. Data acquisition software is usually complicated because you must worry about interrupts generated from the jiffy clock.

A simpler scheme is to append the data acquisition routine to the front of the interrupt service routine which is already func-

tioning in connection with the jiffy clock. Every 16.667 milliseconds, VIC interrupts whatever it is doing to look at the keyboard and update the jiffy timer. Here's how to attach your own program to the jiffy service routine and how to set the jiffy clock to any rate of data acquisition.

To change the number of interrupts per second, just POKE different numbers into the low timer latch (37158) and the high timer latch (37159). Under normal operating conditions, these bytes are loaded with 137 in the low byte and 66 in the high byte. An interrupt is generated and the latches are reloaded into the counters whenever the counters are decremented to zero. The number of cycles between interrupts is two cycles greater than the number in the latches.

You might expect the counter to be loaded with 16667 less two, since the normal interrupts are every 1/60 of a second; but 66*256 +137 =17033 rather than 16665. This means simply that the "1 MHz" counter decrements at 1.022 MHz, not at an even rate of 1.00 MHz. So, to make the jiffy clock interrupt at a rate different than the normal 1/60 per second, just multiply the desired number of microseconds per interrupt by 1.022 and subtract two from that number. Example: for a millisecond interrupt (1000*1.022)-2 = 1020, so you would POKE 3 into the high byte at location 37159, and 252 into the low byte at location 37158 (3*256 +252 =1020) — and now you have an interrupt every millisecond.

There are limits to this method of changing the jiffy clock to produce varied interrupts. At the slow end, the largest number that could be loaded is $FFFF, or 65535. For the longest time interval between interrupts, the number of microseconds would be (65535 +2)/1.022 =64126, or roughly 1/15 of a second. The fast end limit is set by the percent of time remaining for BASIC. This percent is derived by (L-IR)/(L +2), where L is the number POKEd in the timer latch described above, and IR is the number of cycles taken up by the unmodified interrupt service routine.

There are approximately 220 cycles in the unmodified interrupt service routine; thus, if the number POKEd into the timer approaches 220, there will be no time available for anything other than attending to the interrupt service routine.

Here's how to add your own machine language routine to the jiffy clock service routine. Normally, when the decrementing counter hits zero, the operation is transferred to the interrupt service routine whose beginning address ($EABF) is stored in 788 and 789 ($0314 and $0315). By changing the address in 788 and

789, you can tell VIC to do additional instructions in machine language and then go to $EABF to run the normal service routine.

To change the address in 788 and 789, you must disable the interrupt enable register for the jiffy clock to allow the number in these locations to be changed. POKEing location 37166 with 64 will disable the interrupt; after the addresses in 788 and 789 have been changed, POKEing location 37166 with 192 will enable the interrupts again. Here's a sample program:

```
10 POKE52,28:POKE56,28:REM SETTING UPPER
    BOUNDARY FOR BASIC
15 FOR Z=0 TO 9:READ Q:POKE(28*256+Z),Q:
   NEXT Z:REM MACHINE PROGRAM IN PAGE 28
20 POKE 37166,64:POKE 788,0:POKE 789,28:P
    OKE 37166,192
21 REM LINE 20 CAUSES THE INTERRUPT TO N
   OW GO TO PAGE 28
25 DATA 173,16,145,157,0,29,232,76,191,2
   34
30 INPUT"LOW";N1:INPUT"HIGH";N2:POKE3715
   8,N1:POKE37159,N2
31 REM LINE 30 CHANGES THE TIMING OF THE
    INTERRUPT
```

The machine language program in line 25 disassembles to:

```
1C00 LDA $9110;     Get data from user port
1C03 STA $1D00,X;   Store data in page 29 ring buffer
1C06 INX;           Increment pointer for ring buffer
1C07 JMP $EABF;     Jump to normal jiffy service routine
```

This program can be used as a guide for setting up the jiffy clock for timed data acquisition. One additional consideration in terms of the percent of time left for BASIC: the above program has added an additional fourteen cycles which must be added to the IR variable. Exercise caution if data is to be gathered at faster than half-millisecond intervals.

—Chapter 5—

Utilities

—5—

Pause

Doug Ferguson

Here is a revised version of a pause control for your VIC. Once you have it LOADed into memory, you can pause the listing simply by pressing the SHIFT key.

For VIC owners who have not bought a printer yet, studying a BASIC program by using the LIST command can be tedious. The screen displays only about 20 lines at a time if you hit the STOP key. And then the only way to restart at the point where you stopped is to retype LIST — again and again. Even the CONTROL key is not much help; the lines still move by too fast for more than a superficial look. What is needed on the VIC is a PAUSE key.

The program listed here is a new version of an earlier Pause routine that I wrote. Unlike my earlier version, this one patches directly into the LIST routine in ROM without interfering with anything else. Once activated, there is never any need to turn it off.

Type in the program exactly as written and SAVE it before you RUN it. The program deletes itself when run, so you would have to retype it if you run it first. Once it is SAVEd, RUN the utility. Then LOAD in a BASIC program and give it a try. LIST your program to the screen and while it is scrolling press the SHIFT key. Use the SHIFT LOCK to free both hands.

How It Works

Line 20 sets the low-byte/high-byte address of a machine language "patch" at the top of RAM memory. Line 30 redefines memory size to protect the patch and move the LIST vector at 774-775 ($0306-$0307) to renavigate the indirect jump at $A717 in ROM ($C717 in the VIC). The remaining lines create the patch at the top of RAM. Also note that the program assumes the normal LIST vector at power-up; line 20 thus prevents your accidentally trying to RUN the program more than once per power-up.

Pause

```
10 REM PAUSE
20 L=232:H=PEEK(56)-1:Q=PEEK(775):IF Q<1
   67 THEN 80
```

```
30 POKE 51,L:POKE 52,H:POKE 55,L:POKE 56
   ,H:POKE 774,L:POKE 775,H
40 FOR X=L+H*256 TO X+21:READ D:POKE X,D
   :NEXT
50 POKE X,Q
60 DATA 72,152,72,32,159,255,169,1,44,14
   1,2,208,246
70 DATA 169,0,133,198,104,168,104,76,26
80 NEW
```

—5—

Bidirectional Scrolling

Charles Saraceno

How would you like to be able to check and debug your programs by turning your screen into a window which can move anywhere over the listing, stop or start at will, and even move upwards toward the start of the program? All this can be achieved by just touching different keys when using this clever "controlled scrolling" program.

Now that memory expansion modules are readily available, it is possible to write longer VIC programs. This does make it harder, however, to edit the contents without a hard copy from a printer to examine for typing errors. Screen editing is time-consuming, to say the least; with 22 characters per line, you are limited to four or five lines at a time between LIST commands. A very useful LIST would scroll the screen and stop or continue when you want it to. The ideal LIST would also scroll backwards.

This small program efficiently accomplishes all these tasks. Line 63001 determines the starting address (SA) for any memory installed into the VIC. Line 63002 calculates the line number (LN) of your program. Line 63003 sets your screen up to perform the tasks needed to list the line, then continues the program. It is written in white so you won't see the commands and keeps the screen uncluttered for reviewing the listed line.

Once a "list" has been initiated in a program, the program will end. This is where the keyboard buffer commands in line 63004 both control the list and then continue the program with the "go to" 63010 command. Lines 63010-63030 let you review the line just listed and wait for you to press the + key to advance to the next line or the - key to back up to the previous lines listed. Line 63100 looks for the next 0 in BASIC, which indicates the end of that BASIC line, and then sends you back to calculate the next line number. Line 63200 is the routine that looks for the end of the previous line. You have to eliminate the possibility of finding a 0 in the addresses that determine the line number by disallowing a 0 in either of those two addresses.

One other little trick will let you avoid having to type in this program after each main program has been entered. Find the end of BASIC by typing in:

CLR: PRINT PEEK (45),:PRINT PEEK (46)

Now type the following line which moves the beginning of BASIC to two bytes less than the end of the program (either a null or a 0 is needed to start loading in a new program):

POKE 43, PEEK (45)-2:POKE 44,PEEK (46)

Now load in " +/- LIST" program and then reset the BASIC pointers as follows:

Unexpanded	**POKE 43,1:POKE 44,16**
3K Expander	**POKE 43,1:POKE 44,4**
8K or More Expander	**POKE 43,1:POKE 44,18**

Start editing by typing in RUN 63000. You will be able to scrutinize your program on a line-by-line basis. Any mistakes discovered should be noted on paper and corrected after your review.

Bidirectional Scrolling

```
63000 REM** +/- LIST**
63001 SA=PEEK(44)*256+PEEK(43)-1
63002 LN=PEEK(SA+3)+PEEK(SA+4)*256
63003 PRINT"{CLR}{WHT}GOTO 63010":PRINT"L
      IST";LN;
63004 POKE631,19:POKE632,17:POKE633,31:PO
      KE634,13:POKE 635,19:POKE636,13:PO
      KE 198,6:END
63010 IF PEEK(197)=5 THEN 63100:REM TEST
      FOR "-"KEY
63020 IF PEEK(197)=61 THEN 63200:REM TEST
       FOR "+" KEY
63030 GOTO 63010
63100 IF PEEK(SA+5)<>0 THEN SA=SA+1:GOTO
      63100
63110 SA=SA+5:GOTO 63002
63200 SA=SA-1:IF PEEK(SA)=0 AND PEEK(SA-4
      )<>0 AND PEEK (SA-3)<>0 THEN 63002
63210 GOTO 63200
```

—5—

VICword

Mark Niggemann

How would you like to be able to use single-key entry for 52 BASIC commands? With "VICword" running in your VIC, you can hold down the SHIFT key and hit the letter "L" and the word "LOAD" will appear on screen. Hold down the COMMODORE key and hit "L" and "SAVE" writes itself on the screen. Especially helpful when typing in those long BASIC programs, VICword is a clever machine language program that puts itself into memory (expanded or not), protects itself from interference by BASIC, and then tells you how to turn it on or off whenever you want.

Before buying a Commodore VIC, I used my father's PET for most of my programming work. One nice programming aid that I had at my disposal was Charles Brannon's "Keyword." After typing in a couple of long programs on the VIC, I set out to make a revision of Keyword for the VIC.

I was not content with only 26 defined keys. After all, the VIC has both the COMMODORE key and the SHIFT key. So, why not use both to get a total of 52 defined keys? This would prove to be a difficult task. The original Keyword program relied on the fact that the ASCII code values of the SHIFTed letters were in numeric order. On the VIC, the COMMODORE keyed letters are not in that order. This made things very tough.

After looking at Jim Butterfield's memory map (*COMPUTE!'s First Book of VIC*), I noticed a curious link located at $028F and $0290, respectively, that I thought might help. After some further examination, I found that this link points to a routine in ROM that sets up the appropriate keyboard lookup table, depending on whether the SHIFT, COMMODORE, or CONTROL key is being depressed. The lights came on at this point. Since this routine in ROM is part of the interrupt scan for clock updating, cursor flash, and keyboard handling, it is possible to run "VICword" using this link and also to take care of the problem of the COMMODOREd letters.

When you SYS the ON/OFF address given by the loader program, VICword will set the link at $028F and $0290 to point to its scan portion. In scanning, VICword checks to see if the quote mode flag is set. This is done so that you can still get graphics

characters when you need them. If this flag is set, VICword will promptly exit the scan. If it isn't, VICword then checks if the SHIFT or COMMODORE key is being pressed. If either is pressed, then the keyboard lookup table pointer, located at $F5 and $F6, is set to point to the SHIFT key lookup table.

By using this table, and not the COMMODORE key lookup table, the ASCII values are in numeric order. VICword will determine which table of token values it will use and will read the tokenized keyword for the particular key pressed. The rest of VICword is identical to Keyword in function.

Entering VICword

Some precautions should be observed when you type in VICword. Since this is a machine language program, a single mistake in the DATA statements could cause VICword to crash. Generally, it is a good idea to SAVE any machine language program before you try to execute it. Then, if you do crash and you can't get out of it by using RESTORE, you can just load in the version that you saved and recheck the DATA for any erroneous entry.

When I defined the keyword tables used in VICword, I chose the most commonly used keywords in BASIC. I tried to make most of the SHIFT keys complementary to the COMMODORE keys. For example, SHIFT G is GOSUB and COMMODORE G is RETURN. Not all keys could be paired up like this. See the table to find out the key definition.

I have used VICword quite often to help out on those long programs. I hope that VICword is as useful a tool for you as it has been for me.

VICword

```
100 REM** VICWORD LOADER
140 IF PEEK(PEEK(56)*256)<>120THENPOKE56
    ,PEEK(56)-1:CLR
150 HI=PEEK(56):BASE=HI*256
160 PRINT"{CLR}PATIENCE..."
170 FOR AD=0 TO 211: READ BY
180 POKE BASE+AD,BY: NEXT AD
190 :
200 REM RELOCATION ADJUSTMENTS
210 POKE BASE+26,HI: POKE BASE+81,HI
220 POKE BASE+123,HI: POKE BASE+133,HI
230 :
240 PRINT"{CLR}*** VICWORD ***"
```

—5—

Keys into BASIC Commands

KEY	SHIFT	COMMODORE
A	PRINT	PRINT#
B	AND	OR
C	CHR$	ASC
D	READ	DATA
E	GET	END
F	FOR	NEXT
G	GOSUB	RETURN
H	TO	STEP
I	INPUT	INPUT#
J	GOTO	ON
K	DIM	RESTORE
L	LOAD	SAVE
M	MID$	LEN
N	INT	RND
O	OPEN	CLOSE
P	POKE	PEEK
Q	TAB(	SPC(
R	RIGHT$	LEFT$
S	STR$	VAL
T	IF	THEN
U	TAN	SQR
V	VERIFY	CMD
W	DEF	FN
X	LIST	FRE
Y	SIN	COS
Z	RUN	SYS

```
250 PRINT"ON/OFF:{3 SPACES}SYS{RVS}";BAS
    E
260 END
270 DATA 120, 173, 143, 2, 201, 32
280 DATA 208, 12, 169, 220, 141, 143
290 DATA 2, 169, 235, 141, 144, 2
300 DATA 88, 96, 169, 32, 141, 143
310 DATA 2, 169, 0, 141, 144, 2
320 DATA 88, 96, 165, 212, 208, 117
330 DATA 173, 141, 2, 201, 3, 176
340 DATA 110, 201, 0, 240, 106, 169
350 DATA 159, 133, 245, 169, 236, 133
360 DATA 246, 165, 215, 201, 193, 144
370 DATA 95, 201, 219, 176, 91, 56
380 DATA 233, 193, 174, 141, 2, 224
```

```
390 DATA 2, 208, 3, 24, 105, 26
400 DATA 170, 189, 159, 0, 162, 0
410 DATA 134, 198, 170, 160, 158, 132
420 DATA 34, 160, 192, 132, 35, 160
430 DATA 0, 10, 240, 16, 202, 16
440 DATA 12, 230, 34, 208, 2, 230
450 DATA 35, 177, 34, 16, 246, 48
460 DATA 241, 200, 177, 34, 48, 17
470 DATA 8, 142, 211, 0, 230, 198
480 DATA 166, 198, 157, 119, 2, 174
490 DATA 211, 0, 40, 208, 234, 230
500 DATA 198, 166, 198, 41, 127, 157
510 DATA 119, 2, 230, 198, 169, 20
520 DATA 141, 119, 2, 76, 220, 235
530 DATA 76, 67, 236
540 :
550 REM *VICWORD TOKENS FOR SHIFT KEY
560 :
570 DATA 153, 175, 199, 135, 161, 129
580 DATA 141, 164, 133, 137, 134, 147
590 DATA 202, 181, 159, 151, 163, 201
600 DATA 196, 139, 192, 149, 150, 155
610 DATA 191, 138
620 :
630 REM *TOKENS FOR COMMODORE KEY
640 :
650 DATA 152, 176, 198, 131, 128, 130
660 DATA 142, 169, 132, 145, 140, 148
670 DATA 195, 187, 160, 194, 166, 200
680 DATA 197, 167, 186, 157, 165, 184
690 DATA 190, 158, 0
```

—5—

Automatic BASIC

Karl R. Beach

Educators, adventure game writers — anyone who wants to create video displays in a BASIC program — will find this automatic screen generator easy to use and a real timesaver.

This program allows you to compose a page of text, to create text animation exactly as it will later be seen on the screen, or to construct audience interaction programs. The screen display is then automatically converted into bug-free lines of BASIC that can be entered into memory as part of the program. When the programmer is ready, the core is quickly deleted, and the remaining finished program is SAVEd.

This program should be useful to educators who wish to prepare interactive instructional programs as quickly as possible. Other uses include writing "VIC letters" to friends, training children in word processing, and preparing text for interactive adventure games.

Writing the Text Block

Let's use this program to write a BASIC program block beginning at line 2000. You will first be asked to enter a starting line number (which could be any number between 1000 and 7000), and then you will be greeted with a screen display that represents a blank page of text. The red line is the right-hand margin, which limits text-line lengths to 21 characters.

As you begin typing, you'll see that the first line of a text page is indented two spaces. To avoid the indentation, press the "correction key" — the left arrow (←) — and begin typing from the left margin. If you make a mistake, you can correct it before entering the text line by pressing the "correction key" and retyping the entire line. If you discover a mistake after entering the text line, you can correct it at the end of the page when the conversion to BASIC is underway. If you run into the right margin while typing, you must press either the "correction key" or the RETURN key.

When you have completed a satisfactory text line, enter it by pressing the RETURN key. After your ninth line, you will be asked whether you want the reader to proceed to the next page

or, before moving on to the next page, to answer a question which you may have written into the text. Let's assume you've typed the following page of text and a question for the reader:

George Washington was the first President of the United States. What was his wife's name?

(1) Alice
(2) Martha
(3) Melissa

At this point, you will be prompted to enter an answer string. The multiple-choice format is the quickest and most problem-free for use with children, although this program can be easily modified to accommodate a number of different answer formats. If your text page consists of fewer than nine text lines, you need to press the British pound symbol (£) immediately to the left of the CLR/HOME key in order to terminate the page.

Your screen is instantly filled with what appears to be a part of a BASIC program listing beginning at line 2000 and containing the text you just typed. If you have no text line errors to correct, press the HOME key. If you have errors, drive the cursor to the top of the screen using the cursor control keys and correct your errors on the way. Now press the RETURN key until all the program lines have been entered. Note the last line number used in this sequence; then type in RUN again and begin the process anew, using a higher starting line number.

Adding Animation

If you want text animation, first type in up to three lines of static text as you did previously. After you've entered the final line of static text, press the up arrow (↑) between the asterisk and the RESTORE key. You'll first be prompted to enter a color for the animated text lines (default will be black, as are the static text lines), and then you'll see the first in a series of four input prompts that have been "bent" to allow you to type from the left margin. Remember not to exceed 21 characters in an animated text line. In this mode, the text lines cannot include commas or colons. When you are satisfied with each animated text line, press the RETURN key. Here is an example:

—5—

. Computers .
. can help .
. students .
. learn! .

The dots in each animated text line are used to maintain "space" within the string in which each line is stored. The time delay for this animation is set at line 855 and should be adjusted to fit the reading level of your users. A FOR/NEXT time delay can be inserted between the static and animated text lines and also could be used to emphasize an important point repeatedly. Use color and cursor controls carefully to insure the effect you want.

When you are out of memory or when you have finished writing text-pages, delete the core of this program by typing RUN 70. Then, when prompted, enter the number one. Numbers 1-20 will scroll onto the left side of the screen. Press the cursor HOME key and gently tap the RETURN key 20 times. A second RUN 70 will allow you to quickly delete lines 21-40, a third deletes lines 41-60, and a fourth deletes the deletion program block.

A handy tool to put into your computer when writing BASIC programs is the following block:

```
9000 INPUT Q9
9005 PRINT "{CLEAR}":FOR I =1 TO 20:
     PRINT Q9:Q9 =Q9 +1:NEXT I:END
```

This is especially useful if you renumber a program block and wish to delete the "old" block.

The last step before SAVEing your program is to delete line 7000. This line is a "safety net" that allows you to RUN portions of the program you are writing without triggering the LOAD command used in chaining at line 756. If you don't wish to chain, don't delete line 7000.

Program 1. Auto-BASIC

```
1 POKE36869,242:PRINT"{CLR}{BLU}{DOWN}
  {4 RIGHT}AUTO-BASIC"
2 POKE36879,27:K2=7701:K3=38421:INPUT"
  {BLK}{3 DOWN}{3 RIGHT}BEGIN LINE
  {SHIFT-SPACE}#";P
3 PRINT"{CLR}":FORI=1TO23:POKEK3,2:POKEK
  2,92:K2=K2+22:K3=K3+22:NEXTI
4 A$="PRINT":B$=CHR$(34):Q$="{DOWN}":L$=
  "{2 RIGHT}"
```

—5—

```
5 FORZ=1TO9
6 IFZ=1THENPRINT"{HOME}{2 SPACES}{RVS}+
  {OFF}";:F$=F$+"{2 RIGHT}":GOTO12
7 Y=LEN(F$)+1:FORI=1TOY: PRINT"{LEFT}";:
  NEXTI:PRINT"{2 DOWN}{RVS}+{OFF}";
  {11 SPACES}
9 Y=0:F$=""
12 GETE$:IFE$=""THEN12
13 IFE$=CHR$(95)THENY=LEN(F$):FORI=1TOY:
   PRINT"{LEFT}";:NEXTI:F$="":GOTO12
14 IFE$=CHR$(13)THENE$="":PRINT"{LEFT} "
   ;:GOTO23
15 IFE$=CHR$(94)THENE$="":GOTO50
16 IFE$=CHR$(92)THENE$="":GOTO40
17 F$=F$+E$:PRINT"{LEFT} ";:PRINT"{LEFT}
   ";E$;:PRINT"{RVS}+{OFF}";{7 SPACES}
18 IFLEN(F$)>16THENPOKE36878,15:FORI=1TO
   10:POKE36875,225:NEXTI:POKE36878,0:PO
   KE36875,0
19 IFLEN(F$)<21THEN12
20 GET E$:IF E$=CHR$(13)THEN23
21 IFE$=CHR$(95)THEN13
22 GOTO 20
23 F2$=STR$(P)+A$+B$+Q$+F$+B$
24 IFZ=1THENZ$=STR$(P)+A$+B$+"{BLK}"+F$+
   B$
25 IFZ=2THENY$=F2$
26 IFZ=3THENX$=F2$
27 IFZ=4THENW$=F2$
28 IFZ=5THENV$=F2$
29 IFZ=6THENU$=F2$
30 IFZ=7THENT$=F2$
31 IFZ=8THENS$=F2$
32 IFZ=9THENR$=F2$
33 P=P+2
34 NEXTZ
40 PRINT"{HOME}{19 DOWN}1=PAGE 2=ANSWER"
41 GETH$:IFH$=""THEN41
42 P=P+2
43 IFH$="1"THENM3$=STR$(P)+"GOSUB900":GO
   TO46
44 K2=7701:FORI=1TO23:POKEK2,32:K2=K2+22
   :NEXTI:INPUT"ANSWER=";M5$
45 M4$=STR$(P)+"A$="+B$+M5$+B$+":GOSUB82
   5"
46 PRINT"{CLR}":B=0
47 PRINTZ$:PRINTY$:PRINTX$:PRINTW$
48 PRINTV$:PRINTU$:PRINTT$:PRINTS$
49 PRINTR$:PRINTM3$:PRINTM4$:END
```

```
50 PRINT"{CLR}":INPUT"COLOR 3,5,6,7";B9
51 IFB9=3THENC9$="{RED}"
52 IFB9=5THENC9$="{PUR}"
53 IFB9=6THENC9$="{GRN}"
54 IFB9=7THENC9$="{BLU}"
55 C6$=":GOSUB850":E$=""
56 INPUT"{DOWN}{2 LEFT}";U2$
57 INPUT"{DOWN}{2 LEFT}";T2$
58 INPUT"{DOWN}{2 LEFT}";S2$
59 INPUT"{DOWN}{2 LEFT}";R2$
60 IFC9$=""THENC9$="{BLK}"
61 P=P+2:U$=STR$(P)+"GOSUB850:"+A$+B$+"
   {2 DOWN}"+C9$+U2$+B$+C6$
62 P=P+2:T$=STR$(P)+A$+B$+"{UP}"+C9$+T2$
   +B$+C6$
63 P=P+2:S$=STR$(P)+A$+B$+"{UP}"+C9$+S2$
   +B$+C6$
64 P=P+2:R$=STR$(P)+A$+B$+"{UP}"+C9$+R2$
   +B$+C6$
66 GOTO40
68 END
70 INPUT "1-21-41-61";A
71 FORI=1TO20
72 PRINTA
73 A=A+1
74 NEXTI
75 END
88 POKE36869,242:PRINT"{CLR}WHAT'S YOUR
   NAME?"
89 INPUT"{DOWN}";Z$
90 Z$=Z$+""
92 POKE36879,25
95 GOSUB900
100 GOTO 1000
750 PRINT"{CLR}{GRN}PLEASE WAIT WHILE"
752 PRINT"{DOWN}I LOAD MORE PAGES"
754 PRINT"{DOWN}FROM MY CASSETTE!"
756 LOAD
758 END
825 PRINT"{HOME}{20 DOWN}{PUR}{RVS}TYPE
    NUMBER OF ANSWER!{OFF}"
826 GETB$:IFB$=""THEN826
827 IFA$<>B$THEN835
828 PRINT"{3 UP}{BLK}CORRECT, ";Z$;"!"
829 GOSUB890
830 FORI=1TO2000:NEXTI
832 PRINT"{CLR}":RETURN
835 PRINT"{3 UP}{BLK}THE ANSWER IS ";A$
837 FORI=1TO2000:NEXTI
```

```
840 PRINT"{CLR}":RETURN
850 GOSUB890
855 FORI=1TO1000:NEXTI
860 RETURN
890 POKE36878,15:FORI=1TO10:POKE36875,22
    5:NEXTI:POKE36878,0:POKE36875,0:RETU
    RN
900 PRINT"{HOME}{20 DOWN}{RED}{RVS}PRESS
     KEY FOR NEW PAGE{OFF}"
901 GETO$:IFO$=""THEN901
902 PRINT"{CLR}{BLK}":RETURN
905 END
1000 REM
7000 END
7005 GOTO750
```

Program 2. Make These Changes to Program 1 When Using 8K or 16K Expander

```
1 POKE36869,194:PRINT"{CLR}{BLU}{DOWN}
   {3 RIGHT}AUTO BASIC"
2 POKE36879,27:K2=4117:K3=37909:INPUT"
   {BLK}{3 DOWN}{3 RIGHT}BEGIN LINE#";P
44 K2=4117:FOR I=1 TO 23:POKE K2,32:K2=K2
    +22:NEXT I:INPUT"ANSWER=";M5$
88 POKE36869,194:PRINT"{CLR}{DOWN}WHAT'S
    YOUR NAME?"
```

—5—

The VIC Keyboard Redefined

Amihai Glazer

With this short program for the unexpanded VIC, you can make any key on the keyboard represent any other key. This gives you the freedom to make an alphabetic keyboard, a numeric keypad, or any keyboard plan you need.

You might need to use a numeric keyboard on your VIC. As it is, all numerals are situated on the top row of the keyboard instead of being conveniently arranged in a square pattern which makes data entry easy. This program creates just such a keypad in the center of the keyboard, as shown in the figure. Thus, for example, hitting the space bar will be equivalent to hitting 0, and hitting the R key will have the same effect as hitting the 7 key.

Not only will the screen show numerals each time the appropriate keys are pressed, but the computer will actually interpret these alphabetic keys as the corresponding numerals. The program also allows the user to redefine *any* key as any other key. You can, for example, rearrange your keys in alphabetical order, or create any keyboard you like.

Type in the program and SAVE it first, then RUN it. To enable the new interpretation of the keys, type SYS 7424 and hit RETURN. You now have a numeric keypad. To return to a normal keyboard, just hit the RUN and RESTORE keys simultaneously (alternatively, you can execute the statement POKE 655,220: POKE 656,235). Executing a SYS 7424 will bring back the numeric keypad.

You can also redefine keys of your own choosing. Type GOTO 220 and hit RETURN. Now enter pairs of keys: the key you want changed, and then its new meaning. To stop the program, hit the F1 key. Thus, if you want the key labelled = to mean *, hit the = key, then the * key, and then the F1 key. To turn on these new definitions, type SYS 7424 and hit RETURN.

—5—

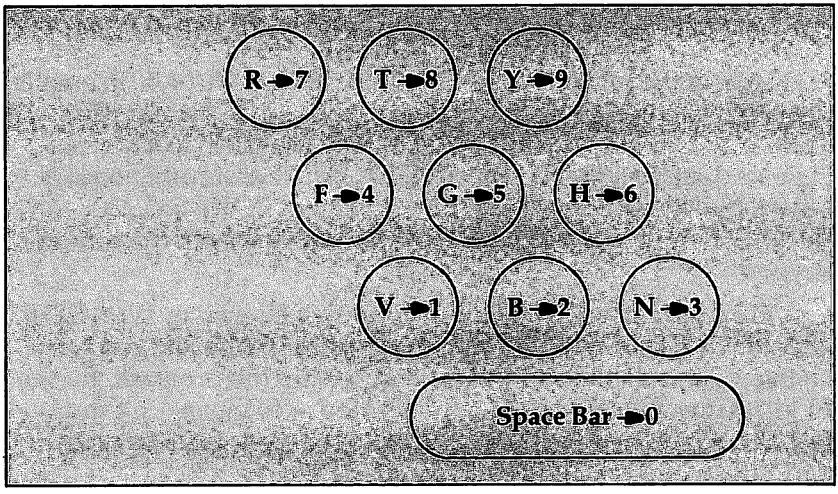
What's Happening

The program works as follows. Normally, during interrupt processing, every sixtieth of a second the VIC calls the decode logic machine language program, whose address ($EBDC) is in the jump vector in locations $028F-$0290. Our machine language program in locations $1D00-$1D14, however, sends the VIC to another machine language program we've put in locations $1D15-$1D24.

This program picks up the code for the key just pressed, given in location $CB. It then indexes into a recode table (beginning in location $1D27, decimal 7463), and puts the new code back into location $CB. Processing continues by jumping into the normal decode logic program in ROM, which is at location $EBDC.

The program's lines 10-110 insert these two machine language programs into memory. Lines 120-140 initialize the recoding table, and lines 150-200 recode the keys in the form shown in the figure. Custom recoding by the user is provided for in lines 220-330. The recoding table is initialized in lines 230-250. CO$ and CN$ get the key that is being redefined and its new definition. The codes the VIC uses for these keys are obtained from location 203 ($CB); CO and CN are assigned these values. A code of 39 (representing the F1 key) stops the program. The appropriate changes in the recoding table, which will be used by the machine language program, are performed in lines 310-320.

Numeric Keypad



Redefined Keyboard

```
10 REM CHANGE KEYBOARD
20 POKE 52,29: POKE 56,29:CLR
30 FOR I=7424 TO 7462
40 READ D
50 POKE I,D
60 NEXT I
70 REM MACHINE LANG.
80 REM PROGRAM
90 DATA 120,8,72,138, 72,169,21,141,143,
   2,169,29,141,144,2,104,170,104
100 DATA 40,88,96,8,72,138,72,166,203,18
    9,39,29,133,203,104,170,104,40
110 DATA 76,220,235
120 FOR I=0 TO 64
130 POKE 7463+I,I
140 NEXT I
150 FOR I=1 TO 10
160 READ CO,CN
170 POKE 7463+CO,CN
180 NEXT I
190 REM RECODED KEYS
200 DATA 32,60,27,0,35,56,28,1, 42,57,19
    ,2,43,58, 10,3,50,59,11,4
210 END
220 REM CUSTOM RECODE
230 FOR I=0 TO 64
240 POKE 7463+I,I
250 NEXT I
260 PRINT "INPUT OLD, NEW"
270 GET CO$:IF CO$="" THEN 270
280 CO=PEEK(203):IF CO=39 THEN STOP
285 PRINT CO$;" ";
290 GET CN$: IF CN$="" THEN 290
300 CN=PEEK(203)
310 PRINT CN$
320 POKE 7463+CO,CN
330 GOTO 270
```

—5—

Block SAVE and LOAD

Sheila Thornton

If you've ever used the VIC's data file functions to do tape saves and loads of machine language, hex tables, or graphics, you'll appreciate the speed, ease, and flexibility with which this program, "Dump/Recover," accomplishes those tasks. You'll also learn a bit about using BASIC's internal machine language routines.

This program is built around four of the Kernal routines, the self-contained machine language software modules in VIC's operating system which can be accessed through a group of JMP instructions located at the top of memory.

These routines — SETLFS, SETNAM, SAVE, and LOAD — are subroutines of the SAVE and LOAD functions in BASIC, but can be used individually to save any size memory block up to location 32766 ($7FFE) and to LOAD the SAVEd matter into its original position or a new one.

To discourage casual copying of their proprietary software, Commodore has inserted code in the SAVE routine which aborts attempted tape saves above 32766 ($7FFE hex). However, a VIC owner who boasts a 1540 disk drive has informed me that, curiously, this prohibition doesn't extend to disk SAVEs.

"Dump/Recover" (Program 1) combines 43 bytes of machine language and ten lines of BASIC to connect you to the Kernal routines and to allow specification of start and end address and name via an INPUT statement.

Understanding the Method

Program 2 is a commented disassembly of the machine language that Dump/Recover must POKE into memory. In the first four instructions, the logical file number, device, and secondary address are selected, and then the SETLFS routine which makes it all happen is called. The second four instructions specify the length of the file name and its location in memory, and then jump to SETNAM, which will expect to find the file name immediately above

the end of the array variables (as pointed to by zero-page locations 49 and 50) and the name length at address 0.

At this point, the SAVE or LOAD routines can be called, but the usual tape messages (other than the PRESS... instructions) will not be displayed. Some sleuthing inside VIC's operating system disclosed that SAVE and LOAD require that bit seven at address 157 ($9D) be set for the messages to be printed. The two instructions following the jump to SETNAM accomplish this.

While these messages are not required for a successful SAVE or LOAD, I find it comforting to see that VIC is indeed SAVING/SEARCHING FOR/LOADING the file I've specified. This feedback also serves as a check for typing errors and helps to spare VIC from doggedly searching through an entire cassette for, say, "OPCODE TABEL" while I've excused myself to make tea. Unfortunately, I wasn't able to find how to turn on the "?LOAD ERROR" message, so this is handled in BASIC.

After completing these preparatory routines, the program returns to BASIC, which checks whether a save or load has been chosen and jumps to the appropriate address. LOAD will look at addresses 251 and 252 ($FB, $FC) to find the start address, and SAVE will additionally use 253 and 254 for the end address.

Since Dump/Recover's purpose is to SAVE and LOAD any permitted section of memory, I decided that the safest place to put the machine language was in the BASIC input buffer (512 to 600 — $0200-$0258), making it necessary to re-POKE the instructions every time the program is run. While this doubles the permanent program length (to 487 bytes), it does add flexibility.

Returning to Program 1, you can see that Dump/Recover's first job is to accept the start and end addresses (in decimal) and the file name, so the input buffer can be freed up for the machine language. The end address entered for a save must be one higher than that of the last byte to be saved. For a load, a 0 must be entered as the end address.

Line 1003 places the name length in location 0 and turns the end-of-arrays pointer, plus the name length, into a decimal number. Because all of the program's variables must be set up before the latter step is taken, U is first set equal to 1. In line 1004, the program puts the file name above the BASIC variables, jumps to the SETLFS and SETNAM routines, POKEs the start address pointer, and tests whether a dump or recovery has been selected. If a dump, line 1005 places the end address in memory, jumps to the appropriate address, and ends the program.

Since a side effect of the LOAD routine is that the numeric and array variable pointers are set to the end address of the loaded material, line 1006 saves the pointers in the input buffer before LOAD is called and restores them afterward. Line 1007 checks the I/O STATUS word and prints a load error message if STATUS reports either an unrecoverable load error or any mismatch.

If the END statements in lines 1005 and 1008 are changed to RETURNs, Dump/Recover can be used as a subroutine; but don't forget that, while RUN restores the DATA pointer, GOSUB does not. I have fashioned short, unique versions of Dump/Recover to include in programs which need to load in binary data and to preface frequently used machine language tapes so they will load in without making BASIC forget where it's put its variables.

Material saved with Dump/Recover can be verified from BASIC using the format, VERIFY "FILENAME",1,1. BASIC will also load these tapes, but the adjustment made to the variable pointers may make it necessary to execute a NEW after the load. You'll often find it necessary to protect the loaded file from BASIC by lowering the string and end-of-memory pointers.

The Kernal routines are pretty thoroughly documented in the *Programmer's Reference Guide* (pp. 182-211), but I'd like to share with you some omissions and errors I discovered there while writing this program. First, the *Guide* neglects to say what the valid secondary addresses are for the SAVE function. I wasn't surprised to discover that they are the same as used in BASIC:

0 =Relocatable save
1 =Nonrelocatable save
2 =Relocatable save with end-of-tape marker
3 =Nonrelocatable save with E-O-T marker

The discussion of the SETLFS routine indicates that 255 ($FF) should be used if *no* secondary address is desired. While this may be true for other I/O operations, a 255 functions exactly like a 3 for a tape save. The *Guide* also gives incorrect secondary addresses for a load. In fact, a 0 will permit a relocating load, and a 1 will inescapably send the file back to its origin.

With just a few bytes of simple "straightline" code, even inexperienced machine language programmers can tap significant programming power and speed from the 36 Kernal routines.

—5—

Program 1. Dump/Recover

```
999 REM "DUMP/RECOVER" FOR VIC-20
1000 PRINT"START,END,NAME":INPUTV,W,V$:R=
     540:FORJ=1TO43:READT:POKER+J+5,T:NE
     XT:GOTO1003
1001 DATA169,1,162,1,160,0,32,186,255,165
     ,0,166,49,164,50,32,189,255,169,128
     ,133,157,96
1002 DATA169,0,166,251,164,252,32,213,255
     ,96,169,251,166,253,164,254,32,216,
     255,96
1003 T=LEN(V$):POKE0,T:U=1:S=256*PEEK(50)
     +PEEK(49)+T
1004 FORJ=1TOT:POKES-J,ASC(RIGHT$(V$,J)):
     NEXT:SYS546:U=V:T=252:GOSUB1009:IFW
     =0THEN1006
1005 U=W:T=254:GOSUB1009:SYS579:END
1006 FORJ=0TO5:POKER+J,PEEK(45+J):NEXT:SY
     S569:FORJ=0TO5:POKE45+J,PEEK(R+J):N
     EXT
1007 IFSTATUSAND48THENPRINT:PRINT"?LOAD":
     PRINT"ERROR";
1008 END
1009 POKET,INT(U/256):POKET-1,U-256*PEEK(
     T):RETURN
```

Program 2. Machine Language Subroutines for Dump/Recover

```
0222 A9  LDA  #01      ;SET FILE NO.
0224 A2  LDX  #01      ;SET DEVICE NO. (TAPE)
0226 A0  LDY  #00      ;SET SEC. ADDR.
                       ; (RELOCATABLE)
0228 20  JSR  FFBA     ;CALL SETLFS
022B A5  LDA  00       ;GE  NAME LENGTH
022D A6  LDX  31       ;GET NAME START ADDR. LO
022F A4  LDY  32       ;GET NAME START ADDR. HI
0231 20  JSR  FFBD     ;CALL SETNAM
0234 A9  LDA  #80      ;
0236 85  STA  9D       ;TURN ON TAPE MESSAGES
0238 60  RTS           ;
0239 A9  LDA  #00      ;SET LOAD FUNCTION
023B A6  LDX  FB       ;GET LOAD START PNT. LO
023D A4  LDY  FC       ;GET LOAD START PNT. HI
023F 20  JSR  FFD5     ;CALL LOAD
0242 60  RTS           ;
0243 A9  LDA  #FB      ;SET SAVE START PNT.
                       ;OFFSET
```

```
Ø245 A6   LDX   FD        ;GET SAVE END PNT. LO
Ø247 A4   LDY   FE        ;GET SAVE END PNT. HI
Ø249 2Ø   JSR   FFD8      ;CALL SAVE
Ø24C 6Ø   RTS             ;
```

—5—

Electric Eraser

Louis F. Sander

If you program in BASIC, you'd sometimes like to delete certain program lines after they've been executed, either to protect your program from piracy or to free up memory for the rest of the program to use. The lines that print your on-screen instructions, for example, are good candidates for deletion as soon as they've been run. Having served their purpose, they do nothing but take up space, which can really be at a premium in small-memory machines like the VIC.

It would be a real help if there were an easy way to delete such lines under program control. Well, now there *is* one: "Electric Eraser" is a two-line routine that deletes itself and all subsequent lines as soon as it's called.

Lines 210 and 220 in the accompanying program are the Electric Eraser. Line 300 activates the Eraser. There is nothing special about this choice of line numbers, and the three lines can be renumbered at will when you use them in other programs. They consume just over 100 bytes of memory.

To use the Eraser, you must set up the lines to be erased as the last lines in your program. There can be as many of them as you wish, and they should preferably include the activator line, since you'll have no need for it once the other lines have been erased. Put the Eraser immediately *before* the first line you want to erase. Then your program can execute any of its lines, except for the activator, to its heart's content.

There's no need to bypass the Eraser, since it has no meaningful effect until it's activated. When it's time for the Electric Eraser to do its work, execute the activator line. This will clear all variables and make the Eraser and everything after it disappear from the program. You can, if you like, replace the END in the Eraser with another statement, and it will be executed after it is deleted. If you leave out the END altogether, the subsequent lines may be executed, depending on what's in them, or your program may crash.

Watch It Work

Right now, let's see the Electric Eraser at work. Type in the demo

program and SAVE it. Don't RUN it first to check your work, or you'll have to type it in again! LIST the program and carefully check lines 210, 220, and 300 for errors. Now RUN the program, and see for yourself that all its lines are actually executed, which should be obvious from the text that prints on the screen. RUN the program again, and you'll see that lines 210 and up do not execute this time, and that you now have several hundred more bytes of free memory. LIST the program to verify that lines 210-300 are no longer there. They have been electrically erased. You could say that these lines were executed, then they were executed. Or maybe they were just RUN to death. Anyway, they are gone without a trace, replaced by usable memory.

Eraser's Secret

Here is where they went. The first two PEEKs in line 210 are the keys to Electric Eraser's success. These locations contain a pointer to the start of the line currently being executed. When activated, the Eraser POKEs zeros into the link for that line and, using the USR vector as a temporary storage area, sets the Start of Variables pointer to the location just above that. As a result, BASIC thinks the program ends with the last line before the Eraser, which of course it now does. If all this is over your head, the System Overview chapter of *The VIC-20 Programmer's Reference Guide* holds the keys to understanding. If you don't care about such matters, don't worry — you can use the Electric Eraser without understanding how and why it works.

Now you've seen the Electric Eraser in all its simple splendor, and maybe you've been impressed. If so, your next step is to add it to your bag of programming tricks, and to make equally impressive use of its powerful erasatorial punch. You could exercise your talents on the demo program, by replacing the END in line 220 with a RUN.

Electric Eraser

```
100 PRINT"{DOWN}FRE(0)=";FRE(0)
110 PRINT"{DOWN}WHERE IS THE REST OF THIS
    "
120 PRINT"LITTLE PROGRAM?"
200 REM ** ERASER FOR THE VIC:
210 A=PEEK(61)+256*PEEK(62)+3:POKE2,INT(A
    /256):POKE1,A-256*PEEK(2)
220 IFERTHENPOKEA-2,0:POKEA-1,0:POKE45,PE
    EK(1):POKE46,PEEK(2):CLR:END
```

```
230 PRINT"{DOWN}IF YOU LIST IT, YOU WON'T
    "
240 PRINT"FIND IT! IF YOU RUN IT ONCE"
250 PRINT"MORE,YOU'LL SEE THAT YOU"
260 PRINT"HAVE GAINED SOME MEMORY."
270 PRINT"{DOWN}THE ELECTRIC ERASER IS"
280 PRINT"POWERFUL MEDICINE!!"
300 ER=1:GOTO210:REM ** ACTIVATOR
```

—Chapter 6—

Joysticks

—6—

VIC Sticks

Jim Butterfield

Butterfield offers a simple program to read the joystick. By studying his example, you can learn to use joysticks in your program.

There is much to be gained by knowing all you can about the working of the joystick on your VIC and how it affects your computer. Try this. Hold the joystick over to the right, and while you're holding it, press the VIC number keys. You'll see that the odd numbers appear correctly on the screen, but the even digits are either missing or butchered. As soon as you release the joystick, the keyboard action returns to normal.

What's going on? In the interests of economy, Commodore has made one of the keyboard lines do "double duty." It tests part of the joystick and performs its normal keyboard-checking functions. This is a two-way interference. We've seen that the joystick can interfere with the keyboard; in addition, the keyboard-servicing routines can make it impossible to check part of the joystick. The routines which read the keyboard are a special type called "interrupt" programs; this makes them hard to control.

Once you know the question, the answer isn't hard. To check the joystick completely, we must shut off part of the keyboard. If we need to keep the whole keyboard live, we must turn it back on again after checking the joystick.

We may shut down part of the keyboard with POKE 37154,127 and restore it with POKE 37154,255. We need to do this only to check the Right position of the joystick, which is done by looking at (PEEK(37152) AND 128).

Solving the Collision

What are our options? First, if we have a program that doesn't need the joystick's Right position, we can ignore the whole question.

If we have a program that doesn't need the keyboard, we can start with POKE 37154,127 as our first statement and restore the full keyboard only when the program ends. It won't matter that the keyboard is partly disabled during the program run. If the user/player stops the program rather than allowing it to end nor-

mally, however, he'll find his keyboard is acting badly. This isn't serious: the RUN/STOP-RESTORE key combination will fix everything up.

If we want to keep the keyboard live during play, each check of the Right position must include the whole set of three: disconnect part of the keyboard, check Right, reconnect keyboard. It will cost us a little more running time, but it's neater. It's not perfect, however; some keys will tend to "hiccup" if held down.

Machine language programs can solve everything, of course. They won't have a speed problem, so the keyboard can be quickly disabled and reestablished. And the "hiccup" will go away if the interrupt is disabled during joystick checking; the interrupt routine won't jump in and be fooled during this check. Even in BASIC, however, you can do a competent job.

Difficult Diagonals

Joysticks are often inexact. You may think you are pushing Up, but you are slightly off true and the joystick might record both Up and Left.

The computer detects this, but your program must make a decision. If your program doesn't want diagonals, you must decide which of the two directions — say, Up and Left — is intended. It's easy to get the wrong one.

Directions are picked up as follows: UP is PEEK(37151) AND 4; DOWN is PEEK(37151) AND 8; LEFT is PEEK(37151) AND 16. The "fire" button is detected with PEEK(37151) AND 32, and RIGHT is checked as above, doing a partial keyboard disable and then working with PEEK(37152) AND 128. Each of these values becomes zero when the appropriate direction/button is activated.

You might write your program to check UP, then DOWN, then LEFT, etc., and to go to the appropriate action when you find an active position. If so, you'll miss the diagonals: UP/LEFT will exit on the UP condition and never check the LEFT, for example. This might be good for your particular game, but think of the human interface: the player might believe that he is pressing LEFT; the joystick is signalling LEFT and UP; and your program is reading only UP.

There's no absolute answer to this kind of question. Depending on the application, you can make certain choices. If you have on the screen a missile which is flying to the right, for example, you might choose to ignore all RIGHT/LEFT signals from the joystick and honor only UP/DOWN. Another approach is to design

your game so as to use diagonals.

It's possible to write programs which "debounce" the joystick — that is, it must be returned to the center or rest position before a new signal will be accepted from it. This gives the effect of an impulse type of stick — action takes place only when the stick is moved.

A Simple Joystick Algorithm

One of the annoying things about joystick testing is that the input is logically inverted: the appropriate input is zero when activated, rather than zero when off. Although the information is the same either way, our minds don't like it. It seems more sensible to us to have bits turned on when the joystick is pushed; this allows us to extract combinations of bits with the logical AND function. A simple conversion statement which allows this is:

```
X=(NOT PEEK(37151))AND 60-((PEEK(37152) AND 128)=0)
```

Don't forget to POKE 37154 with 127 before doing this test, or the Right position won't be detected properly; and remember to POKE 37154 back to 255 after the test.

After executing the above statement, variable X will contain complete information about the joystick. If nothing is active, X will be zero. If we want to check a change in the joystick status, we can see if the value of X has changed since last time.

We may now detect the various control positions with the appropriate AND statements:

Fire Button	**—X AND 32**
Left	**—X AND 16**
Down	**—X AND 8**
Up	**—X AND 4**
Right	**—X AND 1**

In each case, the result of the AND will be zero if this position is not active. Combinations can be used; for example, if we are interested in only UP and DOWN at this moment, we could check X AND 12.

When coding this, use parentheses liberally around the AND statements. For example, to test for Left, code: IF (X AND 16) <> 0 THEN. It won't work properly otherwise.

For motion, we can extract the Left/Right and Up/Down components with coding such as:

H = SGN(X AND 1)-SGN(X AND 16)
V = SGN(X AND 8)-SGN(X AND 4)

This produces values for H and V as follows: 0 for no motion in this direction; +1 or -1 for motion in the appropriate direction.

Putting It All Together

The following simple program gathers together the joystick techniques we have discussed. It's a simple sketching program.

A few comments on the coding. The fire button is used to change color on the screen; the program debounces the button (using variable B) so that holding down the button does not cause a continuous color change.

Lines 310 and 320 compute reverse values of V and H compared to the algorithms given previously. In this case, we're computing an inverse activity — how many places to back the cursor up for a given position.

Lines 330 to 350 are rather "gimmicky"; we force the cursor right and down, and then count our way back to the desired position using cursor-left and cursor-up characters. The intent here is to illustrate the use of the V and H directional values. You may find other ways to achieve the same objective when you write your own programs.

The program prints the "ball" character, CHR$(209); you can switch to another character by making the appropriate change in line 330.

The joystick can indeed be interfaced with your program; all you need is to learn a few rules. You must set your own objectives as to how the joystick best interfaces with the user in your application. Once you have learned the mechanics, it's not hard to make everything work.

If you wish to learn more about joysticks, I suggest you read David Malmberg's article "Using A Joystick" in *COMPUTE!'s First Book of VIC* and for a more formal discussion you might look in the *VIC-20 Programmer's Reference Guide.*

Joystick Sketching

```
100 REM JOYSTICK PROGRAM
110 PRINT CHR$(147);CHR$(142) :REM CLEAR
    SCREEN
120 DATA 5,28,30,31,144,156,158,159
130 DIM C(7) : REM COLOURS
140 FOR J=0 TO 7:READ C(J):NEXT J
```

```
150 S=1:PRINT CHR$(C(S));
200 REM TEST JOYSTICK
210 POKE 37154,127
220 X=(NOT PEEK(37151))AND 60-((PEEK(3715
    2)AND 128)=0)
230 POKE 37154,255 : REM RESTORE KEYBOARD
240 IF (X AND 32)=0 GOTO 300 : REM NO BUT
    TON
250 IF B>0 GOTO 200 : REM DEBOUNCE BUTTON
260 B=1:S=S+1:IF S>7 THEN S=0
270 PRINT CHR$(C(S)); : REM CHANGE COLOUR
280 GOTO 200
300 B=0
310 H=SGN(X AND 16) - SGN(X AND 1)
320 V=SGN(X AND 4) - SGN(X AND 8)
330 PRINT CHR$(209);CHR$(17);CHR$(17);CHR
    $(29);
340 FOR J=0 TO H+1:PRINT CHR$(157);:NEXT
    J
350 FOR J=0 TO V+1:PRINT CHR$(145);:NEXT
    J
360 GOTO 200
```

—6—

Joystick and Keyboard Routine

Michael Kleinert

For VICs without memory expansion, these machine language routines will help speed up BASIC considerably.

Reading a joystick in BASIC can be too slow for some games. My attempts to speed up David Malmberg's joystick routine ("Using A Joystick," *COMPUTE!'s First Book of VIC*) were unsuccessful, so I decided to write one in machine language for reading from the joystick. I designed the routine to be most suitable for game purposes, especially those in which you must guide an object around the screen by using the joystick.

Entering the Machine Language

Type in the BASIC loader provided in Program 1. For those who may not have a joystick or might like to use the keyboard, I have included an identical routine for the keyboard in Program 2.

Using the Routines

Both routines are very similar. Each checks for up, down, left, and right. Accounting for diagonal directions would require longer and more complex programming. The keyboard version will look for the depressing of four keys, which I have defined as I (up), M (down), J (left), and K (right).

I designed the routines for controlling the movement of an object on the screen, and I suggest the following format:

```
10 POKE A,B: SYS 7168: POKE A,32: A =A +PEEK(1)-PEEK(2):
GOTO 10
```

In the above line, A is the memory location of a character's position on the screen, and B is the character code of the desired character. First the character is POKEd onto the screen, and then the subroutine is called with SYS 7168. The subroutine checks for any movement of the joystick (or for keys being pressed). If it detects the joystick being pushed in any direction, it places an appropriate numerical value into location 1 or 2. These values will

be used to update the position of the character being moved. First, the old character must be erased. This is accomplished by the command POKE A,32. The character is erased by POKEing a space onto the same screen position (A). After it has been erased, its position can be updated by adding the contents of memory location 1 and subtracting the contents of memory location 2. Do this as shown above, with the command A =A +PEEK(1)-PEEK(2).

If the routine does not detect the joystick or keyboard being depressed, the values in these two memory locations will be set to zero, and the variable A (character's position) will remain the same.

Avoiding Leaving the Screen

If the joystick is pushed up (or the I key is pressed on the keyboard), the routine will place a value of 22 into memory location 2. This causes the number 22 to be subtracted from the current screen address contained in variable A, and is the basis for accomplishing upward movement of a character on the screen. Similarly, a character is moved right, left, and down in this fashion.

In order to keep the character from going off the top or the bottom of the screen, more complex programming is required. An appropriate method is illustrated in Program 3. The program is not a game, but simply a demonstration for the use of the routines. It will scatter several boxes, as obstacles, on the screen and will enable you only to move your "player" around the screen with the joystick or keyboard. It is the basic structure for a game.

If you are going to use the joystick, enter in lines 10 to 40 from Program 1. If you are using the keyboard, copy the lines from Program 2.

When you are ready to use one of the routines in your own BASIC program, do the following. Place lines 10 to 40 from Program 1 or lines 10 to 30 from Program 2 at the beginning of your program. Then, wherever you wish to utilize the routine in your program, give the command SYS 7168. To update the character's position, use the method which I described above.

Other Applications

There are many other uses for these routines. You may use them in simple delay loops to temporarily stop the program and wait until something is pressed.

To check for a desired direction on the joystick or a key on the

keyboard, use the values from Figures 1 and 2. For example, if you are using the keyboard subroutine and want the program to wait until the letter I is pressed on the keyboard, you PEEK location 2 as follows:

```
100 SYS 7168: IF PEEK(2)<>22 THEN 100
```

This will call the subroutine, and the program will not proceed until the value in location 2 is equal to 22.

If you are using the joystick and want to wait until it is pushed to the right, you follow the same basic format: PEEK memory location 1 for a value of one. For example:

```
100 SYS 7168: IF PEEK(1)<>1 THEN 100
```

The Firing Button

A "firing" button is not accounted for in either of the two routines, since it would require a line of BASIC. If you would like to check for the firing button, you would place the following step into your program:

```
200 IF PEEK(37137) > 69 THEN GOSUB (Line number)
```

After the GOSUB, you would place the line number to which you wish to send the program if it finds the firing button depressed.

If you wish to check for a "firing" button on the keyboard, you may use the following line, which checks for any depressing of the SPACE BAR (the one I usually use).

```
200 IF PEEK(197) =32 THEN GOSUB (LINE#)
```

The Demo Program

Briefly, here's a description of the function of each line in the demonstration, Program 3.

5 Limits the end of BASIC to protect the machine language routine, clears variables, and sets A equal to 7800 (the character's memory location on the screen).

10 READs the machine code from the DATA statements and POKEs the values into memory, starting at 7168.

20-40 Contain the machine code for the routine in DATA statements.

50 Clears the screen and then POKEs the color red onto each screen location.

60 Puts obstacles on the screen in 25 random screen locations.

70 POKEs the character onto the screen, calls the subroutine, and then sets B equal to the updated address.

80 If the new address is found to be off the screen, or if it is occupied by a box, the character remains stationary and the program goes back to line 70.

90 The new screen position has been accepted, so the old character is erased. The program goes back to line 70 to go through the same process.

Both routines can be used on a VIC with any amount of memory and can be placed anywhere in the user's RAM. In order to keep things relatively simple, I wrote the demonstration program for a 3.5K VIC; it will not work on a VIC with any memory expansion. These routines help speed up programs a great deal.

Program 1. Joystick Reader

```
10 FORM=0TO65:READN:POKE7168+M,N:NEXT
20 DATA169,128,141,19,145,169,0,133,1,13
   3,2,169,127,141,34,145,162,119,236,32
   ,145
30 DATA208,4,169,1,133,1,169,255,141,34,
   145,162,118,236,17,145,208,4,169,22,1
   33,1
40 DATA162,110,236,17,145,208,4,169,1,13
   3,2,162,122,236,17,145,208,4,169,22,1
   33,2,96
```

Program 2. Keyboard Reader

```
10 FORA=0TO40:READB:POKE7168+A,B:NEXT
20 DATA169,0,133,1,133,2,165,197,201,12,
   208,4,162,22,134,2,201,36,208,4,162,2
   2,134,1
30 DATA201,44,208,4,162,1,134,1,201,20,2
   08,4,162,1,134,2,96
```

Program 3. Joystick Demonstration

```
5 POKE56,28:POKE52,28:CLR:A=7800
10 FORM=0TO65:READN:POKE7168+M,N:NEXT
20 DATA169,128,141,19,145,169,0,133,1,13
   3,2,169,127,141,34,145,162,119,236,32
   ,145
30 DATA208,4,169,1,133,1,169,255,141,34,
   145,162,118,236,17,145,208,4,169,22,1
   33,1
```

```
40 DATA162,110,236,17,145,208,4,169,1,13
   3,2,162,122,236,17,145,208,4,169,22,1
   33,2,96
50 PRINT"{CLR}":FORX=38400TO38905:POKEX,
   2:NEXT
60 FORX=1TO25:Y=INT(RND(1)*500)+1:POKEY+
   7680,160:NEXT
70 POKEA,42:SYS7168:B=A+PEEK(1)-PEEK(2)
80 IFB>8185ORB<7680ORPEEK(B)=160THEN70
90 POKEA,32:A=B:GOTO70
```

Figure 1. Joystick Location

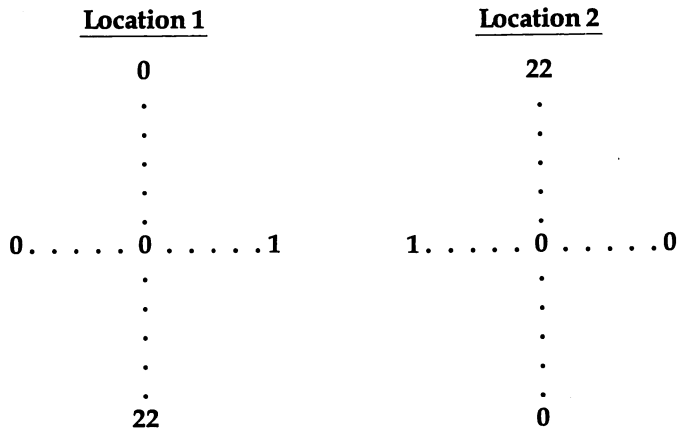
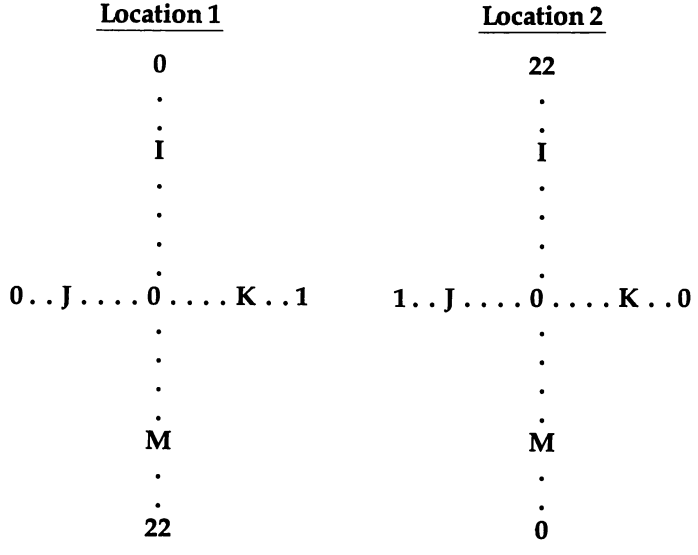


Figure 2. Keyboard Location



—6—

Using Atari Joysticks with Your VIC

Christopher J. Flynn

This discussion explains how the VIC reads the joystick port. Also included is a game called "Doodle."

What is the most inexpensive peripheral that you can buy for your VIC? A color television? Certainly not. Memory expansion? Probably not. No, a joystick. What? You mean one of those gadgets for playing games? That's right!

Perhaps you didn't realize it, but your VIC can use the very same joysticks that are found on the Atari and Sears video games. Absolutely no hardware modifications are needed at all.

To give you an idea of the capabilities of the joystick, we've included a demonstration program called "Doodle." It's a fast-paced game in full sound and color designed for drawing patterns with the joystick. Your kids will love it — if they can get it away from you.

Before we get into the details, an acknowledgment is due. Creative Software of California deserves credit for pointing out to me that Atari joysticks are usable on the VIC.

How We Do It

The figure compares the VIC joystick socket with the Atari's. The similarities are striking.

We need to do a little exploratory surgery first. Since I've already done this, please just follow my description. You don't need to do this to your VIC. First we gently open up VIC's case. Armed with our trusty ohmmeter, we trace the joystick connections. We assume that they must reach the 6522 VIA I/O chips. So that's where we start looking. Voilà! Tracing all the connections, we find that the joystick switches do indeed go to the 6522s.

Finally, we determine that the joystick is connected as follows:

6522#1	E	?	?	?	?	?	?	?
	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
6522#2	?	?	F	W	S	N	?	?
	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

E, W, S, and N represent the four compass directions. F represents the fire button. We won't be concerned with the fire button in this article.

How do we use this information in a program? What we generally have to do is read each I/O port and test the appropriate bits. Then our program can take any action needed. And there are some complications. Don't forget that the 6522 has data direction registers which program each bit for an input or output operation. Also, the signals from the joystick are in what is called an "active low" state. That is, if the joystick is pointing, say, north, the north bit will be low or zero. The other three directions will be high or ones.

That probably sounds a lot harder than it is. We can actually use BASIC to obtain the joystick readings pretty easily. The BASIC statements shown here are the key to using joysticks on the VIC.

```
POKE 37154,127
V1 =PEEK(37152) AND 128
V2 =PEEK(37151) AND 28
POKE 37154,255
JS =V1/16 + V2/4
JS =(NOT JS) AND 15
```

These statements read the I/O ports and manipulate the bits. We end up with a bit configuration like this:

O O O O E W S N

The least significant four bits in the variable JS thus correspond to the four joystick switches. Normally, this would mean that JS could range in value from 0 to 15. In practice, JS will take on values from 0 to 10. This is because some bit patterns just aren't possible. With a properly functioning joystick, you can't press the north and south switches at the same time, for example.

The following table shows the values that JS will assume for each of the valid joystick positions.

Direction	JS Value	Delta X	Delta Y
Neutral	0	0	0
N	1	0	-1
S	2	0	1
Can't occur	3	0	0
W	4	-1	0
NW	5	-1	-1
SW	6	-1	1
Can't occur	7	0	0
E	8	1	0
NW	9	1	-1
SE	10	1	1

Note that JS is 0 in the neutral position. This gives us a handy way to test for joystick movement.

Delta X and Delta Y are variables which will help us if we're trying to move an object around the screen. Suppose we're using an X and Y coordinate system like this:

```
          X
     0 1 2 3 4 5 ... 21
     1
     2
Y    3
     4
     .
     .
     .
    22
```

Y represents a row number, and X represents a position within a row. When the joystick moves, we want to update the values of X and Y so they indicate the new position. We can do this again easily in BASIC:

```
X=X+DX(JS)
Y=Y+DY(JS)
```

DX and DY are arrays where we've saved the list of values for Delta X and Delta Y.

An example will show how this works. Let us assume that we have an object at X=7 and Y=5. We test the joystick and determine that it has moved. Let's assume that it's pointing north. From our table, we know that JS will contain 1. So, the new positions of X and Y will be:

```
X=7+DX(1)
Y=5+DY(1) or
X=7
Y=4
```

Thus, our object is moved up one line closer to the top of the screen. There was no left or right horizontal change.

One last detail we need to think about is how to convert X and Y into something VIC understands. As you know, we can POKE things into VIC's screen memory. But we need a memory location for that. Again, BASIC helps us out:

```
P=22*Y+X
```

That little formula will convert valid X and Y values into a number ranging from 0 to 505. Next, we must add P to the screen and color memory starting locations:

```
POKE 7680+P, code
POKE 38400+P, color
```

Use any screen code and color that you wish.

Doodling

We've covered joysticks pretty quickly; we've only discussed the highlights. There are many other details involved. The best way to pick these up is to study Program 1 and to enjoy the Doodle game.

Doodle is a lot of fun to play. The object is just to enjoy yourself. When you start Doodle, it will display instructions on how to use the special function keys.

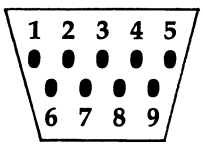
Key	Message	Description
f1	TO QUIT	Ends the game.
f3	MOVE CURSOR	The cursor moves, but does not draw a line. Erases any objects that it crosses.
f5	DRAW LINE	The cursor moves and draws a line.
f7	CLEAR SCREEN	The screen is cleared and the cursor is centered. VIC is ready to doodle again.

You may press any key at any time while doodling. For interesting effects, alternate the f3 and f5 keys. By doing this properly, you can enclose a figure within another figure without any intersecting lines.

—6—

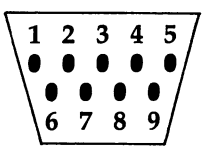
Comparison of VIC and Atari Joystick Sockets (as viewed from the outside)

VIC Joystick Socket



1 JOY 0
2 JOY 1
3 JOY 2
4 JOY 3
5 POT Y
6 LIGHT PEN
7 +5V
8 GROUND
9 POT X

Atari Joystick Socket



1 FORWARD
2 BACK
3 LEFT
4 RIGHT
5 POT B
6 FIRE BUTTON
7 +5V
8 GROUND
9 POT A

Doodle

```
100 REM INITIALIZE
110 GOSUB 30000
120 REM DOODLE
130 GOSUB 2000
140 IF F1=0 THEN 130
150 REM END
160 GOSUB 34000
170 END
300 REM READ JOYSTICK AND KEYBOARD
310 POKE DD,127
320 V1=PEEK(R1)AND128
330 V2=PEEK(R2)AND28
340 POKEDD,255
350 JS=V1/16+V2/4
360 JS=(NOT JS)AND15
```

```
370 GET A$:IF A$<>"" THEN GOSUB 400
380 RETURN
400 REM SERVICE KEYBOARD
410 A=ASC(A$)
420 IF A=133 THEN F1=1
430 IF A=134 THEN CH=32
440 IF A=135 THEN CH=32+128
450 IF A=136 THEN GOSUB 32000
460 RETURN
500 REM CHOOSE COLOR
510 CL=INT(RND(1)*8)
520 IF CL=1 THEN 510
530 RETURN
600 REM VERIFY X&Y
610 IF X<0 THEN X=0
620 IF X>21 THEN X=21
630 IF Y<0 THEN Y=0
640 IF Y>22 THEN Y=22
650 RETURN
700 REM X & Y TO ADDR
710 P=22*Y+X
720 POKE VA+P,BT
730 POKE CA+P,CL
740 RETURN
800 REM SET NOISE AND VOLUME
810 POKE VL,(3+INT(RND(1)*6))
820 POKE S3,(128+INT(RND(1)*110))
830 RETURN
2000 REM DOODLE
2010 TL=TI+60*2.5
2020 GOSUB 300:REM POLL JOYSTICK
2030 IF F1 THEN RETURN
2040 IF JS<>0 THEN 2070
2050 IF TI<TL THEN 2020
2060 GOSUB 800:GOTO 2010
2070 POKE VL,15:POKE S3,220
2080 FOR Z=1 TO 100:NEXT
2090 POKE S3,0:GOSUB 800
2100 REM CLEAR OR FILL CURSOR SPOT
2110 BT=CH
2120 GOSUB 500:REM GET COLOR
2130 GOSUB 700:REM STORE BT
2140 REM NEW CURSOR POSITION
2150 X=X+DX(JS)
2160 Y=Y+DY(JS)
2170 GOSUB 600:REM CHECK X & Y
2180 REM SET CURSOR
2190 BT=CS:CL=0
2200 GOSUB 700:REM STORE BT
```

```
2210 RETURN
30000 REM .BEGIN
30010 PRINT CHR$(147);
30020 PRINT SPC(8);"VIC-20"
30030 PRINT
30040 PRINT SPC(5);"D O O D L E"
30050 PRINT:PRINT
30060 PRINT "PRESS:":PRINT
30070 PRINT "F1- TO QUIT"
30080 PRINT "F3- MOVE CURSOR"
30090 PRINT "F5- DRAW LINE"
30100 PRINT "F7- CLEAR SCREEN"
30110 PRINT:PRINT
30120 PRINT" JOYSTICK PLUGGED IN?"
30130 REM VARIABLES
30140 REM JOYSTICK
30150 DD=37154:R1=37152:R2=37151
30160 REM VIDEO AND SOUND
30170 VA=7680:CA=38400:BG=36879
30180 VL=36878:S3=36876
30190 CS=90:CH=32+128:Z=RND(-TI)
30200 REM DELTA X, DELTA Y FOR JOYSTICK
30210 DIM DX(10),DY(10)
30220 FOR I=0 TO 10:READ DX(I):NEXT
30230 FOR I=0 TO 10:READ DY(I):NEXT
30240 DATA 0,0,0,0,-1,-1,-1,0,1,1,1
30250 DATA 0,-1,1,0,0,-1,1,0,0,-1,1
30260 FOR Z=1 TO 4000:NEXT
30270 REM INITIAL CONFIGURATION
30280 POKE BG,25:REM SET BACKGROUND/BORD
      ER
30290 GOSUB 800:REM GET NOISE
30300 GOSUB 32000:REM CLEAR
30310 RETURN
32000 REM CLEAR SCREEN
32010 PRINT CHR$(147);
32020 X=10:Y=10:BT=CS:CL=0
32030 GOSUB 700
32?040 RETURN
34000 REM.END
34010 PRINT CHR$(147);
34020 POKE BG,27
34030 PRINT:PRINT
34040 PRINT "SO LONG!"
34050 PRINT:PRINT
34060 POKE VL,0:POKE S3,0
34070 RETURN
```

—6—

FIGHTER ACES — Add a Second Joystick

John Parr

This game, "Fighter Aces," is fun in its own right. But it also shows a simple way to add a second joystick to your VIC for two-player games.

I spend many hours in front of the CRT on my VIC, attempting one program or another, but when the work is done, I am not ashamed to play a game or two for relaxation. Many of the games that I like, however, require two joysticks.

Other programmers have circumvented this problem through the use of keys, but I find the use of keys awkward. Besides, most games use the same keys over and over, which I am sure must be wearing on my precious investment. The only answer to my dilemma, therefore, was to find some way of connecting a second joystick.

Before I went to work, I decided that I'd better find out a little bit about how the joysticks worked. As it turns out, the joystick is just a lever connected to four microswitches at its base. When the stick is pressed in one direction, the lever closes the appropriate switch, grounding one of the pins on the games port. For diagonals, two switches are closed simultaneously, grounding two pins in the games port. When a pin is grounded, one bit is turned off in either memory location 37137 or in location 37152.

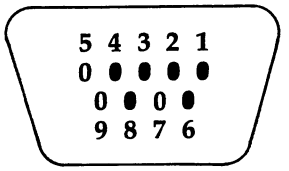
From this understanding, I decided that the best place to hook a second joystick on was through the parallel user port. (As it turns out, PET users have been doing this for years.) After a little checking of my memory map, I decided to connect my second joystick on pins D through J, grounding to pin A. These pins are easily read through memory location 37136.

My next chore was to determine the most logical order in which to make my connections. I finally decided on a system by which any formulas for the first joystick could be used by the sec-

ond. The following hookup is the result of my research.

Looking at the plug on the joystick, you will see this (minus the numbers, of course):

Figure 1. VIC Joystick Plug



The filled-in holes represent pins which are used. You will notice that this is a mirror image of the diagram which is in your VIC book.

The following chart tells what each pin does:

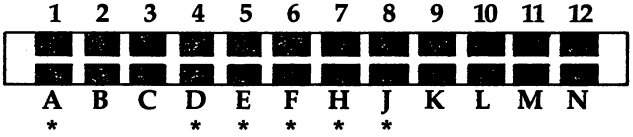
Pin number	Description	
1	Up	—Joy 0
2	Down	—Joy 1
3	Left	—Joy 2
4	Right	—Joy 3
6	Fire Button	
8	Ground	

Simply connect these pins to a 24-pin edge connector as follows:

Joystick		Edge Connector
1	to	E
2	to	F
3	to	H
4	to	D
6	to	J
8	to	A

The 24-pin edge connector then plugs into the User I/O Port on the back of the VIC, which has the configuration shown in Figure 2.

Figure 2. VIC User I/O Port



Pins to which connections are made are marked with an asterisk (*).

These connections can be made either by replacing the existing joystick plug with the edge connector or by using a "patch" cord. A patch cord is an extension cord with one type of plug on one end (such as our edge connector) and with a different type of plug on the other end (such as a connector like the one which is mounted on the side of your computer for the games port). I personally prefer the patch cord method, because then the joysticks can be interchanged if one wears more than the other. Also, if a joystick breaks for some reason, there need be no changes made to the replacement.

From the arrangement I have chosen, all formulas used on one joystick can be used on the other with little modification. Personally, I find the new joystick easier to use because all switches can be read from the same memory location with one simple PEEK. I like it so much, in fact, that it has become my main joystick.

Fighter Aces

Now that I have shown you how to connect this joystick, I will show you how to use it with one of my favorite games, "Fighter Aces." In this game, two players engage in a dogfight across your screen. The game ends at fifteen points; may the best pilot win!

As it is written, this program will run on any memory configuration the VIC can attain.

Before continuing, I must explain the use of the decimal points. The decimal point is a constant for the number zero. The only difference between the use of the decimal point and the use of a zero is that decimal points will speed program execution. If you feel ambitious, try replacing the decimals with zeros to see what I mean.

At last, we have arrived at our program description. The code follows a fairly simple algorithm, so with the explanation, you should be able to understand its workings.

Lines

10-50 Set the program to run with any memory by changing the locations of the screen and color. Also, these lines move the variable storage above the user-defined characters if your computer is expanded by 8K or more; if not, the program sets the end of memory below the special characters, thus protecting them for any memory configuration.

60-150 Set up the variables and the screen before the game begins.

160-170 Get values for each joystick.

180-220 Check for a fire button; see if a shot has already been fired. Each shot is checked here to see if it has gone to the end of its limited range. Note: By eliminating line 180 and the NEXT on line 290, the biplanes will be more responsive, but the shots will be slower. Conversely, if the value of the loop is upped, the shots will move faster, but the planes will be harder to control.

230-280 Move the shots checking for out of bounds, out of range, and a hit.

290-340 Set new direction on each biplane and determine which type of biplane is to be POKEd.

350-400 Move each biplane, checking for out of bounds and crashes.

410-440 Subroutine to determine what a shot hit. (Control tower, another shot, or a biplane.)

450-540 Subroutine for an explosion. Also checks for a midair collision and updates the score. If either score equals fifteen, the ending flag(s) are set.

550-650 Game over routine.

660-790 Create the biplanes and print the title page.

Important Variables:

S The first sound channel.

V% The starting address of the video display.

C The difference between the screen and color locations.

P%( ) Position of each plane on the screen.

SP%( ) Position on the screen of each shot.

SD%( ) Direction of each shot.

SF%( ) Flag to show whether a shot is on the screen and, if it is, how far it has to travel.

D%( ) Direction of each plane.

A%( ) The attitude of each plane.

SC%( ) The score for each player.

E%( ) Flag to show if someone has fifteen points.

G%( ) The number of games that each player has won.

M%( ) Value from each joystick.

L% Flag for the biplane being out of screen limits.

Fighter Aces

```
10 IFFRE(0)>7000THENPOKE46,32:GOTO30
20 POKE56,29
30 CLR:S=36874:POKE4+S,5:POKE36879,25
40 V%=4*(PEEK(36866)AND128)+64*(PEEK(368
   69)AND120):C=37888+4*(PEEK(36866)AND1
   28)-V%
50 GOTO660
60 DIMP%(1),SP%(1),SD%(1),SF%(1),D%(1),A
   %(1),SC%(1),E%(1),G%(1)
70 DEFFNM(X)=((XAND4)=.)*22+((XAND16)=.)
   -((XAND2)=.)-((XAND8)=.)*22
80 GOTO120
90 P%(.)=V%+463:A%(.)=.:D%(.)=1:RETURN
100 P%(1)=V%+482:A%(1)=4:D%(1)=-1:RETURN
110 PRINT"{HOME}{CYN}{RVS}SCORE:":PRINTT
    AB(5)"{RVS}{BLK}"SC%(.)TAB(14)"{WHT}
    "SC%(1):RETURN
120 PRINT"{CLR}{GRN}{2 DOWN}{RVS}*******
    **************":FORX=1TO18:PRINT:NE
    XT:PRINT"{RVS}{CYN} {10 T}";
130 PRINT"{UP}B{UP}{LEFT}B{UP}{LEFT}B
    {UP}{LEFT}B{4 DOWN}{LEFT}{10 T}
    {HOME}"
140 GOSUB90:GOSUB100:GOSUB110
150 POKES+3,200:POKES,200
160 POKE37154,127:X=PEEK(37152):POKE3715
    4,255:M%(1)=2*(X=119)+PEEK(37137)
170 M%(.)=PEEK(37136)-129
180 FORY=1TO2
190 FORX=.TO1:IFM%(X)AND32THENNEXT:GOTO2
    30
200 IFSF%(X)THENNEXT:GOTO230
210 SF%(X)=11:SP%(X)=P%(X)+D%(X):SD%(X)=
    D%(X)
220 IFSP%(X)>V%+483ORSP%(X)<V%+66ORPEEK(
    SP%(X))=194THENSF%(X)=.:NEXT:GOTO230
230 FORX=.TO1:IFSF%(X)=.THENNEXT:GOTO290
```

```
240 SF%(X)=SF%(X)-1:IFSF%(X)=.THENPOKESP
    %(X),32:NEXT:GOTO290
250 POKESP%(X),32:SP%(X)=SP%(X)+SD%(X)
260 IFSP%(X)<V%+66ORSP%(X)>V%+483THENSF%
    (X)=.:NEXT:GOTO290
270 IFPEEK(SP%(X))<>32THENSF%(X)=.:GOTO4
    10:NEXT:GOTO290
280 POKESP%(X)+C,X:POKESP%(X),41:NEXT
290 NEXT:FORX=.TO1:IF(M%(X)AND30)=30THEN
    350
300 D%=FNM(M%(X)):IFD%=D%(X)THEN350
310 D%(X)=D%:A=(D%/11):IFA>2THENA=A+1
320 IFA<-2THENA=A-1
330 IFA<.THENA=A-4
340 A%(X)=ABS(A)
350 IFP%(X)+D%(X)<V%+66ORP%(X)+D%(X)>V%+
    483THENC%=X:L%=1:GOSUB450
360 IFPEEK(P%(X)+D%(X))<>32THENC%=X:GOSU
    B450
370 IFE%(.)ORE%(1)THEN550
380 POKEP%(X),32:P%(X)=P%(X)+D%(X)
390 POKEP%(X)+C,X:POKEP%(X),A%(X)+33
400 NEXT:GOTO160
410 IFPEEK(SP%(X))=194THEN290
420 IFPEEK(SP%(X))=41THENPOKESP%(X),32:S
    F%(.)=.:SF%(1)=.:GOTO290
430 C%=1-X:GOSUB450:IFE%(X)THEN550
440 GOTO290
450 POKEP%(C%),42:POKES+4,15:FORI=1TO70:
    NEXT:POKES+4,5:POKEP%(C%),32
460 H%=PEEK(P%(C%)+D%(C%))
470 IFH%=41THENSF%(1-C%)=.:POKEP%(C%)+D%
    (C%),32:H%=32
480 IFH%<>32ANDH%<>194THENB%=1
490 SC%(1-C%)=SC%(1-C%)+1
500 IFSC%(1-C%)=15THENE%(1-C%)=1
510 ONC%+1GOSUB90,100
520 IFL%THENL%=.:B%=.:GOTO540
530 IFB%THENB%=.:C%=1-C%:GOTO450
540 GOSUB110:RETURN
550 POKES+4,0
560 IFE%(.)ANDE%(1)THENPRINT"{CLR}{RVS}T
    IE GAME !!":GOTO600
570 W%=-(E%(.)=1)-2*(E%(1)=1)
580 PRINT"{CLR}{RVS}PLAYER"W%" WINS."
590 G%(W%-1)=G%(W%-1)+1
```

```
600 PRINT"{2 DOWN}{CYN}{RVS}* CURRENT
    {2 SPACES}STANDINGS *":FORX=.TO1:PRI
    NT"{DOWN}{YEL}{RVS}PLAYER"X+1" -"G%(
    X):NEXT
610 PRINT:PRINT"{BLK}{RVS}PLAY AGAIN?"
620 GETA$:IFA$=""THEN620
630 IFA$<>"N"THENSC%(.)=.:SC%(1)=.:E%(.)
    =.:E%(1)=.:POKES+4,5:GOTO120
640 PRINT"{CLR}{BLU}"
650 POKE36869,240+48*(V%=4096):FORX=.TO4
    :POKES+X,0:NEXT:POKE36879,27:END
660 PRINT"{CLR}{BLU}{DOWN}* * FIGHTER
    {2 SPACES}ACES! * *"
670 FORX=1TO5:PRINT:NEXT
680 PRINTTAB(7)"{BLK}ANOTHER":PRINT:PRIN
    TTAB(9)"JHP":PRINTTAB(9)"VIC":PRINT
690 PRINTTAB(7)"PROGRAM"
700 FORX=.TO10:READY:FORZ=.TO7:READA:POK
    EZ+Y,A:NEXT:NEXT:
710 DATA7464,0,56,145,187,255,187,145,56
    ,7440,4,22,39,88,58,180,72,32
720 DATA7448,60,24,0,90,126,90,0,60,7456
    ,32,104,228,26,92,45,18,4
730 DATA7432,0,28,137,221,255,221,137,28
    ,7472,4,18,45,92,26,228,104,32
740 DATA7480,60,0,90,126,90,0,24,60,7488
    ,32,72,180,58,88,39,22,4,7496,0,0,0,
    24,24,0,0,
750 DATA7504,153,90,60,255,255,60,90,153
    ,7424,0,0,0,0,0,0,0,0
760 FORX=1TO6:PRINT:NEXT
770 PRINT"{GRN}PRESS RETURN TO BEGIN
    {HOME}"
780 GETA$:IFA$<>CHR$(13)THEN780
790 PRINT"{CLR}":POKE36869,255+48*(V%=40
    96):POKE36879,110:GOTO60
```

—Chapter 7—

Machine Language

—7—

HEXEDIT
A BASIC Hex Editor

Bill Yee

"Hexedit" lets you handle hexadecimal-decimal conversions and create and save machine language on your VIC. It works on any VIC, expanded or not.

Are you tired of POKEs, PEEKs, and constant conversion from hexadecimal to decimal and back again? Here's an editor that allows you to roam around memory entirely in hexadecimal. In addition, by changing a BASIC pointer, binary data or machine language entered into RAM via the editor can be saved and loaded on cassette. The normal cassette commands SAVE, LOAD, and VERIFY are used.

The editor is written in BASIC in order to avoid the chicken and egg problem (only an unexpanded VIC is required to create "Hexedit" but it will work without modification on any VIC). Because the unexpanded VIC has only 3500-odd bytes of RAM available, Hexedit contains no REM statements, and GOSUBs are used extensively. Hexadecimal to decimal conversion is done by a subroutine at line 11, and decimal to hexadecimal conversion is done by a subroutine at line 14. This allows you to do conversions outside of Hexedit via direct BASIC statements. For example, entering H$ ="ABCD":GOSUB11:?D displays 43981, and entering D =43981:GOSUB14:?H$ displays ABCD.

Hexedit occupies 615 bytes of memory, and on an unexpanded VIC the end of the program would be at 4713 ($1269). A PEEK of the BASIC pointer for the "end of BASIC program/start of BASIC variable area" at locations 45 and 46 ($2D and $2E) should show 106 and 18 ($6A and $12) after you have created Hexedit.
(104)

Modifying Memory

If you plan to use Hexedit just to look at VIC memory, there is nothing more to do. If you want to create and save data, you need to reserve some space in the RAM following Hexedit. This is done by modifying the BASIC pointer at locations 45 and 46 so that the

"end of BASIC program/start of BASIC variable area" is much higher than it is for Hexedit proper. If you do a POKE45,0 and a POKE46,28 followed by a CLR (to clean up the other BASIC pointers), you would now have the memory space from 4714 to 7167 ($126A to $1BFF) at your disposal.

However, once you have changed the pointer, do not add or delete any BASIC statement. If you do, the BASIC line editor in the VIC will move data around in memory up to the "end of BASIC program" location defined by the pointer as well as relink the data to form linked BASIC statements.

Hexedit is executed with a RUN command. You are prompted for a starting memory location by Hexedit. The address is taken to be hexadecimal if prefixed by $. Otherwise, it is seen as decimal. After the location prompt, all output and input is taken to be hexadecimal. Hexedit displays the current location address followed by the contents. Keying the up CRSR causes a byte walk towards lower memory. The down CRSR is used to byte walk towards higher memory. Depressing the space bar with no other input redisplays the current location. This is useful for looking at the VIC VIA timers or input ports. If the value in the location has not changed, hitting the space bar will appear to have no effect.

A RETURN causes a prompt for a new starting location. If you respond to the prompt with just another RETURN, Hexedit will END.

Data can be entered into memory at the current location whose address and contents are displayed by Hexedit. The digits 0-9 and A-F are accepted for input. Only the last two digits entered are written into memory. So if you make a mistake, just keep on entering digits until it is right. After digit input, write of memory occurs on either up or down CRSR, space bar, or RETURN.

The location pointer is modified after a successful write of memory, as described previously, except for the space bar. In this case, with data entered, the space bar causes the current location to increment. I found this method of data entry with the space bar to be the fastest way. If the current location cannot be written by the data specified, the response will in all cases be the message "R/O" followed by redisplay of the same location.

Once you have finished entering data into reserved memory, you can exit Hexedit by hitting RETURN twice. The VIC cassette commands can then be used to SAVE the new data (along with

Hexedit) to tape. A subsequent LOAD will retrieve the data as well as Hexedit from tape.

I have used Hexedit for entering up to 2K bytes of machine language. For example, it is a way to create "TINYMON1" directly on the VIC rather than doing it via a PET. For those with limited resources, Hexedit provides a way for doing more with what you already have at no further cost.

Hexedit

```
1 GOSUB8:L=D:C=99
2 GOSUB7:L=D
3 GOSUB13:L$=H$:D=PEEK(L):GOSUB14:PRINTL$
   ": "H$" ";:GOSUB18
4 IF H$=""ANDC=32THENPRINT"{UP}":GOTO3
5 IFH$<>""THENH$=RIGHT$(H$,2):GOSUB11:POK
   EL,D:IFPEEK(L)<>DTHENPRINT" R/O":GOTO3
6 GOTO2
7 PRINT:IFC<>13THEND=L+SGN(99-C):RETURN
8 H$="":INPUT"LOC";H$:IFH$=""THENEND
9 IFLEFT$(H$,1)<>"$"THEND=VAL(H$):RETURN
10 H$=MID$(H$,2,LEN(H$)-1):H$=RIGHT$(H$,4
    )
11 N=LEN(H$):D=0:FORM=0TON-1:C$=MID$(H$,N
    -M,1):H=ASC(C$)-48:IFH>9THENH=H-7
12 D=D+H*16↑M:NEXT:RETURN
13 D=L
14 IFD<0ORD>65535THENPRINTD"OOR":END
15 H$="":M=4096:N=3:IFD<256THENM=16:N=1
16 FORH=0TON:C=INT(D/M):D=D-C*M:M=M/16:C=
    C+48:IFC>57THENC=C+7
17 H$=H$+CHR$(C):NEXT:RETURN
18 H$=""
19 GETC$:IFC$=""THEN19
20 C=ASC(C$):IFC=13ORC=17ORC=32ORC=145THE
    NRETURN
21 IFC<48OR(C>57ANDC<65)ORC>70THEN19
22 PRINTC$;:H$=H$+C$:GOTO19
```

—7—

A Tracing Disassembler

Peter Busby

Here is a handy tool to let you look at machine language programs.

Computers are very good at dealing with numbers. In fact, at the most basic level, binary numbers are all the computer can understand. Humans, on the other hand, tend to find numbers confusing and prefer to deal with words. This is why programmers who work with machine language often use assemblers, such as the one presented in the next article. But what if you've got a program in machine language that you want to decipher? It's only logical that there should also be programs to help you work in the opposite direction. Such programs are called *disassemblers.*

What is a disassembler? It is a program which looks at the machine code in memory (RAM or ROM) and displays the equivalent hexadecimal or decimal values. More importantly, it translates these numbers into the *mnemonics* for the 6502 microprocessor instructions. (Mnemonics are abbreviated words which represent machine language program instructions. For instance, RTS is the mnemonic for the "return from subroutine" instruction.)

Some disassemblers allow you only to inspect memory locations sequentially, in the order of their memory location addresses. This, however, is a *tracing* disassembler. This means that if you encounter, for example, a JMP (jump) instruction during disassembly, you may either disassemble the next instruction in memory or make the JUMP to the new location and continue disassembly from there, in the order in which the program is executed. The same holds for JSR (jump to subroutine) and all the branch instructions as well.

Using the Program

"Tracing Disassembler" is written in BASIC. Type in the program carefully, then SAVE a copy. When you RUN the program, you will first be asked for a starting address. This can be in either deci-

mal or hexadecimal. If you enter fewer than five decimal digits or fewer than four hex digits, you'll need to follow the address with a RETURN. The computer will then display the following menu:

Advance one step
Branch/go subroutine
Convert bases
Disassemble
Examine memory
New start address
Quit
Return subroutine
Unbranch/backstep up

Any of the options can be selected at any time while the program is running by typing the first letter of the menu item. As a memory aid, a line listing the letters for the options is printed after each location is disassembled. Whenever you wish to see the menu again, type M. Going back to the menu does not cause you to lose your place in memory.

To start disassembly at the address previously specified, hit A (for Advance). The space bar performs the same function and has the added advantage that it will repeat if held down. The program will display four columns of information. The first column is the address (in hexadecimal) of the data being disassembled. The second column is the mnemonic of the 6502 opcode from that location, plus one or more characters which indicate the addressing mode.

Addressing Modes

The possible addressing modes are:

#	Immediate
Z	Zero page
-A	Accumulator
(X)	Indexed indirect
(Y)	Indirect indexed
Z,X	Zero page indexed,X
Z,Y	Zero page indexed,Y
,X	Absolute indexed,X
,Y	Absolute indexed,Y
(I)	Indirect

If no mode is shown, then the addressing mode is Absolute,

Implied (as with INX and DEY), or Relative (as with BEQ and BCC).

Arguments and Addresses

For all except the JMP, JSR, and Branch instructions, the remaining two columns are the hexadecimal and decimal value of the *argument* of the opcode. For JSRs and Absolute addressing mode JMPs, the two columns are the hexadecimal and decimal value of the address to which the JMP or JSR will take the program. For Indirect addressing mode JMPs, the two columns are the hexadecimal and decimal values of the locations holding the base address for the Indirect JMP. For the various conditional Branch instructions, the third column is the hexadecimal value of the *offset* for the Branch, and the fourth column is the hexadecimal value of the *address* to which the program will go if the Branch is taken.

Examining Memory

Occasionally, as you are disassembling, you will encounter the message ILLEGAL. This means that the location contains a value which has no corresponding 6502 opcode. This may be an array of data or a temporary storage location. Attempting to disassemble an illegal opcode will switch the program to the Examine memory option.

In this option, three columns of information are displayed. The first is the address (in hexadecimal) of the memory byte being examined, and the second and third columns are the hexadecimal and decimal values of the contents of that location.

If you simply wish to look at the contents of a block of memory, you can also select this option by typing E (for Examine memory). You can step sequentially to higher memory addresses by typing A (or the space bar) or to lower addresses by typing U (for Unbranch/backstep). Type D (for Disassemble) to begin disassembling again.

Jumping and Branching

Whenever you encounter a JMP, JSR, or Branch instruction, you can, if you wish, make the jump to the new address by typing B (for Branch). One exception is that the program cannot make Indirect addressing mode JMPs. For JSRs, the program keeps track of the number of subroutine levels and prints the number in the right of the inverse video memory aid line.

The program as presented is limited to ten levels of subrou-

tines. If you have additional memory you can increase this number. For example, to increase the number of levels to 50, add the following to the end of line 530:

:DIM RE(50)

and change the (SR < 10) in line 290 to (SR < 50). Typing R (for Return) will step you back through the JSRs taken. (You can also get back to the previous JSR by typing a B when you encounter an RTS instruction.) Typing a U (for Unbranch) will return you to the last JMP or Branch taken.

You can start disassembling from a different address at any time by typing N to select the New starting address option. You can convert numbers freely between binary, decimal, and hexadecimal by typing C for the Convert base option. To end the Tracing Disassembler program, type Q for the Quit option.

Respect Copyright, Please

No one minds if you look at and disassemble a routine to discover its workings and access points. If you discover that the routine to locate the next tape header begins at $E165, then by all means have your program SYS 57701 in order to use the routine. On the other hand, you may not distribute any portion of copyrighted material, even with the variable names changed, without written permission. This program is intended only to explore the workings of your computer or to verify the assembly of your own programs.

Tracing Disassembler

```
10 PRINT"{CLR}{2 SPACES}6502 DISASSEMBLE
   R"
20 GOSUB530
30 PRINT:PRINT"{RVS}"SE$"{OFF}ENU{RVS}
   {2 SPACES}"RIGHT$(STR$(SR),2)"
   {2 SPACES}{OFF}";:GOSUB70
40 GOSUB80:FORM=1TO12:IFA$<>MID$(SE$,M,1
   )THENNEXT:GOTO40
50 PRINT"{21 SPACES}";:GOSUB70
60 MODE$="{3 SPACES}":LI=PA:ONMGOSUB220,
   220,200,650,210,180,590,999,170,190,2
   20,610:GOTO30
70 FORJ=0TO20:PRINTCHR$(157);:NEXT:RETUR
   N:REM CURSOR LEFT'S
80 POKE198,0:WAIT198,1:GETA$:RETURN:REM
   GET A CHAR FROM KEYBOARD
```

—7—

```
90 J=3+2*(A<LI):K=16
100 A%=A+HI*(A>H):AD$="":FORM=0TOJ:AD$=A
    D$+MID$(H$,(A%/K↑(J-M)AND(K-1))+1,1)
    :NEXT:RETURN
110 AD=0:FORL=1TOR:GOSUB270:A=PEEK(X):AD
    =AD+A*PA↑(L-1):GOSUB90:MN$=AD$+LEFT$
    (MN$,2):NEXT
120 RETURN
130 J=10:A=0:K=A:L=5
140 GOSUB80:IFK=0THENIFA$="$"ORA$="%"THE
    NPRINTA$;:K=1:J=16:IFA$="%"THENJ=2:L
    =9
150 ON-(A$=CHR$(13))GOTO120:FORM=1TOJ:IF
    A$<>MID$(H$,M,1)THENNEXT:GOTO140
160 PRINTA$;:A=A*J+M-1:K=K+1:ON-(K<L)GOT
    O140:RETURN
170 GOSUB300:X=Y-3:GOTO220
180 R=-1:X=Z:GOTO220
190 Y=BR:FL=0:IFR=-1THENX=X+2*(X>1):GOTO
    220
200 IFFL<4THENONFLGOSUB280,290,300:Z=Y
210 X=Z:R=0
220 Z=X:A=X:LIMIT=0:GOSUB90:LI=PA:PRINTA
    D$" ";:A=PEEK(X):IFR>-1THENGOSUB310
230 IFR=-1THENGOSUB90:PRINTAD$;A;IL$;:GO
    TO270
240 PRINTMN$;MO$" "IL$;:MN$="{3 SPACES}"
250 IFFL=1THENGOSUB110:LI=0:A=Y:GOSUB90:
    LI=PA:PRINTMN$" "AD$;:GOTO270
260 IFRTHENGOSUB110:PRINTBS$(R);MN$;AD;B
    S$(R);
270 IL$="":X=X-(X<HI-1):RETURN
280 BR=X-2:RETURN
290 SR=SR-(SR<10):RETURN
300 SR=SR+(SR>0):Y=RE(SR):RETURN
310 R=0:FL=4:ONAAND3GOTO440,460,470:ONFN
    A(4)GOTO380:MN$=MID$(ZE$,(AAND248)/8
    *3+1,3)
320 IF(AAND31)=16THENR=1:Y=PEEK(X+1):Y=X
    +2+Y-2*(YAND128):FL=1
330 ON-((AAND31)>0)GOTO120:ONFNA(128)GOT
    O350:ONFNA(32)GOTO360
340 IL$="{12 SPACES}END OF ROUTINE":RETU
    RN
350 ON-(A=128)GOTO470:R=1:MO$=" # ":RETU
    RN
360 ONFNA(64)GOTO370:R=2.Y=FNX(0):RE(SR)
    =X+3:FL=2:RETURN
370 FL=3:ON-(SR=0)GOTO340:RETURN
```

```
380 ONFNA(128)GOTO420:IF(AAND247)=36THEN
    MN$="BIT":GOTO430
390 ON-((AAND223)<>76)GOTO470:R=2:FL=0:M
    N$="JMP":Y=FNX(0)
400 IFA=108THENMO$="(I)":Y=PEEK(Y)+PEEK(
    Y+1)*PA
410 RETURN
420 MN$=MID$(ZE$,(AAND224)/8*3+1,3):IF(A
    AND80)=80ORA=156THEN470
430 MO$=MID$(MD$,(AAND28)/4*3+1,3):R=1-(
    (AAND31)=25)-((AAND15)>11):RETURN
440 GOSUB430
450 MN$=MID$(OP$(AAND3),(AAND224)/32*3+1
    ,3):ON-(A=137)GOTO470:RETURN
460 ONFNA(4)GOTO510:ONFNA(8)GOTO480:MO$=
    " # ":R=1:IFA=162THEN450
470 R=-1:IL$="ILLEGAL":RETURN
480 ONFNA(16)GOTO500:ONFNA(128)GOTO490:M
    O$="-A ":GOTO450
490 MN$=MID$(TW$,(AAND96)/32*3+1,3):RETU
    RN
500 ON-((AAND208)<>144)GOTO470:MN$="TSX"
    :ONFNA(32)GOTO120:MN$="TXS":RETURN
510 GOSUB440:ON-((AAND208)<>144)GOTO120:
    ONFNA(8)GOTO520:MO$="Z,Y":RETURN
520 ON-(A=158)GOTO470:MO$=",Y ":RETURN
530 H$="0123456789ABCDEF":SE$=" ABCDENQR
    U M"
540 ZE$="BRKPHPBBPLCLCJSRPLPBMISECRTIPHAB
    VCCLIRTSPLABVSSEI"
550 ZE$=ZE$+"STYDEYBCCTYALDYTAYBCSCLVCPY
    INYBNECLDCPXINXBEQSED"
560 OP$(1)="ORAANDEORADCSTALDACMPSBC":OP
    $(2)="ASLROLLSRRORSTXLDXDECINC"
570 MD$="(X) Z{2 SPACES}#{4 SPACES}(Y)Z,
    X,Y ,X ":TWO$="TXATAXDEXNOP":BS$(2)=
    CHR$(157):REM CURSOR LEFT
580 HI=65536:H=32767:PAGE=256:DEFFNX(A)=
    PEEK(X+1)+PEEK(X+2)*PA:DEFFNA(B)=(AA
    NDB)/B
590 PRINT:PRINT"ENTER STARTING ADDRESS (
    PREFIX '$' FOR HEX)":GOSUB130
600 X=A*-(A<HI):Z=X:BR=X:RE(0)=X+3
610 PRINT:PRINT:PRINT" {RVS}A{OFF}DVANCE
     ONE STEP{6 SPACES}{RVS}B{OFF}RANCH/
    GO SUBROUTINE"
620 PRINT" {RVS}C{OFF}ONVERT BASES
    {9 SPACES}{RVS}D{OFF}ISASSEMBLE CODE
    S{5 SPACES}{RVS}E{OFF}XAMINE ADDRESSES"
```

```
630 PRINT" {RVS}N{OFF}EW START ADDRESS
    {5 SPACES}{RVS}Q{OFF}UIT{18 SPACES}
    {RVS}R{OFF}ETURN SUBROUTINE"
640 PRINT" {RVS}U{OFF}NBRANCH/BACKSTEP U
    P{3 SPACES}(SUBROUTINE LEVEL)":RETUR
    N
650 PRINT:PRINT:PRINT"ENTER NUMBER (PREF
    IX{2 SPACES}'$'=HEX, '%'=BINARY)":GO
    SUB130
660 PRINT:IFA>HI-1THENPRINT"OUT OF RANGE
    ";:RETURN
670 GOSUB90:IFA>255THENPRINT"$"AD$;A;:RE
    TURN
680 PRINT"$"AD$" %";:K=2:J=7:GOSUB100:PR
    INTAD$;A;:RETURN
999 END
```

—7—

Customized BASIC Assembler

R.S. Moser

It usually doesn't take a new game programmer long to find the limitations of games written in BASIC. As the games become more complex and interesting, they run slower and slower. Finally, most game programmers realize they have to use machine language in order to make their games play as they envision them.

It's possible, of course, to compose machine language programs directly from the VIC's built-in BASIC. You only have to POKE the numbers of the machine language code directly into memory, in order, and then save that section of memory. Or you can create the machine language program as DATA statements. The trouble is, it's hard to remember, as you scan through your program, that 224 is the "increment (add 1 to) the X register" command and 165 is "load the accumulator with the contents of the following zero-page address." The longer your program is, the harder it is to remember what is going on at any given place within it.

Mnemonics

The solution, of course, is to use an assembly program, written in the form of short *mnemonics* (things that help you remember). The "increment X register" command becomes INX, while "load the accumulator" is LDA. This is much easier to remember, because the letters carry some meaning.

It's important to remember, though, that words you use with an assembler (the mnemonics) are not machine language. LDA means nothing to the computer. Mnemonics are translated into machine language by your *assembler.* The process is simple enough. First, you write your *source code*— the mnemonics and the accompanying addresses and values. Then the assembler scans through your code, recognizing the mnemonics, changing them to their correct numerical values, or *object code,* and storing them in memory. Finally, you save the section of memory that

contains the assembled program — that's the *machine language program.*

Full-fledged, powerful assemblers allow you to use equates and labels instead of calculating every value, address, and branch. But they can be expensive, and you may want to use this program to create your own specialized assembler.

After all, very few programs use even half the available machine language instructions. Some of the shift instructions and addressing modes are rarely used. If you follow good programming practice and diagram your program in advance, then write it down and check it over before entering it in the computer, you can figure out exactly which machine language instructions you are actually going to use, and adapt this assembler so it recognizes the code you need it to recognize — and no other.

The vocabulary that I needed to write "Gumball" and other games consisted of 43 mnemonics:

3-byte Instructions		2-byte Instructions		1-byte Instructions
ADC	LDA.Y	ADC#	INC	CLC
CMP	LDX	BEQ	LDA	DEX
CMP.Y	LDY	BMI	LDA#	DEY
DEC	SBC	BNE	LDX#	INX
INC	STA	BPL	LDY#	INY
JMP	STA.X	CMP#	SDC#	RTS
JSR	STA.Y	CMP()Y	STA	TAX
LDA	STX	CPX#	STA()Y	TXA
	STY	CPY#		TYA

Describing the function of every machine language instruction is beyond the scope of this article. For that, you need a book that teaches 6502 machine language, and specific information about how the VIC works can be found in the VIC-20 reference guide and other sources. (See *Machine Language for Beginners* and *Mapping the VIC* from COMPUTE! Books — *ed.*)

Customizing the Assembler

If you wish, it is a simple matter to change the BASIC Assembler's vocabulary. The mnemonics are paired with the corresponding machine language numbers in the DATA statements of lines 41 through 45 (for example, CMP.Y, 217, DEC, 206, JMP, 76). The mnemonics are further grouped within certain line numbers according to the number of bytes associated with the specific instruction. Instructions with implied addressing take one byte; instructions with zero-page and immediate addressing take two

bytes; instructions with absolute and indirect addressing take three bytes.

Instruction Bytes	Example (mnemonic, operand)	Line Number
3	DLA, 7432	41 and 42
2	LDA#, 0	43 and 44
1	INX	45

With these ground rules in mind, you may replace any number of mnemonics with others that better fit your purposes. Empty vocabulary space is presently held in lines 42 and 44 by Z, 9 data. You may also increase the assembler's vocabulary by adding new DATA statements and revising the program to READ and process the additional data. Consider, however, that additional memory space taken by the assembler will reduce the space available for your game program.

Since machine language is POKEd into the VIC-20 in decimal form, the most direct system is to write the operands (the addresses or values following the instructions) in their decimal rather than hexadecimal form. A hexadecimal-to-decimal converter was not incorporated in the assembler in order to save space.

The instruction mnemonics used by this assembler are very similar to the standard format. The differences, although slight, must not be ignored. They are:

- The operand is always separated from the mnemonic by a comma.
- The operand is never included within the mnemonic.
- Periods rather than commas are used as punctuation within the mnemonics.
- In the immediate mode, there is no space between the # sign and the mnemonic.

Enter the assembler program as listed. To conserve memory, it was written with each line number as full as possible and with no spaces between the statements and variables.

Using the Assembler

Once the BASIC assembler program has been entered and saved on tape, type RUN and press RETURN. The screen will display

PROG MEM LOC
INI ?
LAST ?

The initial and last memory locations refer to the machine language program that you intend to write. You must respond to these prompts only if you are going to use the SAVE or TRANSFER functions. At this point, there is no need to respond. Press RETURN twice.

The screen now displays the assembler's six operating functions, plus index. These are

WRITE, READ, TRANSFER
SAVE, LOAD, END, INDEX

Press the key corresponding to the first letter of the function you wish to use. Each function is described below.

WRITE Function

The screen will display

W INI MEM LOC ?

Enter the first memory location you wish to write to and press RETURN. This location must be in an area of free RAM that is not already occupied by the assembler and will not be overwritten during assembler operation. The beginning of unused memory may be determined by typing

? PEEK(49) +256*PEEK(50) +1

A safe starting location is usually 6400. Now enter your machine language instruction set. Use commas to separate the mnemonics from the operand. Do not use spaces. Press RETURN after each operand. You don't have to calculate the memory locations. They will automatically sequence for each new line. For example,

6400 STX,7432
6403 CPX#,12
6405 BNE,11
6407 INX,
6408 LDA,7428

Repeat this sequence to complete your program. If you make a mistake and catch it before pressing RETURN, simply correct it. If you type in an incorrect instruction mnemonic or one that is not

in the assembler's vocabulary, the screen will display ERROR and prompt you to reenter the mnemonic. If you wish to change an instruction you've already entered, you may write over the old instruction. Start again at the WRITE function by pressing I once, RETURN twice, W once, and entering the line number you wish to write to.

The upper limit of unused memory, and therefore the upper limit of your game program, may be determined by typing

? PEEK(51) +256*PEEK(52)-1

The branch operands (e.g., BNE, *xxx*) are direct offsets. It's easy to calculate the offset. If you wish to branch to a location later in the program, simply subtract the location (line number) of the instruction following the BNE instruction from the location (line number) you wish to branch to. Enter the result as the operand of the branch instruction. If you wish to branch to a location *preceding* the BNE, *xxx* instruction, subtract the location you wish to branch to from the location following BNE, *xxx;* then subtract this result from 256. You can branch up to 127 locations forward and 128 locations backwards, which is usually sufficient. If not, branch to a JMP instruction. When you wish to break to the index, press I once and RETURN twice.

READ Function

The screen will display

R INI MEM LOC ?

Enter the first memory location you wish to read and press RETURN. The screen will display the instruction set starting with your entered memory location and ending when the screen is filled. If the assembler encounters a location with no instruction or with an instruction not in its vocabulary, it will leave a space after the location number. To continue reading additional instructions, press C. If you wish to break to the index, press I.

TRANSFER Function

In the course of developing a machine language program, you will on numerous occasions need to transfer a section of program from one area of memory to another. The transfer function will make this move. It does *not,* however, erase the program from its original location.

It will also find all JMP and JSR instructions elsewhere in the program that jump to the section being transferred. It will revise their operands accordingly to jump to the new location. In order to use this feature, however, you must now respond to the prompts shown on the screen when the assembler is first run. If you ignored those prompts before, simply press RUN/STOP, type RUN, and the program will start over. Your machine language program will still be in memory. Answer the prompts with the initial and last memory location of the entire machine language program.

The screen will display

```
T xxxx to xxxx
  FROM: INI LOC ?
        LAST LOC ?
TO:     INI LOC ?
```

Respond to these three prompts with the starting address of the block you want to move, the ending address of the block, and the starting address of the new location. If the assembler must scan through a large program for JMP and JSR instructions, it will take a few seconds to complete the transfer; then it will automatically return to the index.

SAVE Function

The screen will display

```
S xxxx to xxxx
NAME ?
```

This function was written to save the machine language program to the Datassette (cassette tape). Respond to the NAME prompt with the file name of your program.

Remember, if you began this session without responding to the initial prompts asking for starting and ending locations for the program, you must do so before saving. Press RUN/STOP, then type RUN. The assembler will start over, but the machine language program will still be there in memory, unchanged. Enter the starting and ending addresses as you are prompted, and then choose the SAVE function. The starting and ending addresses will be displayed before the NAME prompt, so you can make sure you're saving the entire program.

LOAD Function

The screen will display

NAME ?

To load a program which was saved through the assembler, follow these steps:

1. LOAD the assembler program from cassette.
2. RUN.
3. Press RETURN twice (until the index appears).
4. Press L, for LOAD.
5. Type the name of the program and RETURN.
6. Put the cassette with the machine language program into the recorder and press PLAY.
7. When the program has been loaded, the index will reappear on the screen.
8. Exit the assembler by pressing RUN/STOP.
9. Type SYS*xxxx*. *xxxx*, of course, is the address where you want the program to begin execution. Depending on how you designed your program, it is not necessarily the *first* address in the program.

It's a good idea to write down on paper the pertinent addresses — the lowest and highest addresses in the program, and where it should begin execution. You can keep updating these addresses on paper as you revise your program.

END Function

The screen will display

NAME ?

This function was written to find the end of a particular program on tape. Enter the name of the program and press RETURN. When the end has been found, the assembler returns to the index.

BASIC Assembler

```
1 PRINT"{CLR}{DOWN}PROG MEM LOC":INPUT"
  {3 SPACES}INI";J:INPUT"{2 SPACES}LAST";
  K
```

-7-

```
2 DIMA$(45):DIMA(45):FORB=ØTO44:READA$(B)
  ,A(B):NEXT
3 PRINT"{CLR}{DOWN} {RVS}W{OFF}RITE,{RVS}
  R{OFF}EAD,{RVS}T{OFF}RANSFER","{DOWN}
  {RVS}S{OFF}AVE,{RVS}L{OFF}OAD,{RVS}E
  {OFF}ND,{RVS}I{OFF}NDEX"
4 C=Ø:D=Ø:GETB$:FORC=1TO6:IFB$=MID$("WRTS
  LE",C,1)THEND=C:C=6
5 NEXT:ONDGOTO6,16,26,33,35,38:GOTO4
6 PRINT"{CLR}":INPUT"{RVS}W{OFF}INI MEM L
  OC";B
7 PRINTB;:G=Ø:C=Ø:D=Ø:INPUTB$,C:IFB$="I"T
  HEN3
8 IFC>255THENE=INT(C/256):F=INT(C-E*256):
  GOTO12
9 FORD=18TO35:IFB$=A$(D)THENG=2:H=D:D=35:
  POKEB+1,C
10 NEXT:FORD=36TO44:IFB$=A$(D)THENG=1:H=D
   :D=44
11 NEXT:GOTO14
12 FORD=ØTO17:IFB$=A$(D)THENG=3:H=D:D=17:
   POKEB+1,F:POKEB+2,E
13 NEXT
14 IFG=ØTHENPRINT"{UP}ERROR":GOTO7
15 POKEB,A(H):B=B+G:GOTO7
16 PRINT"{CLR}":INPUT"{RVS}R{OFF} INI MEM
    LOC";B:C=Ø:D=4Ø
17 E=C:F=PEEK(B+C):PRINTB+C"";:FORG=ØTO17
18 IFF=A(G)THENH=PEEK(B+C+1):I=PEEK(B+C+2
   ):PRINTA$(G)TAB(12)I*256+H:C=C+3:G=17
19 NEXT:FORG=18TO35:IFF=A(G)THENPRINTA$(G
   )TAB(12)PEEK(B+C+1):C=C+2:G=35
2Ø NEXT:FORG=36TO44:IFF=A(G)THENPRINTA$(G
   )"{4 SPACES}--":C=C+1:G=44
21 NEXT:IFE=CTHENPRINT,:C=C+1
22 IFC<DTHEN17
23 PRINT"{UP}":PRINT"{RVS}C{OFF}ONT,{RVS}
   I{OFF}NDEX{UP}",,:GETB$:IFB$="I"THEN3
24 IFB$="C"THEND=D+4Ø:PRINT"{1Ø SPACES}
   {UP}":GOTO17
25 GOTO23
26 PRINT"{CLR}{RVS}T{OFF} "J"TO"K:PRINT:P
   RINT"{DOWN}FROM: INI LOC";:INPUTB:INPU
   T"{5 SPACES}LAST LOC";C
27 INPUT"{DOWN}TO{2 SPACES}: INI LOC";D:I
   FB>DTHENFORE=BTOC:POKEE+D-B,PEEK(E):NE
   XT:GOTO29
28 FORE=CTOBSTEP-1:POKEE+D-B,PEEK(E):NEXT
   :GOTO29
```

```
29 FORE=JTOK:F=PEEK(E):IFF=32ORF=76THENG=
   PEEK(E+1)+256*PEEK(E+2):GOTO31
30 NEXT:GOTO3
31 IFG>=BANDG<=CTHENG=G+D-B:L=INT(G/256):
   M=INT(G-L*256):POKEE+1,M:POKEE+2,L
32 GOTO30
33 PRINT"{CLR}{RVS}S{OFF} "J"TO"K"{DOWN}"
   :INPUT"NAME";B$:OPEN1,1,1,B$
34 PRINT#1,J:FORD=JTOK:E=PEEK(D):PRINT#1,
   E:NEXT:CLOSE1:GOTO3
35 PRINT"{CLR}":INPUT"NAME";B$:OPEN1,1,0,
   B$:INPUT#1,B
36 INPUT#1,C:D=ST:POKEB,C:B=B+1:IFD=0THEN
   36
37 CLOSE1:GOTO3
38 PRINT"{CLR}":INPUT"NAME";B$:OPEN1,1,0,
   B$
39 INPUT#1,B:C=ST:IFC=0THEN39
40 CLOSE1:GOTO3
41 DATACMP,205,CMP.Y,217,DEC,206,JMP,76,L
   DA.Y,185,LDA,173,LDX,174,STA,141,STA.Y
   ,153
42 DATASTA.X,157,STY,140,JSR,32,LDY,172,S
   TX,142,INC,238,SBC,237,ADC,109,Z,9
43 DATABEQ,240,BNE,208,CMP#,201,CPY#,192,
   CPX#,224,LDA#,169,LDY#,160,LDX#,162,ST
   A()Y,145
44 DATAADC#,105,LDA,165,CMP()Y,209,STA,13
   3,INC,230,BPL,16,BMI,48,SBC#,233,Z,9
45 DATADEY,136,INY,200,RTS,96,CLC,24,TXA,
   138,TAX,170,INX,232,DEX,202,TYA,152
```

—7—

Gumball: A Machine Language Game Using the BASIC Assembler

R.S. Moser

"Gumball" is an exciting, all machine language game for the unexpanded VIC. It can be entered into memory using the BASIC Assembler. Even if you don't know machine language, you can enter and play Gumball.

The program that follows is designed to be entered using the BASIC Assembler from the preceding article "Customized BASIC Assembler." If you are using a different assembler, the code can be entered as shown, except in three cases:

This Listing	**Normal Assembly Code**
CPX#,12	CPX #12
STA( )Y,251	STA(251),Y
LDA.Y,8177	LDA 8177,Y

The difference is because, to save memory and time in this custom assembly program, each addressing mode is treated as a separate instruction — which it really is, in machine language.

Also, since this custom assembler doesn't allow for remarks, they aren't listed with leading semicolons, but appear instead under the line or lines they refer to. Do not attempt to enter these explanations in your program — the assembler won't know what to do with them.

All the branch instructions use direct offsets instead of labels to tell the computer where to go. To figure out where the program is branching to, just perform a couple of simple operations using

the number right after the branch instruction. If the number is less than 128, add it to the address of the instruction that immediately follows the branch instruction. If the number is 128 or greater, subtract it from 256 and then subtract the result from the next address in the program. The result, in both cases, will be the address where the program will branch if the branch conditions are met. All numbers are decimal.

Saving Gumball

Once you have finished typing in the assembler listing of "Gumball," you should SAVE it.

1. Press the STOP/RUN key. (Do not turn off the computer.)
2. RUN the Assembler program again.
3. Initial memory location should be entered as 6500, press RETURN.
4. Last memory location should be entered as 7387, press RETURN.
5. Press S for SAVE.
6. Enter a file name and press RETURN.
7. Press PLAY and RECORD on the tape player.

Debugging

After you have entered the game program and saved it on tape, you should check it for possible entry errors. To do this, enter the following checking program in direct mode (no line numbers). It sums up the contents of 24 memory locations at a time. Press the space bar to print the results. You may print a few at a time by holding the space bar down a short time.

```
FOR A=6500 TO 7387 STEP 24:C=0:FOR B=A TO A+23:C=
C+PEEK(B):NEXT:PRINT A;C:WAIT 197,32,0:NEXT
```

The results displayed on the screen should be:

Starting Memory Location	Sum	Starting Memory Location	Sum	Starting Memory Location	Sum
6500	2864	6788	2719	7076	2499
6524	2792	6812	2208	7100	3297
6548	2584	6836	2597	7124	2780
6572	2585	6860	2225	7148	1787

6596	3325	6884	2709	7172	2717
6620	3368	6908	2387	7196	2670
6644	2650	6932	2448	7220	2444
6668	2351	6956	3175	7244	3421
6692	1965	6980	3128	7268	3132
6716	2474	7004	2776	7292	2682
6740	2852	7028	3182	7316	2447
6764	2970	7052	3426	7340	3091
				7364	3692

If you find a discrepancy, read your program at the suspected memory locations and correct any errors. Then SAVE the program again.

Getting Under Way

Once the program has been checked and corrected, run it by entering SYS 6558 and pressing the return key. The screen will appear, surrounded by wall units with the game score (0000) and high score (0000) in the bottom border. The player will be near the bottom center of the screen. Nineteen gumballs will enter the screen from the top left corner, with sound accompaniment.

You can move the player by pressing the following keys:

A =*left*
D =*right*
W =*up*
X =*down*

Fire the gun by pressing the arrow (↑) key.

If the bullet hits a gumball, it is smashed and replaced by a square of gum. Your game score is increased by ten. If the bullet hits a wall unit or gum, you lose one point. Sound accompanies both hits.

Remember: Don't smash *all* the gumballs. At least one must reach the bottom and go out of the gumball machine, or you'll be out of order and no new gumballs will appear.

If a gumball hits the player, the player turns into a heart and the game is over. To start a new game, press the return key. The high score will be maintained from game to game, until you remove the program from memory.

LOADing Gumball from Tape

Follow these instructions to LOAD Gumball from tape:

1. LOAD and RUN the Assembler.
2. Press RETURN twice to get to the menu.

3. Be sure tape is positioned correctly.
4. Type L for LOAD.
5. Type the file name (GUMBALL) and press RETURN.
6. Press play on the recorder.
7. Once the program has been LOADed press STOP/RUN.
8. Type SYS 6558 and the game will appear.

Gumball

X-Coordinate Subroutine

```
6500 STX,7432
     Stash X.
6503 CPX#,12
6505 BPL,11
     If X is greater than 12,
     branch 11.
6507 LDA#,30
6509 STA,252
     Character location is 30.
6511 LDA#,150
6513 STA,254
     Color location is 150.
6515 JMP,6526
6518 LDA#,31
6520 STA,252
     Character location is 31.
6522 LDA#,151
6524 STA,254
     Color location is 151.
6526 LDA#,0
6528 CPX#,0
6530 BEQ,6
6532 DEX,
6533 ADC#,21
6535 JMP,6528
     Each time X is decre-
     mented, add 21 to the ac-
     cumulator until X=0. Then
     branch forward 6 steps (to
     6538).
6538 STA,251
6540 STA,253
     Store accumulator in mem-
     ory locations 251 and 253.
6542 LDX,7432
6545 RTS,
     Restore the original value of
     X; then return from the
     subroutine.
```

Erase Subroutine

```
6546 JSR,6500
6549 LDA#,32
6551 STA()Y,251
6553 LDA#,1
6555 STA()Y,253
6557 RTS,
     Store 32 (a blank) and 1 (the
     color white) at the X-Y co-
     ordinates of the calling
     routine.
```

Set Screen

```
6558 LDY#,0
6560 JSR,6610
6563 LDY#,21
6565 JSR,6610
6568 LDX#,0
6570 JSR,6629
6573 LDX#,22
6575 JSR,6629
     Print wall units at borders.
6578 LDY#,4
6580 DEY,
6581 LDA#,48
6583 STA.Y,8177
6586 CPY#,0
6588 BNE,246
     Set high score to zero.
6590 LDX#,1
6592 LDY#,1
6594 JSR,6546
6597 INY,
```

```
6598 CPY#,21
6600 BNE,248
6602 INX,
6603 CPX#,22
6605 BNE,241
     Clear screen.
6607 JMP,6648
     Jump to initialization
     routine.
6610 LDX#,0
6612 JSR,6500
6615 LDA#,102
6617 STA()Y,251
6619 LDA#,0
6621 STA()Y,253
6623 INX,
6624 CPX#,23
6626 BNE,240
6628 RTS,
     Subroutine to put wall
     units (character 102) at side
     borders.
6629 LDY#,1
6631 JSR,6500
6634 LDA#,102
6636 STA()Y,251
6638 LDA#,3
6640 STA()Y,253
6642 INY,
6643 CPY#,21
6645 BNE,240
6647 RTS,
     Subroutine to print wall
     units at top and bottom
     borders.
```

Initialization

```
6648 LDA#,20
6650 STA,7423
6653 LDA#,10
6655 STA,7424
     Set player at center bottom
     of screen.
6658 LDY#,4
6660 DEY,
6661 LDA#,48
6663 STA.Y,8166
6666 CPY#,0
6668 BNE,246
     Set game score to zero.
6670 LDA#,1
6672 STA,7436
     Set location 7436 to 1, which
     allows new gumballs to
     appear.
6675 LDY#,0
6677 LDA#,0
6679 STA.Y,7440
     Set direction status of gum-
     balls to zero.
6682 LDA#,1
6684 STA.Y,7460
6687 STA.Y,7480
6690 INY,
6691 CPY#,20
6693 BNE,238
     Set gumballs' X and Y co-
     ordinates to 1.
6695 STA,7427
6698 LDA#,15
6700 STA,36878
     Set volume high (15).
6703 JMP,6706
     Jump to timer.
```

Timer

```
6706 LDA#,2
6708 STA,7428
6711 LDA#,2
6713 STA,7429
     Store a 2 at 7428 and 7429.
6716 LDA#,50
6718 STA,7431
6721 STA,7430
     Store 50 in 7431 and 7430.
6724 DEC,7430
6727 BNE,251
     Decrement the contents of
     7430 until it equals 0.
6729 DEC,7431
6732 BNE,243
     Decrement 7431. This is
     nested outside the previous
     loop. To slow down or
```

speed up the game, the user can put different numbers at 7430 and 7431 while entering the program or debugging with a monitor or debugger.

6734 JMP,6823
Jump to bullet routine.
6737 DEC,7429
6740 BNE,230
Decrement 7429.
6742 JMP,6753
Jump to player routine.
6745 DEC,7428
6748 BNE,217
Decrement 7428.
6750 JMP,7137
Jump to controller routine.

Player Routine

6753 LDX,7423
6756 LDY,7424
Load last player location in X and Y.
6759 JSR,6546
Erase last player location.
6762 LDA,197
Look at keyboard.
6764 CMP#,17
6766 BNE,1
6768 DEY,
If *A*, move player left.
6769 CMP#,18
6771 BNE,1
6773 INY,
If *D*, move player right.
6774 CMP#,9
6776 BNE,1
6778 DEX,
If *W*, move player up.
6779 CMP#,26
6781 BNE,5
6783 CPX#,21
6785 BEQ,1
6787 INX,
If *X*, move player down unless already on bottom line.
6788 JSR,6500
6791 LDA#,102
6793 CMP()Y,251
6795 BNE,9
Look to see if new player location is occupied by a wall unit or gum.
6797 LDX,7423
6800 LDY,7424
6803 JSR,6500
If yes, then replace player in last location; otherwise, let new location stand.
6806 LDA#,65
6808 STA()Y,251
6810 LDA#,0
6812 STA()Y,253
Print player.
6814 STX,7423
6817 STY,7424
Store player location.
6820 JMP,6745
Jump to timer.

Bullet Routine

6823 LDX,7425
6826 LDY,7426
Load last bullet location.
6829 LDA#,0
6831 STA,36875
6834 STA,36876
Turn off sound.
6837 LDA,7438
6840 CMP#,1
6842 BEQ,18
If bullet is already moving (7438 is set to 1), then branch.
6844 LDA,197
6846 CMP#,54
6848 BEQ,3
If arrow key is pressed (54), then branch.
6850 JMP,6737
Return to timer.
6853 LDX,7423

```
6856 LDY,7424
     Load initial bullet location
     (which is the same as
     player location).
6859 JMP,6865
     Skip next step.
6862 JSR,6546
     Erase the last bullet
     location.
6865 DEX,
     Move the bullet up one
     step.
6866 LDA#,1
6868 STA,7438
     Set bullet status to moving
     (1).
6871 STX,7425
6874 STY,7426
     Store present bullet
     location.
6877 JSR,6500
6880 LDA#,102
6882 CMP()Y,251
6884 BNE,11
     Look to see if the bullet hits
     a wall unit or gum.
6886 LDA#,201
6888 STA,36876
     If yes, turn on the sound.
6891 JSR,6925
6894 JMP,6943
     Reset bullet and jump to
     bullet-hits-wall routine.
6897 LDA#,81
6899 CMP()Y,251
6901 BNE,11
     Look to see if bullet hits
     gumball.
6903 LDA#,223
6905 STA,36875
     If yes, turn on the sound.
6908 JSR,6925
6911 JMP,7234
     Reset bullet and jump to
     bullet-hits-gumball
     routine.
6914 LDA#,30
6916 STA()Y,251
     If the bullet hits neither,
     then print a new bullet.
6918 LDA#,0
6920 STA()Y,253
6922 JMP,6737
     Jump to timer.
6925 LDA,7423
6928 STA,7425
6931 LDA,7424
6934 STA,7426
6937 LDA#,0
6939 STA,7438
6942 RTS,
     Subroutine to reset bullet
     location to player location
     and reset bullet status to
     stop (0).
```

Bullet-Hits-Wall Routine

```
6943 INX,
6944 JSR,6546
     Erase last bullet.
6947 DEC,8169
6950 LDA,8169
6953 CMP#,47
6955 BNE,50
6957 LDA#,57
6959 STA,8169
     Subtract 1 from game score.
6962 DEC,8168
6965 LDA,8168
6968 CMP#,47
6970 BNE,35
6972 LDA#,57
6974 STA,8168
     "Tens" digit of score.
6977 DEC,8167
6980 LDA,8167
6983 CMP#,47
6985 BNE,20
6987 LDA#,57
6989 STA,8167
     "Hundreds" digit of score.
6992 DEC,8166
6995 LDA,8166
```

```
6998 CMP#,47
7000 BNE,5
7002 LDA#,57
7004 STA,8166
     "Thousands" digit of score.
7007 JMP,6737
     Jump to timer.
```

Gumball Routine

```
7010 STY,7433
     Stash Y.
7013 LDX,7421
7016 LDY,7422
     Load gumball location into
     X and Y.
7019 JSR,6500
7022 LDA#,102
7024 CMP()Y,251
7026 BNE,8
7028 LDA#,0
7030 STA,7420
7033 JMP,7131
     Eliminate hit gumball (store
     0 at 7420); recognize that
     gumball was hit because
     there is a square of gum at
     the gumball's location.
7036 LDA#,32
7038 STA()Y,251
7040 LDA#,1
7042 STA()Y,253
     Erase last gumball.
7044 LDA,7420
7047 CMP#,1
7049 BEQ,10
     Look for left or right gum-
     ball direction (left =1).
7051 INY,
7052 LDA#,102
7054 CMP()Y,251
7056 BEQ,23
7058 JMP,7088
     Look one space to the right
     for wall or gum.
7061 DEY,
7062 LDA#,102
7064 CMP()Y,251
7066 BEQ,3
7068 JMP,7088
     Look one space to the left
     for wall or gum.
7071 INY,
7072 LDA#,2
7074 STA,7420
7077 INX,
7078 JMP,7088
     Move gumball to the right
     one space; set gumball
     status to right(2).
7081 DEY,
7082 LDA#,1
7084 STA,7420
     Move gumball to the left
     one space; set gumball
     status to left(1).
7087 INX,
7088 CPX#,22
7090 BNE,3
7092 JMP,6670
     If gumball reaches bottom
     (22), then jump to
     initialization.
7095 JSR,6500
7098 LDA#,102
7100 CMP()Y,251
7102 BNE,4
7104 INX,
7105 JMP,7095
     If a wall unit is at the new
     gumball location, then
     drop gumball down one
     space.
7108 LDA#,65
7110 CMP()Y,251
7112 BNE,3
7114 JMP,7302
     If gumball hits player, then
     jump to gumball-hits-
     player routine.
7117 LDA#,81
7119 STA()Y,251
7121 LDA#,4
```

```
7123 STA()Y,253
     Print new gumball.
7125 STX,7421
7128 STY,7422
     Store new gumball
     location.
7131 LDY,7433
     Restore original Y.
7134 JMP,7213
     Jump to controller routine.
```

Controller Routine

```
7137 LDA#,235
7139 STA,36874
     Turn on sound for gumball.
7142 LDA,7436
7145 CMP#,1
7147 BNE,23
7149 LDY,7427
7152 CPY#,19
7154 BEQ,11
7156 LDA#,2
7158 STA.Y,7440
7161 INC,7427
7164 JMP,7172
     If 7436 contains 1, then gen-
     erate new gumballs by stor-
     ing 2 at 7440 plus offset (Y).
7167 LDA#,0
7169 STA,7436
     When complete, turn off
     gumball generator by stor-
     ing 2 in 7436.
7172 LDY#,255
7174 INY,
7175 LDA.Y,7440
7178 BNE,12
7180 CPY#,19
7182 BNE,246
     Look at status of each gum-
     ball. If 0, then skip to next
     gumball. If 1 or 2, then
     branch.
7184 LDA#,0
7186 STA,36874
7189 JMP,6706
     When completed, turn off
     sound and jump to timer.
7192 LDA.Y,7440
7195 STA,7420
7198 LDA.Y,7460
7201 STA,7421
7204 LDA.Y,7480
7207 STA,7422
     Transfer direction status
     and X-Y coordinates from
     matrix memory to individ-
     ual gumball memory.
7210 JMP,7010
     Jump to gumball routine.
7213 LDA,7420
7216 STA.Y,7440
7219 LDA,7421
7222 STA.Y,7460
7225 LDA,7422
7228 STA.Y,7480
     Transfer new direction
     status and new X-Y coordi-
     nates from individual gum-
     ball memory to matrix
     memory.
7231 JMP,7174
     Continue sequence.
```

Bullet Hits Gumball

```
7234 JSR,6500
7237 LDA#,102
7239 STA()Y,251
7241 LDA#,4
7243 STA()Y,253
     Print wall unit over hit
     gumball.
7245 INX,
7246 JSR,6546
     Erase bullet under hit
     gumball.
7249 INC,8168
7252 LDA,8168
7255 CMP#,58
7257 BNE,35
7259 LDA#,48
```

```
7261 STA,8168
     Increase game score by 10.
7264 INC,8167
7267 LDA,8167
7270 CMP#,58
7272 BNE,20
7274 LDA#,48
7276 STA,8167
     Hundreds digit.
7279 INC,8166
7282 LDA,8166
7285 CMP#,58
7287 BNE,5
7289 LDA#,48
7291 STA,8166
     Thousands digit.
7294 LDA#,0
7296 STA,7438
     Stop bullet by storing 0 at
     7438.
7299 JMP,6737
     Jump to timer.
```

Gumball Hits Player

```
7302 JSR,6500
7305 LDA#,83
7307 STA()Y,251
7309 LDA#,2
7311 STA()Y,253
     Print heart over player.
7313 LDA#,201
7315 STA,36876
     Set sound frequency.
7318 LDA#,15
7320 STA,36878
     Set volume high.
7323 LDA#,120
7325 STA,7434
7328 STA,7435
7331 DEC,7435
7334 BNE,251
7336 DEC,7434
7339 BNE,243
7341 DEC,36878
7344 BNE,233
     Slowly diminish volume.
7346 LDA#,0
7348 STA,36876
     Turn sound frequency off.
7351 LDY#,0
7353 LDA.Y,8166
7356 CMP.Y,8177
7359 BMI,18
7361 BNE,8
7363 INY,
7364 CPY#,4
7366 BNE,241
     Compare game score with
     high score.
7368 LDA.Y,8166
7371 STA.Y,8177
7374 INY,
7375 CPY#,4
7377 BNE,245
     If game score is higher,
     then print new high score.
7379 LDA,197
7381 CMP#,15
7383 BNE,250
     Game over. Wait until re-
     turn key is pressed to re-
     peat the game.
7385 JMP,6590
     Jump to set screen.
```

—Chapter 8—

Memory Map

—8—

What Is a Memory Map?

G. Russ Davies

A memory map is a consolidated description of all memory locations that can be addressed by a particular computer (in this case the Commodore VIC-20).

It represents all of the various memory locations and their usage, and it is the ultimate source you will need for any memory information. The map is a reference resource just as the dictionary is.

The more experience you have on the VIC-20, the more you will find the map useful. It is essentially a guided tour of the internals of the VIC-20.

The map will help you to:

- Find the areas of memory that control a particular function
- Grasp the interrelationship of memory locations
- Discover new ways of doing things
- Understand how something works
- Change memory so it does what you need it to do
- Understand what other programmers' instructions are doing
- Find and use existing Kernal and BASIC routines that you can use for your own needs
- Remember the location of that one field you need to find

How The Map Is Formatted

A	B	C	D	E
00034	$0022	L=7	Memory location description	XXXX

A = The decimal address of the beginning of the field (as used in PEEK and POKE). An asterisk here indicates a zero-page usable location (very useful in 6502 machine language programs).

B = The hexadecimal address of the beginning of the field (used in 6502 machine language).

C = The length of the field (in decimal); L =1 assumed.
D = Description of the field's use.
E = Notation or typical (default) value in 5K VIC-20 (included when of value).

BASIC Working Storage
00-143 $00-$8F L =144

Note: When you're not using BASIC, ML Code could be stored in this area.

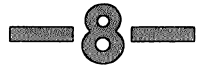
Decimal	*Hex*	*Field Length*	*Description*	*Default*
0*	0		6502 JMP operation for following USR vector.	76
1*	1	L =2	USR jump vector; set this with POKEs before issuing USR. Also used as vector for BASIC error message routines.	$D248
3	3	L =2	*Vector:* Floating point to fixed point BASIC conversion routine.	$D1AA
5	5	L =2	*Vector:* Fixed point to floating point BASIC conversion routine.	$D394
7	7		Search character, observed to be 0 if line entered to BASIC, 34 if entered in immediate mode, probably set to : or 0 by BASIC scan routine.	
8	8		*Flag:* Scan-quotes, observed to be 34 if colon in immediate or BASIC line.	
9	9		Column cursor is currently on in line; 0-87	
10	A		*Flag:* 0 =LOAD, 1 =VERIFY.	
11	B		Input buffer pointer/subscript number, observed to be 76 unless line has subscripted variable, then set to 0.	76
12	C		*Flag:* Default DIM or first character of variable name.	
13	D		*Flag:* Variable type (FF =string, 00 =numeric).	
14	E		*Flag:* Numeric variable (80 =integer, 00 =floating point).	
15	F		*Flags:* DATA scan / LIST quote / memory flag observed to contain 4 if no : in immediate line or 2 if quotes in line and no : .	

16	10		Subscript or FN X flag.	
17	11		*Flag:* Where is data coming to BASIC from? 0 =INPUT, $40 =GET, $98 =READ.	0
18	12		ATN sign/comparison evaluation flag > is 1, =is 2, < is 4, and in combination 0 0 if AND or OR in comparison.	
19	13		*Flag:* Current I/O prompt, 20 if GET, INPUT or READ is done.	4
20	14	L =2	Integer value for GOTO, SYS, POKE, etc. Observed to be not normally used.	
22	16		*Pointer:* Next available descriptor in the below temporary storage stack.	25
23	17	L =2	*Pointer:* Last temporary string descriptor below (will be 25, 0; 28, 0; or 31, 0).	22,0
25	19	L =9	Descriptor stack for 3 temporary strings each descriptor: length, top, bottom (expressed as displacements within BASIC string storage).	
34	22	L =4	Miscellaneous temporary pointers and indirect index register 1 and 2 temporary save.	
38	26	L =5	Multiplication product area for some functions.	
43	2B	L =2	*Pointer:* Start of BASIC program (byte containing 0 must precede the location that this points to). LOAD puts program where this is pointing; issue NEW after changing this pointer.	1,16
45	2D	L =2	*Pointer:* End of BASIC program, start of variables. Variables are built from low to high memory. This pointer is reset by LOAD.	
47	2F	L =2	*Pointer:* End of BASIC variables, start of arrays; arrays are built from low to high memory. This pointer is reset by LOAD.	

49	31	L=2	*Pointer:* End of BASIC arrays, start of free RAM.	
51	33	L=2	*Pointer:* Bottom of BASIC active strings; strings are built from high memory down.	
53	35	L=2	*Pointer:* Top of BASIC active strings; strings are built from high memory down.	
55	37	L=2	*Pointer:* End of BASIC memory; issue CLR after changing this pointer.	0,30
57	39	L=2	Current BASIC line number being executed.	
59	3B	L=2	Previous BASIC line number executed, in form low,high (for example, 148,19 =5012).	
61	3D	L=2	*Pointer:* BASIC statement to CONTinue with.	
63	3F	L=2	Current DATA line number in low,high form.	
65	41	L=2	*Pointer:* Current BASIC DATA item.	
67	43	L=2	BASIC input vector, source of input address. READ: Pointer to one past last item read. GET: 0,2 or 1,2 if key struck. INPUT: LL,2 where LL = number of characters entered.	
69	45	L=2	Current BASIC variable name with type flags, one or two characters. $ or % is not included in name. FLOATING POINT: Character 1, character 2 or 0. INTEGER: Character 1, character 2 or 0 (both ORed with 128). STRING: Character 1, character 2 or 0 (character ANDed with 128). (Also see $D and $E.)	
71	47	L=2	*Pointer:* Current BASIC variable or string descriptor.	
73	49	L=2	*Pointer:* BASIC variable used in current FOR loop.	

75	4B	L=2	Y-SAVE ; OP-SAVE ; BASIC Pointer SAVE; Pointer to current operation in operation table.
77	4D		BASIC comparison symbol.
78	4E	L=2	*Pointer:* BASIC subject function definition.
80	4F	L=2	*Pointer:* BASIC subject string descriptor.
82	52		BASIC length of subject string.
83	53		Constant for BASIC garbage collection (3 or 7).
84	54	L=3	JUMP opcode, vector to function routine.
87 *	57	L=10	BASIC numeric work area (ML can use this area).
97	61		*Floating Point ACCUM1:* Exponent.
98	62	L=4	*Floating Point ACCUM1:* Mantissa.
102	66		*Floating Point ACCUM1:* Sign.
103	67		*Pointer:* Series evaluation constant.
104	68		*ACCUM1:* High-order propagation word (overflow).
105	69		*Floating Point ACCUM2:* Exponent.
106	6A	L=4	*Floating Point ACCUM2:* Mantissa.
110	6E		*Floating Point ACCUM2:* Sign.
111	6F		Sign comparison: ACCUM1 vs ACCUM2.
112	70		*ACCUM1:* Low order of mantissa (for rounding).
113	71		Tape buffer length / series pointer, observed to be 239 during tape I/O.
115	73	L=24	Get-BASIC-character routine (CHRGET) placed here at power-up. You can insert an ML JSR/JMP in this ML routine to intercept the interpretation of BASIC words. CHRGOT pointer at 122 ($7A) to character retrieved.
139	8B	L=5	BASIC RND work area, last random number, initialized at power-up.

11



Kernal Working Storage
144-255 $90-$FF L = 112

144	90		ST: I/O status word.	
145	91		*Keyswitch PIA:* STOP key sensed, left SHIFT key and every alternate bottom key can also be sensed: 255 = NONE 239 = N 253 = LEFT SHIFT 223 = , 251 = X 191 = / 247 = V 127 = CRSRUP.	
146	92		Timing constant for tape servo.	
147	93		*Flag:* LOAD = 0, VERIFY = 1.	
148	94		*Serial:* Output deferred character flag.	
149	95		*Serial:* Buffered character.	
150	96		Tape block sync flags, tape sync number.	
151	97		Register save area, temporary area for serial input.	
152	98		How many OPEN files, as displacement into file table at 601.	
153	99		INPUT device number. Common Devices: 0 = keyboard, 1 = tape, 2 = RS-232, 3 = screen, 4/5 = printer, 8-11 = disk, 4-127 = serial.	0
154	9A		OUTPUT device number (also set by CMD).	3
155	9B		*Tape:* Character parity.	
156	9C		*Tape:* Dipole switch/byte received flag.	
157	9D		*Flag:* Kernal message control: 128 = direct, 0 = RUN.	
158	9E		*Tape:* Pass 1 error log.	
159	9F		*Tape:* Pass 2 error log corrected.	
160	A0	L = 3	JIFFY CLOCK: Access/change realtime with TI$; access as count of jiffies with TI. High byte incremented every 18.2 minutes, mid byte incremented every 4.2 seconds, low byte incremented every 0.016 second (Jiffy).	
163	A3		*Serial:* Bit count/end of input (EOI) flag.	

164	A4		*Serial:* Cycle counter/*Tape:* Dipole number.
165	A5		*Tape:* Countdown for sync on tape header.
166	A6		*Tape:* Count of characters in tape buffer (see 178); POKE 166,191 to force buffer to tape.
167	A7		*Tape:* Short COUNTER1. *RS-232:* Receiver input bit temporary storage. *RS-232:* Write leader count.
168	A8		*Tape:* Error flags. *RS-232:* Receiver bit count in. *RS-232:* Write new byte.
169	A9		*Tape:* Counter for zeros. *RS-232:* Receiver flag: start bit check. *RS-232:* Write start bit.
170	AA		*Tape:* Bits 7-6:function, bits 5-0 sync countdown. *RS-232:* Receiver pointer: Byte/buffer assembly.
171	AB		*Tape:* Short COUNTER2. *RS-232:* Receiver parity/checksum bit storage. *RS-232:* Write leader length.
172	AC	L=2	*Tape:* Pointer to start of LOAD/SAVE area (in ML), or pointer to tape buffer if BASIC program I/O.
174	AE	L=2	*Tape:* Pointer to END +1 of LOAD/SAVE area (in ML), or pointer to tape buffer end if BASIC program I/O.
176	B0	L=2	*Tape:* Timing constants.
178	B2	L=2	*Tape:* Pointer to tape buffer (user could change). 828
180	B4		*Tape:* Miscellaneous flags. *RS-232:* transmit bit count out, timer enable flag.
181	B5		*Tape:* Sync save area. *RS-232:* Transmit next bit to be sent or EOT.

182	B6		*Tape:* Error accumulator. *RS-232:* Transmit pointer into byte disassembly area.	
183	B7		Number of characters in file name, 0-188; only first 16 will show in "found" message.	
184	B8		Current logical file number.	
185	B9		Current secondary address.	
186	BA		Current device number.	
187	BB	L=2	*Pointer:* Current file name, usually stored in BASIC area below active strings, above arrays. After LOAD, pointer to start of area loaded.	
189	BD		*Tape:* Input character read, write shift character.	
190	BE		*Tape:* Number of blocks remaining to READ/WRITE.	
191	BF		*Serial:/Tape:* Byte being built.	
192	C0		*Tape:* Motor interlock switch.	
193	C1	L=2	*Pointer:* I/O start address, start address for LOAD; normally would point to tape buffer.	
195	C3	L=2	*Pointer:* Kernal setup routine. *Tape:* Temporary start address for LOAD.	$FD6D
197	C5		Matrix coordinate of key pressed (64 if none).	64
198	C6		Number of characters (0-10) in keyboard buffer at 631 ($277).	
199	C7		*Flag:* Reverse mode 18 =on 0 =off.	
200	C8		*Pointer:* End of line for input.	
201	C9	L=2	*Cursor:* Current position (logical line, column); logical line could be 22, 44, 66, or 88 columns; see 213 ($D5). Column is actually the number of characters on line (screen line links 217 ($D9) flag continued lines).	
203	CB		Matrix coordinate of last key pressed.	
204	CC		*Cursor:* 1 =off 0 =flash.	
205	CD		*Cursor:* Countdown before blink.	

206	CE		*Cursor:* Character under cursor (ASCII value).	
207	CF		*Cursor:* Blink flag, 1 =off.	
208	D0		Input from screen/keyboard.	
209	D1	L =2	*Cursor:* Pointer to start of screen line cursor is on.	
211	D3		*Cursor:* Column cursor is on in screen line.	
212	D4		*Flag:* Quote mode 0 =off, 1 =on.	
213	D5		Current screen line length (21, 43, 65, 87).	
214	D6		*Cursor:* Current physical screen line cursor on; to change the position of the cursor you must alter 202, 210, 211, and 214.	
215	D7		ASCII value of last key pressed. *Tape:* Most recent dipole.	
216	D8		Number of outstanding inserts; POKE 216,0 to turn off insert mode.	
217	D9	L =24	Screen line link table. If screen is at 7680, then link bytes are 158; continuation screen line is 30. If screen is at 4096 then 144, continuation =16. If 6144 =152/15, etc., see 648.	158
241	F1	L =2	Screen line link table temporary line index.	
243	F3	L =2	*Pointer:* Current screen character color nybble; 244 can be used as a color page number after a PRINT"{HOME}" is done.	0-506, 150/151
245	F5	L =2	*Pointer:* Which keyboard table being used of four starting at $EC5E, see 655.	$EC5E
247	F7	L =2	*RS-232:* Pointer to receiver buffer base location.	
249	F9	L =2	*RS-232:* Pointer to transmit buffer base location; each of 2 buffers is 256 bytes, allocated from the top of memory pointer down. You should OPEN RS-232 (device 2) before strings or arrays are used.	

251*	FB	L=3	Unused page 0 space.
254	FE		STOP key test pattern for $F770 to compare to value in 145.
255	FF		Stack pointer points here when stack is full.

Stack Area
00256-00511 $0100-$01FF L=256

BASIC, Kernal, and processor all use this Last In, First Out (LIFO) stack. Built from $1FF down to $100. Size therefore limits nesting to 127 levels of subroutines in ML. Much less in BASIC.
"OUT OF MEMORY" is received if stack fills up.
Example of BASIC use of stack for FOR keyword: Value of $81, pointer to variable, 5-byte STEP value, sign, 5-byte TO value, 2-byte RETURN line number, pointer to loop RETURN point = 18 bytes.

256	100	L=10	Temporary floating point to ASCII work area.
256	100	L=64	Temporary tape error log of READ errors, top of this stack at 320.

BASIC and Kernal Working Storage
00512-00827 $0200-$033B L=316

512	200	L=89	BASIC screen editor input buffer, tokenization is done here for immediate or numbered line statements, 0 marks end of BASIC line.
601	259	L=10	Logical file number table (file number greater than 127 causes line feed after carriage return).
611	263	L=10	Device number number table in same order as 601. See 184-186 for current I/O table entries.

621	26D	L=10	Secondary address table in same order as 601.	
631	277	L=10	Keyboard buffer (see 198) filled by IRQ interrupt routines.	
641	281	L=2	*Pointer:* Start of memory.	
643	283	L=2	*Pointer:* End of memory.	
645	285		*Serial:* Timeout flag.	
646	286		Contents of current color nybble (see 243)	6
647	287		*Cursor:* Original color at this screen location (the character's or screen color).	1
648	288		Screen Memory page. If screen is at 7680, then this is 30. . If at 4096 then 16, at 6144 =24, 5632 =22.	30
649	289		Maximum number of characters that the keyboard buffer at 631 will hold.	10
650	28A		Key repeat flags: 0 =norm 127 =none 128 =all.	0
651	28B		Delay before first repeat of key held down.	3
652	28C		Delay between following repeats of key.	16
653	28D		Flag for SHIFT/CONTROL keys: 1 =SHIFT, 2 =LOGO, 4 =CTRL, and in combination.	0
654	28E		Last SHIFT pattern, same as above.	
655	28F	L=2	*Pointer:* Keyboard table setup routine, controls which keyboard table used based on shift pattern (see 245).	$EBDC
657	291		*Flag:* Shift keys: 0 =Enabled, 128 =Disabled. See 36869 for programmed shift or PRINT CHR$ (14) for lower-case, 142 for upper. CHR$ (8) and 9 can also be printed to disable and enable character set2 (LC/UC).	0
658	292		*Flag:* Screen scroll enabled =0.	0

659 293

RS-232: Pseudo 6551 control register.
(The VIC emulates a 6551 UART chip with software.)
To set: OPEN N,2,0,CHR$(X) + CHR$(Y) where X is the value for here and Y for 660 ($294). N is file number: 128-up causes linefeed and carriage return.

Bit(s)	Usage	Values
7	stop bits	0 =1 Stop Bit, 1 =2 Stop Bits
6 =5	word length	00 =8, 01 =7, 10 =6, 11 =5
4	unused	
3-0	baud rate	0000 =user 0001 =50 0010 =75 0011 =110 0100 =134.5 0101 =150 0110 =300 0111 =600 1000 =1200 1001 =1800 1010 =2400

660 294

RS-232: Pseudo 6551 command register.
(The VIC emulates a 6551 UART chip with software.)

Bit(s)	Usage	Values
7-5	parity:	XX0 = Disabled 001 =Odd 011 =Even 101 =Mark Xmit 111 =Space Xmit
4	duplex:	0 =Full 1 =Half
3-1	unused	
0	handshaking:	0 =3line 1 =Xline.

661 295 L =2

RS-232: Nonstandard bit timing.

663	297		*RS-232:* Status register. Bit Meaning 7 BREAK detected 6 DSR missing 5 Unused 4 Clear To Send missing 3 Unused 2 Receive buffer overrun 1 Framing error 0 Parity error.	
664	298		*RS-232:* Number of bits to be sent/received.	
665	299	L=2	*RS-232:* System clock divided by baud rate.	
667	29B		*RS-232:* INDEX: End of receive First In, First Out (FIFO) buffer.	
668	29C		*RS-232:* INDEX: Start receive FIFO buffer.	
669	29D		*RS-232:* INDEX: Start transmit FIFO buffer.	
670	29E		*RS-232:* INDEX: End of transmit FIFO buffer.	
671	29F	L=2	Save area for IRQ vector during tape I/O (tape I/O skips update of clock, TI, TI$).	
673	2A1	L=95	User program indirect link addresses.	
768	300	L=2	*Link:* Error message (user ML could intercept errors by changing this link; error message number is in the 6502 X register).	$C43A
770	302	L=2	*Link:* BASIC warm start (warm start goes back to the main BASIC command handling routine).	$C483
772	304	L=2	*Link:* Crunch BASIC into tokens.	$C57C
774	306	L=2	*Link:* Print detokenized BASIC keywords.	$C71A
776	308	L=2	*Link:* Execute new BASIC line.	$C7E4
778	30A	L=2	*Link:* Arithmetic symbol evaluation.	$CE86
780	30C	L=4	SYS temporary storage for 6502 registers (A,X,Y,SR).	
784	310	L=4	????? unknown ????? (observed to be zeros).	

788	314	L=2	*Vector:* IRQ interrupt entry. This is where you would put the address of your own ML IRQ routine to be executed every jiffy. SAVE and RESTORE A, X, and Y registers, then exit to $EABF. POKE 788,194 to disable RUN/STOP, TI, TI$. POKE 788,191 to reenable them (see 808 below to disable only STOP key).	$EABF
790	316	L=2	*Vector:* BREAK interrupt entry.	$FED2
792	318	L=2	*Vector:* NMI interrupt entry.	$FEAD
794	31A	L=2	*Vector:* OPEN logical file.	$F40A
796	31C	L=2	*Vector:* CLOSE logical file.	$F34A
798	31E	L=2	*Vector:* Set INPUT device.	$F2C7
800	320	L=2	*Vector:* Set OUTPUT device.	$F309
802	322	L=2	*Vector:* Reset default I/O.	$F3F3
804	324	L=2	*Vector:* INPUT from device.	$F20E
806	326	L=2	*Vector:* OUTPUT to device.	$F27A
808	328	L=2	*Vector:* Test STOP key. POKE 808,114 to disable, 112 to enable.	$F770
810	32A	L=2	*Vector:* GET from keyboard.	$F1F5
812	32C	L=2	*Vector:* CLOSE all files (abort).	$F3EF
814	32E	L=2	*Vector:* BREAK routine.	$FED2
816	330	L=2	*Vector:* LOAD from device.	$F549
818	332	L=2	*Vector:* SAVE to device.	$F685
820	334	L=8	????? unknown ????? (observed to be zeros).	

Cassette Tape Buffer
828-1023 $33C-$3FF L=195

Note: During BASIC SAVE or LOAD or ML SAVE or LOAD, this area is only used for a header. SAVE/LOAD data is not blocked, but is saved as one large block of memory from where 172,173 points up to where 174,175 points.

As Used for Tape Header I/O

828	33C		*Tape:* After OPEN,01 = PGM, else data tape.
829	33D	L=2	*Tape:* Starting address of program.
831	33F	L=2	*Tape:* Ending address of program.

833	341	L=187	*Tape:* File name of last tape file SAVEd/LOADed. Overlaid by any later BASIC program tape I/O (only first 16 characters are shown in FOUND message), 187,188 point to current file name in memory.
1021	3FD	L=3	?? unused ??

As Used for Tape I/O From BASIC Program

828	33C	L=192	*Tape:* Block buffer for PRINT/INPUT/GET. 178,179 point to this buffer, 166 contains number of characters in it.

3K Expansion RAM
1024-4095 $400-$FFF L=3072

1024	400	L=3072	If 3K Expanded (only) VIC, BASIC program, strings, variables, and arrays start here. (If 5K VIC, they start at 4096 $1000. If 13K+ VIC, they start at 4608 $1200.) 43,44 point to begin / 55,56 to end. If 8K or more expansion is used, this area is not addressed by BASIC, and you may use it to POKE DATA into or for ML routines.

4K Built-In RAM
4096-8191 $1000-$1FFF L=4096

4096-07679	1000-1DFF	L=3584	On an unexpanded VIC, BASIC program, variables, strings, and arrays are stored in this area. (If 8K VIC, they start at 1024 $400. If 13K+ VIC, they start at 4608 $1200.) 43,44 point to beginning of BASIC; 55,56 point to end. In any case, user can change the BASIC area pointers.

4096-4607	1000-11FF	L=512	Screen RAM on 13K+ VIC, contains index values into character tables for 23 lines of 22 characters. If bit 7 is on, then character is in inverse video. 8 bytes of character table form a single character (16 bytes if double-size is in effect; see 36867). If 5K or 8K VIC, screen RAM is at 7680 ($1E00).
7168-7679	1C00-1DFF	L=512	May contain 64 user characters on an unexpanded or 8K VIC, if area taken from BASIC, protected, and 36869 set to 255.
4608-????	1200-????	L=????	On a 13K+ VIC, BASIC program, variables, strings, and arrays are stored in this area. (If 5K VIC, they start at 4096 $1000. If 8K VIC, they start at 1024 $400.) 43,44 point to beginning of BASIC; 55,56 point to end.
7680-8191	1E00-1FFF	L=512	Screen RAM on unexpanded or 8K VIC, contains index values into character tables for 23 lines of 22 characters. If bit 7 is on, then character is in inverse video. 8 bytes of character table form a single character (16 bytes if double-size is in effect; see 36867). If 13K+ VIC, screen RAM moves to 4096 ($1000). See 32768, 36866, 36867.

8K Expansion RAM/ROM
8192-16383 $2000-$3FFF L=8192

RAM Expansion Block 1.

8K Expansion RAM/ROM
16384-24575 $4000-$5FFF L=8192

RAM Expansion Block 2.

8K Expansion RAM/ROM
24576-32767 $6000-$7FFF L =8192

RAM Expansion Block 3. Programmer's aid, some ML monitors, word processors, and games will use this area. Autostart is not done for this area; only 40960 ($A000) resident cartridges autostart. Some cartridges reside at $A000 and also use this area.

Can't SAVE to Tape Above Here!

The Kernal even keeps monitors from saving to tape past here.

Character Sets ROM
32768-36863 $8000-$8FFF L =4096

Note: 8 bytes form 1 character, each of the four maps holds 128 characters, giving a grand total of 512 possible characters. See also 36869, 36879, and 657.

Character set 1

32768	8000	L = 1024	Uppercase (0-63), graphics (64-127).
33792	8400	L = 1024	Reversed uppercase, reversed graphics. Select this set with CNTL +REV ON.

Character set 2

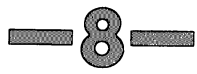
34816	8800	L = 1024	Lowercase (0-63), uppercase (64-127).
35840	8C00	L = 1024	Reversed lowercase and uppercase. Select this with CNTL +REV ON.

The 6561 VIC Chip Registers
36864-36879 9000-900F L =16

36864	9000			5
		bit 7	Interlace scan (for overlays); try turning on to see effect. Some TV sets may require turning this on to correct "jitters."	0
		bits 0-6	Left frame origin on screen 0-127, adding 2 moves frame 1 character right. If 0, puts column 15 on left edge.	5
36865	9001		Top frame origin on screen 0-255; adding 4 moves down 1 line. 255 to blank.	25

36866	9002			150
		bit 7	Bit 9 of 14 bit screen address	128
		bits 6-0	Number of screen columns 1-27. By reducing lines, up to 27 columns can be specified, but editor won't handle lines. Columns times lines can never exceed 506.	22
36867	9003			46
		bit 7	Raster beam location bit 0.	
		bits 6-1	Number of screen lines 1-32 (times 2). By reducing columns, up to 32 lines can be specified, but editor won't handle lines. Columns times lines can never exceed 506.	46
		bit 0	8 x 8 or 16 x 8 character size. Specify 16 x 8 to bitmap screen. Bottom of frame drops out of sight. You may not be able to bitmap the entire screen. You will need to have a custom character set that corresponds to the size of the bitmapped screen. Each 16 bytes builds 1 character, 16 high by 8 across.	
36868	9004		Raster beam location bits 8-1. A light pen on the VIC senses this location; see 36870, 36871.	
36869	9005			240
		bits 7-4	Bits 13-10 of screen address	240
		bits 3-0	Bits 13-10 of character set address 0 =U/C +graph 1 =revU/C + graph 2 =L/C +U/C 3 =revL/C +U/C 12 =4096 13 =5120 14 =6144 15 =user 64 CHARS at 7168 POKE 36869,242 to switch to lowercase. POKE 36869,240 to go back to normal. Character sets must start on 1024 byte boundary.	0

• 36870	9006		Light pen horizontal screen location. You may need to debounce this value as it is very sensitive to imperceptible movement.	0
• 36871	9007		Light pen vertical screen location. You may need to debounce this value as it is very sensitive to imperceptible movement.	0
• 36872	9008		Paddle-X value : right =0, left =255.	255
• 36873	9009		Paddle-Y value : right =0, left =255.	255
• 36874	900A		*SOUND:* Low tone 128-255 (softest voice); you can turn off bit 7 to silence it.	
• 36875	900B		*SOUND:* Medium tone 128-255; you can turn off bit 7 to silence it.	0
• 36876	900C		*SOUND:* High tone 128-255 (sharpest voice); you can turn off bit 7 to silence it.	0
• 36877	900D		*SOUND:* Noise tone 128-255; you can turn off bit 7 to silence it.	0
• 36878	900E			0
		bits 7-4	Auxiliary color for multicolor mode	
		bits 3-0	*SOUND:* Volume 0-15.	
• 36879	900F			27
		bits 7-4	Background color	16
		bit 3	If off, character color becomes screen color and vice versa for every character (gives patchwork background effect).	4
		bits 2-0	Border color.	3
• 36880	9010	L =256	As a result of incomplete address decoding for the VIC chip, locations 36880-37135 are duplications of 36864-36879 and have no particular use.	



6522 VIA Number One
37136-37151 $9110-$911F L=16

This VIA Controls/Senses a Nonmaskable Interrupt — NMI

37136 9110 Port B I/O Register (parallel user port/RS-232): this port controls handshake lines CB1 and CB2. The voltages on this port are not the standard -12 to +12 volts, only 0 to +5 volts. OK for short cables. 247

Bit	Pin	RS-232 Usage	Direction	DB25 Pin
PB7	L	Data Set Ready (If PB7 used as OUTPUT, timer1 can clock it)	I	21
PB6	K	Clear To Send (If PB6 used as INPUT, timer can clock it)		
PB5	J	Unused		
PB4	H	Data Carrier Detect		
PB3	F	Ring Indicator		
PB2	E	Data Terminal Ready	O	5
PB1	D	Request To Send (VIC always keeps this on)	O	6
PB0	C	Received Data	I	2
CB1	B	Interrupt for Received Data		
CB2	M	Transmitted Data (CB1, CB2 can act as serial port when controlled by the shift register at 37146.)	O	3

This port could also be used for second joystick: PB1 =JOY3,

			PB2 =JOY0, PB3 =JOY1, PB4 =JOY2, PB5 =FIRE.	
37137	9111		Port A I/O Register (serial/game ports): this port controls hand-shake lines CA1 and CA2	124
			Bit Pin Usage	
			PA7 S3 Serial attention in	
			PA6 F6 Sense tape button down (tape pins 6/F): 1 =none down, 0 =some down	
			PA5 G6 light pen / FIRE button	
			PA4 G3 JOY2	
			PA3 G2 JOY1	
			PA2 G1 JOY0	
			PA1 S5 Serial data in	
			PA0 S4 Serial clock in	
			CA1 C INPUT interrupt, RESTORE key	
			CA2 D I/O line.	
37138	9112		Data Direction Register for Port B: 0 =in, 1 =out.	0
37139	9113		Data Direction Register for Port A: 0 =in, 1 =out.	128
37140	9114	L =2	Timer 1 low, high count (RS-232/ user port output speed) Timer can time up to 0.0591 second, then interrupt.	
37142	9116	L =2	Timer 1 low, high latch (tape write timing).	
37144	9118		Timer 2 low count/latch (RS-232/ user port input speed).	
37145	9119		Timer 2 high count (RS-232/user port input speed).	
37146	911A		Shift Register (parallel-to-serial and serial-to-parallel conversion done through here with CB1, CB2).	0
37147	911B		Auxiliary Register.	64
		bit 7-6	Timer 1 control 7 =0 No PB7 timing 7 =1 Time PB7 6 =0 One-shot timing 6 =1 Free running timer	
		bit 5	Timer 2 control 5 =0 One-shot timing 5 =1 Time pulses	

		bit 4-2 Shift register control 000 = Shift register disabled 010 = System clock: data in CB2, pulse out on CB1 100 = Timer 2: data in CB2, pulse out on CB1	
		bit 1 PORT B latch enable	
		bit 0 PORT A latch enable.	
37148	911C	Peripheral Handshaking Control Register	254
		bit 7-5 CB2, RS-232 sent (Shift register serial I/O) (Interrupt input or peripheral output)	
		bit 4 CB1, Interrupt for received data (Output for shift register pulses) (High-to-low or low-to-high interrupt)	
		bit 3-1 CA2, Tape Motor: 111 = ON, 110 = OFF (Interrupt input or peripheral output)	
		bit 0 CA1, Interrupt if high/low (restore to high) (High-to-low or low-to-high interrupt).	
37149	911D	Interrupt Flag Register	0
		bit 7 NMI occurred: one of below conditions:	
		bit 6 Timer 1 interrupt	
		bit 5 Timer 2 interrupt	
		bit 4 CB1 RS-232 input interrupt	
		bit 3 CB2 RS-232 output interrupt	
		bit 2 Shift register interrupt	
		bit 1 CA1 RESTORE key was pressed	
		bit 0 CA2 data line interrupt.	
37150	911E	Interrupt Enable Register	130
		bit 7 NMI 1 = enable 6-0 below, 0 = disable	
		bit 6 Timer 1 interrupt can happen	
		bit 5 Timer 2 interrupt can happen	
		bit 4 CB1 interrupt can happen	
		bit 3 CB2 interrupt can happen	
		bit 2 Shift register interrupt can happen	

bit 1 CA1 (RESTORE) interrupt can happen
bit 0 CA2 interrupt can happen.

37151 911F Port A I/O Register. Set the same as 37137 ($9111), but has no CA1 or CA2 control capabilities. (CA1 interrupt flag is cleared by reading from here.) 124

Bit	Pin	Usage
PA7	S3	Serial attention in
PA6	F6	Sense tape button down (tape pins 6/F) 1 =none down. 0 =some down
PA5	G6	Light pen / fire button
PA4	G3	JOY2
PA3	G2	JOY1
PA2	G1	JOY0
PA1	S5	Serial data in
PA0	S4	Serial clock in.

6522 VIA Number Two
37152-37167 $9120-$912F L =16

This VIA generates the 1/60 second interrupt request — IRQ for realtime clocking, keyboard scanning. Realtime clock loses time during tape I/O.

37152 9120 Port B I/O Register (primarily keyboard scan); this port controls handshake lines CB1 and CB2. 247
PB7-0 Keyboard column scan select
PB7 JOY3 (game port pin 4)
PB3 Tape write line, tape port pin E/5 left on for STOP key scan
CB1 Serial service request in
CB2 Serial data out.

37153 9121 Port A I/O Register (keyboard scanning); this port controls handshake lines CA1 and CA2. 255
PA7-0 Keyboard row scan select
CA1 Tape I/O
CA2 Serial clock out.

37154 9122 Data Direction Register for Port B: 0 =IN, 1 =OUT 255
POKE 37154,127 to read JOY3, then restore to 255.

37155	9123		Data Direction Register for Port A: 0 =IN, 1 =OUT.	0
37156	9124	L =2	Timer 1 low, high latch (tape read timing).	
37158	9126	L =2	Timer 1 low, high count (keyboard interrupt) Timer 1 used for 1/60 second IRQ interrupt.	
37160	9128		Timer 2 low count/latch (serial timeout, tape R/W).	
37161	9129		Timer 2 high count/latch (serial time-out, tape R/W).	
37162	912A		Shift Register (parallel-to-serial/ serial-to-parallel conversion done through here with CB1, CB2).	0
37163	912B		Auxiliary Register	64
		bit 7-6	Timer 1 control 7 =0 no PB7 timing 7 =1 time PB7 6 =0 one-shot timing 6 =1 free-running timer	
		bit 5	Timer 2 control 5 =0 one-shot timing 5 =1 time pulses	
		bit 4-2	Shift register control 000 =Shift register disabled 010 =System clock:data in CB2, pulse out on CB1 100 =Timer 2:data in CB2, pulse out on CB1	
		bit 1	Port B latch enable	
		bit 0	Port A latch enable.	
37164	912C		Peripheral Handshaking Control Register	222
		bit 7-5	CB2, Serial data out (Shift register serial I/O) (Interrupt input or peripheral output)	
		bit 4	CB1, Serial service request in (Output for shift register pulses) (High-to-low or low-to-high interrupt)	
		bit 3-1	CA2, Serial clock out (Interrupt input or peripheral output)	

		bit 0	CA1, Interrupt if high/low (tape I/O) (High-to-low or low-to-high interrupt).	
37165	912D		Interrupt Flag Register	32
		bit 7	IRQ occurred: One of below conditions	
		bit 6	Timer 1 interrupt	
		bit 5	Timer 2 interrupt	
		bit 4	CB1 Serial service request interrupt	
		bit 3	CB2 Serial data output interrupt	
		bit 2	Shift register interrupt	
		bit 1	CA1 Tape I/O interrupt	
		bit 0	CA2 Serial clock out interrupt.	
37166	912E		Interrupt Enable Register	192
		bit 7	IRQ 1 =enable 6-0 below, 0 =disable	
		bit 6	Timer 1 interrupt can happen	
		bit 5	Timer 2 interrupt can happen	
		bit 4	CB1 Interrupt can happen	
		bit 3	CB2 Interrupt can happen	
		bit 2	Shift register interrupt can happen	
		bit 1	CA1 (tape I/O) Interrupt can happen	
		bit 0	CA2 Interrupt can happen.	
37167	912F		Port A I/O Register Set the same as 37137 ($9121), but has no CA1 or CA2 control capabilities. (CA1 interrupt flag is cleared by reading from here.) PA7-0 Keyboard row scan select CA1 Tape I/O CA2 Serial clock out.	255
37168	9130	L =720	As a result of incomplete address decoding for the VIA chips, locations 37168-37887 are duplications of 37136-37167 and have no particular use.	

Screen Color Maps
37888-38911 $9400-$97FF L =1024

Contents of 36866 and memory size affect which segment used.

37888	9400	L=512	Color map if 13K or more VIC.
38400	9600	L=512	Color map if unexpanded or 8K VIC.
	bits 7-4		Used by VIC
	bit 3		Multicolor=1 or normal=0
	bits 2-0		Color value 0-7

I/O Block 2
38912-39935 $9800-$9BFF L=1024

Memory mapped I/O (unused)

I/O Block 3
39936-40959 $9C00-$9FFF L=1024

Memory mapped I/O (unused)

8K Expansion RAM/ROM
40960-49151 A000-BFFF L=8192

RAM BLOCK 5

RAM Expansion Block 4. This block is primarily used for autostart cartridges such as games and other cartridge-based software. The cartridge is not required to be autostarting. The following must be present to activate the autostart code in the Kernal:

40960	A000	L=2	*Vector:* Initialization and reset routines.
40962	A002	L=2	*Vector:* NMI (restore) routines.
40964	A004	L=5	A0CBM (with high order bit set in last 3 bytes)

The user may add expansion RAM in this area, although BASIC will never see it. You can use this RAM for ML code or POKE data in from BASIC.

The ROM routines of the Super Expander cartridge reside in this area; see "Super Expander Memory Map."

8K BASIC ROM
49152-57343 $C000-$DFFF L=8192

Note that routines may have multiple valid entry points, depending on the function desired.

49152	C000	L=2	*Vector:* Cold start address.	E378
49154	C002	L=2	*Vector:* Warm start address.	E467
49156	C004	L=8	"CBMBASIC".	
49164	C00C		Keyword dispatch vector table, in token order (i.e., END, FOR, NEXT, DATA, INPUT, etc.).	

49234	C052	Function dispatch vector table, in token order (i.e., SGN, INT, ABS, ... STR$, CHR$, LEFT$, etc.).	
49280	C080	Math operators dispatch vector table, in token order (i.e., +, -, *, /, AND, OR, >, =, <, etc.).	
49310	C09E	BASIC keywords table in token order ended with a byte of 0. Last letter of keyword has high order bit on.	
49566	C19E	BASIC messages table. Last letter of message has high order bit on.	
49960	C328	Error message table vector.	$C19E
50021	C365	Miscellaneous Messages: OK, ERROR, READY, BREAK.	
50058	C38A	Scan stack for GOSUB and FOR/NEXT entries.	
50104	C3B8	Open space in memory for new BASIC line.	
50171	C3FB	Check stack depth not exceeded.	
50184	C408	Check memory space for available area.	
50229	C435	*Vector:* OUT OF MEMORY routine.	
50231	C437	Error message handler routine, error number in 6502 register X.	
50281	C469	Break entry point.	
50292	C474	Print READY and GOTO main BASIC loop.	
50304	C480	Main BASIC loop, execute or store BASIC line.	
50332	C49C	Handle new BASIC line.	
50483	C533	Rechain BASIC lines.	
50528	C560	Receive input BASIC line and fill $200.	
50553	C579	Tokenize BASIC line.	
50707	C613	Find the BASIC line from its line number.	
50754	C642	BASIC's NEW, then do CLR.	
50782	C65E	BASIC's CLR.	
50830	C68E	Back up text pointer to start of program.	
50844	C69C	BASIC's LIST.	

50970	C71A	Print detokenized keyword/ function.
51010	C742	BASIC's FOR.
51172	C7E4	Execute the BASIC statement.
51229	C81D	BASIC's RESTORE.
51244	C82C	Break entry, then do STOP.
51247	C82F	BASIC's STOP.
51249	C831	BASIC's END.
51287	C857	BASIC's CONT.
51313	C871	BASIC's RUN.
51331	C883	BASIC's GOSUB.
51360	C8A0	BASIC's GOTO.
51410	C8D2	BASIC's RETURN.
51448	C8F8	BASIC's DATA, i.e., skip to next statement.
51462	C906	Scan for next BASIC statement.
51496	C928	BASIC's IF, and perhaps skip rest of statement.
51515	C93B	BASIC's REM, i.e., skip rest of statement.
51531	C94B	BASIC's ON.
51563	C96B	Get fixed point number.
51621	C9A5	BASIC's LET.
51741	CA1D	Add ASCII digit to ACCUM1.
51756	CA2C	Continue BASIC's LET.
51840	CA80	BASIC's PRINT#.
51846	CA86	BASIC's CMD.
51872	CAA0	BASIC's PRINT.
51998	CB1E	Print message from table at 49566, indexed by 6502 X and A registers (end of message is 0 byte).
52027	CB3B	Print format characters: SPACE, CURSOR RIGHT.
52045	CB4D	Bad input routines.
52091	CB7B	BASIC's GET.
52133	CBA5	BASIC's INPUT#.
52159	CBBF	BASIC's INPUT.
52217	CBF9	PROMPT and INPUT.
52230	CC06	BASIC's READ, also common routines for GET and INPUT.
52476	CCFC	Input error messages: ?EXTRA IGNORED, ?REDO FROM START.
52510	CD1E	BASIC's NEXT.
52600	CD78	Type-match checking.

52638	CD9E	Evaluate formula using $7A, $7B, stack, ACCUMs.
52870	CE86	Evaluate expression.
52904	CEA8	Constant pi.
52977	CEF1	Evaluate within parentheses.
52983	CEF7	Check for ')'.
52986	CEFA	Check for '('.
52991	CEFF	Check for ', '.
53000	CF08	Print SYNTAX ERROR message.
53005	CF0D	Set up function for later evaluation.
53012	CF14	Check range of variable.
53032	CF28	Search for variable.
53159	CFA7	Set up FN reference.
53222	CFE6	BASIC's OR.
53225	CFE9	BASIC's AND.
53270	D016	Compare numerics or strings.
53377	D081	BASIC's DIM.
53387	D08B	Locate variable, sets $44, $45, $46, $47, $D, $E.
53523	D113	Check if ASCII character is alphabetic.
53533	D11D	Create new variable.
53652	D194	Array pointer subroutine.
53669	D1A5	Value 32768 in floating point.
53764	D1AA	Evaluate for positive integer.
53682	D1B2	Floating point-to-integer conversion.
53713	D1D1	Find or create an array.
53829	D245	Print BAD SUBSCRIPT message.
53832	D248	Print ILLEGAL QUANTITY message.
54092	D34C	Compute array subscript size.
54141	D37D	BASIC's FRE.
54164	D394	Integer-to-floating point conversion.
54174	D39E	BASIC's POS.
54182	D3A6	Check if immediate statement legal or illegal.
54195	D3B3	BASIC's DEF.
54241	D3E1	Check FN syntax.
54260	D3F4	BASIC's FN.
54373	D465	BASIC's STR$.
54389	D475	Calculate string vector.
54407	D487	Scan and set up string.

54516	D4F4	Make room for string, build string vector.
54566	D526	Garbage collection.
54717	D5BD	Check for most eligible string to collect.
54790	D606	Garbage collect a string.
54845	D63D	Concatenate string.
54906	D67A	Build string into memory.
54947	D6A3	Discard an unwanted string.
55003	D6DB	Clean up the string descriptor stack.
55020	D6EC	BASIC's CHR$.
55040	D700	BASIC's LEFT$.
55084	D72C	BASIC's RIGHT$.
55095	D737	BASIC's MID$.
55137	D761	Pull string parameters from stack.
55164	D77C	BASIC's LEN.
55170	D782	Exit string-mode.
55179	D78B	BASIC's ASC.
55195	D79B	Input byte parameter.
55213	D7AD	BASIC's VAL.
55275	D7EB	Get parameters for POKE/WAIT.
55287	D7F7	Floating-to-fixed conversion.
55309	D80D	BASIC's PEEK.
55332	D824	BASIC's POKE.
55341	D82D	BASIC's WAIT.
55369	D849	Add 0.5 to ACCUM.
55376	D850	Subtract ACCUM2 from ACCUM1.
55379	D853	BASIC's SUBTRACT.
55402	D86A	BASIC's ADD.
55623	D947	Complement ACCUM1.
55678	D97E	Print OVERFLOW message and exit.
55683	D983	Multiply a byte.
55740	D9BC	Constants for functions.
55786	D9EA	BASIC's LOG.
55848	DA28	Multiply floating point in memory with ACCUM1.
55859	DA33	Multiply ACCUM2 with ACCUM1.
55897	DA59	Multiply-a-bit subroutine.
55948	DA8C	Move memory to ACCUM2.
55991	DAB7	Test and adjust ACCUM1 and ACCUM2.

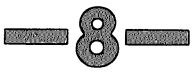
56020	DAD4	Handle underflow/overflow.
56034	DAE2	Multiply ACCUM1 by 10.
56057	DAF9	+10 in floating point.
56062	DAFE	Divide ACCUM1 by 10.
56082	DB12	Divide ACCUM2 by ACCUM1.
56226	BDA2	Move memory to ACCUM1.
56263	DBC7	Move ACCUM1 to memory.
56316	DBFC	Move ACCUM2 to ACCUM1.
56332	DC0C	Move ACCUM1 to ACCUM2, with rounding.
56335	DC0F	Move ACCUM1 to ACCUM2, without rounding.
56347	DC1B	Round ACCUM1.
56363	DC2B	Get sign from ACCUM1.
56377	DC39	BASIC's SGN.
56408	DC58	BASIC's ABS.
56411	DC5B	Compare ACCUM1 to memory.
56475	DC9B	Floating point-to-fixed conversion.
56524	DCCC	BASIC's INT.
56563	DCF3	String-to-floating point conversion.
56702	DD7E	Get ASCII digit.
56755	DDB3	String to floating point conversion constants.
56768	DDC0	Print message IN.
56781	DDCD	Decimal output routine, print a number.
56797	DDDD	Convert floating point to TI$ or ASCII.
57110	DF16	Decimal constants for conversion.
57146	DF3A	TI constants for conversion.
57201	DF71	BASIC's SQR.
57211	DF7B	BASIC's POWER.
57268	DFB4	BASIC's NOT.
57325	DFED	BASIC's EXP.

8K Kernal ROM
57344-65535 $E000-$FFFF L=8192

BASIC routines spill over into this ROM; note that routines may have multiple valid entry points, depending on the function desired.

57408	E040	Function series evaluation subroutine 1.

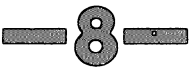
57430	E056	Function series evaluation subroutine 2.
57482	E08A	Manipulate constants for RND.
57492	E094	BASIC's RND.
57590	E0F6	Kernal patch routines.
57639	E127	BASIC's SYS.
57683	E153	BASIC's SAVE.
57698	E162	BASIC's VERIFY.
57701	E165	BASIC's LOAD.
57787	E1BB	BASIC's OPEN.
57796	E1C4	BASIC's CLOSE.
57809	E1D1	Handle parameters for LOAD and SAVE.
57859	E203	Check default parameters.
57867	E20B	Check for comma.
57878	E216	Handle parameters for OPEN and CLOSE.
57953	E261	BASIC's COS.
57960	E268	BASIC's SIN.
58033	E2B1	BASIC's TAN.
58077	E2DD	Trig evaluation constants: pi/2, .25, etc.
58123	E30B	BASIC's ATN.
58171	E33B	Constants for ATN evaluation.
58232	E378	Cold start BASIC.
58247	E387	CHRGET routine to be copied to 115-138 ($73-8A).
58276	E3A4	Initialize BASIC: Restore CHRGET and page zero pointers.
58409	E429	Power-up message "bytes free", "**** CBM BASIC V2 ****".
58447	E44F	Vectors to copy to $300.
58459	E45B	Initialize vectors.
58471	E467	Warm restart.
58486	E476	Program patch area.
58528	E4A0	*Serial:* Output a 1.
58537	E4A9	*Serial:* Output a 0.
58546	E4B2	*Serial:* Get input and clock.
58549	E4B5	Restore vectors.
58556	E4BC	Program patch area.
58624	E500	Return address of 6522.
58629	E505	Set screen limits, max lines, columns.
58634	E50A	Read/Plot cursor location.



58648	E518	Initialize I/O.
58650	E51A	Initialize/restore VIC chip defaults.
58700	E54C	Normalize screen.
58719	E55F	Clear screen.
58753	E581	HOME cursor.
58759	E587	Set screen line link table pointers.
58805	E5B5	NMI entry for restore key.
58811	E5BB	Set I/O defaults.
58819	E5C3	Set VIC chip defaults.
58831	E5CF	Get character from keyboard queue.
58959	E64F	Input from queue until carriage return.
59064	E6B8	Quote mark test.
59077	E6C5	Set up screen print.
59114	E6EA	Advance cursor on screen.
59157	E715	Retreat cursor on screen.
59181	E72D	Back up into previous screen line.
59202	E742	Output a character to the screen.
59392	E800	Handle shift keys.
59587	E8C3	Go to next screen line.
59608	E8D8	Handle return key.
59624	E8E8	Check for screen line index pointer decrement.
59642	E8FA	Check for screen line index pointer increment.
59666	E912	Set color code.
59681	E921	Color code table.
59689	E929	Screen code to ASCII code conversion.
59765	E975	Scroll screen.
59886	E9EE	Open space on screen.
59990	EA56	Move screen line.
60014	EA6E	Sync color transfer.
60030	EA7E	Set start of screen line.
60045	EA8D	Clear screen line.
60065	EAA1	Print to screen.
60074	EAAA	Store on screen.
60082	EAB2	Sync color to character.
60095	EABF	IRQ interrupt entry pointed to by 788,789 ($314,$315).
60190	EB1E	Scan keyboard using VIA2.

60830	EBDC	Decode keyboard from 203 ($CB) to ASCII in keyboard queue.
60416	EC00	Set keyboard mode.
60486	EC46	Keyboard vectors.
60510	EC5E	Keyboard matrix.
60705	ED21	Graphics/text control.
60720	ED30	Set graphics mode.
60763	ED5B	Wrap up screen line.
60778	ED6A	Shifted key matrix.
60835	EDA3	Control key matrix.
60900	EDE4	Initial values for VIC regs.
60916	EDF4	"LOAD".
60921	EDF9	"RUN".
60925	EDFD	Screen line adds low.
60948	EE14	*Serial:* Send talk.
60951	EE17	*Serial:* Command serial to listen.
60956	EE1C	*Serial:* Send control character.
61001	EE49	*Serial:* Send to serial.
61111	EEB7	*Serial:* Timeout on serial.
61120	EEC0	*Serial:* Send secondary address after listen.
61125	EEC5	*Serial:* Clear attention.
61134	EECE	*Serial:* Send secondary address after talk.
61156	EEE4	*Serial:* Send serial deferred.
61174	EEF6	*Serial:* Send untalk.
61188	EF04	*Serial:* Command serial to unlisten.
61209	EF19	*Serial:* Receive byte from serial.
61316	EF84	*Serial:* Set clock line on.
61525	EF8D	*Serial:* Set clock line off.
61334	EF96	Delay 1 millisecond.
61347	EFA3	*RS-232:* Send (NMI continuation routine).
61375	EFBF	*RS-232:* Calculate parity.
61416	EFE8	*RS-232:* Count stop bits.
61422	EFEE	*RS-232:* Send new byte.
61435	EFFB	*RS-232:* Set up to send next byte.
61462	F016	*RS-232:* Error or quit.
61479	F027	*RS-232:* Compute bit count.
61494	F036	*RS-232:* Receive (NMI).
61504	F040	*RS-232:* Calculate parity.
61510	F046	*RS-232:* Shift data bit in.
61515	F04B	*RS-232:* Store byte in buffer.
61531	F05B	*RS-232:* Set up to receive.

61544	F068	*RS-232:* Receiver start bit checking.
61551	F06F	*RS-232:* Receiver put data in buffer.
61551	F06F	*RS-232:* Receiver parity checking.
61588	F094	*RS-232:* Receiver calculate parity.
61597	F09D	*RS-232:* Receiver parity error.
61599	F09F	*RS-232:* Receiver errors repeated.
61602	F0A2	*RS-232:* Receiver overrun error.
61605	F0A5	*RS-232:* Receiver break error.
61608	F0A8	*RS-232:* Receiver frame error.
61625	F0B9	*RS-232:* Bad device.
61628	F0BC	*RS-232:* File to RS-232 buffer.
61636	F0C4	*RS-232:* Check Data Set Ready, Request To Send.
61645	F0CD	*RS-232:* Check request to send (low active input).
61652	F0D4	*RS-232:* Wait for Clear To Send to turn off.
61657	F0D9	*RS-232:* Turn on request to send.
61665	F0E1	*RS-232:* Wait for clear to send to turn on.
61677	F0ED	*RS-232:* Send character to RS-232 from buffer.
61692	F0FC	*RS-232:* Set up output.
61698	F102	*RS-232:* Set up first byte out.
61710	F10E	*RS-232:* Set up VIA1 Timer 1 NMI's.
61718	F116	*RS-232:* Input character from RS-232 to buffer.
61730	F122	*RS-232:* Check for Data Set Ready and no Request To Send.
61739	F12B	*RS-232:* Wait for output to be done.
61744	F130	*RS-232:* Turn off Request To Send.
61752	F138	*RS-232:* Wait for data carrier to turn on.
61759	F13F	*RS-232:* Enable VIA1 CB1 for RS-232 input.
61766	F146	*RS-232:* If not 3-line handshaking, see if CB1 needs to be on.



61775	F14F	*RS-232:* Get character from RS-232 buffer.
61788	F15C	*RS-232:* Receiver always runs.
61792	F160	*RS-232:* Check serial idle, to protect from RS-232.
61812	F174	Handle Kernal messages.
61922	F1E2	Print message if immediate.
61941	F1F5	Get from device.
61966	F20E	Input from device, up to 88 characters.
62074	F27A	Output to device.
62151	F2C7	Set device for input.
62217	F309	Set device for output.
62282	F34A	Close logical file.
62415	F3CF	Find file characters.
62431	F3DF	Set file characteristics.
62447	F3EF	Close all open files.
62451	F3F3	Reset default I/O, reset devices.
62474	F40A	OPEN a logical file.
62613	F495	Send secondary address.
62663	F4C7	*RS-232:* OPEN RS-232 device.
62786	F542	LOAD program.
62793	F549	LOAD program to RAM from device designated in 186 ($BA), or verify.
63047	F647	Print "SEARCHING...."
63065	F659	Print file name.
63082	F66A	Print "LOADING/VERIFYING."
63093	F675	SAVE RAM to device designated in 186 ($BA).
63109	F685	SAVE to device.
63272	F728	Print "SAVING...."
63284	F734	Increment realtime clock by 1 jiffy.
63328	F760	Get time.
63335	F767	Set time.
63344	F770	Check for STOP key in 145 ($91), purge queue and channels.
63358	F77E	File error message handler.
63407	F7AF	*Tape:* Find next tape header.
63463	F7E7	*Tape:* Write tape header.
63565	F84D	*Tape:* Get buffer address.
63472	F854	*Tape:* Set buffer start and end.
63591	F867	*Tape:* Find a specified header.
63626	F88A	*Tape:* Bump tape pointer.
63636	F894	*Tape:* Print "PRESS PLAY...."

63659	F8AS	*Tape:* Check tape play/ rewind/forward status.
63671	F8B7	*Tape:* Print "PRESS RECORD..." and check.
63680	F8C0	*Tape:* Initiate tape header read.
63689	F8C9	*Tape:* Read tape load block.
63715	F8E3	*Tape:* Initiate tape header write.
63732	F8F4	*Tape:* Common tape read/write, start tape operation.
63189	F94B	*Tape:* Check tape STOP key.
63837	F59D	*Tape:* Set timing for tape dipole.
63886	F98E	*Tape:* Read bits into buffer (IRQ driven).
64189	FABD	*Tape:* Byte handler.
64173	FAAD	*Tape:* Store characters.
64466	FBD2	*Tape:* Reset pointer.
64475	FBDB	*Tape:* New tape character setup.
64490	FBEA	*Tape:* Toggle tape.
64518	FC06	*Tape:* Data write.
64523	FC0B	*Tape:* Tape write (IRQ driven).
64661	FC95	*Tape:* Leader write (IRQ driven).
64719	FCCF	*Tape:* Restore vectors.
64758	FCF6	*Tape:* Set vector.
64776	FD08	*Tape:* Stop motor.
64785	FD11	*Tape:* Check read/write pointer.
64795	FD1B	*Tape:* Bump read/write pointer.
64802	FD22	Power on restart (checks for autostart). Clear $0-$FF, $200-$3FF, set the pointer to tape buffer, from $400 search up for start and end of RAM, move screen, set top and bottom of memory pointers, set default I/O vectors, initialize the page zero jump table, build CHRGET routine, clear screen, print startup message and bytes free, go to BASIC ($C000).
64831	FD3F	Check $A000 for autostart ROM.
64850	FD52	Restore old I/O vectors.
64855	FD57	Read/set vectored I/O.
64909	FD8D	Initialize system constants.
65009	FDF1	Initialize IRQ vectors.
65017	FDF9	Initialize I/O registers.
65097	FE49	SAVE data name.

65104	FE50	SAVE file details.	
65111	FE57	Read I/O status.	
65126	FE66	Control Kernal messages.	
65135	FE6F	*Serial:* Set timeout value.	
65139	FE73	Read/set top of memory.	
65154	FE82	Read/set bottom of memory.	
65169	FE91	Test memory location.	
65193	FEA9	NMI interrupt entry handler.	
65197	FEAD	NMI interrupt entry.	
65234	FED2	Break interrupt entry.	
65246	FEDE	*RS-232:* NMI RS-232 sequences.	
65366	FF56	Restore 6502 registers and return to interrupt.	
65372	FF5C	*RS-232:* Timing and baud rate tables.	
65394	FF72	IRQ handler, JUMP on $314 vector (IRQ entry) or JUMP on $316 vector (BREAK entry).	
65413	FF85	5 bytes of FF.	

The following have a JMP opcode followed by the vector:

65418	FF8A	*JUMPTO:* Restore old I/O vectors.	$FD52
65421	FF8D	*JUMPTO:* Read/set vectored I/O.	$FD57
65424	FF90	*JUMPTO:* Control Kernal messages.	$FE66
65427	FF93	*JUMPTO:* Send secondary address (after listen).	$EEC0
65430	FF96	*JUMPTO:* Send secondary address (after talk).	$EECE
65433	FF99	*JUMPTO:* Read/set top of memory.	$FE73
65436	FF9C	*JUMPTO:* Read/set bottom of memory.	$FE82
65439	FF9F	*JUMPTO:* Scan keyboard.	$EB1E
65442	FFA2	*JUMPTO:* Set timeout on Serial.	$FE6F
65445	FFA5	*JUMPTO:* Receive byte from Serial.	$EF19
65448	FFA8	*JUMPTO:* Output byte to Serial.	$EEE4
65451	FFAB	*JUMPTO:* Command serial to untalk.	$EEF6
65454	FFAE	*JUMPTO:* Command serial to unlisten.	$EF04
65457	FFB1	*JUMPTO:* Command serial to listen.	$EE17

65460	FFB4	*JUMPTO:* Command serial to talk.	$EE14
65463	FFB7	*JUMPTO:* Read I/O status word.	$FE57
65466	FFBA	*JUMPTO:* Set logical first, second address.	$FE50
65469	FFBD	*JUMPTO:* Set file name.	$FE49
		Following are indirect JMPs off of a $300 vector. The $300 vectors can be set to go to your code.	
65472	FFC0	*JUMPIND:* OPEN file.	$31A
65475	FFC3	*JUMPIND:* CLOSE file (a reg).	$31C
65478	FFC6	*JUMPIND:* OPEN INPUT device; changes GET, INPUT to .X file num device.	$31E
65481	FFC9	*JUMPIND:* OPEN output device; changes print to .X file num device.	$320
65484	FFCC	*JUMPIND:* CLOSE input and output devices; by not using this, multiple devices can listen.	$322
65487	FFCF	*JUMPIND:* INPUT character from device.	$324
65490	FFD2	*JUMPIND:* OUTPUT character to device (a reg).	$326
		The following have a JMP OPCODE followed by the vector:	
65493	FFD5	*JUMPTO:* LOAD/VERIFY.	$F542
65496	FFD8	*JUMPTO:* SAVE RAM to device.	$F675
65499	FFDB	*JUMPTO:* Set realtime clock.	$F767
65502	FFDE	*JUMPTO:* Read realtime clock.	$F760
		Following are indirect JMPS off a $300 vector. The $300 vectors can be set to go to your code.	
65505	FFE1	*JUMPIND:* Test STOP key.	$328
65508	FFE4	*JUMPIND:* Get from device.	$32A
65511	FFE7	*JUMPIND:* Close all files.	$32C
		The following have a JMP OPCODE followed by the vector.	
65514	FFEA	*JUMPTO:* Increment realtime clock.	$F734
65517	FFED	*JUMPTO:* Return X,Y origin of screen.	$E505
65520	FFF0	*JUMPTO:* Read/set X,Y cursor position.	$E50A
65523	FFF3	*JUMPTO:* Return base address of page for I/O devices.	$E500
65526	FFF6	4 bytes of FF.	
65530	FFFA	6502 NMI Initial instruction (med priority).	$FEA9

65532	FFFC	6502 RESET Initial instruction (hi priority).	$FD22
65534	FFFE	6502 IRQ Initial instruction (low priority).	$FF72

—8—

Super Expander Memory Map

Chuan Chee

A map of the significant machine language routines in the VIC Super Expander. You can translate these hexadecimal numbers into decimal, then SYS to them and watch the effects.

General Input/Output Routines

A000-A001 **Vector:** RESET ($A044).
A002-A003 **Vector:** NMI ($A077).
A004-A008 ROM identification ('A0CBM').
A009-A010 **Table:** function key numbers.
A011-A043 **Table:** initial function key definitions.
A044-A076 RESET routine.
A077-A08A NMI routine.
A08B-A0BE Parse KEY (get parameters and check syntax).
A0BF-A131 Display all function key definitions.
A110-A11C Print ' °+chr$(34)' and an optional ' +'.
A11D-A131 Print ' °+chr$(13)' and an optional ' +'.
A132-A135 **Table:** ASCII string for output ('key' backwards).
A136-A13F **Table:** ASCII string for output (' °+chr$(13)' backwards).
A140-A149 **Table:** ASCII string for output (' °+chr$(34)' backwards).
A14A-A17A Delete current function key string (key number in 6502 register X).
A17B-A1B0 Insert string into function key definition area.
A1B1-A1BE Locate function key definition (key number in 6502 register X, return index in register Y).
A1BF-A213 **Table:** new BASIC keywords in ASCII form.
A214-A237 **Table:** vectors corresponding to new BASIC tokens ($CC to $DD).
A238-A2A1 Initialize Kernal vectors, I/O, RAM.
A2A2-A2C1 **Table:** Kernal vectors (L,H).
A2C2-A2C7 Warm start routine.
A2C8-A317 Output a character to device 3 (character in 6502 register A).

A318-A336 End music mode.
A337-A365 Interpret keyboard matrix input.
A366-A369 **Table:** keyboard matrix code for function keys.
A36A-A371 **Table:** conversion pattern for function keys.
A372-A394 IRQ routine.
A395-A3A5 Input a character from any device (device number in $99).
A3A6-A3B3 Output a character to any device (character in 6502 register A, device number in $9A).
A3B4-A3F1 Input each character from keyboard buffer.
A3B4-A3E7 Handle 'RUN' key.
A3E8-A3F1 Handle 'RETURN' key.
A3F2-A3FC Input from device 0.
A3FD-A406 Print an error message in GRAPHIC 0 mode (error token in 6502 register A).
A407-A4B9 Tokenize BASIC source line.
A4BA-A503 Print BASIC tokens in ASCII form.
A504-A529 Start new BASIC statement.
A515-A523 Handle new tokens ($CC to $D6).
A52A-A58A Get and evaluate an expression.
A558-A58A Handle new function tokens ($D7 to $DD).
A58B-A596 **Table:** BASIC vectors for RAM.
A597-A5A4 Change BASIC vectors during RESET.

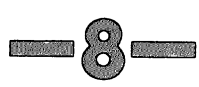
Music Routines

A5A5-A5D0 Save current sound table (address of table in 6502 registers X,Y).
A5D1-A601 IRQ music driver.
A602-A625 **Table:** conversion for note index to frequency.
A626-A6E5 Interpret music mode characters (character in 6502 register A).
A629-A643 Execute 'O' command (default 3).
A644-A65D Execute 'T' command (default 0).
A65E-A674 Execute 'S' command (default 4).
A675-A686 Execute 'V' command (default 7).
A687-A693 Execute 'R' command.
A694-A69B Execute 'P' command.
A69C-A6A7 Execute 'Q' command.
A6A8-A6AA Play new note (note index in 6502 register Y).
A6AB-A6B3 Save new sound table when previous note finishes.

A6B4-A6B9 Common return routine.
A6BA-A6CD Play notes 'A' to 'G'.
A6CE-A6DA Execute '#' command.
A6DB-A6E5 Execute '$' command.
A6E6-A6EC **Table:** conversion for notes to note index.
A6ED-A6EF **Table:** conversion for octave to base note index.
A6F0-A6F9 **Table:** conversion for tempo to duration (jiffies).

Parsing New Command Routines

A6FA-A713 Look for and evaluate first 1-byte and two 2-byte parameters.
A6FD-A713 Look for and evaluate two 2-byte parameters.
A700-A713 Look for and evaluate one 2-byte parameter.
A714-A71B Save one 1-byte parameter (parameter in 6502 register A, index in register Y).
A71C-A72B Look for and evaulate two 1-byte parameters.
A71F-A72B Look for and evaluate one 1-byte parameter.
A72C-A73F Parse GRAPHIC (get parameters and check syntax).
A740-A762 Parse CIRCLE.
A763-A7A4 Parse DRAW.
A7A5-A7BC Parse POINT.
A7BD-A7C7 Parse COLOR.
A7C8-A7CE Go to execute commands after parsing.
A7CF-A7D8 Parse REGION.
A7D9-A7DC Parse SCNCLR.
A7DD-A7E9 Parse SOUND.
A7EA-A809 Parse CHAR.
A80A-A810 Parse PAINT.
A811-A817 Parse RPOT.
A818-A81B Parse RPEN.
A81C-A81F Parse RSND.
A820-A823 Parse RCOL.
A824-A827 Parse RGR.
A828-A842 Parse RDOT.
A843-A846 Parse RJOY.
A847-A84E Look for first 1-byte parameter.
A84F-A866 Indirect jump to execute new commands (pointer to parameter save area in 6502 registers X,Y, command index in register A).
A867-A878 **Table:** vector to execute new commands (H).
A879-A88A **Table:** vector to execute new commands (L).



Execute New Command Routines

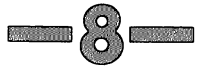
A88B-AA22 Execute GRAPHIC.
- **A8AB-A94E** Handle GRAPHIC 1,2,3, if previous was 0
  - **A8D4-A942** Transfer BASIC program to above $2000 and execute CLR.
  - **A943-A94E** Make screen at $1E00 and character set at $1000.
- **A94F-A9AB** Handle GRAPHIC 4.
  - **A967-A9AB** Transfer BASIC program down to old location and execute CLR.
- **A9AC-A9B7** Handle GRAPHIC 0 if previous was 1,2,3.
- **A9B8-AA22** Set up proper GRAPHIC screen.

AA23-AA28 Execute RGR.
AA29-AA6A Execute COLOR.
AA6B-AA84 Execute REGION.
AA85-AA8B Execute RCOL.
AA8C-AAE6 Execute RDOT.
AAE7-AAF1 Execute POINT.
AAF2-AB12 Execute SCNCLR.
AB13-AB22 Execute DRAW (c TO x,y ...).
AB23-AB34 Execute DRAW (c,x,y TO x,y ...).
AB35-AB54 Execute SOUND.
AB55-AB69 Execute RSND.
AB6A-AB76 Execute RPOT.
AB77-AB7D Execute RPEN.
AB7E-ABE4 Plot a single point from parameter save area.
AB86-ABE4 Plot a single point from beginning scaled X,Y coordinates.
ABE5-AC0A Set up pointers to character and color memory.
ABFA-AC0A Set up pointer to color memory.
AC0B-AC92 Draw a line with a new starting coordinate.
AC11-AC92 Draw a line starting from previous coordinate (using a version of Bresenham's DDA algorithm).
AC93-AD12 Execute CIRCLE (using principle of digital differential analyzer (DDA)).
AD13-AD18 Convert starting angle to radians.
AD19-AD22 Divide FAC#1 by 16.
AD23-AD6B Calculate new scaled X and Y coordinate on locus.
AD39-AD6B Calculate unit offset * scaled radius.
AD6C-ADDE Execute PAINT.
ADDF-AE01 Check for possible new lower bound pivot coordinate.

ADE8-AE01	Save pivot coordinate.
AE02-AE0B	Check for possible new upper bound pivot coordinate.
AE0C-AE1E	Check if able to PAINT a coordinate.
AE0F-AE1E	Check if able to PAINT a coordinate (X,Y in 6502 registers A,Y).
AE1F-AE23	Move beginning scaled X,Y coordinate to 6502 registers A,Y.
AE24-AE3B	Check if coordinate has been already plotted.
AE3C-AE44	Move beginning scaled X coordinate to the right.
AE45-AE51	Move beginning scaled X coordinate 2 to the left.
AE52-AE56	**Flag:** 'FORMULA TOO COMPLEX' error message.
AE57-AED9	Execute CHAR.
AEDA-AF13	Execute RJOY.
AF14-AF33	Set up correct VIC chip screen registers.
AF34-AF3E	Save number of coordinates and color register.
AF39-AF3E	Save color register.
AF3F-AF47	Copy beginning from ending scaled X,Y coordinate.
AF48-AF75	Scale X and Y coordinates.
AF76-AFB0	Scale X or Y coordinate to the range 0 to 159 (6502 register X =register A*coordinate*2/256) (number of columns or rows in register A).
AFB1-AFBA	**Table:** vector to map Y coordinate to color memory (L).
AFBB-AFCE	**Table:** vector to map X coordinate to character memory (L).
AFCF-AFE2	**Table:** vector to map X coordinate to character memory (H).
AFE3-AFE5	**Table:** bit set for color memory.
AFE6	(Not used — contains $00.)
AFE7-AFEE	**Table:** bit mask for high-resolution mode.
AFEF-AFF6	**Table:** bit mask for multicolor mode.
AFF7-AFFA	**Table:** bytes to plot in multicolor mode.
AFFB-AFFE	**Table:** conversion factor for VIC chip screen registers.
AFFF	(Not used — contains $AA.)

Note:

(H):	high byte of a two-byte address
(L):	low byte of a two-byte address
Vector:	two-byte address used for indirection of execution
Pointer:	two-byte address for data
Index:	one-byte offset for a table



General RAM Area Used by Super Expander

0024	Number of coordinates.
0024	**Flag:** color register mode ($FF =multicolor, $00 =high resolution).
0024-0025	**Pointer:** New start of variables/start of BASIC memory.
0026	Temporary area for building VIC chip registers/for building character byte/for saving start of BASIC (L).

Current Coordinates

0062	Ending scaled X coordinate (0 to 159).
0063	Beginning scaled X coordinate (0 to 159).
0064	Scaled X difference (absolute value).
0065	Ending scaled Y coordinate (0 to 159).
0066	Beginning scaled Y coordinate (0 to 159).
0067	Scaled Y difference (absolute value -1).

For Scaling Coordinates

0069	Multiplicand -1.
006A	16-bit product.
006B-006C	10-bit multiplier.

For DRAW

0069	Scaled X unit direction -1.
006A	Scaled Y unit direction.
006B-006C	Number of scaled Y units left to plot before next scaled X unit (count up).
006D-006E	Number of points left to plot (count up).

For PAINT

0069	**Index:** pivot coordinates save area.

For CHAR

0069	Current row (0 to 19).
006A	Current column (0 to 19).
006B	Length of string.
006C-006D	**Pointer:** string location.

Other Zero-Page Usage

009B	**Index:** beginning of current function key definition.
009B-009C	**Pointer:** current character set address/byte in character set/position in screen memory/destination of byte of BASIC program to transfer.

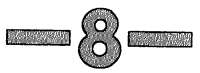
009D	**Index:** end of function key definition area.
009E	Current function key number/length of current function key string.
009F	Length of current function key string (count down).
009E-009F	**Pointer:** byte in color memory.
00AC-00AD	**Pointer:** current byte (function key definition, tape, scrolling).
00C3	**Flag:** 0 =transferred BASIC program to a new location.
00C3-00C4	**Pointer:** Kernal setup/current music table/ parameter save area ($033C).
00FB-00FC	**Pointer:** top of BASIC memory (usually same as $0284-$0285).

For Keyboard Input

028F-0290	**Vector:** interpret keyboard input ($A337).
02A1	Number of bytes taken by Super Expander in high memory ($88).
02A2	Number of characters in function key definition.
02A3	**Index:** current byte of function key string.
02A4	Length of function key string (amount left to output).

For Music

02A5	Previous character in music mode.
02A6	Music mode ($80 =on).
02A7	Screen echo ($50 =on, $00 =off).
02A8	Current voice (sound register -1).
02A9	Current note index.
02AA	Current duration (jiffies).
02AB	Current sound amplitude (volume *2).
02AC	Current octave (base note index).
02AD	Voice 1 note index (+$80).
02AE	Voice 1 duration count down (jiffies).
02AF	Voice 2 note index (+$80).
02B0	Voice 2 duration count down (jiffies).
02B1	Voice 3 note index (+$80).
02B2	Voice 3 duration count down (jiffies).
02B3	Voice 4 note index (+$80).
02B4	Voice 4 duration count down (jiffies).
02B5-02BF	(For expansion.)



For Execution Of New Commands

02C0-02C2	**Vector:** execute new commands (JMP $A84F).
02C3	Current VIC chip left margin register.
02C4	Current VIC chip top margin register.
02C5	Current VIC chip number of columns register.
02C6	Current VIC chip number of rows register.
02C7	Current row of cursor.
02C8	Current GRAPHIC mode.
02C9	(For expansion.)
02CA	Current color register parameter (while plotting).
02CB	Current screen color.
02CC	Current border color.
02CD	Current character color.
02CE	Current auxiliary color.
02CF	**Index:** parameter save area (while plotting).
02D0	Current character set address page.
02D1	Usual character set address page ($80).
02D2-02D3	**Pointer:** old limit of BASIC memory.
02D4	Old screen memory page.
02D5	Last scaled X coordinate (0 to 159).
02D6	Last scaled Y coordinate (0 to 159).
02D7	**Flag:** $00 =DRAW c,x,y TO, $01 =DRAW c TO/current number of out-of-range coordinates ($00 =within range).
02D8	Old number of out-of-range coordinates ($00 =within range).
02D9	**Index:** parameter save area (while getting parameters).
02DA-02FF	(For expansion.)

Operating System Vectors

0300-0301	**Vector:** error message.	($A3FD)
0302-0303	**Vector:** BASIC warm start.	($C483)
0304-0305	**Vector:** analyze BASIC source line.	($A407)
0306-0307	**Vector:** print BASIC tokens in ASCII form.	($A4BA)
0308-0309	**Vector:** start new BASIC statement.	($A504)
030A-030B	**Vector:** get and evaluate an expression.	($A52A)
0314-0315	**Vector:** IRQ.	($A372)
0316-0317	**Vector:** BRK instruction.	($A2C2)
0318-0319	**Vector:** NMI.	($FEAD)
031A-031B	**Vector:** BASIC OPEN statement.	($F40A)
031C-031D	**Vector:** BASIC CLOSE statement.	($F34A)

031E-031F	**Vector:** set input.	($F2C7)
0320-0321	**Vector:** set output.	($F309)
0322-0323	**Vector:** restore I/O.	($F3F3)
0324-0325	**Vector:** input a character.	($A395)
0326-0327	**Vector:** output a character.	($A3A6)
0328-0329	**Vector:** test STOP key.	($F770)
032A-032B	**Vector:** BASIC GET statement.	($F1F5)
032C-032D	**Vector:** abort I/O.	($F3EF)
032E-032F	**Vector:** user BRK instruction.	($A2C2)
0330-0331	**Vector:** BASIC LOAD statement.	($F549)
0332-0333	**Vector:** BASIC SAVE statement.	($F685)
0334-033B	(For expansion.)	

Save Area

033C-03F8	**Save area:** parameter passing/pivot coordinates (PAINT).

For Circle

033C	**Index:** X or Y.
0347-0348	Old scaled X coordinate on locus.
0349-034A	Old scaled Y coordinate on locus.
034B-034C	New scaled X coordinate on locus.
034D-034E	New scaled Y coordinate on locus.
034F-0353	Floating point unit offset X coordinate.
0355-0359	Floating point unit offset Y coordinate.

—8—

Memory Map Index

This is an index by subject. *The references are to decimal memory locations,* not to page numbers.

—Appendix A—

A Beginner's Guide to Typing In Programs

—Appendix A—

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 COMPUTE! Books for Commodore are written in a computer language called BASIC. BASIC is easy to learn and is built into all VIC-20s.

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 "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 the VIC'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 characters, lowercase, and control characters? It's all explained in your VIC-20's manual, *Personal Computing with the VIC.*

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 INST/DEL key to erase mistakes.

2) Check the line you've typed against the line in the magazine. You can check the entire program again if you get an error when you RUN the program.

3) Make sure you've entered statements in brackets as the appropriate control key (see "How to Type In Programs").

—Appendix B—

How to Type In Programs

—Appendix B—

How to Type In Programs

Many of the programs which are listed in this book contain special control characters (cursor control, color keys, inverse video, etc.). 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, any VIC-20 program listings will contain words in 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. You should never have to enter such a character on the VIC-20, but if you do, you would have to leave the quote mode (press RETURN and cursor back up to the position where the control character should go), press CTRL-9 (RVS ON), the letter in braces, and then CTRL-0 (RVS OFF).

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:	When You Read:	Press:	See:
{CLEAR}	SHIFT + CLR/HOME		{GRN}	CTRL + 6	
{HOME}	CLR/HOME		{BLU}	CTRL + 7	
{UP}	SHIFT + CRSR ↑		{YEL}	CTRL + 8	
{DOWN}	CRSR ↓		{F1}	f1	
{LEFT}	SHIFT + CRSR ←		{F2}	f2	
{RIGHT}	CRSR →		{F3}	f3	
{RVS}	CTRL + 9		{F4}	f4	
{OFF}	CTRL + 0		{F5}	f5	
{BLK}	CTRL + 1		{F6}	f6	
{WHT}	CTRL + 2		{F7}	f7	
{RED}	CTRL + 3		{F8}	f8	
{CYN}	CTRL + 4		←	←	
{PUR}	CTRL + 5		↑	SHIFT + ↑	

Index

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:
☐ PET ☐ Apple ☐ Atari ☐ VIC ☐ Other _____ ☐ Don't yet have one...

☐ $20.00 One Year US Subscription
☐ $36.00 Two Year US Subscription
☐ $54.00 Three Year US Subscription

Subscription rates outside the US:
☐ $25.00 Canada
☐ $38.00 Europe, Australia, New Zealand/Air Delivery
☐ $48.00 Middle East, North Africa, Central America/Air Mail
☐ $68.00 Elsewhere/Air Mail
☐ $25.00 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 ____/____

16-7

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
_____	The Beginner's Guide to Buying A Personal Computer	$ 3.95*	_____
_____	COMPUTE!'s First Book of Atari	$12.95†	_____
_____	Inside Atari DOS	$19.95†	_____
_____	COMPUTE!'s First Book of PET/CBM	$12.95†	_____
_____	Programming the PET/CBM	$24.95‡	_____
_____	Every Kid's First Book of Robots and Computers	$ 4.95*	_____
_____	COMPUTE!'s Second Book of Atari	$12.95†	_____
_____	COMPUTE!'s First Book of VIC	$12.95†	_____
_____	COMPUTE!'s First Book of Atari Graphics	$12.95†	_____
_____	Mapping the Atari	$14.95†	_____
_____	Home Energy Applications On Your Personal Computer	$14.95†	_____
_____	Machine Language for Beginners	$12.95†	_____

* Add $1 shipping and handling. Outside US add $5 air mail; $2 surface mail.

† Add $2 shipping and handling. Outside US add $5 air mail; $2 surface mail.

‡ Add $3 shipping and handling. Outside US add $10 air mail; $3 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 Acct. No. _____ Expires ___/___

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Allow 4-5 weeks for delivery.

16-7

COMPUTE!'s
Second Book of VIC

COMPUTE!'s First Book of VIC has been a number one best seller in 1983. Here's *COMPUTE!'s Second Book of VIC,* including some of the best articles and programs from *COMPUTE!* Magazine and *COMPUTE!'s Gazette,* plus many more that have never before appeared in print. And none of these articles were published in the *First Book.*

Here's a sample of what you'll find inside:

- A system for creating realistic sound effects
- "Pixelator," a utility program that makes it easy to redefine characters
- An extensive memory map of the VIC
- A mini word processor
- A cassette-based file system
- A machine language assembler written in BASIC
- An extraordinary all-machine-language game
- How to program the function keys

No matter whether you are an advanced programmer or just starting out, *COMPUTE!'s Second Book of VIC* has much that you will find useful. Edited with the clarity and care which has made COMPUTE! Publications today's leading publisher of personal computing magazines and books.

ISBN 0-942386-16-7

$12.95