COMPUTE!'s FIRST BOOK OF

VIC

Games, Programs, And Other Helpful Information
For Owners And Users Of
The Commodore VIC-20™ Personal Computer.

A **COMPUTE! Books** Publication

$12.95

Extended Input Devices: Paddles And The Keyboard

MIKE BASSMAN / SALOMON LEDERMAN

You can use game paddles with the VIC, even from a BASIC program. Also included is information on VIC's "polled keyboard."

The VIC-20 has some remarkable capabilities not documented by the manual. Specifically, you can use game paddles with the VIC-20 as well as make better use of the keyboard.

What A Paddle Does

Have you ever seen the little nine-pin port right next to the power switch? This port can be used with paddles. To make life easy, it can be used with the widely available Atari game paddles (which are used with their video games and home computers). Just plug in a pair, and we'll be ready to begin. These paddles are *linear* devices. This means the paddle is a much more sensitive device than a directional joystick, which can only point in eight or so directions. You may think the paddle is not even as good, pointing only left or right. This is not true.

What the paddle actually does is isolate one position out of the 256 possible ones. When the paddle is turned to the far right, this value is 0. Every time you turn the paddle in either direction the number is increased or decreased accordingly. The VIC-20 allows up to two paddles. For each of them, we can obtain a position value. These values are in memory locations $9008 for the first paddle, and $9009 for the second. In decimal these are 36872 and 36873, respectively. (A number preceded by a "$" signifies it is hexadecimal.)

How To Do It

Shown below is a quick one-liner that prints out the values of both the paddle registers.

```
10 PRINT PEEK(36872);PEEK(36873):GOTO 10
```

Try typing and running this program now. You should see a continuous stream of two numbers flying by. Fiddle with the paddles. The numbers should change accordingly. The more you turn a paddle left, the higher the number goes (the opposite for right, of course).

Next, we'll try something a little more complicated and which might be more applicable. Program 1 will move a little ball across the screen according to your paddle position. It will also slide a musical tone up and down at the same time.

The first two lines are just set-up, setting volume for the tone generator and clearing the screen. Line 20 gets the initial paddle position. The next line, 30, determines the ball's position on the screen. The ball can move from the far left edge of the screen (memory location 7900) to the far right (7921). Logically, the thing to do is to move the ball a little bit left whenever the paddle value goes a little bit up (turning towards the left). The problem is that the paddle is much more accurate than one line of the screen.

While the paddle has 256 possible calibrations, one line of the screen is only 22 characters long. What we do is to make a proportion of paddle calibrations per screen character, in this case 11.64 (obtained by dividing 256 by 22). Now we have the position of the ball on the screen. Line 40 does almost the same thing, finding an appropriate tone for the paddle position. We have 128 possible tones, so the proportion of the calibrations to tones is only 2 to 1. The next three lines just put the ball on the screen, tack a color onto it, and turn on the proper tone. Only the clean-up work remains to be done now: a small delay loop so the ball doesn't flicker badly, and erasing the ball and the color. After this, we get a new paddle reading and start all over again.

If you have run this little demonstration, the advantage of a linear device should be obvious. You can just whip the paddle back and forth without having to worry whether the computer is fast enough to keep up with you, and the ball will follow because the paddles determine an absolute position, rather than just a direction. This could be very convenient in games where speed is indispensable. In the near future, I'm sure we'll see many clever and innovative ways to use the paddles.

The Keyboard

There are two types of keyboards: ASCII, or hardware keyboards, and *polled*, or software keyboards. The ASCII keyboard is a separate device from the computer, which just sends out the ASCII value of

the key being pressed. The polled keyboard is a little more subtle. A polled keyboard is split up into sections of eight keys, called rows. Generally, a polled keyboard has eight rows. The computer can test one row at a time, and detect which key along which row is being pressed, if any. The polled keyboard can also detect any number of key combinations along any particular row. Consequently, polled keyboards need a fair amount of system software to do what comes naturally to an ASCII keyboard.

Most microcomputers today, the VIC-20 included, use polled keyboards because of the added flexibility and lower price. Unfortunately, the VIC-20 normally does not let us get at some of those nice features. To us, from BASIC, it seems just like an ASCII keyboard. We can only obtain one character at a time using the GET command. If two keys are being pressed down at once, the GET command will almost randomly choose one of those two as the value that gets sent back to the user. If you wanted to do a two-player game or a game requiring simultaneous depressing of more than one key, life would be very difficult. But here's how it can be done.

Polled Keyboard Encoding

The VIC-20 polled keyboard has eight rows of eight keys each. Each row can be selected by a particular value. The eight values for the eight rows are all shown in Table 1. These values are by no means arbitrary. If you examine the table, you can see that the values are given in binary, as well as decimal and hexadecimal. Row values were made by turning on all the bits in the byte, then turning off the bit which the row represents. For example, the first row has all the bits on (set to 1) except for the one on the far left, which is off (or 0). Then this binary number is simply used in its hexadecimal or decimal form to represent the row. Each key along the row is handled in exactly the same manner as the rows (for example, the value representing the first row would be the same as the one representing the first key in that row). This is a little confusing, but it works out well in the end. Table 2 is the keyboard encoding matrix. It shows all the row values going down, and all the keys along each row, and their values. For instance, the keys on row 223 are F3, =, :, K, H, F, S, and Commodore. The value of the Commodore key would be 254.

Implementing Keyboard Theory

Using an individual row on the keyboard is accomplished as follows. You select a row by POKEing its value into a memory location we'll call the row select register. Then you can get the information as to which key(s) is hit by PEEKing another location, the keyboard data

register. The row select register is located at $9120 (37152), and the data register at $9121 (37153).

Things don't work out as easily as doing just one POKE, then another PEEK. The problem, in this case, is the RUN STOP routine. This part of BASIC checks if you hit this key during execution of a program. If you have, the program stops. After every command executed, the routine puts a 247 in the row select register (the row which has the RUN STOP key) and checks the data register for a value of 254 (eighth key over). If the data register is 254, then you have hit the RUN STOP key, and program execution terminates.

This means that even after we have just chosen a row by POKEing a value into the select register, the RUN STOP routine will change it right back to a 247. Very bad news indeed, unless you only want to use row 247. Not only that, but you can't use the RUN STOP key for your own purposes. There is a way to disable the RUN STOP key. POKEing 808 with 114 turns off the RUN STOP key, and POKEing 808 with 112 turns it back on again. This does not solve our problem. Turning off the RUN STOP key will prevent it from ending program execution when that key is hit, but the routine still stores that 247 in the select register. However, when we clear up the major problem, turning off the RUN STOP key will allow us to use that key in our programs.

A Solution

The way to solve this problem is by noticing that this routine operates after every BASIC command. What must be done is to POKE in our select value, then PEEK the data register, all in the time of less than one BASIC command. Machine language is the answer. The VIC-20 can use machine language even though it has no direct facilities for entering or saving it. [*See Jim Butterfield's TINYMON1, reprinted in this book, which provides a monitor for VIC.*] We are going to use a very short machine language routine that simply puts our row into the select register, looks at the data register, then puts the contents of the data register into a RAM location into which the BASIC program can look. Program 2 shows just such a machine language program. Not much to it at all, just five lines of code. The first instruction loads in the row value, in this case a $7F (127). The second stores it in the select register. The next picks up a value from the data register. The last two just store that value in accessible RAM (at $1DFF, or 7679), then return to BASIC.

This routine will do the trick because it does what we want in less than one BASIC command. Even though the VIC-20 has no real

method for entering machine code data, it can be done anyway. You just take the machine code values, convert them into decimal, and stick them into a BASIC DATA statement. Then just add a line of BASIC that reads the values and puts them in the correct place. In Program 3, we have a complete demonstration. Lines 30 and 40 are the aforementioned DATA statement and reader/POKEr. Line 5 turns off the RUN STOP.

Lines 10 and 20 need a little bit of explanation. We are going to put the machine language routine into the top of available memory. Unfortunately, BASIC also wants to use this space. These lines tell BASIC not to use the highest 21 bytes of RAM. Locations 51 and 52, as well as 55 and 56, contain the top of BASIC RAM in low, high format. Low, high format is when the low byte of an address precedes its high byte. To calculate an address from this format, just use this formula: (256*high byte + low byte = address). Normally the low and high bytes for the top of BASIC are 00 and 30, respectively (yielding an address of 7680). These we change to 235 and 29, giving an address of 7659. Line 50 goes to our machine code routine, line 60 prints the result, and 70 repeats the process. Try it now. I'll wait. If you press one of the keys from the first row, the appropriate value will be printed. No key is indicated by its printing 255. As it is now, this program will print first row values. To change the row, just change the second item of data in line 30. I used this program, incidentally, to make the keyboard matrix chart.

All this may seem pretty useless to you at this point. Our next program will do something that cannot be done with regular old BASIC. Program 4 will play a tone of varying pitch depending on which of two keys you hit. Doesn't sound too exciting, but it will play the two tones one after the other even if both keys are pressed at the same time. This is the basis of two-player games, where the computer can fairly give one turn to each player. All the material in this program should be old hat to you now, so I won't bother to explain it.

Possibly you've learned to use your paddles and keyboard now. Put them to good use!

Table 1.

127 - 7F - 0111 1111
191 - BF - 1011 1111
223 - DF - 1101 1111
239 - EF - 1110 1111
247 - F7 - 1111 0111
251 - FB - 1111 1011
253 - FD - 1111 1101
254 - FE - 1111 1110

Table 2. Keyboard Matrix Table

Column (POKE) ↓ Row (PEEK) of Keys →

	127	191	223	239	247	251	253	254
127	F7	Home	—	∅	8	6	4	2
191	F5		@	O	U	T	E	Q
223	F3	=	:	K	H	F	S	COMMO-DORE
239	F1	RIGHT SHIFT	.	M	B	C	Z	SPACE
247	CURSOR	/	,	N	V	X	LEFT SHIFT	RUN STOP
251	CURSOR	;	L	J	G	D	A	CTRL
253	RETURN	*	P	I	Y	R	W	
254	DEL	£	+	9	7	5	3	1

Program 1.

```
5 POKE36878,3
10 PRINT"{CLEAR}"
20 X=PEEK(36872)
30 L=7921-INT(X/11.64)
40 T=255-INT(X/2):IFT=255THENT=254
50 POKEL,81
55 POKEL+30720,2
```

```
60 POKE36874,T
70 FORK=1TO10:NEXT
80 POKEL,32:POKEL+30720,1
90 GOTO20
```

Program 2.

```
A9 7F     LDA #$7F
8D 20 91  STA $9120
AD 21 91  LDA $9121
8D FF 1D  STA $1DFF
60        RTS
```

Program 3.

```
5 POKE808,114
10 POKE51,235:POKE52,29
20 POKE55,235:POKE56,29
30 DATA169,127,141,32,145,173,33,145,141,255,
    29,96
40 FORK=1TO12:READX:POKE7659+K,X:NEXTK
50 SYS7660
60 PRINTPEEK(7679);
70 GOTO50
```

Program 4.

```
10 POKE808,114:POKE51,235:POKE52,29:POKE55,23
    5:POKE56,29:POKE36878,3
20 DATA169,127,141,32,145,173,33,145,141,255,
    29,96
30 FORK=1TO12:READX:POKE7659+K,X:NEXTK
40 POKE7661,127:SYS7660
50 IFPEEK(7679)=254THENPOKE36874,200:FORK=1TO
    500:NEXTK:POKE36874,0
60 POKE7661,191:SYS7660
70 IFPEEK(7679)=127THENPOKE36875,200:FORK=1TO
    500:NEXTK:POKE36875,0
80 GOTO40
```