VIPER

August - September 1982

Volume 4, Number 3   Journal of the VIP Hobby Computer Assn.

The VIPER was founded by ARESCO, Inc., in June 1978

****

Contents

VIPHCA INFO...

The VIPER, founded by ARESCO, Inc. in June 1978, is the Official Journal of the VIP Hobby Computer Association. Acknowledgement and appreciation is extended to ARESCO for permission to use the VIPER name. The Association is composed of people interested in the VIP and Computers using the 1802 microprocessor. The Association was founded by Raymond C. Sills and created by a constitution, with by-laws to govern the operation of the Association. Mr. Sills is serving as director of the Association, aw well as editor and publisher of the VIPER.

VIP and COSMAC are registered trademarks of RCA Corp. The VIP Hobby Computer Association is in no way associated with RCA, and RCA is not responsible for the contents of this newsletter. Please send all inquiries relating to the VIPER to VIPHCA, 32 Ainsworth Avenue, East Brunswick, NJ 08816.

The VIPER will be published six times per year and sent to all members in good standing. Issues of the VIPER will not carry over from one volume to another. Annual dues to the Association, which includes six issues of the VIPER, is $12 per year. Membership in the VIP Hobby Computer Association is open to all people who desire to promote and enjoy the VIP and other 1802 based systems. Send a check for $12 in U.S. funds payable to "VIP Hobby Computer Assn." c/o Raymond Sills, 32 Ainsworth Avenue, East Brunswick, NJ 08816. People outside the U.S., Canada and Mexico please send $18, due to additional postage charges. The VIPER is normally sent via first class mail, and airmail to members outside North America.

Contributions by members or interested people are welcome at any time. Material submitted by you is assumed to be free of copyright restrictions, and will be considered for publication in the VIPER. An honorarium payment is made to those whose material is published in VIPER to nelp cover the cost of a submission. Articles, letters, programs, etc., in camera-ready from on 21.5 x 28 cm (8.5 x 11 inch) paper will be given preferential consideration. Please send enough information about any program so that readers can operate the program properly. Fully documented programs are best, but memory dumps are okay if you provide enough information to run the program.

If you write to VIPER/VIPHCA, please indicate that it is okay to print your address in letters to the editor, if you want your address revealed to VIPER readers. Otherwise, we will not print your address in VIPER.

Editorial

I have to appologize to you all out there for the delay in mailing VIPER 4.02. Things sometimes get put on the "back burner," particularly around vacation time, and I wasn't able to pick up the finished copies of VIPER from our printer until after my vacation. This issue will be going to the printer around the 24th of August, so allowing 10 days or so for the printing to be completed, plus a day or two for stapling and colating, and then the normal delay for mail delivery, you should have this issue about two weeks after it goes to the printer.

There hasn't been a lot of mail about the "cassette" idea, but everyone I have heard from has favored the idea. So I will assume, unless I get a sudden deluge of mail opposing the idea, that most of you would tend to favor having casettes available of VIPER material. But I will hold off for a bit on this project in case it does turn out that you don't want to bother with it. It would be a big help for me if those of you who send in programs, particularly those that run more than a couple CHIP-8 pages, would also send in a cassette of the program. If you do, I will either return the original or replace it with a blank tape, whichever you would prefer. It would probably be a good idea to send the tape in a separate mailer, unless you are using a large mailer. Radio Shack has some inexpensive cardboard mailers (catalog # 44-632) which should work just fine. You can mail even a C-60 in this box for only 2 ozs. (37 cents, US) postage.

Any of you published authors out there who have had material published in VIPER are also welcome to send in casettes, so that we might build a library. I "dink-in" a good deal of VIPER material myself, but not ALL of it. And I don't have either of the ELF machines here at VIPHCA HQ, so I can't check out ELF programs.

By the way, have any of you ELF folks tried Paul Moews' ELFISH? Paul had an ad in VIPER 4.01.06 and sent in a copy of his booklet and a tape of the program. Due to the lack of the ELF machine here, I couldn't try it out myself, but perhaps one of you already has. It sure looks interesting, and if you like CHIP-8, I think you'll love ELFISH. It looks like a "Super CHIP-8," with 16 bit variables, built-in editor, and more extensive functions and commands.

73,

Ray

HOW TO UPGRADE THE COLOR BOARD
by Jeff Jones

Wouldn't it be nice to be able to control the color board in higher resolution modes instead of the just the original CHIP 8-X mode? Even better, without mirroring effects like those that occur when the Floating Point BASIC is used. Well, by replacing one chip from the color board with the circuit described here you can have it all- High res color with no mirroring effects, all under the control of CHIP 8.

The circuit consists of 3 IC's; a 2114 4K static RAM, a 4050 Hex Buffer, and a 4042 Quad Latch. All IC's can be purchased at Radio Shack for less than $12.

The frist thing to do is remove the 1822 RAM chip from the color board and install a 22 pin socket in its place. Make sure that it is installed with pin 1 of the socket where pin 1 of the 1822 was. Next install a jumper from pin 10 of the socket to the TPA pad on the color board. Thic completes all modifications to the color board.

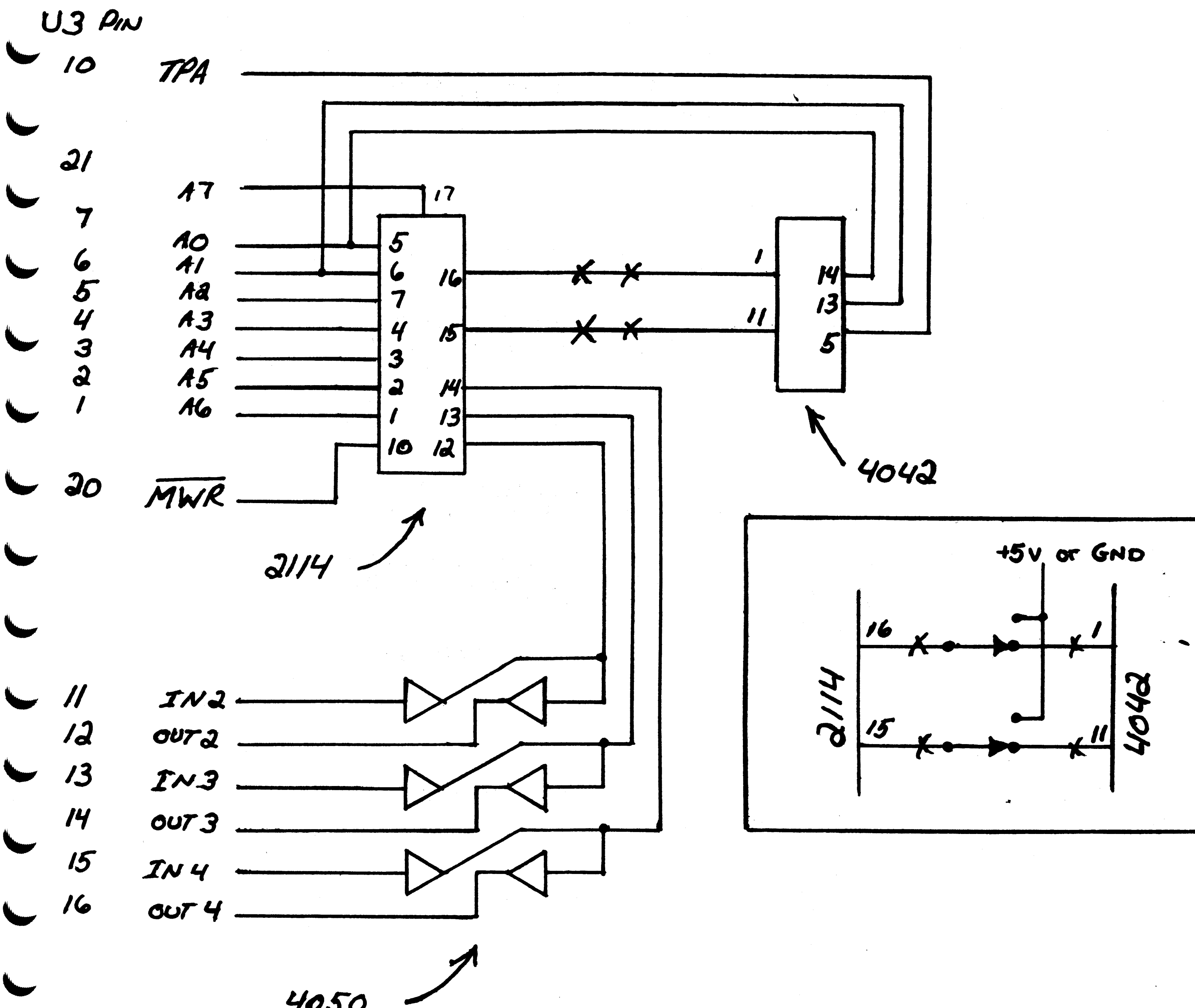
Assemble the color expansion circuit using the method which you prefer. If you plan on using the Tiny Basic Board or any color programs that you don't know where the color subroutine is or if you don't have access to it because it is in some form of ROM, install the DPDT switch circuit at points X.X. Now, using a piece of ribon cable, connect the circuit to a 22 pin dip header. Use the chart below to help you connect the wires. This completes the assembly of the color expansion circuit.

To use the expansion circuit simply plug the dip header into the socket on the color board. When the DPDT switch is in the 5 volt position, it acts just like a color board with no modifications. If it is thrown to the other position, you have control over the color of the entire display area.

On the following pages are modifications to the 2 page Chip 8 (VIPER 1.03) and the HI-RES CHIP 8(VIPER 2.06) that allow them to operate like high resolution CHIP 8-X interpreters. These interpreters operate the same as CHIP 8-X with the following exceptions:

1. Niether has a BXY0 instruction
2. The 2 page interpreter uses 00F0 instead of 00EE
3. The 5XY1 instruction can handle numbers up to FF

Modifications to CHIP 8-X have been provided to allow proper operation if the DPDT switch is not installed(it must be thrown in to the 5 volt position if these mods are not uesd and the switch has been installed).



17,22 +5V →

8,9,18,19 GND →

PINS	2114	4050	4042
+5V	18,11	1	16,6
GND	8,9	8	8,7,4

CHIP 8-X MODS

01A7	FF	Allows hex digits up to FF in 5XY1 instruction.
01AB	FF	
01AE	FF	
0224	D3*	Relocates color map to new address.
023E	C3*	

TWO PAGE CHIP 8-X MODS

0015	02	New starting address for CHIP 8 instructions.
0018	F8	
0055	00	New starting addresses for 5 and B instructions.
005B	02	
0065	FE	
006B	50	
0092	F5	
00EC	3A B3	Relocated portion of 2 page erase screen (00E0) instruction.
00EE	30 D9	
00F0	42 B5	Relocated return from CHIP 8 subroutine instruction. Use 00F0 instead of 00EE!
00F2	42 A5	
00F4	D4	
00F5	8D A7	Relocated part of display routine.
00F7	87 32	
00F9	AC 2A	
00FB	27 30	
00FD	F7	
00FE	05 F6	Go to 5XY1 or other 5XYN instructions.
0100	33 A4	
0102	30 95	
01A4	E6 06	5XY1 instruction.
01A6	FA FF	
01A8	56 07	
01AA	FA FF	
01AC	F4 FA	
01AE	FF D4	
01B0	15 D4	
01B2	00 00	

*not required when DPDT switch is used in the color expansion circuit.

TWO PAGE CHIP 8-X MODS cont.

01F2	37 88	EXF2 and EXF5 instructions.
01F4	D4 3F	
01F6	88 D4	
01F8	E6 63	FXF8 and FXFB instructions.
01FA	D4 E6	
01FC	3F FC	
01FE	6B D4	
0250	92 BD	Color subroutine (BXYN).
0252	F8 9F AD	
0255	0D	
0256	32 5A	
0258	45 D4	
025A	00 45	
025C	FA 0F AF	
025F	32 8E	
0261	46 FA 3F	
0264	F6 F6 F6	
0267	22 52 E2	
026A	06 FA 3F	
026D	FE FE FE	
0270	F1 AC 12	
0273	F8 D2 BC	
0276	3B 84	
0278	9C	
0279	FF D2	
027B	32 81	
027D	F8 D2	
027F	30 83	
0281	F8 D3	
0283	BC	
0284	07 5C 8C	
0287	FC 08 AC	
028A	2F 8F	
028C	3A 76	
028E	D4	
0290	92 BD	Scan for Color Board.
0292	F8 9F AD	
0295	F8 C0 BC	
0298	94 AC	
029A	F8 AA 5C	
029D	94 BC AC	
02A0	0C FB 91	
02A3	32 AC	
02A5	F8 91 5C	
02A8	F8 01 5D	
02AB	D4	
02AC	F8 00 5D	
02AF	D4	

TWO PAGE CHIP 8-X MODS cont.

02F0	E2 65	Switch background color subroutine (02F0).
02F2	22 D4	

02F8	0245	Adjust to 2 page display	
02FA	0290	Scan for color board	
02FC	0230	Erase display pages	CHIP 8 initialization
02FE	004B	TV on	

HI-RES CHIP 8-X MODS

0015	02	New hi address for start of CHIP 8 instructions.
0055	02	New starting addresses for 5 and B instructions.
005B	02	
0065	44	
006B	50	
01A4 to 01B3		Same as in TWO PAGE CHIP 8-X
01F2 to 01FF		Same as in TWO PAGE CHIP 8-X
0244	05 F6	Go to 5XY1 or other 5XYN instructions.
0246	C3 01	
0248	A4 C0	
024A	01 95	
0250	92 BD	Color subroutine (BXYN).
0252	F8 9F AD	
0255	0D	
0256	32 5A	
0258	45 D4	
025A	00 45	
025C	FA 0F AF	
025F	32 98	
0261	46 FA 3F	
0264	F6 F6 F6	
0267	22 52 E2	
026A	06 FA 3F	
026D	FE FE	
026F	33 74	
0271	F8 D0	
0273	30 77	
0275	F8 D2	
0277	BC	
0278	06 FA 7F	
027B	FE FE FE	
027E	F1 AC 12	
0281	3B 8E	
0283	9C	

HI-RES CHIP 8-MODS cont.

0284	FC 01 BC	
0287	FF D4	
0289	3A 8E	
028B	F8 D0 BC	
028E	07 5C 8C	
0291	FC 08 AC	
0294	2F 8F	
0296	3A 81	
0298	D4	
02A0	92 BD	Scan for Color Board.
02A2	F8 9F AD	
02A5	F8 C0 BC	
02A8	94 AC	
02AA	F8 AA 5C	
02AD	94 BC AC	
02B0	0C FB 91	
02B3	32 BC	
02B5	F8 91 5C	
02B8	F8 01 5D	
02BB	D4	
02BC	F8 00 5D	
02BF	D4	
02F0	E2 65	Switch background color subroutine (02F0).
02F2	22 D4	
02FA	02A0	Scan for Color Board
02FC	023F	Adjust for 4 page display
02FE	004B	TV on and erase display pages

SAMPLE PROGRAM

0300	A32E 6200 6100 D124 7108 3140 1306 7204
0310	32XX 1304 C43F C57F CC07 B4C8 C707 4701
0320	02F0 6820 F815 F807 3800 1326 1314 FFFF
0330	FFFF

XX use 40 for 2 page Chip 8-X and 80 for Hi-Res Chip 8-X.

Additional note: A slight disturbance will occur on the screen each time a "5XYN" instruction is used on the Hi-Res interpreter.

NOTES ON OPERATION

1. Both the 2 page and Hi-res interpreters can be used without color board modifications. This will only result in mirrored displays.

2. The original RAM can be installed at any time as long as the # 10 pin does not make contact with pin 10 of the socket.

3. The Hi-Res color subroutine may be modified to work on with the Floating Point BASIC.

WIRING CHART

socket pin #	color expansion function
1	A6
2	A5
3	A4
4	A3
5	A2
6	A1
7	A0
21	A7
10	$\overline{\text{TPA}}$
20	$\overline{\text{MWR}}$
11	IN 2
12	OUT 2
13	IN 3
14	OUT 3
15	IN 4
16	OUT 4
17, 22	+5 volts
8,9,18,19	GND

ERROR TRAP

As mentioned in the last VIPER, there was some code missing from the MINI-CALCULATOR program in VIPER 4.01.02. The missing lines were machine language subroutines for multiplication and division. Here is the missing code:

```
0630  E6 45 A6 45 A7 07 AF 06  BF F8 00 56 8F 32 47 9F
0640  F4 56 33 47 2F 30 3C F8  FF A6 F8 00 7E 56 D4 00
0650  E6 45 A6 45 A7 06 FE 56  FE FE F4 56 33 61 07 F4
0660  56 F8 FF A6 F8 00 7E 56  D4 00 00 *END MLS*
```

Very sorry about that goof, but all's well that ends well.

Little Loops -- Follow the Bouncing Ball

by Tom Swan

"Ready kids? Just follow the bouncing ball!" says the announcer as the intro swells in the background, the TV screen darkens, and the words "I'm in the Mood for Love" roll in from top right. Then the star appears, nothing more than a simple white bouncing ball with a happy face, flattening a little on each downbeat to take us cantillating through to the end of the sing-along.

This image, familiar to everyone within range of a television of movie screen, came to mind when a friend suggested that a program to test a person's ability to keep time with music would make an interesting experiment.

The result is a program written in Chip-8 with one machine language subroutine (MLS) to process the test scores. A line in center screen -- I call it the impact line -- represents the point of reference or downbeat, midway between the highest ball position and the lowest. When the ball passes through the impact line, the computer beeps giving an audible reference point of the "beat" to go along with the visual. When the ball reaches the end of its travel in either direction, it automatically reverses.

Try to keep time with the ball's movement by pressing key 5 on the VIP's hex pad at the moment you think the ball is passing through the impact line. How close you come forms the measure of your timing ability. The computer controls the display and timing, collects the data and processes the test results. New tests are initiated by the press of a button.

The test does not begin until the first key press. This duplicates the count musicians are given before they start playing. Watch the display for a moment to "get used" to the timing, press the 5 key to begin and then every time the ball passes through the line thereafter.

Tracking a 64th Note

The VIP's interrupt driven timer resets at both the top and the bottom of the ball's travel. The timer exists as a dedicated 8-bit register half (R8.0) that automatically decrements once during each interrupt. The interrupt routine is provided on ROM with the VIP. Because interrupts occur once every 1/60th second, the frame rate of the TV screen, the timer is more than adequate for the test. If a quarter note lasts for one second, the test conforms to the ability to track somewhere in the range of a 64th note.

At the time the hex pad key is pressed, the program jumps to a routine (at location 0244) that records the current value of the timer. A flag is set which allows only one key press to be read on each swing of the ball. When key presses are not sensed, a small sync loop at location 0234 keeps the time "right" by equalizing the time needed to record key presses. This is the only sync loop needed.

The program halts the test and displays the results after 64 key presses. I tried longer tests, but one minute is a long time to sit in front of a TV screen punching a button. My subjects became bored after about 100 punches. Also, 64 is a nice number to deal with when calculating the average score.

Following the collection of the 64 tests, the program passes to a two-part analysis. A machine language subroutine finds the average of the 64 results by first adding all test scores then logically performing a 16-bit divide by 64 with two shift left instructions taking the high 8 bits as the answer. This is shorter than a shift right by 6 which would accomplish the same thing but with the 8-bit answer in the low portion of the register.

The average is displayed in the upper right corner. This average represents an individual's accuracy in matching beats over a period of time. As the timer is decrementing, higher values mean the person was on the average early while lower values represent lateness, with 30 being a perfect score.

Following the average, a point-by-point analysis of the data is displayed taking advantage of the VIP's graphics capabilities, and forming the heart of the test's usefulness. While accuracy is certainly a needed rhythmical quality, consistency cannot be neglected. A perfect average could be composed of widely varying data while good consistency could be achieved by someone who, unfortunately, is consistently late or early. Therefore, both parts of the test are essential to a meaningful interpretation of the results.

By setting the Y coordinate to equal each test result (adjusting to center screen by adding a constant value) while incrementing the X coordinate horizontally from 0 to 63 (one for each key press), a visual graph of all the data is seen.

On the VIP, proximate points on the same line either vertical or horizontal appear stuck together. The vertical aspect is ignored here. The degree of the subject's consistency becomes a matter of viewing for the largest number of horizontally connected lines. The idea was to give a quick visual indication of consistency rather than assign a conventional statistical analysis technique to the data. The point-by-point display does this very well and could be adapted to other data needing a quick overall analysis for consistency.

Conclusion

More sophisticated processing of the data could be programmed, and you may want to analyze the data in other ways or compare sets, etc. Turning the TV or tone off could demonstrate the relation between the two references, audio and visual, and form the basis for studying the importance of these factors in helping a person keep time to music. A pure consistency test could be performed with both references turned off.

In addition, the next time you have the opportunity to demonstrate your computer to a skeptical what-does-it-do, what's-it-for audience, you need not break out the Space Wars program right away. Ask instead if your guests would like a new challenge, to learn something about themselves, then announce in your best voice; "Ready? Just follow the bouncing ball!" (continued next pg.)

PIN-8 Super sound Music by David Ruth

AMERICA

Step 1: Load the PIN-8 interpreter.

Step 2: Load the following:

```
0259   BF
02E0   0101 0101 0102 0202 0202 0303 0303 0300
0300   0104 070A 0D10 1114 171A 1D22 2529 0000
0310-037F  0000
0380   0104 070A 0D10 1114 171A 1D20 2327 0000
0390-03FF  0000
0400   006B 6B6D 8A2B 6D6F 6F70 8F2D 6B6D 6B6A
0410   CB72 7272 9230 6F70 7070 902F 6D6F 302F
0420   2D2F 8F30 7234 306F 6DCB 0000 0000 0000
0430-04FF  0000
0500   0066 6668 8628 6A6B 6B6B 8B2A 6B68 6666
0510   C66F 6F6F 8F2D 6B6D 6D6D 8D2B 6A6B 6B6B
0520   8B2B 6B2B 2D6B 6ACB 0000 0000 0000 0000
0530-05FF  0000
0600-06FE  0000
06FF   ED
```

Break Table:

```
0270   1201 E016 01E0 FE12 01E0 1601 E0FE 1201
0280   E016 01E0 FE12 01E0 1601 E0FF 0000 0000
```

Step 3: Store on tape 7 pages.

Listing -

Hexadecimal Address	Chip-8 Instruction	Comment
0200	6A1C	VA=1C X coordinate impact line
02	6B0F	VB=0F . Y " " "
04	A2C0	"I" (a memory pointer) points to impact line
06	DAB1	Display impact line pattern
08	6A1F	VA=1F X coordinate ball-start position
0A	6B00	VB=00 Y " " "
0C	6C00	VC=00 Index value for storing times
0E	6D3D	VD=3D Timer start
0210	3B00	Skip if VB (ball X)=00 (ball at top of display)
12	121A	Go to 021A-test if ball is at bottom
14	6805	V8=05-for later "Key Pressed?" check
16	6901	V9=01-for adding to ball Y coordinate (will go down)
18	FD15	Reset timer (auto decrement x 01 every 1/60 secon
1A	3B1E	Skip if VB (ball Y)=1E (ball at bottom of display)
1C	1224	Go to 0224-ball not at bottom (or at top)
1E	6805	V8=05-for later "Key Pressed?" check
0220	69FF	V9=FF-for adding to ball Y coordinate (will go up)
22	FD15	Reset timer (auto decrement x 01 every 1/60 sec.)
24	A2C1	I="ball" pattern stored in memory
26	DAB1	Display ball @ VA VB XY coordinates
28	4F01	Skip if VF≠01 (when VF=01, ball hit impact line)
2A	FF18	Sound tone for VF (causes short beep)
2C	E8A1	Skip V8≠ key pressed (test if Key 5 is pressed)
2E	1244	Go to 0244 record time
0230	6E02	VE-02 set a utility variable for a loop count
32	6E02	VE=02 repeat (no operation-compensate for record
34	7EFF	VE+FF (same as minus 01)(Loop count -01) loop)
36	3E00	Skip if VE=00 - when loop completed
38	1234	Go to 0234 - equalize timing
3A	A2C1	I=pattern for ball
3C	DAB1	Display @ VA VB (displaying 2nd time erases via Exclusive OR logic
3E	8B94	VB+V9 - add direction value (V9) to ball X (VB) for movement
0240	1210	Go to 0210 - loop until a condition met or done
42	0000	Unused

Record Time When Key 5 is Pressed

0244	F007	V0 = current timer value
46	A400	I = base address of data storage area (64 bytes)
48	FC1E	I = I+VC - index "I" to next empty space
4A	F055	Store timer value (V0) in memory @I
4C	7C01	VC+01 Index variable + 01 for next time

Hexadecimal Address	Chip-8 Instruction	Comment
024E	6800	V8=00 Display key check value (only record once per pass)
0250	3C40	Skip if VC=40 (skips nect after 64 tests done)
52	123A	Go to 023A - continue with test
		Display Average Score
0254	00E0	Erase screen after test is completed
56	0290	Do sub - Average the Test Results
58	A2D0	I=work area @ 02D0 for number conversion
5A	F033	Convert hex value to 3 digit decimal
5C	F265	Load the 3 decimal digits into V0 V1 V2
5E	6A30	VA=30 X coordinate for number display
0260	6B00	VB=00 Y " " " "
62	F129	Point to bit pattern for the number in V1 (V0 ignored)
64	DAB5	Display one digit
66	7A05	VA+5 (move X coordinate over for 2nd digit)
68	F229	Point to bit pattern for the number in V2
6A	DAB5	Display one digit (average is now displayed)
		Point by Point Analysis
026C	6C00	VC=00 Index for cycling through data
6E	6A00	VA=00 X coordinate for data display
0270	A400	I=0400 Point to first data
72	FC1E	I=I+VC Point to next data
74	F065	Get data into V0
76	70F0	V0+F0 hex - adjust data for display purposes
78	A2C2	Point to bit pattern for display
7A	DA01	Display a point
7C	4F01	Skip next if VF≠01 (bit did <u>not</u> hit another)
7E	DA01	Erase the bit- in the off chance data interferes with score
0280	7C01	VC+01 Index for next data
82	7A01	VA+01 X coordinate + 01
84	3C40	Skip when VC=40 after displaying 64 bits
86	1270	Go to 0270 to loop until done
88	FF0A	Wait for key to be pressed (end)
8A	00E0	Erase screen
8C	1200	Go 0200 for restart
		Display Patterns
02C0	FFC0	(FF=Impact line/C0=ball)
C2	8000	(80=point for data display)

Machine Language Subroutine

Average the Test Results

Address	Hex Code	Mnemonic		Comment
0290	F804	LDI		;Begin add test results
92	BF	PHI	RF	
93	F800	LDI		
95	AF	PLO	RF	;Set register RF to 0400-data storage
96	BE	PHI	RE	
97	AE	PLO	RE	;Preset RE (answer) = 00
98	E2	SEX	2	;X (stack pointer) = 2
99	22	DEC	R2	;Decrement stack pointer to free location
9A	4F	LDA	RF	;Get data
9B	52	STR	R2	;Push onto stack
9C	8E	GLO	RE	;
9D	F4	ADD		;Add low 8 bits RE to byte on stack
9E	AE	PLO	RE	;
9F	9E	GHI	RE	;Adding possible carry to
02A0	7C00	ADCI		;
A2	BE	PHI	RE	;High 8 bits RE
A3	8F	GLO	RF	;
A4	FB40	XRI		;Loop until all 64 test results
A6	3A9A	BNZ		;Are added to RE
A8	F802	LDI		;
AA	A6	PLO	R6	;Set R6 to a loop count of 2
AB	8E	GLO	RE	;
AC	FE	SHL		;Shift low 8 bits RE left
AD	AE	PLO	RE	;
AE	9E	GHI	RE	;Shifting the carry bit
AF	7E	SHLC		;
02B0	BE	PHI	RE	;Into the high 8 bits RE
B1	26	DEC	R6	;Decrement the loop counter
B2	86	GLO	R6	;
B3	3AAB	BNZ		;Loop until double precision shift left x 2 completed
B5	9E	GHI	RE	;The high 8 bits RE = answer ÷ 64
B6	56	STR	R6	;Store as Chip-8's variable V0
B7	12	INC	R2	;Increment stack to original position
B8	D4	SEP	R4	;Return

MACHINE CODE Part 7

P. V. Piescik, 157 Charter Rd., Wethersfield, CT 06109

DRVRIN-parallel looks very much like DRVROUT-parallel. We just substitute an input instruction for the output instruction and add another instruction to strip off the parity-bit.

```
00A0                  DRVRIN:  ORG *
00A0   3F A0 R                 BN4 *           ..wait for keypress
00A2   6B                      INP 3           ..read VIP port
00A3   FA 7F                   ANI #7F         ..stip parity
00A5   7B                      SEQ             ..audible feedback
00A6   37 A2 R                 B4 DRVRIN+2 ..wait for release
00A8   7A                      REQ             ..kill feedback
00A9   D5                      SEP 5
```

If the EF-line for the keyboard is not EF4, change the bytes at 00A0 and 00A6 to match. If you're not using port 3, change the input instruction at 00A2. The 7B/7A instructions may be omitted if you want a silent keyboard, or if it has its own beep. For pulsed data strobes, the branch at 00A6 may be omitted.

The branch at 00A6 goes all the way back to 00A2 so if you have an elastomer (ugh!) keyboard, the longer loop will absorb the noise of contact bounce to prevent multiple characters. For clean signals this branch may loop back on itself (37 A6), and for very noisy signals it can loop all the way back to DRVRIN (37 A0). Any signal lasting less than 10 cy. is then ignored. Sometimes this will tend to hang up too much, but you can experiment to see what works best on your system.

Once we turn on Q for the tone, we explicitly force it off! The VIP video interrupt routine will shut it off, but if you don't have a VIP or get away from using the 1861, you won't want Q on forever once you hit a key.

Let's do DODEL next. There are 3 variations to handle different types of devices we've already mentioned. Hopefully if you've scrounged up something else you'll be able to kluge this over to it.

```
00CC                  DODEL:   ORG *          ..for hardcopy
00CC   8E 32 D6 R              GLO E;BZ DODELX ..empty--ignore
00CF   2E 27                   DEC E;DEC 7     ..decr ct, ptr
00D1   F8 5C                   LDI #5C         ..backslash
00D3   D4 00 36 V              SEP 4,A(DRVROUT)..  to terminal
00D6                  DODELX:  ORG *
00D6   D5                      SEP 5
```

```
00CC                 DODEL:    ORG *               ..DEL function
00CC  8E 32 D6 R               GLO E;BZ DODELX ..empty--ignore
00CF  2E 27                    DEC E;DEC 7       ..decr ct, ptr
00D1  F8 7F                    LDI #7F           ..DEL code
00D3  D4 00 36 V               SEP 4,A(DRVROUT)..  to video bd
00D6                 DODELX:   ORG *
00D6  D5                       SEP 5

00CC                 DODEL:    ORG *             ..BS SP BS type
00CC  8E 32 E0 R               GLO E;BZ DODELX ..empty--ignore
00CF  2E 27                    DEC E;DEC 7       ..decr ct, ptr
00D1  F8 08                    LDI 8             ..BS code
00D3  D4 00 36 V               SEP 4,A(DRVROUT)
00D6  F8 20                    LDI #20           ..SP code
00D8  D4 00 36 V               SEP 4,A(DRVROUT)
00DB  F8 08                    LDI 8             ..BS code again
00DD  D4 00 36 V               SEP 4,A(DRVROUT)
00E0                 DODELX:   ORG *
00E0  D5                       SEP 5
```

The code at 00CC-00D0 stays the same in all cases, except for the address in the branch instruction if DODELX moves to 00E0. Select the variation of the code you need. If you need other codes for your device, the method should be obvious. Do NOT, however, attemtp to use PUTLINE for a sequence of codes UNLESS you save and restore R7 (which is in use, pointing to the user's <u>input</u> buffer).

DOCAN code:

```
00E1                 DOCAN:    ORG *
00E1  F8 5C                    LDI #5C           ..backslash
00E3  D4 00 36 V               SEP 4,A(DRVROUT)
00E6  D4 00 63 V               SEP 4,A(PUTCR)
00E9  8E 52                    GLO E;STR 2       ..reset R7 ptr
00EB  87 F7 A7                 GLO 7;SM;PLO 7    ..by subtracting
00EE  97                       GHI 7             ..count from ptr
00EF  7F 00                    SMBI 0            ..16-bit arith!!
00F1  B7                       PHI 7
00F2  F8 00 AE                 LDI 0;PLO E       ..reset count
00F5  D5                       SEP 5
```

You may not have expected to play with R7.1 to subtract an 8-bit value! We have not restricted the user in the location of his buffer, so we must be prepared for the case where his buffer crosses a page boundary.

You may also have thought of decrementing R7 in a loop, "count" times. It would work, but it would take 4+(8*CT) cycles. The code above always takes 20 cy. to reset the pointer and count. The break-even point is a count of 2--the loop is faster only for cancelling a 1-char. line! If the user has a 32-char buffer, the loop takes up to 260 cy.; for a 64-char buffer, up to 516; up to a maximum of 255 chars in 2044 cy.! That's 9 ms.; while you won't notice it in any one place, if we continue to program like that we'll end up with a real dog of a system!

GETLINE comes next, and I want to think aloud before getting into the code. We have only ten bytes left on this memory page, and the entire routine obviously won't fit. The first thing we have to do is check the buffer size the user gives us, and return if it's zero. In PUTLINE and DODEL, we branched to the return instruction at the end of the routine. If we do that here, we'll need a long branch to cross the page boundary, and that will cause glitching if the 1861 is in use. To get around this using a short branch, we'll test for the opposite condition. If it's met, we'll branch around a return instruction; if it's not met, we'll fall through to the return. In PUTLINE we programmed, "IF zero THEN GOTO return ELSE putline"; here we'll program, "IF not zero THEN GOTO getline ELSE return." It's the same logic!

```
00F6                GETLINE: ORG *
00F6   3A F9 R               BNZ *+3
00F8   D5                    SEP 5
00F9   .. .. ..     ..code to do GETLINE starts here
```

We also know that GETLINE will involve a loop to handle a series of incoming characters, and we are concerned whether all of the loop will be on the next memory page. If not, we'll need a long branch to loop, and can expect a LOT of glitching. So we'll start laying out the code for things we have to do before the loop starts to see if we make it to the next page. First, we're required to save R7 (user's buffer address) as part of the GETLINE problem:

```
00F9   87 73 97 73           GLO 7;STXD;GHI 7;STXD..save R7
00FD   .. .. .. ..
```

Three bytes to go. We have to keep a char count, which we decided will be in RE.0. We have to loop for a maximum of n chars, which implies that we will decrement the count as in previous loops. We also know that with DOCAN and DODEL the count may also be incremented and sometimes reset. DOCAN already resets the count to 0, so we must be incrementing the char count; if we control the loop by decrementing its count, we'll need two counter running in opposite directions. (To be consistent with a loop counter being decremented, DOCAN would have to reset it to n, and we didn't do that.) We do not need the double trouble of an up-count and a down-count, so let's loop until the char count reaches the limit, n. We need a copy of n for comparison, so let's keep it in RE.1. The copy of n in D has been clobbered by saving R7, and our subroutine calls will clobber the copy in RF.1. Since we're using RE and we're polite, we have to save it.

```
00FD   8E 73 9E 73           GLO E;STXD;GHI E;STXD..save RE
0101   9F BE                 GHI F;PHI E..copy n
0103   F8 00 AE              LDI 0;PLO E..init char ct to 0
```

OK--we made it! We crossed the page while saving RE, and I tossed in the next two steps before I forget them. If we had fallen short of the next page, we'd simply start the routine at a higher location to keep the entire loop on the same page.

Before we do the middle, let's look at the other end of the loop. Unlike our previous loops, this index (counter) is incremented and will be non-zero (equal to n) when we're done. For normal chars, the count is incremented by 1; for DEL it is decremented by 1 unless it's already 0; for CAN it will be decremented by its entire value (reset to 0). For CR it will be incremented by 1, but CR means we're done so we won't check the count. Until now, we've exited the loops on a condition of "equal" (index=0), which is a small target. Here, where the count isn't always changed by ±1, it's much safer to exit on some other condition (less than, less than or equal, greater than or equal, greater than). Now if something goes wrong and the count becomes greater than n, we stand a better chance of avoiding an infinite loop than if we attempt to match a single value.

The count will reflect the actual number of chars in the buffer. Some of the chars we read will not be counted, and may decrement or reset the count, so we will NOT simply bump the count every time we read the keyboard! Only after we find that the char is not special and is stored in the buffer, can we bump the count. When the count is equal to the limit (n) we have no room left in the buffer and we don't want to loop again. The count should never be greater than the limit, but if that happens, we definitely don't want to loop! Our condition is: IF count LESS THAN limit THEN GOTO start of loop ELSE don't. We don't affect DF if we use XOR to compare the count and the limit, so we'll subtract instead. We have to set up the subtraction so DF will be different for "less than" than it is for "greater than or equal." We're dealing with three possible conditions: less than, equal, and greater than. DF will indicate either less than or greater than after we subtract; D=0 will indicate equal. We'd like to distinguish between the "less than" condition we want and the "less than or equal" condition we don't want, by using only one conditional branch instruction. We don't want to examine both DF and D if we can avoid it. We can subtract either the limit from the count, or the count from the limit; our choice will be the way that results in a different value of DF for "less than" than for "equal."

I check this by doing a little math on the side. Let's assume that the limit is #10. Counts of #0F (nearest less than), #10 (equal) and #11 (nearest greater than) will be tested. If we subtract the limit from the count we get (=DF.D): 0F-10=0.FF; 10-10=1.00; 11-10=1.01. DF=0 for less than, and DF=1 for equal, so we'll subtract the limit from the count.

When you're trying to set up the end of a loop like this, remember that you're dealing with three conditions: less than, equal, and greater than. "Less than or equal" and "greater than or equal" are logical expressions (logical OR) built up from the three conditions! Also note that "less than or equal" is the same as "not greater than"; "less than" is the same as "not greater than or equal"; etc. Finally, if we check the blue card (MPM-920) or manual (MPM-201), we find that a 33 instruction may be BDF (branch on DF=1), BPZ (br on positive or zero), or BGE (br on greater than or equal). This implies that zero is a positive number.

The end of the loop will look like this:

```
xxxx  9E 52              GHI E;STR 2   ..M(RX)=limit
      8E F7              GLO E;SM      ..get ct; ct - limit
      3B 06 R            BL GETLOOP    ..loop if less than
```

We'll know the address (xxxx) when we fill in the rest of GETLINE. During the loop we first read the keyboard. Next we check for special characters with a series of XRI instructions which give us D=0 if the char matches. If we find CR, we store it in the buffer, echo with PUTCR, bump the count, and exit. If we find CAN, we just call DOCAN and loop. If we find DEL, we just call DODEL and loop. If we don't match a special character, we store the char, bump the buffer pointer and count, then check the count to see if we can continue to loop. With CAN and DEL we don't check the count, since it had to be OK (less than) for us to be in the body of the loop. DOCAN and DODEL will decrement the count by at least 1, so it must still be less than the limit.

```
00F6                   GETLINE: ORG *
00F6  3A F9 R                   BNZ *+3            ..return if n=0
00F8  D5                        SEP 5
00F9  87 73                     GLO 7;STXD         ..else save R7,RE
00FB  97 73                     GHI 7;STXD
00FD  8E 73                     GLO E;STXD
00FF  9E 73                     GHI E;STXD
0101  9F BE                     GHI F;PHI E        ..copy n to RE.1
0103  F8 00 AE                  LDI 0;PLO E        ..init count=0
0106                  GETLOOP:  ORG *
0106  D4 00 A0 V                SEP 4,A(DRVRIN)..read keyboard
0109  FB 0D                     XRI #0D            ..is it CR?
010B  32 2B R                   BZ GLCR            ..yes, branch
010D  FB 15                     XRI #15            ..is it CAN?
010F  3A 16 R                   BNZ GLCKDEL        ..  no, check DEL
0111  D4 00 E1 V                SEP 4,A(DOCAN)     ..  yes, DOCAN
0114  30 06                     BR GETLOOP         ..    and loop
0116                   GLCKDEL: ORG *
0116  FB 67                     XRI #67            ..is it DEL?
0118  3A 1F R                   BNZ GLCHAR         ..  no, norm char
011A  D4 00 CC V                SEP 4,A(DODEL)     ..  yes, DODEL
011D  30 06 R                   BR GETLOOP         ..    and loop
```

```
011F                GLCHAR:  ORG *
011F   9F 57                 GHI F;STR 7       ..store char in bfr
0121   17 1E                 INC 7;INC E       ..bump ptr, count
0123   9E 52                 GHI E;STR 2       ..  end loop here
0125   8E F7                 GLO E;SM          ..count - limit
0127   3B 06     R           BL GETLOOP        ..loop if less
0129   27 38                 DEC 7;SKP         ..decr ptr to jam CR
012B                GLCR:    ORG *             ..  don't bump count
012B   1E                    INC E             ..  exc. for real CR
012C   F8 0D 57              LDI #0D;STR 7     ..store CR in bfr
012F   D4 00 63 V            SEP 4,A(PUTCR)    ..echo CR LF NULs
0132   8E BF                 GLO E;PHI F       ..save count in RF.1
0134   60                    IRX               ..restore regs
0135   72 BE                 LDXA;PHI E
0137   72 AE                 LDXA;PLO E
0139   72 B7                 LDXA;PHI 7
013B   F0 A7                 LDX;PLO 7
013D   9F                    GHI F             ..D=count
013E   D5                    SEP 5             ..return
```

There are two ways to check the chars. We can restore the input char from RF.1 each time with the code we want to match as the immediate data in the XRI instruction. This costs us an extra instruction (1 byte, 2cy.) for each check after the first. Or as we did above, we can chain the data in the XRIs. This saves time and memory but the code is less readable without comments. The check for CR (0D) is obvious. Next, instead of XRI'ng with the actual code for CAN (18), we XRI with CR XOR CAN, 15; 0D XOR 15 = 18, the code we want. For DEL, we XRI with CAN XOR DEL, 67; 18 XOR 67 = 7F. If the author didn't provide us with comments, we can determine the codes he's trying to match by performing the XOR operations. 0D is the first code; the second code is 0D XOR 15 = 18; and 18 XOR 67 = 7F, the last code. He's looking for CR, CAN, and DEL. We could have subtracted to match the codes: SMI #0D; BZ GLCR; SMI #0B (18-0D=0B); BNZ GLCKDEL; SMI #67 (7F-18=67); BNZ GLCHAR. Since subtraction also affects DF, it's useful for checking that the input code is within some range of codes without having to know the exact code or checking every possibility.

At 0129 we have dropped out of the loop. The last thing we do before returning is always drop a CR into the buffer and echo it. This is done at GLCR, but there are two ways we could have gotten here. Both situations differ slightly. If we drop out of the loop, the buffer is full and the pointer has been bumped to the next location. Usually this is the next free slot, but now it's beyond the buffer. So we back up the pointer to drop the CR into the buffer. The non-special char which filled the buffer also bumped the count. While we want to count the CR, we don't want to count both the CR and the char it replaces, so we skip the INC E instr. On the other hand, if we got to GLCR because a CR was entered, the pointer is OK, but we have to count the CR.

To use GETLINE we have to have a buffer, pointer, size, and then call:

```
013F  F8 01     R            LDI A.1(TXTBFR)
0141  B7                     PHI 7
0142  F8 4E     R            LDI A.0(TXTBFR)
0144  A7                     PLO 7              ..R7=bfr addr
0145  F8 50                  LDI #50            ..size=80 chars
0147  D4 00 F6  V            SEP 4,A(GETLINE)..read keyboard
014A  D4 00 7C  V            SEP 4,A(PUTLINE)..dup echo
014D  23                     DEC 3              ..halt main pgm
014E              TXTBFR:    ORG *              ..buffer
019E                         ORG TXTBFR+80      ..next available
```

We don't need to call PUTLINE since the chars are echoed one by one as you enter them, but you might like to see what an expert typist (the computer) does when you give your two fingers a rest!

STORAGE MAP

LOAD	ENTR	SIZE	NAME
0000	0000	0015	INIT
0015	0016	0012	CALL
0027	0028	000F	RET
0036	0037	000A	BAUD
0040	0040	0023	DRVROUT
0063	0063	0019	PUTCR
007C	007C	0024	PUTLINE
00A0	00A0	002C	DRVRIN
00CC	00CC	0015	DODEL
00E1	00E1	0015	DOCAN
00F6	00F6	0049	GETLINE
013F			next available

Your storage map may be different if you used shorter versions of any routines which are device-dependent.

The next steps would be to get the next character from the input buffer for the program to process (GETCHAR), and the next word from the line (GETWORD). All these involve is the handling of two buffers via pointers and the same kind of character checking to determine when the end of a word or line has been reached. If you've learned anything so far, you should be able to write GETCHAR and GETWORD without too much trouble.

The next few things on the agenda are converting from ASCII characters to binary numbers and back, for both decimal and hexadecimal, and multiplication and division of binary integers. You should try to forge ahead on these projects, using the steps we have here (Donovan's). Then you'll be able to compare your efforts with the discussion in Machine Code. You may even find that you're ready to wander through the land of computing on your own!!

VIP Hobby Computer Assn.
32 Ainsworth Avenue
East Brunswick, N. J. 08816

FIRST CLASS

A final word:

A software note: the August '82 issue of Popular Electronics magazine has an article entitled "A 16-bit Math Package for ELF Computers," by R. S. Fitzgerald (page 60) for those of you who might be interested. Although the program is designed to fit into a 256-byte ELF, it should be not too difficult to modify for the VIP. The article describes in detail how the 134-byte package of routines works and gives a very good example of how a stack is used to store, process, and pass data. It was Pop'tronics that perhaps got the whole ball rolling for the 1802 when it came out with the construction article for the ELF Computer about 5 years ago. This particular issue of Popular Electronics may no longer be available at your local newsstand, but it should be available at most libraries, or perhaps at an electronics supply house that stocks magazines until they sell, rather then returning the out of date issues.

Also, I had a note from George E. Frater, 1730 Mariposa Dr., Las Cruces, NM 88001. George was looking into the possibility of substituting super-low-power 5114 RAM chips for the 2114s. The 5114 is supposed to be a pin-for-pin replacement and would make battery oeration for the VIP very practical, due to its very low current drain. The only problem is: where can you find a 5114? None of the usual mail-order houses seem to carry them. Anybody out there have any sources for that chip? George does know of the 6514, but that chip requires some logic changes on the VIP.