VIPER

April - May 1982

Volume 4, Number 1 Journal of the VIP Hobby Computer Assn.
The VIPER was founded by Aresco, Inc. in June 1978

****

Contents

EDITORIAL

With this issue of VIPER we begin the second year of publication for VIPHCA. All the questionaires have not yet been returned, but the consensus is that VIPER is just fine in its present format. A few people wished that material would not be crowded right up to the paper edge, so that they might punch holes in the VIPER for storage in a 3-ring binder. And that is a very reasonable request. So, henceforth I'll try to see to it that there is enough room on the inside margins to allow holes to be punched without obliterating any text.

The opinion was divided on the idea of an honorium for authors. The people returning their questionaires early on were almost universally against any payment to contributors. Later respondents were more in favor. I purposely did not editorialize either for or against the idea to encourage you to give a candid answer. My own opinion favors giving the honorium. The money involved is not really very much compared to our other expenses, and it would at least partially compensate authors for at least the expense of mailing in their material. There is almost no way we could honestly pay full value for the material which has been published in VIPER, so authors' contributions will still be a labor of love even if we do give the honorium. I also favor some sort of compensation for the purely selfish reason that so much of the material is so good that it can be used as is, requiring little additional work for yours truly to get it ready for publication.

Even of those in favor of the honorium, quite a few did not want it retroactive. Perhaps a compromise might be to have the honorium, but make it effective with material published from volume 4 on. One member suggested that perhaps instead of a cash payment, full or partial credit toward annual dues would be appropriate. This is also a fine idea. Those of you not in a position to contribute material would only have to send in dues. And those who have material published, and without whom there could be no VIPER at all, would not be called upon to do more than their fair share. There are, as I said, still more members to be heard from, so a decision won't be made until all the returns are in.

You may also have noticed that some of this issue's material, including this editorial, have been printed with a dot-matrix printer, rather than a typewriter. This has been due to the kindness and generosity of my good friend Jeff Buerger, WB2WIH, who has loaned me his Microline 82A printer for several weeks. This text is being prepared on a Radio Shack Color Computer with the Telewriter word processing program. I'd like to solicit your opinions regarding this type of print, and whether you feel it is better or worse than conventional typewritten material. The people who create word processing programs and systems woul have us believe that this is the only really effective way to prepare written material. I don't think that the final word has been heard on that, but from what I can see with this system, they have a very compelling argument! It sure is a lot easier and quicker to let a computer do the hard part of preparing material and let you spend your energies on the content. I'd like to seriously consider buying a printer for use with the VIPER if you feel that it would be worth while. For a while I had the use of a Daisy Wheel type printer which was donated by a friend. Unfortunately, the machine was in poor shape and finally went kaput altogether. And that was a shame, since you can't ask for a better printer that a Daisy Wheel type for this application. Anyway, let me know what you think. And here's to a great Volume 4!

MINI-CALCULATOR

Using machine language subroutines @0600, this program turns the VIP into a simple calculator with four hexpad functions. These are:

```
Key C = Multiply     :NOTE -- Following each
Key D = Divide       :  numerical entry, you
Key E = Add/equals   :  may press any of
Key F = Subtract     :  these
```

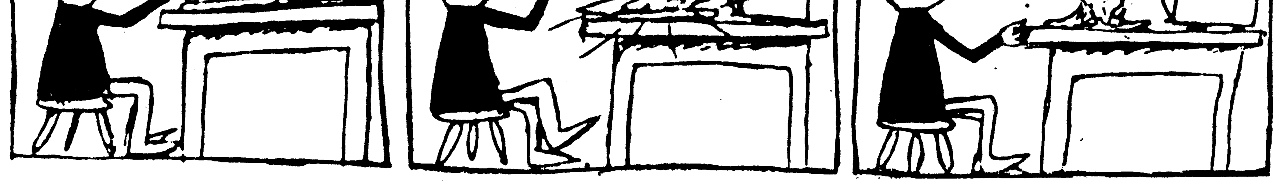
The capabilities of the calculator are limited to positive values less than or equal to 255. For results greater than that an E error message is given. To restart for new calculations, press Key Ø.

MINI-CALCULATOR LISTING

```
0200   BEGIN: 6C00   ;VC=00 -- VX for display
  02          6D00   ;VD=00 -- VY for display
  04          2276    GTKEY -- Do user input sub
  06          3F00   ;SK=00 -- Skip on no overflow
  08          126A    ERROR -- Jump to signal error
  0A          8430   ;V4=V3 -- Put input into V4
  0C          8E00   ;VE=V0 --   and last digit into VE

       ;DISP Function

  0E          A300   ;ZEROS -- I addresses zero bytes @ 0300
0210          400C   ;NE OC -- On key C, set
  12          A292    MUL   --   I=Multiply sign
  14          400D   ;SK≠D  -- On key D, set
  16          A29E    DIV   --   I=Divide sign
  18          400E   ;SK≠E  -- On key E, set
  1A          A298    ADD   --   I=Add sign
  1C          400F   ;SK≠F  -- On key F, set
  1E          A2A4    SUB   --   I=Subtract sign
0220          DCD5   ;SHOW  -- Display function
  22          7C06   ;VC+6  -- Adjust X coordinate
  24          2276    GTKEY -- Get next input
  26          4F01   ;SK≠1  -- Skip on no error
  28          126A    ERROR -- Jump to signal error
  2A          A2AA    EQUAL -- Set I to equal sign
  2C          DCD5   ;SHOW  -- Display equal sign

       ;Functions

  2E          3E0C   ;SK=C  -- Skip into multiply
0230          1238    NEXT1 -- Go next function
  32          0630   ;MLS   -- Do multiply MLS
  34          F4F3          -- For V4*V3
  36          1256    OVER? -- Go check answer
```

```
      ;DIVIDE

0238  NEXT1:  3E0D   ;SK=D   -- Skip into DIVIDE
  3A          1242    NEXT2  -- Go next function
  3C          0600   ;MLS    -- Do DIVIDE MLS
  3E          F4F3           -- For V4/V3
0240          1256    OVER?  -- Go check answer

      ;ADD

  42  NEXT2:  3E0E   ;SK=E   -- Skip into ADD
  44          124A    NEXT3  -- Go next function
  46          8434   ;V4+V3  -- ADD
  48          1256    OVER?  -- Go check answer

      ;SUBTRACT

  4A  NEXT3:  3E0F   ;SK=F   -- Skip into SUBTRACT
  4C          126A    ERROR  -- Go ERROR message
  4E          8435   ;V4-V3  -- SUBTRACT
0250          4F00   ;SK≠0   -- Skip on no underflow
  52          126A    ERROR  -- Go signal ERROR
  54          125A    ANSR   -- Go DISPLAY answer

  56  OVER?:  4F01   ;SK≠1   -- Skip on no error (not subtract)
  58          126A    ERROR  -- Go signal ERROR

  5A  ANSR:   7C06   ;VC+6   -- Adjust X coordinate
  5C          8040   ;V0=V4  -- Answer is in V4
  5E          22B0    NUMB3  -- Go DISPLAY V0=V4=ANSWER

0260  END:    F00A   ;KEY?   -- Wait for restart
  62          3000   ;SK=0   -- On Ø, go restart
  64          1260    END    -- Else other keys loop
  66          00E0   ;ERASE  -- Clear display
  68          1200    BEGIN  -- Restart

  6A  ERROR:  600E   ;V0=E   -- "E" for Error
  6C          F018   ;TONE   -- Sound tone
  6E          F029   ;SET I  -- I=bits for V0 (E)
0270          7C0C   ;VC+C   -- Adjust VX
  72          DCD5   ;SHOW   -- DISPLAY E
  74          1260    END    -- Go      restart

      ;SUBROUTINES

  76  GTKEY:  6300   ;V3=00  -- Preset answer
  78  DIGIT:  F00A   ;KEY ?  -- Get input
  7A          6109   ;V1=9   -- Test if greater than
  7C          8105   ;9-V0   --   9 by subtracting
  7E          4F00   ;SK≠0   -- If ok, skip
0280          00EE   ;RET    -- Else return 0 in V3
  82          F029   ;SET I  -- I=bits for input digit
  84          DCD5   ;SHOW   -- Show the digit
  86          7C05   ;VC+5   -- Adjust X coordinate
```

```
0288         0650   ;DOSUB -- Combine input (MLS)
 8A          F3F0          -- V3 from V0 (for the MLS)
 8C          4F00   ;SK≠0  -- Skip on no overflow
 8E          1278    DIGIT -- Loop next press
0290         00EE   ;RET   -- Return

      ;DISPLAY PATTERNS

  92  MUL:   00 50 20 50 00 00
  98  ADD:   00 20 70 20 00 00
  9E  DIV:   20 00 70 00 20 00
  A4  SUB:   00 00 70 00 00 00
  AA  EQUAL: 00 70 00 70 00 00

02B0  NUMB3: A2CA    C-3DD -- I addresses work space
 B2          F033   ;CNVRT -- Convert V0 to 3 digit decimal
 B4          F265   ;GET   -- Get digits into V0,V1,V2
 B6          F029   ;SET I -- I addresses bits for V0 value
 B8          DCD5   ;SHOW  -- Show 1st digit
 BA          7C05   ;VC+5  -- Add 5 to X coordinate
 BC          F129   ;SET I -- I addresses bits for V1 value
 BE          DCD5   ;SHOW  -- Show 2nd digit
02C0         7C05   ;VC+5  -- Add 5 to X coordinate
 C2          F229   ;SET I -- I addresses bits for V2 value
 C4          DCD5   ;SHOW  -- Show 3rd digit
 C6          7CF6   ;VC-A  -- Reset X coordinate
 C8          00EE   ;RETN  -- Return from subroutine
 CA   C-3DD: 0000
 CC          0000

                   MLS - DIVIDE & MULTIPLY

0600  E6 45 A6 45 A7 F8 00 AE 07 3A 12 F8 FF A6 F8 01
  10  56 D4 07 F5 56 3B 1A 1E 30 12 07 F4 AD 8E 56 F8
  20  00 A6 8D 56 F8 FF A6 F8 00 56 D4 00 00 00 00 00

      ;END of CALCULATOR PROGRAM
```



The "DO NOT USE" Instruction

by Bob Casey

Those of you who have worked with 1802 machine code probably have noticed that all 256 possible bytes, except one, represent an instruction to the 1802. This one unused byte is "68" Hex and is merely named "Do Not Use." You may have wondered what it might do if you used it in a program.

Code "68" turns out to be an "Input 0" instruction, just like the other input instructions. When the 1802 sees a "68" instruction, it tries to select input port 0 by setting the N lines to all low and setting MRD high. The N lines are always low for any non-I/O instruction and MRD is high when the 1802 doesn't want to read memory. These conditions occur for instructions like "Put High Register N" where the data bus has stuff on it. There's no real way for the outside world to know the difference between "Input 0" and "Put High Register N" and be able to feed input 0 at the right time.

This phantom input instruction exists because the PLA in the 1802 that decodes the instructions thinks "68" is an I/O instruction and determines that it IS input because the N part of the byte (lower nibble) has a high in its most significant bit (8 hex = 1000 binary) and because the other N bits are low, decides that it means "Input 0." By the way, a phantom "Output 0" instruction doesn't exist because code "60" is decoded by the PLA to mean "Increment Register X." This is true for both the RCA and the Hughes 1802 chips.

Here's a program to demonstrate the "68" instruction:

```
Address

0000-1FFF  CHIP-8 Interpreter              MLS

0200       6000 Set Display                0600     F8
           6108 Position                            06   Load Reg. E with 06FA
           0600 Go to MLS                           BE
           FA29                                     F8
           D105 Display LSNibble                    FA        "       "
           8AA6                                     AE
           8AA6                                     EE   Make Reg. E the X Reg.
           8AA6 Shift MSN to LSN                    68   Input 0
           8AA6                                     D4   Return
           FA29
           D005 Display LSN
           00E0 Erase display
           1204 Return for another input
```

Flip to RUN. You should se "FF" flickering on the screen. Take a 5K Ohm resistor, one end tied to ground, and touch the other end to line D7. You should now see "7F displayed. Touch D6 and you should get "BF." What you're doing is causing selected data bits to be pulled down softly, rather than pulled up as is normally done by the 22K Ohm resistors inside the VIP. You can't actually ground the bits or else you'll kill the program (and maybe a chip).

As you can see, the "68" instruction is rather useless, but it won't blow up your VIP!

NOW AVAILABLE

ELFISH

AN INTERPRETIVE DEVELOPMENT SYSTEM

This 2K package was written for 4K and larger Super Elf's and Elf II's. The package is page relocatable and can reside in ROM. It contains a 1K editor designed to make program entry and modification easy, as well as a 1K interpreter. The editor displays 4 lines, each consisting of a two byte address followed by two bytes of memory content as is shown on the right. REPLA in the diagram indicates that the editor is in the replace mode; successive two byte instructions are entered and replace the memory contents at the indicated line. Other commands are insert, delete, go to, scan up, scan down, execute, assemble, and change address mode (absolute or relative). The interpreter features 16 bit variables and 64 ASCII display symbols; interpretive code is fully relocatable.

A 32 page booklet which includes annotated hex dumps of both the interpreter and the editor, together with a sample program, and instructions for use is available for $6.00 from the address below. Also available are casette tape versions in either Super Elf or Elf II format (please specify) for $6.00 postpaid. A special price is offered when both the booklet and casette tape are ordered at the same time, $10.00 postpaid. Further information can be obtained by writing to:

Paul Moews
34 Circle Drive, RFD 3
Willimantic, CT 06226 USA

```
 0000 0000
>0002 7233<A
 0004 4000
 0006 5000

REPLA FOFE
```

```
 0002 FOFE
>0004 4000<A
 0006 5000
 0008 0000

REPLA FE
```

```
 0002 FOFE
>0004 4000<A
 0006 5000
 0008 0000

INSER 8ABC
```

```
 0004 8ABC
>0006 4000<A
 0008 5000
 000A 0000

INSER BC
```

```
 0004 8ABC
>0006 4000<A
 0008 5000
 000A 0000

DELET  3
```

```
 0004 8ABC
>0006 5000<A
 0008 0000
 000A 0000

DELET  3
```

What Can You Do with One Page of Memory ?

by Steven Vincent Gunhouse
2629 T. R. 247
Arcadia, Ohio 44804

In the past few months I have written several programs for the RCA VIP microcomputer, many of which take up only one page. I have decided to send in a couple of the more interesting ones so that other people could use them or the techniques which were employed in their writing.

Both of these programs are in machine language and tend to be very involved, so unless you know the 1802 op codes as well as you do English I would reccommend that you simply load them and run. It is doubtful that you would get any further than initializing the registers without much help.

I say this because I would probably have trouble interpretting them if I had not written them myself and I personally do know the 1802 that well. I wrote these and many other programs without the use of flowcharting or an assembler in only about half an hour for each successive version.

My first version of any program is generally fairly simple, at least compared to the next one. With each successive version the program becomes more compact, complicated, and often faster. Take these two programs, for example.

The first program is a Morse Code keyboard program with a 256 character buffer and easy to access speed adjustments. The first version was written up by David Barber, WD8AJQ, a blind radio amateur from Defiance, Ohio. I had volunteered to type it up for him and decided I could do a better job than he did. The version here is the sixth I wrote and bears only a slight resemblance in function to Dave's original, though in structure it bears almost none.

The second program is a simplified version of life as described in VIPER, volume 1, issue 5, page 10, by Brian Astle. I have not seen his program, so I can not say how mine compares other than that it is only a tenth the size and takes about 25 times as long for each generation. Also, it works only with full wrap-around. The program itself is the tenth generation and is about 15 times as fast as my original version, so I am quite satisfied with it. Some versions of life on other computers take over 100 times as long as this and do not have anywhere near these features. If I expanded this program beyond one page I could probably get it as fast as Brian's, but I see no need for that. Here are the programs.

MORSE CODE KEYBOARD

```
Page 00
0000    80 B2 B4 90 B1 B3 B6 F8  FF A2 A4 F8 80 BE F8 66
0010    A6 F8 97 A3 22 9E A1 D6  82 51 84 F3 32 17 04 FB
0020    92 32 2A FB 01 3A 31 F8  FF 38 84 A2 F8 FF A4 30
0030    15 04 FC 20 C7 04 C4 24  A1 92 B4 01 3A 41 FE 92
0040    38 FE BF 32 56 D6 9E 7E  7E 7E AE 7B D6 2E 8E 3A
0050    4C 7A D6 9F 30 41 F8 05  AE D6 2E 8E 3A 59 30 15
0060    7A 37 61 E2 37 61 3F 66  E2 3F 66 6B FA OF BB 38
0070    12 12 30 82 73 FB 91 32  60 FB 19 32 70 94 B2 C8
0080    ws DO D3 37 81 DO 37 82  38 DO D3 3F 89 DO 3F 8A
0090    E2 6B F9 80 30 74 D6 E1  9B B5 95 25 3A 9A 30 96
00A0    44 80 80 16 54 B4 54 80  00 80 80 80 80 8C 80 80
00B0    75 80 80 80 80 80 80 16  00 80 80 80 80 80 80 80
00C0    80 AC 4A 80 80 80 80 7A  B6 B6 80 80 CE 14 56 94
00D0    FC 7C 3C 1C OC 04 84 C4  E4 F4 E2 AA 80 80 80 32
00E0    44 60 88 A8 90 40 28 DO  08 20 78 BO 48 EO AO FO
00F0    68 D8 50 10 CO 30 18 70  98 B8 C8 B6 94 B6 8C 00
```

Speed is controlled by pressing CONTROL 1, then desired speed. The first entry from the ASCII keyboard is regarded as a speed value no matter what is pressed. A speed value of 7 is equivalent to 11 words per minute. The fastest speed value, 1, is equivalent to 77 wpm, a speed which I sincerely doubt any person could comprehend.

CONTROL 2 repeats whatever message was last entered into the buffer. CONTROL 3 resets the buffer for entry of another message. Any message you enter is automatically stored in the buffer and sent out over the Q line.

The backspace actually does backspace in the buffer for purposes of correcting errors. However, do not use this key if the character you are trying to correct has already been sent by the VIP as this will mess up the buffer. A delete is valid under all conditions as it simply sends eight dots.

Addresses 00A0-FF are an upper-case only look-up table. If you do not like the code (or lack thereof) which I have used for any of the standard ASCII character set it is simple to create your own, if it consists of no more than 7 dots/dashes and contains no letter- or word-spaces. Simply write down the code you desire on a piece of paper. Now, write a 1 over every dash and a 0 over every dot. Follow this by a 1 and enough 0's to bring the total number of digits to 8. Now you simply convert this to hexadecimal and put it at the appropriate location in the table(e.g., D is -.. ; which is 10010000 or 90 in loc. 00E4). The program accepts either upper or lower case ASCII.

LIFE

```
Page 00
0000    92 A4 90 B3 B4 B5 F8 01  B2 F8 81 B1 81 A2 F8 46
0010    A1 F8 18 A3 F8 02 BB D3  9B BE 82 AE EE 95 73 8E
0020    3A 1D 5E E2 69 9B D4 5C  00 D4 86 D4 86 E2 F8 10
0030    FF 01 3B 25 52 62 22 E2  3E 30 FB FF A5 05 A5 EE      out 2
0040    D5 30 25 26 27 D3 27 16  D3 16 17 D3 17 26 D3 9C      EF3
0050    F1 5E D3 9C FB FF F2 5E  D3 36 59 D3 E2 BE 86 FA
0060    07 AF 86 FA 3F A6 F6 F6  F6 52 87 FA 1F A7 FE FE
0070    FE F4 52 AE EE 84 BC 8F  32 58 9C F6 2F 30 75 D5
0080    BF 43 A5 9F 30 7F 9C F3  5E 00 EE D3 F8 91 A3 D3
0090    61 9B BE 90 BA 95 AE AA  4E 5A 1A 8E 3A 98 95 A6
00A0    A7 38 16 16 17 95 AC 90  BD D4 5C 02 FF 08 52 AE
00B0    9C AF 9B AD 8F FE 3A C4  8E FA 07 3A C2 8E F9 07
00C0    AE 38 2E 92 AF F2 32 C9  1C 8D 2D 3A B4 9D 3A AB
00D0    26 26 27 27 9B D4 5C 8C  FF 03 FA 0E 3A E2 D4 4F
00E0    30 E4 D4 53 87 3A A2 86  3A A3 30 23 F8 90 A3 D3
00F0    58 59 EC 11 86 8C 49 4A  4C 47 4F 4D 46 44 43 53
```

Key	Function	Stored at
0	Erase dot at present location(routine @ 53)	FF
1	Move cursor up and left (@ 43)	FE
2	Move cursor up only (@ 44)	FD
3	Move cursor up and right (@ 46)	FC
4	Move cursor left only (@ 4D)	FB
5	Place dot in present location(@ 4F)	FA
6	Move cursor right only (@ 47)	F9
7	Move cursor down and left (@ 4C)	F8
8	Move cursor down only (@ 4A)	F7
9	Move cursor down and right (@ 49)	F6
A	Begin finding next generation; video on (@ 8C)	F5
B	Invert dot at present location (@ 86)	F4
C	Clear screen (@ 11)	F3
D	Find next generation; video off (@ EC)	F2
E	Wait (cursor stops flashing while held (@ 59)	F1
F	Not used	F0

If the placement of the key functions does not appeal to you, simply change the byte at the location indicated to the address of the desired routine. The machine only responds to keypress when the cursor is present on screen and flashing (i.e., when the machine is not busy finding the next generation). It takes the machine 4.5 sec. to find the next generation regardless of how many dots are on the screen when the display is off, or 9 seconds with it on. After finding a generation the cursor will always appear in the top left corner of the screen.

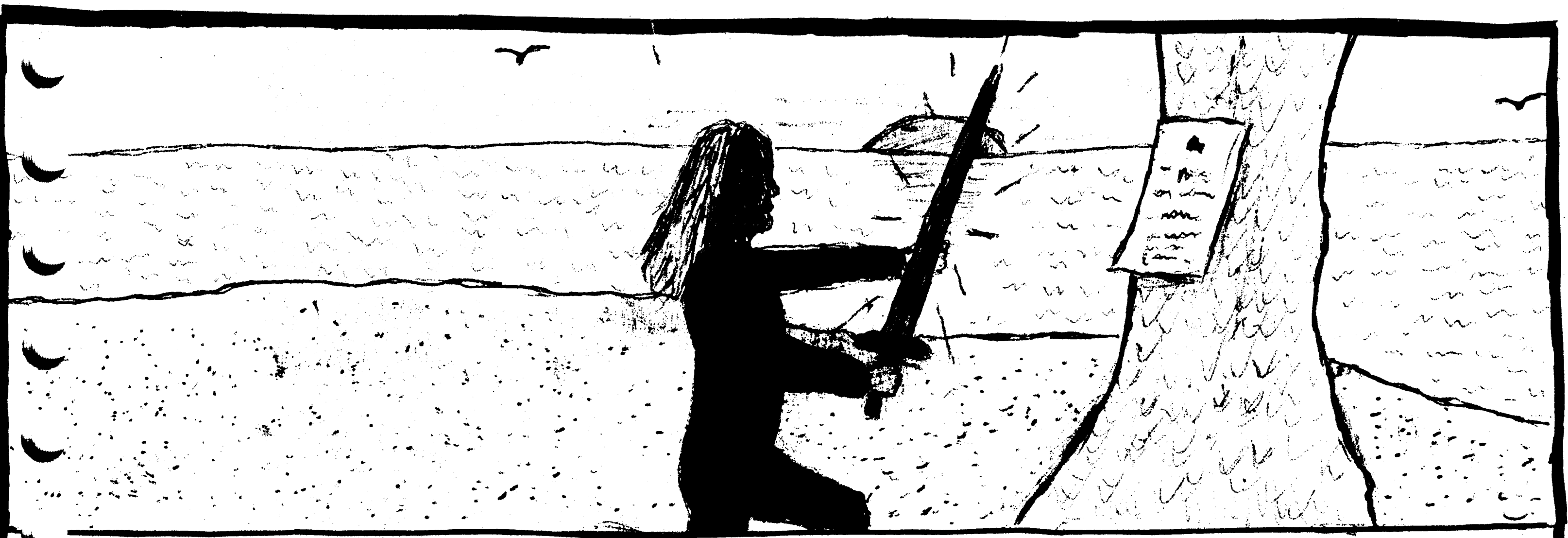
If somehow you get bored of life as it is, you can change the rules by changing the four bytes at locations 00D8-DB. However, be careful to make sure that only some of the numbers 0-9 return a 0 while some don't. If you don't watch this, you will have the screen totally one color after only one generation and remain that way till you enter other dots or change the rules again.

I am willing to help other programmers with their programs in either machine language or CHIP-8. I have written several games, a character generator, and have even converted several ELF programs to run in a VIP, including their BASIC; so I can probably handle almost any sort of program you can dream up.

The programs by David Barber should be available in this issue of VIPER.* I will probably write to VIPER later with some of my earlier programs and will probably furnish the software library (if such is still in existence) with a copy of my BASIC. I'll have to see how much time I have once school starts again in September.

* published in VIPER 3.06

Until next time,
Steven Vincent Gunhouse
WD8NVS

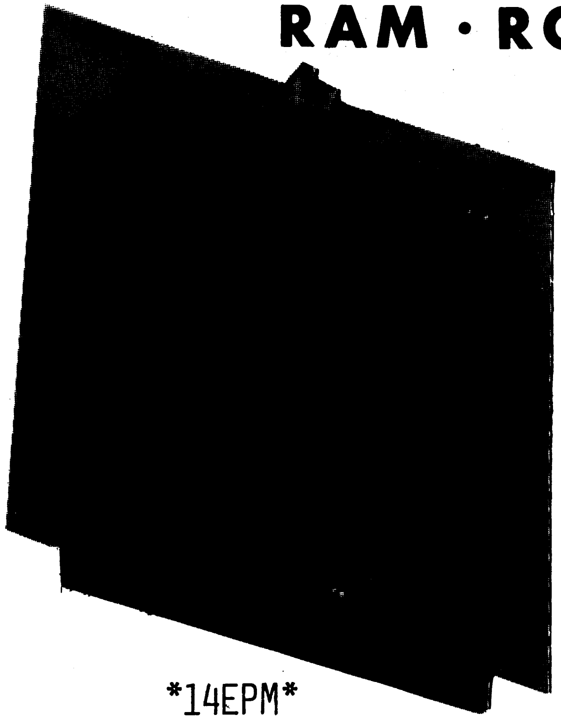


QUEST OF THE ENCHANTED SWORD

If you own a 4K VIP with an ASCII keyboard, this is the program for you. QUEST OF THE ENCHANTED SWORD is an adventure program that takes place in the legendary kingdom of Camelot, shortly after King Arthur's death. In addition, three "mini-adventures" in Tiny Basic are included. Send $8.95 to:
VIP ADVENTURE,UNLTD. 168 Pond St. Sharon,MA. 02067

- Soon to come: VIP-MAN, a VIP version of the popular arcade game PAC-MAN by Midway. Features 64 x 64 resolution, color, and sound effects! -

RAM · ROM for VIP


14EPM

THE 14EPM GIVES YOU ALMOST INSTANT VP-701 FLOATING POINT BASIC. JUST SWITCH TO RUN, PRESS "1" ON HEX KEYPAD, AND PRESS "W" OR "C" ON YOUR ASCII KEYBOARD TO SELECT WARM OR COLD START. USES 5V 2716S.

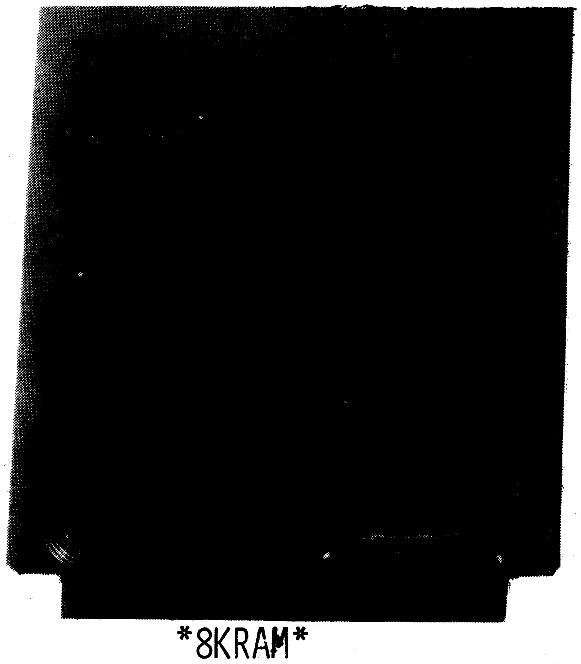
BARE BOARD WITH DATA...$ 39.00
A&T LESS EPROMS........$ 59.00
A&T W/ EPROMS PROGRAMED WITH
DATA PROVIDED BY YOU*..$119.00
A&T W/ RCA'S VP-701**..$139.00

*DATA MUST BE ON VIP COMPATIBLE CASSETTE IN 2K BLOCKS.
**STATE YOUR HIGHEST RAM ADDRESS AND IF YOU WANT COLOR COMMANDS.

THE 8KRAM CARD GIVES YOU TWO 4K BLOCKS OF STATIC RAM ADDRESSABLE TO ANY 4K BLOCK IN VIP MEMORY. USES POPULAR 2114 RAMS.

BARE BOARD WITH DATA.$ 49.00
A&T LESS RAMS........$ 79.00
A&T W/ RAMS..........$129.00

NOTE: 8KRAM CARD REQUIRES APROX 600MA FROM 5V LINE.


8KRAM

PACKAGE SPECIAL

2-8KRAM, 1-14EPM BARE BOARDS
WITH DATA...................$ 99.00
2-8KRAM, 1-14EPM A&T W/ RAMS
AND VP-701 BASIC............$349.00

(CALIF. RESIDENTS ADD SALES TAX.)

G. J. KRIZEK
722 N. MORADA AVE.
WEST COVINA, CA.
91790

MACHINE CODE

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

TOP-DOWN DESIGN and BOTTOM-UP CODING "Top-down" design is simply what we did with the algorithm for the ADD routine. We began with a broad abstraction and few details, then broke it down into smaller, more detailed, parts. To "add," we needed to do some housekeeping, to determine where the data would be, what it would look like, etc. We figured out what to do step-by-step and finally arrived at the smallest details-- 1802 instructions. The design phase ends just short of writing those instructions.

Writing the instructions is "coding." I prefer to do this from the "bottom up" for two reasons. First, working in machine code, we have almost no established conventions (protocols) for dealing with subroutines to call, return, and pass data. There is no compiler or assembler to create linkages between routines, nor any operating system to which we must conform. These details are ours to invent and write. Second, with all these details and our assumed lack of experience, we're likely to overlook something during the design phase, so bottom-up coding gives us a second chance.

If routine A calls routine B, we'd design A first, then B. If we also write A first, we may find details in B which we left out, or did awkwardly in A. We may have forgotten to pass some data, or passed it in an inconvenient order, or .... Whatever the headache, we might have to rewrite routine A when we're done with B. If writing B (lower level) first will help us to get A (higher level) right the first time, why not? One more advantage is that the calling routine must know the address of the sub, but if the sub is written later, we won't know its address when writing the caller. Having to go back and insert the addresses of all the subs into all the callers is an opportunity to miss something and build in a bug!

Before we get into a problem to design and code, let's think about a protocol for passing data. Using SCRT, we've seen that R6 can be used for passing in-line data <u>to</u> a subroutine. If the subroutine has data to return, R6 is still usable. But since the caller, upon return, has no readily available pointer to the data, other than a backwards displacement from R(P), this method is awful. Except for constants, inline data cannot be used in ROM-resident software, and passing data through a level (A to B to C) is a lot of work.

We can pass data by value or by address. Passing by value, we load D, one or more registers, or inline locations with the actual data. This data is quickly available, but is subject to limitations of the size and availability of registers and D. Numerical data is often 16 or 32 bits, and strings and arrays can be enormous.

Passing by address, we load D, one or more registers, or inline locations with the memory addresses of the values. We give the subroutine pointers to the data which is in memory. If you're wondering how we can pass a 16-bit address in an 8-bit accumulator (D), we'd have to ensure that the subroutine knows the high-order byte of the address (memory page) and D would contain the low-order byte.

Let's try a protocol which matches the complexity of passing data to the size of the data. 1) Values of 8 bits of less will be passed in D, which is copied into RF.1 by SCRT for a convenient second copy! 1a) We'll use RF.0 to report error codes. #00 means OK--no error; we'll determine the meanings of the other 255 values as we need them. 2) Data of 9-16 bits will be passed in registers if it's convenient and registers are available. 3) Data larger than 16 bits will be passed by address using R7. 4) At our option, any data may be passed by address using R7.

We can substitute another register for R7, use more than one register for addresses, or invent a variation or extension of this protocol if we need to.

The problem: terminal I/O. We can sit and think, and the machine can compute, but it's a big waste of time unless we can communicate between man and machine. Since there are quite a few things to communicate, we are going to proceed slowly, lest we confuse ourselves. We may want to enter letters, numbers, punctuation, non-printing control codes, and values (in decimal, hexadecimal, and maybe other number bases). The terminal works only in "characters" coded in ASCII (we won't worry about other codes yet). While the terminal will always see "1802" as #31 38 30 32, the contents of memory which result from this I/O will vary with the context: If this is your house number, it will be treated as characters: 31 38 30 32. If this is a decimal number, it will appear in memory as 07 0A. If it's hex, it will appear as 18 02. Obviously, we'll need some routines to do the conversion eventually. For now, we'll just get the ASCII into and out of memory, to and from the terminal, and let the user do the conversions.

Besides getting ASCII in and out, we should provide a means to correct mistakes, both by deleting and replacing one erroneous character at a time, and by cancelling an entire line of input (for those of us who spot boo-boos about 50 characters later). The idea of "lines" of input and output comes from the early days of 80-column cards, and the width of devices like TTYs, printers, and video screens. The input and output devices may have different widths (e.g., 80-col. card reader and 132-col. line-printer), or the idea of a "line" may be inappropriate to whatever the user is doing. So we'll let the user determine how long his line will be when he calls us. We'll also let the user provide his own buffer in memory. That way we don't have to provide each user with a buffer, worry about two users needing our only buffer at the same time, or do a lot of moving data between the user and us. Our "user" may be any routine within a program--we don't need two programs, two people, etc.. As I/O, we're part of the "system."

To be able to correct mistakes, the user must be able to see what he's typed in. This doesn't happen automatically, since a terminal is not a single, integrated piece of equipment! Even in the same enclosure, the input and output are separate devices to the software. To show the user what he's doing, we'll "echo" the data we just got from the input device back to the output device.

1) Specify the problem. Send and receive ASCII codes to/from a terminal, providing for a line of n characters in the user's buffer. Provide echo and backspace-erase editing as well as line-cancel. Line-cancel both clears the buffer and starts a new line on the echo device. A line is

terminated by a carriage-return entered no later than the n-th character. If no CR has been received by the n-th character, line termination will be forced by replacing the n-th character with a CR. (The number of characters is determined by the contents of the buffer, not keystrokes. Deleting characters (backspace-erase) and canceling lines reduces the character-count, so more than n keystrokes per line are possible.) A count of the characters received will be returned to the user (line length) with the input. For output, CR will signify the end of a line, or a CR will be forced as the n-th character. A line-feed and timing nulls will be generated after CR as necessary (device-dependent). The user will provide the buffer address and size (n); the address will be restored before returning, and n will be replaced by the character-count.

2) Data Structures. ASCII codes with and without parity; special codes for NUL, BS, LF, CR, CAN, DEL; line buffer, buffer address, buffer size.

3) Formats. ASCII w/parity: 8 bits; ASCII w/o parity: 8 bits, MSB=0; special codes (1-byte constants): NUL=00, BS=08, LF=0A, CR=0D, CAN=18, DEL=7F; line buffer is "size" bytes in memory beginning at "address," where size is 8 bits unsigned and address is 16 bits unsigned.

4) Algorithm. Until we start to refine some of the details, we can just recopy (1) here. Obviously, that doesn't fit our criterion of a one-sentence description, so we have some modularity to find.

5) Modularity. First, we'll split this into input and output halves. Output is easier--given an n-byte buffer at buffer-address, we send the characters to the device one at a time, "typewriter" style. If the buffer contains a CR, or we reach the n-th character without a CR, we generate CR, LF, and NULs as needed by a particular device. The characters are sent through a driver routine which is device-dependent and may be either serial or parallel. The top-down hierarchy is: PUTLINE to send up to n chars or until CR; which calls PUTCR to generate and send CR, LF, and NULs; both PUTLINE and PUTCR call DRVROUT which handles a specific device.

For input--given an n-byte buffer at buffer-address, we request characters from the device one at a time, "typewriter" style. Since the buffer-address must be restored for the caller, it must be saved before we do anything to it. If n chars or CR are received, the input line is ended and the buffer address is restored. The buffer size (n) is replaced by the character count and we return. Also, if CR, CAN, or DEL is received, special actions are taken. CR was discussed above. For CAN, the char count is reset to 0, the buffer pointer is reset to buffer-address, and CR is echoed via PUTCR. For DEL, the buffer pointer is backed up one location, the char count is decremented by one, and some indication of the deletion is echoed. For a video display, we probably need to send BS SP BS, or maybe DEL, depending on the device. For hardcopy, we may not be able to back up the mechanism, and will use backslash (#5C) to indicate the deletion. All other characters are both stored in the buffer and echoed, and the buffer pointer and char count are incremented. As with output, chars are received through a driver routine which is device-specific. The top-down hierarchy is: GETLINE to save and restore the buffer-address, return the char count, receive and store normal chars, and detect special chars. It will call DOCAN (DO CANcel), DODEL (DO DELete), and PUTCR as necessary to perform special character functions. GETLINE, DOCAN, and DODEL will also call DRVROUT and DRVRIN to handle the device(s).

Before we continue with steps 1-5 on all these modules, I have a question for you to think about. What should the program do if the input contains CAN followed by DEL, or more DELs than there are characters in the buffer? Why? If we have received n-1 chars without a CR and will force a CR for the next (n-th) char anyway, can we force the CR after n-1 non-CR chars without waiting for the next key? Why, or why not? (The answers and reasons will appear in the appropriate part of the discussion.)

Let's diagram the hierarchies to get an idea of how many routines and levels we'll have, and what goes where:

```
PUTLINE-┬-PUTCR   ---DRVROUT        GETLINE-┬-PUTCR   ---DRVROUT
        └-DRVROUT                           ├-DOCAN   ---DRVROUT
                                            ├-DODEL   ---DRVROUT
                                            └-DRVRIN
```

PUTLINE and GETLINE are on the first level (leftmost column of each side). PUTCR, DOCAN, and DODEL are on the second level; DRVROUT and DRVRIN are on the third level. Note that we assign the routines to the lowest level on which they appear, so we can preserve the idea that calling is "down." Otherwise, if DRVROUT were on the second level, PUTCR would call it on the same level. We also make an exception for DRVRIN, which does not appear lower than the second level, but since it's the converse of DRVROUT, we'll keep them together.

I'll run through PUTLINE all the way, then do GETLINE.

1) Specify the problem. Given a buffer address and buffer size (n), send up to n characters to the terminal using DRVROUT. Upon detection of CR before the nth char., or as the nth char. if no CR is found, end the line using PUTCR. Restore the user's buffer address.

2) Data Structures. ASCII codes without parity; special code CR; buffer address, buffer size.

3) Formats. ASCII w/o parity is 8 bits w/MSB=0, in buffer; CR=0D (constant); buffer is "size" bytes in memory beginning at "address." We'll also tie these down: size is 8 bits in D; address is 16 bits in R7. The constant, CR, is immediate data within an instruction.

4) Algorithm. Check buffer size, if 0, return. Else save R7 (buffer address). For size-1 chars., pick up a char. and advance the buffer pointer, check if it's CR then break (out of this loop). Else send char. via DRVROUT. The loop ends here, which implies that the char. count (size) is decremented and checked. Call PUTCR to end the line. Restore the buffer address and return.

5) Modularity. For PUTLINE, we're done!

PUTCR. 1) Problem. Send, via DRVROUT, CR to terminal to end the line; also send LF and NULs as needed for the particular device. We'll assume that the device is a TTY or similar, and requires the LF along with 5 NULs to allow for mechanical motion.

2) Data Structures. Constants: CR=0D, LF=0A, NUL=00.

3) Formats. All data are 1 byte, immediate within instructions.

4) Algorithm. Generate and send, via DRVROUT: CR LF (5) NULs.

5) Modularity. For PUTCR, we're done!

Admittedly, PUTCR is trivial, but it's handy. It allows many routines to end a line with a minimum of effort, and it allows memory savings by needing to store only CR to terminate a line.

DRVROUT. This is device-dependent since it babysits one particular device. All the characteristics of the device are handled here--parallel (including serial with a hardware UART), serial, baud rate, handshaking, as well as which EF, port, or Q line is involved. I'll go through this for both parallel and serial, but you are likely to have only one or the other.

DRVROUT-parallel. 1) Problem. Given a character in D (there's also a copy in RF.1), sent it through a port to the device.

2) Data structures. ASCII code.

3) Format. 1 byte in D (use RF.1 at your option).

4) Algorithm. Check handshake (EF usually; may also be a "status" port) until device is ready. Output ASCII through the port. Check handshake until acknowledged, if this signal is available.

(Whether the handshake after the data is sent is available, and if it's acknowledge, busy, not ready, is device-dependent.)

5) Modularity. None.

For a serial device with a hardware UART, we'd still use a parallel DRVROUT, only with two ports: status and data, instead of one EF and one port. Checking the handshake is then a process of reading the status port until the TBMT (transmit buffer empty) bit comes true.

DRVROUT-serial. 1) Problem. Given a char. in D (again, we have a copy in RF.1), send it bit-by-bit, LSB first, to the terminal. A start-bit and one or more stop-bits, as required by the device, must be generated. Parity will be ignored if the device allows, but a "parity-bit" must be part of the timing. All bits must be generated at the baud rate of the device.

2) Data Structures. Start-, data-, parity-, and stop-bits. ASCII w/o parity; byte-in-progress; bit-count.

3) Formats. Bits are 1 bit, generated on Q-line for start; data-bits and stops are shifted to DF enroute to Q; ASCII and byte-in-progress are 8 bits in D and RF.1, respectively; bit count is 8 bits in RF.0.

4) Algorithm. Check handshake (if available) until "ready." Set bit-count to 9 or 10 (1 or 2 stops; we'll see that the start-bit isn't part of the count). Generate and send start-bit. At intervals of 1 bit-time, get byte-in-progress, shift out new LSB to DF while shifting "1" into MSB, send bit, and decrement bit count, for 9 or 10 iterations.

I don't distinguish between data-, parity-, and stop-bits, for reasons which will become clear when we get to the code. The MSB is the parity position, and that bit comes to us as "0"; we're ignoring parity, so nothing has to be done with it. As a result of the timing, DF will be "1" when we go to shift out a new bit to send, and stop-bits are 1s, so we'll just fill in stops as the new MSB by using a ring shift.

5) Modularity. Timing of the bits will be done by a separate routine, BAUD, called by SEP Technique. BAUD will delay for the remaining machine cycles of a bit-time after DRVROUT has performed its function and housekeeping.

The "top-down" hierarchy for output: PUTLINE, PUTCR, DRVROUT, and BAUD (if we need it). We're ready to begin coding!

BAUD simply eats machine cycles to stretch the time of the bit-loop in DRVROUT, and will be used by DRVRIN as well, to a full bit-time. A bit-time is simply 1 sec./baud. Since we have no clock available to the program except the machine cycle, we have to write a combination of BAUD and DRVROUT which loops in the desired number of machine cycles to approximate a bit-time at the required baud rate and stay within tolerance. For a VIP, there are 220,080 machine cycles per second (CLK/8). The loop must take 1/baud of these cycles for a bit-time. A bit-time ranges from 45.85 (rounded to 46) cycles at 4800 baud to 2000.73 (2001) cycles at 110 baud.

It will be easiest to set up a loop which simply decrements D, but that means that we have a maximum of only 256 iterations. Can we eat enough time? How tightly can we loop, in terms of the minimum number of cycles per iteration? We need at least 2 instructions: SMI 1 and BNZ *-2 (back to SMI 1) which gives us a 4-cy. loop. We could toss in a NOP to add 3 more cy. if necessary. 256 iterations at 7.85 cy./loop will eat 2001 cy., and 6 or 7 iterations at 7.85 cy./loop will eat 46 cy.. It appears that we can do it!

Since BAUD is called by SEP Tech., and its caller (DRVROUT or DRVRIN) uses SCRT, the caller's PC is R3 and the return will be a D3 instruction in front of (lower memory address) the start of BAUD. The first thing we must do is load D with the number of loops we need, with an F8 xx instr.. We don't know how many loops yet--we have to account for every cycle in the DRVROUT loop, which includes all of BAUD, even the SEP call and return! Next comes the loop: FF 01 3A yy (we don't know the address yet, either). Finally, we exit back to the D3 instr., at BAUD-1: 30 zz. From 'way back, the next available location after our standard initialization is 0036, so here's the code:

```
0036  D3                    SEP 3               ..return to caller
0037             BAUD:      ORG *
0037  F8 __                 LDI #__             ..delay constant
0039             BAUD1:     ORG *
0039  FF 01                 SMI 1               ..time killing
003B  3A 39                 BNZ BAUD1           ..  loop
003D  30 36                 BR BAUD - 1         ..go back
```

To count the timing, we separate the instructions within the loop and those without. This will give us an execution time, in cycles, of the form: mx+b, where m is the number of cycles for 1 iteration of the loop, x is the constant (no. of loops), and b is the number of cycles for the remaining instructions. The loop (39-3B) has two 2-cy. instructions for a total of 4 cy./loop; the rest of BAUD is 3 2-cy. instrs. for another 6 cy.. BAUD runs in 4x+6 cycles.

The resolution of BAUD, i.e., the smallest change in total execution cycles for a change of ±1 in the constant, is ±4 cy. With a small resolution, we have the most precise control of the timing. For some baud rates, however, we may have a lot of time to kill, and we may have to sacrifice some precision by going to a larger resolution (longer loop).

I use three additional variations of BAUD to get resolutions of 6, 7, or 9 cy./loop in addition to the 4-cy. version above. At most, BAUD becomes 1 byte longer, and we'll reserve that byte in case we make future changes to our hardware.

```
6 CY:                                7 CY:
003B  32 36 30 39                    0039  C4 FF 01 3A 39

                                     9 CY:
                                     0039  C4 FF 01 32 36 30 39
```

The 6 CY version is the same length as the 4-cy., with a formula of 6x+4 cycles. The 7 CY ver. is 1 byte longer with a formula of 7x+6. The 9 CY ver. is 1 byte longer than the 4-cy., with a formula of 9x+4.

DRVROUT-serial. First, we need a register for the PC of BAUD, which we will save and restore. We also need to realize that since we're counting cycles, we can't allow cycle-stealing by interrupts and DMAs; if the 1861 is on we must shut it off. Saving the register will clobber D, so we'll rely on the copy of the data in RF.1, which becomes our byte-in-progress. We also need a bit count, in RF.0, to tell when we've sent all the bits. The initialization of DRVROUT will be completed by sending the start-bit.

Before we try to do it, let's see what serial transmission is supposed to look like. When we first turn on the device, and anytime no characters are being sent, the line is in "idle" condition. This is a fail-safe condition, as if the phone lines blew down. For a current-loop, no current flows; for RS232, no voltage (GND for us). We're also using negative logic, so this is a logical "1" and is called "mark" in communications. The opposite condition, current flowing or high voltage, is a logical "0" called "space."

The start-bit wakes up the device by changing the line from idle, so it must be a "0," "space," high voltage--i.e., Q is on. At the other end of the character, 1 or 2 stop-bits allow the terminal mechanism to come to rest. Additional stop-bits are interpreted as idle-time, so stop-bits are "marks," "1", GND, i.e., Q is off. Actually, it's not too important that stop-bits are correctly sent as marks, as long as the line is at mark by the time the terminal is ready for the next character if no more characters are sent immediately. The stops are mainly to delay the next start until the device is ready when characters are sent continuously.

Once we've sent the start-bit, we'll pick up the data from RF.1, and shift the next LSB into DF. At the same time, we'll shift the "1" in DF (from BAUD) into the MSB, then replace the byte-in-progress in RF.1. Then we will set/reset Q depending on the bit. By keeping the byte-in-progress as it's left by the previous shift, we can get each bit in the same amount of time, instead of shifting once for the first bit, twice for the second, etc., up to 8 shifts for the last bit! Now, once we've sent the bit to Q, we'll kill time by calling BAUD. When we're back, we'll decrement and check the bit count, looping until all bits have been sent. Then we'll restore the register used as PC for BAUD and return to the caller.

Note that when BAUD exits, the contents of D are 0 as a result of a subtraction, so DF is always 1 at this point. By using a ring shift to get the next data-bit, we automatically ensure that we have high-order 1's in the byte-in-progress to use as stop bits.

```
0040                  DRVROUT: ORG *
0040  22 61                    DEC 2;OUT 1            ..TV off
0042  87 73 97 73              GLO 7;STXD;GHI 7;STXD..save R7
0046  93 B7                    GHI 3;PHI 7            ..BAUD is on same page
0048  F8 37 A7                 LDI A.0(BAUD);PLO 7    ..R7 is BAUD PC
004B  F8 0A AF                 LDI 10;PLO F           ..RF.0=bit count
004E  7B                       SEQ                    ..jam start-bit
004F  D7                       SEP 7                  ..call BAUD for time
0050                  DRVRO:   ORG *                  ..bit loop
0050  9F 76 BF                 GHI F;RSHR;PHI F       ..get byte-in-prog.,
                        ..                              shift out LSB, 1 into
                        ..                              MSB, replace
0053  CF                       LSDF                   ..skip if DF=1
0054  7B 38                    SEQ;SKP                ..  else Q on, skip
0056  7A                       REQ                    ..  Q off if DF=1
0057  D7                       SEP 7                  ..call BAUD for time
0058  2F 8F                    DEC F;GLO F            ..decr. and check bit ct
005A  3A 50                    BNZ DRVRO              ..loop on count
005C  60                       IRX                    ..prime stack for pop
005D  72 B7 F0 A7              LDXA;PLI 7;LDX;PLO 7   ..restore R7
0061  69                       INP 1                  ..TV on
0062  D5                       SEP 5                  ..return via SCRT
```

The housekeeping is almost done when we get to the instruction at 004B: the 1861 is off, R7 has been saved and set up as BAUD's PC. Next, we set up the bit count in RF.0, as 1 less than the number of bits we're sending, since we're jamming the start-bit outside of the loop which counts bits. After the loop (at 005C) we clean up after ourselves. R7 is restored and the 1861 is turned back on. We might want to leave the 1861 to the user, since he may not have had it on in the first place; however, we do NOT trust the user to turn it off for us!! If you do not have an 1861, or you never use it, you can leave out the 22 61 instructions at the beginning.

The timing of the bits is governed by the number of machine cycles in the loop (50-5A), which includes BAUD. BAUD takes 4x+6 cycles (use the formula for the version you are using). The loop takes 19-21 cycles depending on the bit. For each iteration, we'll execute 9F 76 BF CF (9 cy.), either 7B 38 (4 cy.) or 7A (2 cy.), then D7 2F 8F 3A 50 (8 cy.). We'll use the average, 20 cycles; added to BAUD, we have a combined formula of 4x+26 cycles (4x+6 + 20).

The number os cycles we need is a bit-time, or (CLK / 8) / baud. For 110 baud on a VIP this is 1.76064 MHz / 8 / 110, or 2000.7272 (round to 2001). For 300 baud it comes to 733.6 (734) cycles. You can use this formula for other machines and/or other baud rates.

Now we solve for x in the 4x+26 formula so that 4x+26=cycles needed for 1 bit. At 110 baud, this is 4x+26=2001, or x=(2001-26)/4= 493.75 (494). We have a problem! 494 (#1EE) doesn't fit in a 1-byte constant. We'll try the 6 CY ver. of BAUD: 6x+4+20=2001, or x≑(2001-24)/6= 329.5 (330); it's still too big. For the 7 CY ver.: 7x+6+20=2001, x=(2001-26)/7= 282.14 (283). Finally, the 9 CY ver.: 9x+4+20=2001, x=(2001-24)/9= 219.67 (220), which fits nicely as #DC.

For 300 baud, 4x+26=734, x≑(734-26)/4= 177, which fits as #B1.

Use the version of BAUD with the smallest resolution and most precision which can handle the constant you need. The grosser the resolution, the more risk you run of being out of tolerance with your timing. The tolerance is rather liberal, though, so only a very "touchy" device will act up. Ideally, each bit is read at the center of its time-slice, so we're as far as possible from the transitions of the line as the bits change. As long as we don't drift into the transitions for any bit, we're OK! That gives us a tolerance of somewhat less than 50% of the bit-slice, for any individual bit, and cumulatively over 10 or 11 bits. In case the error accumulates in one direction, and that's more likely here than with an oscillator generating the timing since we're going to be a little long or a little short all the time, try to stay within 2-4% of the desired number of cycles.

For parallel DRVROUT, I'll list the code so you can use it, and we'll talk about it next time.

```
0036                DRVROUT: ORG *          ..BAUD won't be here!
0036   37 36                 B4  *          ..wait while "busy"
0038   22 52                 DEC 2;STR 2    ..data from D to stack
003A   63                    OUT 3          ..output to port 3
003B   3F 3B                 BN4 *          ..wait for "busy" as
                ..                            acknowledge (opt.)
003D   D5                    SEP 5          ..return via SCRT
```

The EF, port, and direction (hi/lo) of the signals are dependent on your system and device, so check what you have. The wait for acknowledge may be unnecessary, but depends on the way your device does a byte-cycle and handshakes.

LINES AND CIRCLES

BY NATHAN GOPEN

```
        LINE DRAWING PROGRAM--(Memory Locations 0C00-0C5B)
0C00 84 20 68 02 84 81 65 01            0C30 1C 36 F8 03 30 18 66 FF
0C08 66 01 68 00 82 85 3F 00            0C38 67 01 80 54 84 34 88 40
0C10 1C 26 0C 16 1C 24 F8 02            0C40 88 25 3F 01 1C 4C 81 64
0C18 AC F8 0E BC 0C FB FF FC            0C48 84 25 AC 3A D0 11 77 01
0C20 01 5C D4 00 65 FF 68 00            0C50 88 70 78 FF 88 25 3F 01
0C28 83 85 3F 00 1C 38 0C 32            0C58 1C 3A 00 EE
***************************************************************
        CIRCLE DRAWING PROGRAM--(Memory Locations 0D00-0D8C)
0D00 83 E0 AE FE 0D 0A 8B 00            0D48 1D 52 45 00 1D 52 75 FF
0D08 1D 26 F8 00 AC EA F8 F3            0D50 1D 3C 88 40 4A 00 1D 5C
0D10 AE F8 0E BE 0E AF 8F 32            0D58 62 00 1D 86 87 E0 87 D4
0D18 1F 8C F4 AC 2F 30 16 F8            0D60 86 80 86 C4 AD 8C D7 61
0D20 F0 AE 8C 5E E2 D4 6A 00            0D68 93 90 1D 7A 3A 01 1D 2E
0D28 69 00 89 E5 3A 00 7E FF            0D70 82 E0 72 FF 92 90 00 EE
0D30 85 B0 83 E0 AE FE 0D 0A            0D78 1D 2E 4A 01 00 EE 6A 01
0D38 85 05 64 FF 74 01 85 45            0D80 6E 00 8E 95 1D 2C 82 85
0D40 4F 00 1D 52 85 45 4F 00            0D88 88 20 1D 5C 80 00
```

To use these programs, you do as follows: To draw a circle on the screen, you must first save any variables that you are using in the program. (This step can be skipped if you don't care about what happens to the variables). This is easily accomplished with two instructions. First, set "I" to a convenient memory location, i.e. "AE00" to store variables starting at 0E00. Then execute an FX55 instruction to save the variables you want ("FF55" for all the variables). Now set VE to the radius of the circle begin drawn (00-0F), VD to the x-origin (00-3F), and VC to the y-origin. Now execute the instruction "2D00" and the desired circle will be drawn. Note that this routine only plots the points, so it is a dotted circle. Now the variables can be recovered.

For the line drawing program, save the variables in the same way. Next, decide what type of line you want. The lines normally slant down from left to right, so if the one you want slants the other way, you'll have to have it drawn from right to left. Basically, the x-origin is specified in V0 and the y-origin in V1, the length in V2, and the height in V3. Then "2C00" is executed to draw the specified vector.

Note: This unsolicited announcement was received by VIPER and is here F.Y.I. VIPER can't vouch for this outfit, but the announcement speaks for itself.

Date: March 12, 1982

The Practical Computer Contest

or

How I Can Pay For My New Home Computer

Enter Now and Win $100

If you are interested in home computers, then this contest is for you. The object is to find the most practical way an individual can justify the purchase of a home computer. We are interested in finding practical revenue producing ideas for typical home computers, such as the TRS-80, Apple, IBM Personal Computer, Pet, Heath, etc. On the other hand, we are not interested in the obvious (ie: knowing that a small business can use one to run accounts payable, payroll, etc.)

A possible example might be to use a computer to create some type of data base which could be maintained by one person. Customers could then pay a fee for access which could be by means of an auto-answer modem which replies via a voice terminal unit. Don't worry whether or not there is any available software. Just describe the application. It must, however, utilize current technology that is readily available.

Poor examples might be listing all telephone numbers in a data base which users could dial up to get the corresponding name and address. (Possible, but who would pay for it?) Or, putting all books in the library on a disk and letting the users key in their requests. (Great, but then it wouldn't be your computer.)

RULES: Entries must be typed (not handwritten) on one side of a single 8½ x 11 inch sheet of typing paper. No photo reductions. Use a standard typewriter (or equivalent printout). Use generic terms, such as computer, disk, auto-answer modem, tone-pad entry, etc. Don't use trade names unless, of course, the application won't work without that particular unit. Enter as many times as you wish, but each entry must be on a separate piece of paper and must be mailed in a separate envelope. Be sure your name, address and telephone number appear on the page. Only entries postmarked before June 30, 1982 will be considered. The single one hundred dollar ($100) award will be made before July 1, 1982. In the event of a duplicate entry, the earliest postmark will rule.

The judging will be based on the simplest revenue producing application which has possibilities of actually succeeding (based on our judgement), and how well the author explained the application. Confusing, ambigious, non-intelligent or extremely high technological discussions just won't make it. Entries not following the above rules (ie: handwritten or having "supplemental" or "amplifying" pages or "notes" on the back of the page) will not be considered. The decision of the judges is final.

The winner will be notified by telephone. Others will be notified only if their request includes a self-addressed stamped envelope. All entries become the property of OCEAN and none can be returned. Void where prohibited and/or regulated by law.

OCEAN, P.O. Box 2331, Springfield, Virginia 22152

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

ALWAYS USE YOUR ZIP CODE

Robert Millikan 37c USA

KILMER FACILITY, NJ
P M
JUN 1
1982

FIRST CLASS

7

Vol 1 & 2 being sent under separate cover,
and Welcome to VIPHCA!
Ray

A Final Word:

Here's news for those of you who have been thinking of adding a Floppy Disk to your VIP. Don King has a Floppy running with his VIP and wants to know if there is enough interest in our group to enable him to manufacture an interface board. Don's system is designed for an 8" drive and includes a "mini-DOS" which will FORMAT a disk, READ, and WRITE to disk in the manner of the cassette I/O. There are no file handling routines or fancy stuff like that of a full-featured DOS. The control program for the Floppy is in EPROM and will plug right into the VIP. Don estimates that the cost for PC board artwork will run around $1100, and he is willing to make the boards at COST, so the more people interested in his system, the cheaper it will be. The bare boards should cost around $60 or so, and the parts, including the NEC Floppy controller chip, will run around $75. You should be able to store around 400K on one 8" disk, and his system will let you run up to four disk drives! But Don needs to know how much interest there will be in his system before he puts out any money. The more people interested, the cheaper it will be. For more info or to indicate your interest, contact:

Donald A. King, 4269 Trenton, Detroit, MI 48210

Also,

There are still a few of you who have not renewed membership for 1982. Some of you have returned the questionaire, but not sent in dues. I guess I should have been more explicit about the fact that this is now a new year as far as membership is concerned! So, please check to see if you sent in your $12 ($18 outside North America). We love you all and want you as members! RS