VIPER

August-September 1981

Volume 3, number 3 Journal of the VIP Hobby Computer Assn.
The VIPER was founded by ARESCO, Inc. in June, 1978

****

Contents

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 which govern the operation of the Association. Mr. Sills is serving as Director of the Association, as well as editor and publisher of the VIPER.

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. Individual copies of the VIPER and past issues, where they are available, may be sent to interested people for $3 each. Annual dues to the Association, which includes six issues of the VIPER, are $12 per year.

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

Membership in the VIP Hobby Computer Association is open to all people who desire to promote and enjoy the VIP and other 1802 based computers. Send a check for $12 payable to "The VIP Hobby Computer Association" c/o Raymond C. Sills 32 Ainsworth Avenue, East Brunswick, NJ 08816 USA. People outside of U.S. Canada, and Mexico, include $6 extra for postage. All funds in U.S. dollars, please.

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. Articles, letters, programs, etc., in camera ready form on 8½ by 11 inch paper will be given preferential consideration. Many dot-matrix printers will not copy well, so this material has to be re-typed before it can be used. Please send enough information about any programs so that the readers can operate the program. Fully documented programs are best, but "memory dumps" are OK if you provide enough information to run the program. Please indicate in your material key memory locations and data sections.

Advertising Rates:

1. Non-commercial classified ads from members, 5¢ per word, minimum of 20 words. Your address or phone number is free.

2. Commercial ads and ads from non-members, 10¢ per word, minimum of 20 words. Your address or phone number is free.

3. Display ads from camera ready copy, $6/half page, $10/page.

Payment in full must accompany all ads. Rates are subject to change.

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

Simple Music Program Part 2

Udo Pernisz

The demonstration program for the RCA Simple Sound Board VP 595 that was listed in Part 1 of this series as "Simple Music 0.0" (VIPER vol.1, iss.10, p.20) suffers from a few inconveniences in regard to the flexibility of the tone/note code employed. When a tune is encoded the duration of the tone had to be given in units of 1/60 seconds and without recoding the whole list of notes there was no way of changing the tempo of a piece. Also, only one piece could be played by the program unless, by a program change, a different starting address was entered. And there were only two ways of playing a note: standard (non-legato) and legato.

To provide more experimental possibilities for the development of a musical piece the following features were added:

(1) A note can have one of four agogic accents: standard (non-legato), legato, staccato, and pizziccato.
(2) The duration of a tone is encoded in units of the duration of the (1/4)-note.
(3) The duration of the (1/4) note is given by the metronome setting M.M. in units of 1/60 sec and is stored separately from the musical piece in a Table of Contents.
(4) Prior to reproducing the piece, the metronome setting can be changed by a timed input. If no input is made the program defaults to the stored value of M.M.
(5) Any one out of sixteen pieces can be selected from the Table of Contents at the hex key pad.
(6) If a number is selected under which no piece was entered before, the program goes into "compose"-mode and allows to enter the code along with the metronome setting (and the starting address of the piece in RAM).

The coding of the (high order) tone byte is the same as before and is described in the manual for the VP 595. The coding of the note byte which provides features (1) and (2) becomes very straightforward and adds great transparency to the coding.

The format of the note byte is as follows:
The first six (high order) bits represent the duration values of the (1/2) note down to the (1/64) note. The values of those bits that are set to 1 are then summed up to the note duration. The range is thus from a (1/64) to a (63/64) note. A full note (exactly one bar) is rendered by e.g. two half notes played legato. Any length of a note can be played using legato. The table shows the bits with their associated note values:

bit	8	7	6	5	4	3
note value	1/2	1/4	1/8	1/16	1/32	1/64

Since most pieces use notes in the (1/16) range and longer, the coding is mainly done with the high-order nybble of the note byte, e.g. 4X is a (1/4) note, 8X a (1/2) note, 2X a (1/8) note and so forth, or 6X is lengthened (1/4) note, i.e. a (3/8) note etc. where X stands for the low-order nybble. The two high-order bits of this nybble, i.e. bits 4 and 3, are then mostly used for augmentation and diminution of a note, and for triple notes and the acciaccatura.

The two last (least significant) bits determine the agogic accent or mode of playing a note as set forth in the table below:

value	bit 2	bit 1	mode	duration
0	0	0	standard	L - s
1	0	1	legato	L
2	1	0	pizziccato	L/4
3	1	1	staccato	L/2

where L is the nominal duration as given by the sum of bits 8 through 3, and s is a small "non-legating" pause for the standard note that can be adjusted in the program (and is currently set at 03, i.e. 1/20 seconds). Of course, the rest of the nominal duration of pizziccato and staccato notes is a pause to keep the duration of the bars constant.

The numbers in the value column are added to the low-order nybble which was empty so far (if no (1/32)- of (1/64) note is being played) or which may be 4, 8, or C.

A pause is encoded by a tone byte of 00 with the note byte carrying the duration as usual. A cautionary remark: Do not use a legato note in front of a pause but only a standard one (or one of the others which make no difference, and neither sense). If you do you will hear the switching off of the VP 595.

Metronome settings that the program supports fully range approximately from 60 to 240 quarter notes played per minute which corresponds to the actual duration of a (1/4) note of between 3F-hex to 10-hex in units of 1/60 sec as explained in Part 1 or about 1 sec to .25 sec in real time. When playing a tune the program takes the metronome setting (either as given when entering the music bytes or optionally as entered at the key board) and calculates the note duration by multiplying and dividing it into the fractions indicated by the bits set to 1 in the note byte locations 8 to 3. These fractions are added up and supplied as the actual duration in units of 1/60 sec to the replay routine. This means that although any metronome setting can be used, only those that are divisible without remainder will give exact durations. Deviations from the exact value are for example equivalent to a (1/32) note if the M.M. setting is divisible by 4 only and if (1/32) note durations are actually part of the note to be played - if bits 4 and/or 3 are set.

Since the execution of the program takes up some time itself, part of the missing time is made up for anyway; this will be more important for very short notes. Moreover, in the coding of a tune it is often desirable to vary the agogic accent (i.e. the note duration) to put emphasis on a note or express a certain mood. The Russian folk song listed as tune #0 at the end of the program listing is an example for this technique.

Both when entering the code of a tune and while replaying it, a music (double) byte entry of 0000 terminates the music piece and the program returns to command mode expecting a hex-digit again for dialling another tune. If it finds a tune has been already entered under this number the metronome setting can be changed before replaying within one second after selecting the tune by pressing key C. Then you enter a byte, e.g 18, for the metronome and after another second (during which you can even change the metronome again with C) the piece will be reproduced.

If under the selected number no piece has yet been entered a long beep is issued to prompt you for entering a starting address as PQR where the 0 in OPQR is implied, then the metronome setting as MM, and then you can start entering the music bytes. After the last tone/note byte you enter 0000 which returns you to the command mode.

The information for the tunes is stored as part of the program in the four blocks starting at address 02A0 as explained below: The first two bytes are AP and QR where OPQR is the address of the first music byte of a tune. The next byte is the metronome setting for the (1/4) note in units of 1/60 seconds. The forth byte is the number of the tune under which it can be dialed from the keyboard. Since the information for tune number N is position-encoded starting at address 02A0 + 4*N the program is able to determine whether a tune has been already entered by comparing the keyboard entry with the last one of the four bytes, at location 02A0 + 4*N + 3. If the two bytes are equal it is assumed that a tune will be found at address APQR. If the two bytes are not equal the entry subroutine is selected.

Although it is highly unlikely that by pure chance the contents of memory will contain byte ON at location 02A0 + 4*N + 3 for N=0...F this might be checked initially to be sure.

A note of precaution is indicated here: If you are going to use the Hersh editor to enter and modify the code (setting the address to the last address of the Table of Contents) and then use the Pernisz renumbering routine, all the start address data APQR are being renumbered properly (which is what you want) but

all those metronome settings of the form XY with X = 0 , 1 , 2 A or B and with Y >= 2 are converted into X(Y-2) or X(Y+2) when numbering down or up, respectively. This, though, is mostly negligible. In addition, one would change the GOTO statements in 024E, 0272, 0288 and 0296 to 1200 when working with the editor and exit the program into the editor rather than to its begin. (And before renumbering down, change it to 1400 to end up with 1200 again.) In the present version since no editor or screen message prompt program is provided the statement in location 0288 is changed from 1200 to 11F8 and the following "key catch" inserted into location 01F8: F00A, 1200.

The whole program works like a manual record player with random access to the pieces recorded - and with the possibility of modifying and overwriting the "recordings". To demonstrate the facilities the listing of the program also contains three themes from Serge Prokofiev's "Peter and the Wolf" which can be selected for replay by numbers 1, 2, and 3. This is the present Listing of Contents:

```
#0:     Russian folk song
#1:     Peter (main theme)
#2:     The Cat
#3:     The Duck
```

Now tune numbers 4 through hex-F are available for entry. In order to make up a different Table of Contents one would have to go into the program (using the editor or VIP's own operating system) and replace the tune numbers with "impossible" entries first, e.g. FF_hex or in general, with ST where S > 0 and T is any hex-digit.

A suggestion for the use of tune #0 is to assign to it unused RAM and play through memory as powered up listening to the random data. One enters the four tune information bits "manually", e.g. for tune #0 one enters into the four memory locations starting at 02A0 the address (e.g. PX00), the metronome setting (e.g. 20-hex) and finally 00. The address can be changed to higher values if the memory space is needed for additional tunes. In this way, if 0X is the display page, one can "see" what one hears as the program converts the bit pattern into music.

One can also assign one's "Identi-Tune" to #0 and locate it right after the program before tunes #1...#F begin.

Simple Music 1.1

Machine:	COSMAC VIP with Simple Sound Board VP 595
Language:	CHIP-8I
Author:	Udo Pernisz, June 1979
Program location:	0200 through 0347
Main program:	0200 through 029F
Subroutines:	02E0 through 02F6 Initialization and Utilities
	02FF through 030F Write Duration List
	0310 through 032B Calculate Note Duration
	032C through 0347 Calculate Agogic Accent
Table of contents:	02A0 through 02DF
Duration list:	02F8 through 02FD
Begin of tunes:	0348 and higher

Features:

Up to sixteen tunes are selectable from the keyboard for both entry and replay.
Zero duration notes are ignored.
End-of-tune is by bytes 00 00.
Replay can be stopped at any time by holding down key 0 on the keyboard.
The metronome setting is stored initially but can be temporarily changed at replay.
Notes have two accents: agogic (duration) and tonic (pitch)

Functions:

0...F selects one out of 16 tunes
C within 1 second after a selection is made enters metronome setting mode (temporary)
Long sound after a selection is made indicates that no tune is available under the entry and enters input mode for tone-note byte list.
a) enter start address PQR for OPQR
b) enter metronome setting
c) enter coded tune
d) exit with 00 00.
0 stops replay of tune

Data structure:

- A musical note is encoded as a two-byte word
- The first byte is the tone byte and contains a number from 00 to hex-FF that is interpreted as a frequency according to the table in the VP 595 manual. 00 is played as pause.
- The second byte is the note byte and contains the duration of the tone as the sum of fractional durations given by bits set to 1. The MSB, bit 8, corresponds to a duration of a (1/2) note, bit 7 to a (1/4) note, etc. Reference to this byte is made by the metronome setting of the (1/4) note.

```
ADDRESS  CODE    COMMENT
                                 SELECT TUNE
0200     A2A0    I=02A0          :Point to begin of Table of Contents
0202     F00A    V0=KEY          :Input # of tune to be played (entered)
0204     8400    V4=V0           :Store entry in V4
0206     02E0    DO MLS @02E0    :Initialization subroutine; V0=4*V0
0208     F01E    I=I+V0          :Point to selected tune info in table
020A     F365    V0:V3=MI        :Read tune address, M.M. and tune #
020C     8345    V3=V3-V4        :Compare selected with stored tune# and
020E     4300    SKIP;V3.NE.00   :skip to Tune Entry Mode if not equal,
0210     1250    GO 0250         :else go to Replay Mode
                                 TUNE ENTRY MODE
0212     60A0    V0=A0           :Prepare CHIP-8 instruction APQR, also
0214     F018    TONE=V0         :sound the prompt for Tune Entry
                                       Generate APQR
0216     F10A    V1=KEY          :Input P: page where tune begins
0218     8014    V0=V0+V1        :V0 now is AP - first instruction byte
021A     F10A    V1=KEY          :Input Q: high nybble (h.n.) of address
021C     02F0    DO MLS @ 02F0   :Shifts entry into h.n. position in V1
021E     F20A    V2=KEY          :Input R: l.n. of address in page P
0220     8124    V1=V1+V2        :V1 now is QR - second instruction byte
                                       Get temporary M.M.
0222     F20A    V2=KEY          :Input metronome setting - h.n. first
0224     02E9    DO MLS @ 02E9   :Resets pointer in tune table, shifts
                                  entry into h.n. position in V2
0226     F30A    V3=KEY          :Input l.n. of metronome setting
0228     8234    V2=V2+V3        :Makes V2=M.M. (metronome setting)
                                       Store entries
022A     8340    V3=V4           :Put tune # (V4) into V3
022C     F355    MI=V0:V3        :Write address, M.M., tune # into table
022E     A232    I=0232          :Provide the program itself with this
0230     F155    MI=V0:V1        :address for entering the coded tune
                                       Get tone-note codes
0232     xxxx    I=0PQR          :Set pointer to start address, see 0230
0234     F10A    V1=KEY          :Enter h.n. of tone byte
0236     02F0    DO MLS @ 02F0   :Shifts it left into h.n. position
0238     F00A    V0=KEY          :Enter l.n. of tone byte
023A     8014    V0=V0+V1        :Add; now V0 = tone byte (frequency)
023C     F10A    V1=KEY          :Enter h.n. of note byte
023E     02F0    DO MLS @ 02F0   :Shift left
0240     F20A    V2=KEY          :Enter l.n. of note byte
0242     8124    V1=V1+V2        :Add, V1=note byte (duration, accent)
0244     F155    MI=V0:V1        :Write into memory, advance pointer
                                       Check for end of entry
0246     3100    SKIP;V1.EQ.00   :Both note and tone byte must be zero,
0248     1234    GO 0234         :Loop back to entry if first is not
024A     3000    SKIP;V0.EQ.00   :Checks second byte if first one was 0
024C     1234    GO 0234         :Loops back to entry if 2nd one is not
024E     1200    GO 0200         :Restart from begin. Can be used to
                                  jump to other program, e.g. an editor.
```

```
                                    REPLAY MODE
0250    A266    I=0266              :Supply start address (as read from
0252    F155    MI=V0:V1            :table of contents) to replay section
0254    1294    GO 0294             :Set up duration list from M.M. setting
0256    600C    V0=0C               :Defines key that allows change of M.M.
0258    6188    V1=88               :Set wait time for pressing key C in
025A    F115    V1=TIME             :Start timer          \_timed input
025C    E0A1    SKIP;KEY.NE.V0      :Check whether key (V0) is pressed
025E    128A    GO 028A             :Enter new M.M. if pressed, then set up
                                     the new duration list; returns to 0256
0260    F107    V1=TIME             :Else, check timer
0262    3100    SKIP;V1.EQ.00       :Skip to begin of replay if time up
0264    125C    GO 025C             :Else, loop back to check keyboard
                                           Start of replay
0266    xxxx    I=APGR              :Start address was supplied from 0250
0268    F165    V0:V1=MI            :Read in tone-note double byte
026A    3100    SKIP;V1.EQ.00       :Detect zero-duration note
026C    1274    GO 0274             :Go to play the tone-note if not zero
026E    3000    SKIP;V0.EQ.00       :Check tone if note was 0
0270    1268    GO 0268             :Read in next one if only note was 0
0272    1200    GO 0200             :Else restart from begin (or other)
                                           Play the byte
0274    B100    I/O=V0              :Send tone to I/O port
0276    0310    DO MLS @ 0310       :Calculate duration of note from V1:
                                     returns with note value in V1 and with
                                     note value modified by accent in V2
0278    F115    TIME=V1             :Set timer to duration of note
027A    3000    SKIP;V0.EQ.00       :Do not switch on the sound if pause
027C    F218    TONE=V2             :Play the note if no pause
027E    F107    V1=TIME             :Count down timer
0280    3100    SKIP;V1.EQ.00       :Go on if time up and check for key 0
0282    127E    GO 027E             :Else, loop on if time not up
0284    E19E    SKIP;KEY.EQ.V1      :Look for key 0 being pressed
0286    1268    GO 0268             :Play on if key 0 not pressed,
0288    11F8    GO 01F8             :Else exit replay and restart
                                    TEMPORARY M.M. AND DURATION LIST
028A    F00A    V0=KEY              :Dummy statement to catch pressed key C
                                     when exiting timed entry loop at 025E
028C    F10A    V1=KEY              :Input h.n. of M.M.
028E    02F0    DO MLS @ 02F0       :Shifts nybble left into h.n. position
0290    F20A    V2=KEY              :Input l.n. of M.M.
0292    8214    V2=V2+V1            :Add, makes V2=M.M.
0294    4200    SKIP;V2.NE.00       :Check for valid (non-zero) M.M. entry
0296    1200    GOTO 0200           :and restart if entry was 00
0298    822E    V2=(SHL)V2          :x2 to make V2 the (1/2) note duration
029A    A2F8    I=02F8              :Point to begin of duration table
029C    02FF    DO MLS @ 02FF       :Write duration table for this M.M.
029E    1256    GO 0256             :Back to Play Tune Routine (wait loop)
```

```
                                    TABLE OF CONTENTS
02A0    A348  1C 00                 :Tune #0 - Russian folk song
02A4    A3D0  16 01                 :Tune #1 - Peter,     themes from Peter
02A8    A434  20 02                 :Tune #2 - The Cat       and the Wolf
02AC    A49C  22 03                 :Tune #3 - The Duck      by Prokofiev
02B0                                :(empty for Tunes #4 ... #F-hex)

02DC    Axxx  mm nn                 :Tune #F-hex (user defined)

                                    MACHINE LANGUAGE SUBROUTINES
                                            Initialization and Utilities
02E0    F8 02    LDI 02             :Initialize RD.1 for page of Dura-
02E2    BD       PHI RD             :tion List which is from 02F8 to 02FD
02E3    F8 F0    LDI F0             :Load R6.0 with address of V0. R6.1 is
02E5    A6       PLO R6             :fixed at 0Y in a CHIP-8 program
02E6    06       LDN R6             :Load V0 via R6 and
02E7    30 F3    BR  F3             :branch to shift and store back @ F3
02E9    2A 2A    DEC RA             :Second entry point: decrement memory
02EB    2A 2A    DEC RA             :pointer back to begin of tune's data
02ED    F8 F2    LDI F2             :Load address of V2  \_ in Contents
02EF    A6       PLO R6             :into R6.0 and
02F0    06       LDN R6             :load V2 via R6
02F1    FE FE    SHL      2x        :Shift left to get entry into the
02F3    FE FE    SHL      2x        :high nybble position (total 4x)
02F5    56       STR R6             :Store shifted value back into V2
02F6    D4       SEP R4             :Set program counter to R4 = return
02F7    x        (not used)
                                            Duration List
02F8    x x x x x x                 :Six loacations will be filled in with
                                     actual durations of bits from (1/2)
                                     to (1/64) note in terms of M.M. setting
02FE    x        (not used)
                                            Write Duration List
02FF    F8 06    LDI 06             :Set RC.0 as counter for six note dura-
0301    AC       PLO RC             :tions, to write into the Duration List
0302    F8 F2    LDI F2             :Load address of V2 which is (0Y)F2
0304    A6       PLO R6             :into R6.0 - V2 is the 1/2 note
0305    06       LDN R6             :Load via R6          \_duration
0306    5A       STR RA             :Store value at top of Duration List
0307    2C       DEC RC             :Decrement counter RC.0
0308    8C       GLO RC             :Load counter value and
0309    3A 0C    BNZ 0C             :branch to continue routine if not done
030B    D4       SEP R4             :else, reset program counter = return
030C    4A       LDA RA             :Load value stored in Duration List and
030D    F6       SHR                :Divide by 2 - halve duration \_advance
030E    30 06    BR 06              :and loop back for next value.
```

```
                                        Calculate Note Duration
0310    F8 F1    LDI F1          :Load R6 with address of V1 which is
0312    A6       PLO R6          :the note byte (symbolic duration) code
0313    F8 06    LDI 06          :Set up RC.0 as counter for summing the
0315    AC       PLO RC          :duration according to note byte
0316    F8 F8    LDI F8          :Load RD.0 with start address of Dura-
0318    AD       PLO RD          :tion List
0319    06       LDN R6          :Load V1 via R6 and temporarily
031A    BC       PHI RC          :store in RC.1 - note byte
031B    F8 00    LDI 00          :Reset variable V1 in memory to 0 to
031D    56       STR R6          :initialize the summation of durations
031E    E6       SEX R6          :Set up R6 as the data pointer
031F    9C       GHI RC          :Load back the note byte from RC.1 and
0320    FE       SHL             :shift MSB into DF to check its value
0321    BC       PHI RC          :and store back into RC.1.
0322    3B 27    BNF 27          :Check DF - if 0 advance to next bit,
0324    0D       LDN RD          :else load its corresponding duration
0325    F4       ADD             :and add V1 in memory via RX to it.
0326    56       STR R6          :Then store sum back into V1 in memory
0327    1D       INC RD          :Advance pointer in Duration List
0328    2C       DEC RC          :Decrement counter and
0329    8C       GLO RC          :load RC.0 to check its value
032A    3A 1F    BNZ 1F          :Loop back if not yet done, else
                                        Calculate Agogic Accent
                                              Standard/Legato
032C    9C       GHI RC          :Load back last two bits of note byte,
032D    FE       SHL             :shift first one into DF \_i.e. accent
032E    BC       PHI RC          :Store rest back into RC.1
032F    33 3D    BDF 3D          :Check DF - if 1 jump to staccato part
0331    32 37    BZ  37          :Check last bit, if 0 then jump to 37
0333    F8 00    LDI 00          :If 1, accent is 01=legato - load 0 and
0335    30 39    BR  39          :go to subtract zero =leave alone at 39
0337    F8 03    LDI 03          :accent 00=standard - subtract 3/60 sec
0339    F5       SD              :from value in V1 for short interruot
033A    16       INC R6          :Increment R6 from V1 to V2 and
033B    56       STR R6          :store actual duration in variable V2
033C    D4       SEP R4          :Set program counter back = return
                                              Staccato/Pizziccato
033D    06       LDN R6          :Load V1 via R6 if accent = staccato or
033E    F6       SHR             :and divide by 2 and      \_pizziccato
033F    16       INC R6          :increment R6 from V1 to V2 and
0340    56       STR R6          :store back as staccato tone duration
0341    9C       GHI RC          :Load back rest (last bit) of note byte
0342    3A 47    BNZ 47          :and branch to return if 0 = staccato
0344    06       LDN R6          :else accent=pizzicato, load V2 via R6
0345    F6       SHL             :divide by 2 once more, then
0346    56       STR R6          :store back into V2 and
0347    D4       SEP R4          :Set R4 program counter = return
```

BEGIN OF TUNES: CODE LISTS
(each line is one bar long)

#0: Russian Folk Song

```
0348    7C70 6F28 6860 7C20
0350    6823 6823 6F23 7C23 6F4C A64F
035C    6F60 6824 5D60 6F20
0364    5D23 5D23 6823 6F23 7C6C 0024
0370    5340 3E40 4544 3E21 4520
037A    4E23 4E23 5323 5D23 5348 7C41
0386    7C20 5323 5323 6823 5360 6820
0392    5D23 5D23 6823 6F23 7C70 0010
039E    5340 3E40 4544 3E21 4520
03A8    4E20 4E20 5321 5D20 5349 7C41
03B4    7C20 5320 5320 6820 5360 6820
03C0    5D20 5D24 6824 6F2C 7CA0 000F
03CC    0000
03CE    xxxx  (not used)
```

#1: Peter
(Main theme from
Peter and the Wolf
by Serge Prokofiev)

```
03D0    8B80 6860 5320
03D6    4540 3E40 4560 5320
03DE    4540 3E40 3760 3420
03E6    4540 5340 6840 5D40
03EE    5880 5840 3740
03F4    5880 5840 3740
03FA    5880 7580
03FE    75C0 7C40
0402    7580 5860 4520
0408    3A40 3440 3A60 4520
0410    3A40 3440 2E60 2B20
0418    3A40 4540 5840 4E40
0420    4980 4940 2E40
0426    4980 4940 2E40
042C    4580 4580
0430    45F0
0432    000C
```

#2: The Cat

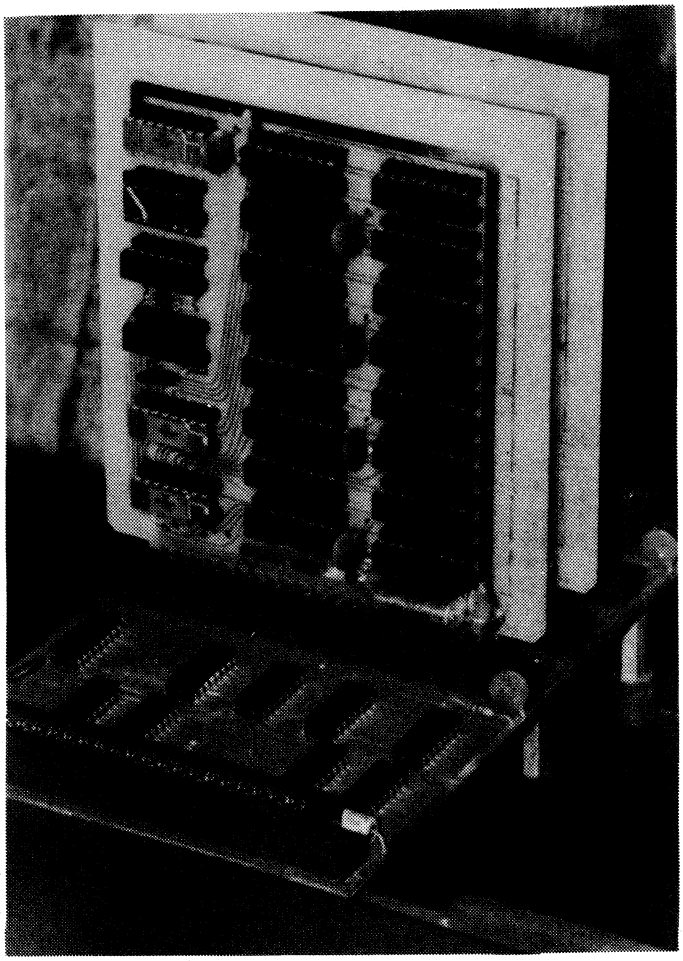
```
0434    8B23 6823
0438    5343 6823 8B23 9443 8B23 6823
0444    5323 4523 4E41 4E20 5320 6820 5D20
0452    5321 5D20 6F23 6823 5D21 6820 7C23 6F23
0462    6880 5343 8B23 6823
046A    5343 6823 8B23 9443 8B23 6823
0476    5323 4523 4E41 4E21 5321 4521 4E21
0484    5320 5D21 4E20 5321 5D20 6821 5320 5D20
0494    6F81 6820 00A0
049A    0000
```

#3: The Duck

```
049C    5309 58C0
04A0    5805 5D21 6221 6821 6221 5D05 4521 4E20
04B0    5305 58C0
04B4    5805 5D21 6221 6821 6221 5D05 2E21 3720
04C4    3441 2B61 3421
04CA    3A41 3140 4E21 4521
04D2    4141 3441 5840 58C0
04DA    1405 1520 1520 1510 1410 1510 1610 2B73
04EA    5305 58C0
04EE    5809 5D21 6221 6821 6221 5D05 4521 4E20
04FE    5305 58C0
0502    5809 5D21 6221 5D21 4521 2E21 2620
0510    2641 2B40 3421 2621
0518    2B41 3140 4E21 4521
0520    4141 3441 5820 5820
0528    5305 58C0
052C    1405 1520 1520 1510 1410 1510 1610 2B5C
053C    0040 2BB0
0540    0000
```

ADVERTISEMENT

8K RAM BOARD FOR YOUR RCA VIP



*USES 2114 RAMS

*EACH 4K BLOCK SEPARATELY ADDRESSABLE

*G10 GLASS BOARD

*DOUBLE SIDED 2 OZ COPPER

*SOLDER PLATED

*GOLD PLATED CONTACTS

*4-1/2" W x 5-3/8" H

*PLUGS INTO VIP EXPANSION CONNECTOR OR SYSTEM EXPANSION BOARD

*DRAWS APPROXIMATELY 600 mA FROM +5 V LINE

*BARE BOARD WITH DATA $ 49.00

*ASSEMBLED AND TESTED $ 149.00

CALIFORNIA RESIDENTS ADD STATE SALES TAX

WITH THE RCA VP-575 SYSTEM EXPANSION BOARD YOU CAN PLUG IN TWO OF THESE 8K RAM BOARDS, ONE 12K ROM BOARD* PROGRAMMED WITH YOUR VERSION OF RCA VP-701 FLOATING POINT BASIC, AND STILL HAVE ROOM FOR A VP-590 COLOR BOARD AND A SPARE.

*FUTURE PRODUCT

SEND CHECK OR MONEY ORDER TO:

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

3.03.13

NOTICE: Cuddly Software is no longer in business. All projections indicate that heavy losses can be expected for even the most popular programs. Thinking, "Hey, I took a shot," I would like to thank all my customers for their patronage.

P. V. Piescik, Prop.

MACHINE CODE

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

Well, the feedback from the first column "complained" that what we did was not immediately obvious, but it seems that no one was hopelessly lost. I'm inclined to believe that this is OK, since computing always involves a lot of brain-burn, and if you expected to read the column and the example programs like a novel, you were surprised to find that it doesn't work that way! Unfortunately, it won't, either, especially since I have only 6-8 pages per issue and 6 issues a year, which amounts to about 1 night's reading for a single college course. But this isn't college, so hopefully with your feedback I'll be able to strike an exact balance between what I say and what I leave out, to stimulate your thought process in the right direction.

You should expect to work with a pencil in one hand and a manual in the other, at least for a while; things will eventually stick. It's also helpful to have a sheet of paper on which to keep a running list of the contents of X, P, D, DF, whatever registers are in use, and probably a few important memory locations, especially the stack.

STACKS seem to be a major source of confusion on the 1802, judging by the misinformation and bad advice I've seen in several articles. A stack is a "data structure," which is a set of data items and a set of axioms (rules) for manipulating those items. The stack is also called a LIFO (last-in, first-out), since that's how the data moves to and from the stack. Remember LIFO, and that stacks are for temporary storage, and you've got it!

You've seen how a stack works if you've ever filled your driveway with your guests' cars. You've also encountered the restrictions of the stack if you've had to run out for more of those cheese snacks and your car is still in the garage, or an early guest has to run because the kids locked the babysitter in a closet (with a telephone). Obviously, if you have to get at a car (data item) without following LIFO order, the stack is an inappropriate choice.

The stack is very handy, however, for getting data out of the way when we won't need it for a while, but we do need the register(s) or accumulator that it's tying up. The stack provides a way to free the registers by storing their contents in memory with a minimum of fussing. We'll replace "fussing" with the axioms for handling the stack. (Apparently, this is the hairy part of using stacks--we tend to be more confident with fussing than with rules.)

As a data structure, the stack involves a set of data items and a set of axioms. The set of data items simply consists of single bytes, and we don't care what they represent; ASCII codes, instructions, binary values, addresses, etc. are all alike to us! The axioms are simply allowable operations, and there are only 3 of them: Create a stack defined as empty; PUSH an item onto the stack except if it's full; POP an item off the stack except if it's empty.

I mentioned that the stacked data would be memory-resident, and from last time, we recall that we'll need a register to point to it, called the "stack pointer" (what else?). We could select any register for this, but we'll use R2. This is almost an arbitrary choice, but we'll find that all of the instructions we'll be using refer to RX; an Interrupt sets X=2, and MARK (79) uses R2 to "Push X, P to Stack." If we don't use R2, we'll have to reserve it for Interrupt and MARK anyway. There is also no reason why we cannot share one stack among several users, so let's not complicate matters by having more than one stack. We'll also find that the stack grows toward lower memory addresses as items are added to it, so we'll want to start it at a high address, like 00FF (since we assumed only 1 page of RAM). So let's CREATE an empty stack by initializing R2 to contain 00FF and X to contain 2:

```
0000  F8 00          LDI 0     ..THIS PAGE
0002  B2             PHI 2
0003  F8 FF          LDI #FF   ..LAST BYTE
0005  A2             PLO 2     ..(R2)=00FF
0006  E2             SEX 2     ..X=2
```

We now have an empty stack! It's empty because we haven't PUSH-ed anything onto it, and since we didn't store anything at 00FF previously, we must consider that location to contain garbage. With that established, the location to which R2 (which is also RX, and sometimes called "SP" for stack pointer) points is called TOS (top of stack) and is ALWAYS EMPTY, so we're free to store an item there.

The stack pointer, R2, always points to TOS, the next free location (which is not a fixed address), and is our only window to the stack. We do all our PUSH and POP operations there, not caring what the actual memory location is at any time. That's scary--not knowing the location; but if we don't change R2 except during PUSH and POP, and then only in accordance with the axioms, it works just fine! We avoid all the fuss of figuring out where TOS happens to be in memory, or searching for it, etc.

PUSH is easy. Since TOS is free all we do is store data there. Since TOS is also supposed to be free after we add to the stack, R2 must be changed to complete a PUSH. It turns out that STXD (73) does both operations with a single instruction. Since STXD decrements R2, TOS will be one location lower in memory following a PUSH, which is why I mentioned that we'll see the stack grow downward, and started it at a high address.

Let's see PUSH in action. If we're writing a subroutine, it will probably need to use one or more registers, and we can expect that the routine which called it is also using registers for something. As a matter of programming practise, we write all routines to be "dumb" and "polite." They are dumb because they know as little as necessary about the routines which call them, or they, in turn, call, except to communicate data back and forth. If routine "A" calls routine "B," neither knows which registers the other is using, or for what purpose, except that A will leave RD, for example, pointing to data in memory on which B will perform some function, and B will leave RD pointing to the result. If B changes any other registers, A may not be able to continue afterward. So, B will be polite, and save all of the registers it's going to use, on the stack, and restore them before it returns. OK, you're saying, "but I'm writing all of these routines, and I know which registers to avoid," right? Well, just consider that you may have to modify one of the routines to correct a bug, or you may want to use a subroutine in another program--how many routines do you want to re-write, anyway?

Taking the last example from the last MACHINE CODE, let's make it into a subroutine at 0080. It uses only RE, so we have only 1 register to save:

```
     0080  8E              GLO E     ..(D)=(RE.0)
 (1) 0081  73              STXD      ..(M(R2))=(D); R2=00FE
     0082  9E              GHI E     ..(D)=(RE.1)
 (2) 0083  73              STXD      ..(M(R2))=(D); R2=00FD
```

Now let's look at the stack as we started (fig. 0), with X=2 and R2=00FF from our CREATE operation, and after each PUSH (figs. 1 and 2, corresponding to the numbered lines above):

```
(0) R2 = 00FF  M(00FF) = xx

(1) R2 = 00FE  M(00FE) = xx
               M(00FF) = (RE.0)

(2) R2 = 00FD  M(00FD) = xx
               M(00FE) = (RE.1)
               M(00FF) = (RE.0)
```

No surprises! R2 changed by 2, which is the number of bytes we pushed. By definition, TOS moved with it. At each step the contents at TOS are unknown, as we'd expect for a free location. Since we're looking at the addresses, it's obvious that each time, the contents of the accumulator are stored at the old TOS location. The previously stacked items are not affected, but in looking through our R2 "window," everything appears to have been pushed down relative to the window; actually, the window moved, but you may see this called a "push-down stack" anyway.

If you consider that lower addresses are "down," and therefore that TOS is at the bottom and everything was pushed "up," you're getting confused, right? Well, up and down are relative in any case. If a guy in California and I (in Connecticut) point "up" at the same time, we're pointing about 44 degrees different! As long as PUSH moves TOS one way and POP moves it the other way, we're OK. Let's ignore the addresses as we look back over figs. 0-2, and we'll see that this works just as well if we don't know the direction or the starting address. If we're writing a subroutine which may be invoked by callers at several nesting levels, or by different callers who may themselves stack various amounts of data, we obviously didn't check R2 to see where we started. But it worked, and we always expect it will.

I won't recopy the example routine, but let's say that we're now 11 (decimal) bytes from where we finished saving the register and our addition is done. Before we leave, we must restore RE to its original contents.

Now we want to POP data off the stack, which is EXACTLY the reverse, step-by-step, of a PUSH. Since R2 points to TOS (a free location) we must increment R2 first, knowing that the last thing we did to PUSH was decrement. Then we'll load the data from memory into the accumulator, and finally put it back where we got it. Aw, nuts! We don't have an instruction "ALDX" which would advance-and-load via RX, so this is going to take more than one instruction. (If you're wondering, the 1802 simply cannot increment or decrement a register first, in time to get the high-order address on the bus by TPA in the same cycle.) So, we've got IRX (60) which increments RX; LDX (F0) which loads via RX without

changing it; and LDXA (72) which loads via RX and then increments. We need to advance-and-load, but it'll work nicely if we have to advance (IRX), then load (LDX). We can also get sneaky by using LDXA to POP all but the last byte. We'll prime the pump with IRX, end with LDX (remember: our last operation must be load, not advance), and use LDXA in between (to load and "re-prime"):

```
008F  60          IRX      ..R2=00FE
0090  72          LDXA     ..(D)=(M(R2)); R2=00FF
0091  BE          PHI E    ..RE.1 RESTORED
0092  F0          LDX      ..(D)=(M(R2)); R2 UNCHANGED
0093  AE          PLO E    ..RE.0 RESTORED
```

This time we started with fig. 2 and worked back to fig. 0; however, the figures cannot be exactly matched with the state of the machine after an instruction is completed. After the IRX, we're halfway between fig. 2 and fig.1--R2 has been changed, but we have not yet read (RE.1) from that location, so we can't consider it free. The LDXA reads and frees the location to complete the transition to fig.1, but simultaneously (at least as far as we can tell) has advanced R2 again to leave us straddling figs. 1 and 0. Finally, the LDX reads and frees the last location, and once again we match fig. 0 exactly. Note that we have been true to LIFO, POP-ing the halves of RE in the opposite order from the way they were PUSH-ed.

That's important, because unlike the cars in the driveway, we can't tell one byte from another when we POP them. We didn't check the values when we PUSH-ed them, and if we did, so what? We'd have to run another stack to keep track of this one, and a third to keep track of the second... ...and we could still put the data in the wrong place! You simply have to keep track of where you got the data to PUSH so you can put it back later. When you run up and yell, "POP," you take what's there.

It's very tempting to consider that when we've POP-ed the last of a series of items, leaving the stack pointer at that location, that the data is still there, and is available. Well, if nothing has clobbered that location, the data IS there. However, the POP axiom says that the first step is to increment the stack pointer, so we cannot read that location again without violating the rules. The penalty is blowing the stack and your program.

Now, I have to admit to a lie. I said that TOS is ALWAYS FREE, but there's an exception. When we are straddling any two of our status figures after the "advance" operation of one instruction but before the "load" of the next, TOS contains live data which we still need and must not be clobbered. It's very unlikely that we'd clobber it ourselves on purpose (we might, accidentally, by using LDXA instead of the final LDX), but an impolite interrupt service routine could get us easily!

"Interrupt service routine" is a mouthful, so I call it "interrupt," meaning the software which sets up the R0 DMA pointer for the 1861 or whatever is required for some other interruptor device. The hardware signal is an "interrupt request," and since I need that term less frequently, I don't mind the extra typing and will call it by it's full name. The operative word is request, since if IE=0, no interrupt (routine) will be invoked; the request signal may or may not remain pending, but that's up to the requestor device.

The definition of an interrupt request is "an unexpected subroutine call." Since it IS unexpected, it is also unpredictable! We may know that the 1861 will hit us 60 times a second, but we virtually never know at which instruction, or even in which routine, we'll be when it happens. That means we cannot possibly prepare for it. Our concern is that, while POP and PUSH ideally should be indivisible operations, on the 1802, POP takes two instructions. POP is therefore divisible and interruptible. The problem is that in between those two instructions, we have a shared resource in a vulnerable condition, i.e., another process (the interrupt) may want to share that resource and could change it on us! If the interrupt comes in between the two instructions of our POP, we have the stack in an inconsistent state--TOS has been moved to a location which still contains live data, but TOS points only to dead data by definition. The interrupt, believing that TOS is always free, will clobber our data.

The interrupt will have to use the stack, since it must at least get our values of X and P from the T register to memory to effect a return to us. And it will undoubtediy have to save D as well, since not much can be done on the 1802 without going through the accumulator. Since it is unexpected, the interrupt MUST BE a polite subroutine, regardless of how badly we abuse good practice elsewhere! The interrupt also has no way of knowing that we are in a critical region of our program (a critical region is the code which deals with a shared resource, during which no other process is allowed to share it), the interrupt must allow for the possibility that it could mess us up. So, the interrupt follows rules which are the opposite of ours: TOS is NEVER FREE. This implies that PUSH means decrement-and-store, and POP means load-and-advance. If you examine a video routine you'll see it done this way; if you also follow through what it would do for the possible states in which we might be caught with the stack, you'll see that everything works just fine.

I mentioned that we neither PUSH onto a full stack, nor POP from an empty one. But how do we tell? Well, we don't! We always pair a PUSH with a POP, although we do all our PUSH-ing, do something else, then do all the POP-ing. By always PUSH-ing first, we will never have an empty stack when we POP. Unless, of course, we do an unpaired POP, but if we consider that the POP completes a PUSH/POP pair, and completed pairs leave the stack as though they'd never happened (think about it), then an unpaired POP is the same as POP-first. That's a no-no.

Not PUSH-ing onto a full stack is avoided differently. Unlike completing a pair of operations, we are beginning a pair, so that's no way to tell. And we never check for the actual location of TOS, so how do we tell? Again, we don't. We recognize that stack room has priority when we're allocating memory space for it, program code and other data. If we don't allow enough room, the stack may grow into another area and clobber something we need for our program to run. There's also the possibility that the data on the stack may be clobbered if the program stores data in the area we've invaded. Fortunately, the amount of room the stack will need is predictable. In our example, we stacked 2 bytes. While we haven't covered it yet, let's assume that we are using Standard Call and Return Technique, which stacked 2 bytes in calling us; we're up to 4 bytes. A video interrupt would need another 2 for a total of 6. Assuming that's all we have, we simply reserve 00F9-00FF for the stack.

Allocating room for the stack is simply a matter of determining the worst case (most data on the stack) by adding up how many bytes we can expect from each routine which will actually have data on the stack. It's a matter of

determining the hierarchy of routines (who calls whom), how many bytes each will stack, how many bytes will be stacked during a call, etc. It's been my experience that #10 bytes are usually enough, and #30 bytes will do for a reasonably monstrous program. That's if you'd prefer to use a rule of thumb instead of counting!

Now let's finish off a couple of details to complete the conversion of last issue's example from a stand-alone program to a subroutine. Right away, I see a problem! I started out by PUSH-ing RE onto the stack using R2 with X=2 and instructions which refer to the index register, RX. Later, I POP-ed RE using more RX-instructions, so what's wrong? That routine changed X (at what was 0006 then, and has been moved to 008A, but not shown, this time), so I'm not POP-ing from the same place in memory as I PUSH-ed. This is a common error, so keep an eye out. To fix it, we'll insert an SEX 2 (E2) instruction at 008F, and move the rest of the instructions one location higher.

The POP code starts where the BR * was, and extends through that to where the data (ONE, TWO, SUM) would be. We'll have to replace the BR * with an instruction to return from this subroutine after we've POP-ed the stack. The specific instruction used will depend on which subroutine technique we use, so let's just note that there will be a "return" and find out what it looks like when we discuss the various subroutine techniques. At most, we'll need 2 bytes, and if you're lost, they'll go at 0095-0096. We can put the data at 0097-009B, then change A.0(ONE) where it appears at 0088. I realize I'm getting a little obscure with all these changes in mid-air, but I'm also getting off the topic--stacks.

There's a trick you may see used, and use yourself, which appears very much like a stack operation, but it isn't! Many times you'll have an arithmetic or logic operation to perform which requires that one operand be at M(RX). So you use an STR 2 (52) to put it there, get the other operand into the accumulator, and perform the operation. You're taking advantage of the fact that TOS is free, and that means that you have a byte of memory available for temporary storage without having to set up a register to point to it. So you borrow the stack pointer and the TOS location for another purpose. We will even say that we've put the data on the stack, but we haven't! To have used the stack means that we must have performed some operation according to the stack axioms. Have we CREAT-ed a stack? No, we didn't set up a register. Have we PUSH-ed? No, we didn't decrement. Have we POP-ed? No, we didn't increment, and with the result in the accumulator, we probably didn't read the location, either. So, it was NOT a stack operation.

OK, this trick works with one provision, that we do not use the stack while we still need that byte. An interrupt won't touch it because we gave interrupts different axioms to keep them out of our hair. But if we PUSH anything, our first step will be to store at TOS--bye-bye, data. If we POP, we'll not only change RX so it points to a different location, but we'll also be allowing an interrupt to clobber the data, now at TOS-1 (or lower). If you'll need that constant after calling a subroutine, be sure to decrement RX first so the data becomes part of the stack (we're completing a PUSH, actually). And, be sure to increment RX afterward.

There are several instructions which increment RX along with whatever else they do: RET (70), DIS (71), and OUT (61-67). Be aware of this, since you may have to offset that advance (we'll see this in the MARK technique) just because it happened, or to get back to our data. Oh-oh. Getting back to our data is a

lost cause! This is a common complaint following output, since you won't always find your data there. You shouldn't expect to, since that advance left your data at a lower address than TOS, so it's no longer on the stack. If you want the data later, you have to stack it twice, so one copy is left when the other has been output.

OK, catch ya in the next issue! Meanwhile, this column is intended for you--the beginner--so if you're getting lost, tell me, and be specific! You'll be amazed to see how fast I can turn around to get you caught up!

EDITORIAL

I was very sorry to hear from Paul that he will not be able to keep his Cuddly Software business going. Paul had perhaps the only software house devoted to the 1802 processor exclusively. But you cannot continue to operate a business that does not show a profit, and Paul had to make his decision as a businessman. However, at least for us, Paul has indicated that he will continue with his new column, Machine Code, which has been very informative and useful to many people, if the sampling of mail that I have received is any indication. If you have a specific question relating to anything Paul has written in his column, drop him a line, so that he can have a chance to cover it in the next column. Or send it along to VIPER directly, if you prefer.

I, for one, am very glad to see Paul's column. The mere mention of machine language has caused some computer hobbyists to break into a cold sweat, cause palpitations of the heart, and a glassy-eyed look to the face. Anyone who can help those who suffer from the aforementioned symptoms is doing a great service indeed!

ANNOUNCEMENTS

VIPHCA headquarters has received a large number of VIPER volume 2 issues. Infact we have dozens of complete sets. For those of you who would like a set of VIPER Volume 2, VIPHCA will send them to you for $8.00 postage paid in the U.S. and $10 for other countries. Send payment to VIP Hobby Computer Assn. at the usual address. U.S. Funds.

Tom Swan has told me that there may be a PIPS IV. It depends on whether we can get enough people interested in it to make publication worth while. No hints yet on the contents, but there are supposed to be <u>many</u> pages of material. No firm price either, but I suspect that it will probably be around $20. I'll keep you posted on developments.

READER I/O

Gary C. Grinnell of 9 Laurel Park, Northampton, MA. 01060 could use some suggestions on a software technique to generate a random number for use in a program. He thinks that a shift register in software might be a useful approach to the problem. Also, the program will be used <u>without</u> a TV monitor. If anyone can suggest a machine code routine and perhaps provide a listing, he would appreciate it very much.

Yours,

Ray

Raymond C. Sills
Director, VIPHCA

3.03.20

An Overlay Editor/Assembler

by

William Lindley

As an owner of the PIPS for VIPS I and II books, I have been quite pleased with the programs in both books, particularly the editor and the CHIP-8 assembler. I purchased a video terminal and made it work with the editor, acquired an additional 8K RAM, and modified the editor so that I could use the full 4K capability of text editing. The use of only 4K stemmed from the fact that the highest page number for the page addressing command is Hex #F.

I was quite contented with my mods and the assembler until a friend showed me how fast the overlay assembler ran on his Data General machine. "If he can do it," said I to myself,"so can I." With this in mind, I set about modifying the assembler to use the same data space as the editor. Now I can write my program with the editor, load 4 blocks of the assembler, type in "0200 E, 1, 1," and presto, the assembler has fully assembled the program in about five seconds! The only tape operation involved is writing the final program to tape, and naturally, writing the source to tape also. Below is a detailed history of the modifications.

EDITOR MODIFICATIONS

I first modified the editor to be only six blocks long, not 12. This cuts the loading time for the editor about in half. This was done by relocating the character set data. Why should the text be in the middle of the program inbetween the main program and the character set? By simply changing the byte byte at 0364 to point to the start of the character set data at 0400, and then writing the character set (originally 0A00 to 0BFF) into locations 0400 to 05FF, the size of the editor program was cut in half.

Then I set about moving the text buffer so that it did not overwrite the character set. This was a more tedious task, as the address of the first and last pages appear several times in the program. Since the modifications for this are more lengthly, I will present them in the form of a table. "Ending page" refers to the last page that contains text. The values for text starting at 1000H and a 4K buffer are given.

Location	Data		Location	Data
001A	10	first text pg.	0170	1F ending pg.
0056	10	" " "	01D0	10 first pg.
005C	0F	#pgs for tape	01DD	20 end pg. +1
008B	10	first text pg.	0213	1F end pg.
00AA	20	ending pg. +1	024D	0F first pg. -1
012A	1F	ending pg.	02B7	1F end pg.
0139	10	first text pg.	0364	04 character set pg.
014C	1F	ending pg.	03D8	10 first pg.

With these modifications, the editor will work with text at location 1000H and a 4K or 16 page buffer. The page select function, (Escape 3 or C/3), will work up to "F" pages. The relocation of text allows changes to be made quickly to CHIP-8 programs with the modification to the assembler, which follows.

ASSEMBLER MODIFICATIONS

After changing the editor to edit the text at 1000H, an area which is not used by most of my utility programs, I changed the Tom Swan PIPS for VIPS assembler to use this space for its source, rather than the tape. This speeds up assembly so much that it is hard to notice whether it is assembling or waiting for a response. After selecting first or second pass modes and running the pass, the time between the start and end of the past is about two seconds. The only large changes were the addition of two short branches to circumvent the tape read instructions. In addition, some data for checking for the beginning or end of source text had to be changed. These were the only changes necessary for the assembler to use the overlay technique of having the text in memory, rather than on a mass storage system (cassette tape) at all times. Here are the changes:

Location	Data	
0038	30	branch (to bypass tape I/O)
0039	3E	to 3E
003F	10	starting page of text
006C	20	ending page of text +1 (check for text end)
008F	30	branch (to bypass tape I/O)
0090	95	to 95
0096	10	starting page of text
00F0	20	ending page of text +1

These changes are all that is necessary to have the assembler work on the data at 1000H, rather than the tape as source. The assembler works exactly as before, but does not use tape input. The taping of the output still works exactly as before. The only new limitation introduced by this modification is that the text file can only be "F" blocks long. This is still quite a long source file, considering that the output file can only be 3 blocks long. Also, you must have more than 4K of RAM, preferably 8K, for editing long files.

There are several advantages to using a RAM buffer: new changes can be made in a flash without having to reload the edited file, and running CHIP-8 and loading the object file will not kill the text in memory. And assembly is speeded up quite considerably. And, by the way, this modified assembler runs faster than the one on my friend's NOVA!

"SCROLL UP" MODIFICATIONS

Tom Swan's "Scroll Up" program (VIPER 3.01.06) uses only four lines of display. After looking at the scroll routine, I changed the byte at 0336, the byte determining the amount of scroll, to 06; this allows five lines of text each with six pixel rows. However, two unwanted pixel lines from the previous data appear at the top of the screen. If you change the byte at 0200 (initialization of VD) from 1A to 19 (hex), this will set the VY variable for display of the data after a scroll has been performed, to a location two lines higher, eliminating the unwanted lines. This change also allows five lines of data to be displayed, increasing the amount of text that can be displayed on the screen.

VIP Hobby Computer Association
32 Ainsworth Avenue
East Brunswick, NJ 08816

A Final word:

There were a few more things that would have been nice to include in this issue, but there just wasn't any room for them. We can send out 24 pages of material for 35¢ worth of postage. If four more pages are added (you'll notice that they are printed in fours) the postage will go up to 53¢--almost 50% more. For that amount of postage we can send out 36 pages of material. And I didn't have 36 pages of stuff ready to send right now. However, who knows what might happen in the future? So, Leo Hood's VIP Operating system was not printed in this issue, even though it is ready. Next issue, for sure. Promise. And we can still use material from you guys in the field. (Yes, all of your fellow VIPHCA members are males: not one single female member on the roster.)

ARTICLES PLANNED FOR COMING ISSUES:

1. VIP Operating System for ELF by Leo F. Hood
2. CHIP-8 for ELF by Leo F. Hood
3. CHIP-8 Editor by William Lindley
4. Dive Bomber, Mastermind and Battle of Numbers by Gaylord M. Ellerman
5. Step by Step: a single stepper/tracer for CHIP-8 by Tom Swan