VIPER

The only comprehensive monthly newsletter devoted exclusively to

THE RCA COSMAC VIP

PRESENTS

TIPS FOR VIPS

VOLUME II

by TOM SWAN

ARESCO
P.O. Box 1142
Columbia, MD 21044
THE PAPER · VIPER · RAINBOW · SOURCE

TO ANNE

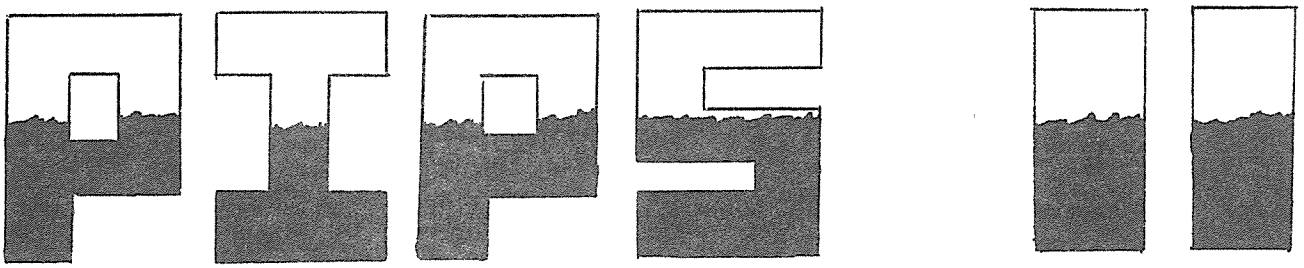


TABLE OF CONTENTS

** The material and information provided in this manual is believed to be accurate. No guarantees are made, however, and ARESCO is not responsible or liable for any consequential or inconsequential damages which may arise from the use of these programs.

Please address all inquiries to ARESCO, P.O. Box 1142, Columbia, MD 21044; or telephone (301) 730-5186.

C H I P - 8 A S S E M B L E R - 3

C H I P - 8 A S S E M B L E R - 3

INTRODUCTION

Entering CHIP-8 programs using the RCA COSMAC VIP's system monitor has obvious disadvantages. No doubt you have used one of the editors available through VIPER*, or have written an editor of your own to display addresses and whole instructions on the screen. While this is an improvement, you are still programming in hex. If you're at all like me, you have by now acquired a ream of notes - all of which are more or less (more than likely "less") understandable since the time you wrote them.

CHIP-8 Assembler-3 allows programming in CHIP-8 by using the Text Editor-21* (or any other editor) to prepare a source listing for your program. Instead of calculating GOTO target addresses, CHIP-8 Assembler-3 allows the use of English (or other) language labels to direct the program flow. Comment lines may be written into the program, sections may be joined together <u>regardless</u> of their position in memory, and commonly used subroutines may be stored on tape in source form to be later assembled <u>anywhere</u> you wish in RAM. Debugging is facilitated by the comments which, instead of being buried in that ream of dog-eared, yellowing paper, are stored <u>along with</u> your program. Complete instructions, hints for modifications, a ne game "Get It Together", and program listings are provided. Error messages during assembly detect common mistakes and direct you to them before you test your program. A Subroutine Library starter of five subroutines is also included. This is a Total Programming System!

---

* VIPER is a newsletter dedicated to the VIP. A subscription blank and more information is at the end of this book. Text Editor-21 is part of Tom's first book, PIPS FOR VIPS, which is available for $19.95 (with cassette) from ARESCO, Box 1142, Columbia MD 21044.

CHIP-8 Assembler-3 provides a professional approach to programming in CHIP-8, and its features, plus the terminology used to describe it, demand full treatment. I will therefore proceed very carefully into its operation in order to give you every opportunity to appreciate its full value.

My hope is that you will gain not only a valuable new programming aid, but that you'll learn more about the workings of your microprocessor and the operation of utility programs such as these. (Machine language assemblers use the same terminology and format as CHIP-8 Assembler-3, and if you have not had experience with such a program, the following will introduce the concepts involved. You need no special knowledge to get full use out of CHIP-8 Assembler-3.)

Before reading further, however, you should be familiar with the operation of Text Editor-21, which will be used to prepare CHIP-8 programs. If you already own a text editor for your VIP (so long as it can prepare 16-character-a-line ASCII text), you will be able to use your own editor to prepare the source listing. In fact, by using only the first 16 characters of a line, and by following the format for the assembler described here, <u>any other computer</u> could be used to prepare VIP source listings!

In order to do so, you would need detailed knowledge of both systems. For example, tape I/O compatibility might be a problem. But ASCII <u>itself</u> can be considered to be a machine-independent language. Any computer that <u>speaks</u> ASCII could conconceivably talk to any other computer, no matter how different, that <u>understands</u> ASCII. The concept is valid, and the implementation is a challenge.

Experienced programmers will want to use CHIP-8 Assembler-3 to assemble programs that use an entirely different interpreter as well. The only requirement is that your interpreter's instructions have the same basic two-byte CHIP-8 format, with addressing from 0000 to 0FFF, taking the last three nybbles (a nybble is 4 bits) of the hex code instruction. Other

formats are possible with minor modifications. Full 16-bit addressing could easily be accomplished for more comples interpreters (see the program description for details).

If you have modified your CHIP-8 interpreter, CHIP-8 Assembler-3 needs no modifications to handle the changes. The system ROM must be intact, however, for the Assembler to operate. Such versatility will be appreciated by those who have developed their own interpreters or have added new CHIP-8 instructions to their old one.

VIPpers suffer from a lack of s ftware, particularly utility programs such as CHIP-8 Assembler-3, Text Editor-21, and Disassembler-7*. The availability of such tools -- and that's exactly what they are: tools -- will precede the eventual acceptance of the 1802 microprocessor into the arena now shared only by the 8080, the Z-80, the 6502, etc. I have a firm belief that this will happen, and am committed to its progress.

You are the owner of a well built, well designed, fairly well (and improving!) documented microprocessor system. I hope the following programs will add to your knowledge and enjoyment of your RCA COSMAC VIP.

---

* The Disassembler-7 is another part of Tom's PIPS FOR VIPS book.

GLOSSARY

It will be important to your understanding of Chip-8 Assembler-3 for you to understand the terms used in describing its functions. The following section is a glossary presenting the terminology used in the operating instructions of the Assembler. I have tried to stay as close as possible to the general meaning of each word as many are used to describe the operation of other assembler programs too. This then will also serve as a general introduction into utility programs of this nature. Words in the glossary used to describe others are underlined. This glossary is intended as a general reference. In order to operate Chip-8 Assembler-3, you will need to read the Instruction section thoroughly.

ARGUMENT -- In Chip-8 Assembler-3, an argument refers to the use of a label following a Chip-8 instruction. In that case, the assembler knows that the instruction is an addressing type (i.e. AMMM, 1MMM, etc.) and it will look for an address to be inserted in that

instruction (see label).

ASCII -- A one byte (8-bit) code from 00 to 7F hex that represents one of 128 standard codes for each letter of the alphabet, number, punctuation mark or control character normally used in text applications. Ex. 41-A; 42=B; etc.

ASSEMBLER -- A program that allows programming usually one level higher than absolute hex -- the method used to enter programs with the Cosmac VIP system monitor. Instead of addresses, labels are used to specify positions in the program, and in the case of machine language assemblers, mnemonics are used instead of the hex codes for instructions. Assemblers also usually include pseudo instructions as aids to programming.

COMMENTS -- Text in the program. Clarifies instructions, allows titles to be written before subroutines, etc. Format for Chip-8 Assembler-3 must be

strictly followed and comments may take two forms. 1) On any single line, if the first character of that line is a semicolon ";" (ASCII 3B) the rest of that line will be ignored by the Assembler. 2) If the character immediately following an instruction is a semicolon, the rest of that line will be ignored (i.e. 6102;V1=02).

CONTROL CHARACTER -- Usually one or more of the first 32 ASCII codes 00-1F. Specifies a function such as line feed, or carriage return. Not used, recognized or printed (unless as user-defined graphics codes -- see A Character Designer for the VIP by the author) by Text Editor-21 or Chip-8 Assembler-3.

LABEL -- Used to specify positions in a source listing instead of actual addresses. In Chip-8 Assembler-3, labels must begin on the first character position of the line and may be any length up to 5 characters or design.

In order for labels to be output to the symbol table, however, the first character of any label must be a capital letter. (ASCII 41-5A) This may be easily changed, however, (see program description). Except for this restriction, any combination of the 128 ASCII codes may be used for labels -- including punctuation, lower case, spaces and numbers. But even if the capital letter restriction is changed, one firm rule must be observed -- the first character may not be an ASCII space (i.e. 20 hex) or an ASCII semicolon (3B) (see comments). Labels specify position in the program, and come before the instruction. Arguments specify instructions that need to know those addresses, and follow the instruction. Labels and arguments have the identical form. (Note--the label "LINK" is reserved.)

LINK ADDRESS -- The word link has many meanings in computer terminology. (For instance the link bit for DEC's PDP computers is analogous to the single bit DF register

of the 1802.) Here we use the standard assembly definition of link -- the process of joining sections of source listings together. Each new section's start address is the link address.

LINK TABLE -- Automatically output at the end of the second pass, the link table in Chip-8 Assembler-3 is a table of the starting addresses for each source listing section joined together. (Fourteen sections may be linked before overflow occurs.) The last address in the link table specifies the first available byte following the last instruction in the program. Linking may also be achieved without the use of the link table although its computation is automatic.

MNEMONICS -- Alphabetic representations of machine language binary, or hex, codes. (i.e. FE SHL; the "SHL" is the mnemonic while the FE is the hex code.) Mnemonics are easier to read and remember and for that reason most assemblers are based on

their use. (Excluding Chip-8 Assembler-3 -- it is not a machine language assembler.) Disassembler-7 for instance, translates hex codes into mnemonics for ease in reading otherwise indecipherable machine language programs.

OBJECT PROGRAM or OBJECT CODE -- The actual program in a form that will run (assembled). Following the automatic output to tape of the symbol table and the link table, Chip-8 Assembler-3 waits about 4 seconds, then records the Chip-8 Object Program ready to reload at the start address and run.

PASS -- The process of loading the source listing into Chip-8 Assembler-3 so it may be assembled. As do most assemblers, Chip-8 Assembler-3 requires two passes -- or two "looks" -- at the same program in the same order to do its work. During the first pass, the symbol table is created. During the second pass, the symbol table is searched

and addresses inserted in their proper locations. It is during the second pass that the object program is created.

PSEUDO INSTRUCTION -- An instruction whose only purpose is to instruct the assembler to perform a special operation. Chip-8 Assembler-3 contains one pseudo instruction which is usually preceded by a label, though this is not essential. It takes the form RMKK where KK = a number from 00-FF hex. Using this instruction will reserve KK bytes at that particular position in the program. (KK + 1 bytes if KK is an odd number, thus conforming with the Chip-8 format of even-numbered addressing.)

SOURCE LISTING -- The prepared program usually written using a text editor such as Text Editor-21. The source listing exists in ASCII (or other encoded form for other assemblers) and is not capable of being run in that form on a computer. Chip-8 Assembler-3 reads the source listing in

two passes creating (assembling) it into the object program which will then run on the computer. Source means just that -- the original form or "source" of the program.

START ADDRESS -- Chip-8 Assembler-3 requires you to specify from where you want assembly to originate. For a normal Chip-8 program, ML 0200 would be the start address. Any address may be specified -- even in the middle of a page. The assembler will begin to create object code using the start address for the first instruction of the source listing it encounters.

SYMBOL TABLE -- A collection of all labels and their associated addresses which after assembly may be used to debug the object program without having to reassemble. Chip-8 Assembler-3 alphabetizes (by first letter only) all the labels and automatically outputs the symbol table to tape along

with the link table. Both tables may then be examined using Text Editor-21. Up to 73 different labels may be used before overflow occurs.

I - FORMAT FOR SOURCE LISTINGS

I A. FORMAT EXAMPLE

LINE#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	;	S	A	M	P	L	E		F	O	R	M	A	T		
3	B	E	G	I	N	;	A	X	X	X		V	A	R	S	
4							F	6	6	5	;	G	E	T		

Above, you see a blocked-out sample describing the format for creating Chip-8 source listings. Each block horizontally represents one of sixteen character positions numbered from left to right, corresponding with the capabilities of Text Editor-21. We will refer to the position of each element in a line of the source listing by these numbers. THIS FORMAT MUST BE FOLLOWED PRECISELY FOR CHIP-8 ASSEMBLER-3 TO WORK.

If you are using another text editor, please be sure that all spaces are ASCII 20's (hex) codes and that control characters are not stored in memory.

Lines 2, 3 and 4 show an example of Chip-8 source code. At the end of this section, the full

source listing for a new Chip-8 game "Get it Together" is also shown, following the above format. The Sample Program will be referred to in these sections along with the above Format Example, though the game itself will remain a mystery until later.

I B. THE SOURCE

I've always said I wish I had written "The Source" by James Michener, and I could not resist using the title. (At no risk, by the way, titles may not be copyrighted. Just think of the lawyer fees BYTE would be paying if they could!)

COMMENTS

The ability to enter and store comments on tape along with your programs greatly increases their legibility. Chip-8 Assembler-3 allows comments to be entered in two different ways.

1)If a semicolon (;) is entered at position #1 on any line, the next 15 characters will be ignored by the assembler. Each line must begin with a semicolon in position #1 if it is intended as a comment line. Line #2 in the Format Example is a comment line.

2) Following any Chip-8 instruction or two bytes of data, if a semicolon is entered in position #11 (the character immediately following the last character of the instruction), the remaining five characters are ignored by the assembler. This space may be used to describe the preceding instruction. An example of this type of comment is given in the Format Example, line #4. The Chip-8 instruction, F665, is followed by a semicolon in position #11, which is in turn followed by the comment, GET.

Only the above two methods may be used to enter comments. In no case may comment lines start in the middle of lines. In the first instance, the semicolon must be the first character of the line, and in the second, it must immediately follow a Chip-8 instruction, or a pseudo instruction, or a piece of data. The sample source listing at the end of this section includes examples of both types of comments.

SPACE LINES

Lines may be left blank in order to break up the program making it easier to read, and marking sections of the program to aid in debugging. Blank lines are

defined as a string of 16 ASCII spaces (20 hex). As Text Editor-21 automatically inserts spaces in all unused character positions, this function is automatic. Skipping down a couple of lines in other words will have no effect on the assembly of your program.

CHIP-8 INSTRUCTIONS

Chip-8 instructions and blocks of data such as the bit patterns for rocket ships and race cars must begin at position #7. (Pseudo instructions too, but this is described later. See section I,C.) All Chip-8 instructions are written exactly as you have in the past, four characters wide, except for those instructions requiring an address such as the 1MMM or as in line #3 of the Format Example, the AMMM. The resolution of the Cosmac VIP created a problem with certain characters -- M, N, W, B in particular -- and I have chosen to write the AMMM as AXXX because I think it looks better on the monitor (see line #3).

However, the choice is entirely up to you, and you may write addressing type instructions in any way you want as only the first character distinguishes these from other instruction types. (A--- would be all right for instance, or A..., etc.) The reason for

doing so is that we use labels instead of addresses now to specify where we want the "I" pointer to point, or the jump to occur. Chip-8 Assembler-3 will insert the proper addresses -- in place of the XXX or whatever -- for you. In the next section, labels will be discussed in depth.

If you happen to know the address you want for a particular instruction, simply write the instruction as you normally would. Only if an instruction is followed by an argument (see next section) will that instruction be affected by the assembler. This gives you the option of keeping data blocks or subroutines in memory where you know the addresses rather than having to include the data along with the source listing.

The above rules apply to all the addressing type instructions -- 1XXX; BXXX; 2XXX; AXXX; and 0XXX. If you have modified your interpreter, or if you are using a different one, the rules apply automatically to your new instructions too, even if you use a different letter for the first character. This is because the instruction will be modified and an address inserted if and only if it is followed by an argument.

Remember, begin all instructions on position #7.

LABELS AND ARGUMENTS

Please read the glossary definitions for labels and arguments if you have not done so. This will give you a general picture of what they are.

Labels always start on position #1, and may extend up to and including position #5 of the line with the first character being a capital letter. (This restriction may be easily modified, though you may not of course use an ASCII space (20 hex) or a semicolon (3B hex) for the first character of any label.)

The remaining four of the maximum five characters may be composed of any of the 128 ASCII codes 00-7F. This means you may use any combination of numbers, spaces, punctuation -- anything -- as long as the first character is a capital letter. (See the sample source listing (section I,D.) for more examples of labels.)

In the Format Example, line #3 begins with the label, BEGIN. It meets the requirements above, being five characters in length, with the first character a capital letter, positioned on the line from position #1-5.

Labels may have less than five characters. Even single letter labels may be used. It is a good idea, however, to think up labels that approximate the

function of the section of program they identify. In the Format Example, for instance, BEGIN is easily remembered as the first line of the program. Later, when a program restart is desired, it becomes a simple matter to perform a Go-To BEGIN rather than have to remember at which address this occurs.

You will notice in the Format Example that the label "BEGIN" is followed by a colon (:) in position #6. This colon is purely cosmetic to the assembler, and by that I mean position #6 of a normal instruction (vs. a comment) line is inactive and transparent to the assembler. You may leave the colon out or replace it with another character -- a dash or period for instance if that looks better to you. The sole purpose of the colon in position #6 is to make that line clearer to read. The label BEGIN obviously goes with the instruction AXXX in the example, line #3. If you want, all instructions may be preceded with a colon or other punctuation (or character). Position #6 is totally inactive -- only if the assembler sees a capital letter in position #1 will it treat the next five characters as labels.

(This is not completely true as I have hinted, however. The assembler actually recognizes everything but spaces and semicolons in position #1 as the first characters of labels. Unless labels start with a capital

letter, though, they will not be output to the symbol table during taping (section III). But see the program description to modify.)

ARGUMENTS

The rules for constructing arguments are identical to those for constructing labels. Only the purpose and line position differ.

All arguments must follow an addressing instruction of the type discussed before (i.e. AXXX). Arguments always begin on position #12 of the line and may (or may not) continue to the end of the line at position #16 (see the Format Example line #3). In every case, the end of the instruction and the first character of the argument are separated by a space. No other character but a space may go in position #11 preceding any argument.

The purpose of the argument can best be demonstrated by a new example.

FORMAT EXAMPLE #2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
						1	X	X	X		B	E	G	I	N

Here we have the instruction -- an addressing type which is the Chip-8 Go-To instruction -- beginning at position #7. Position #11 is a space and at position #12 is the first character of the argument "BEGIN" which then fills out the line. Compare the argument "BEGIN" with the label in the original Format Example line #3. You will see they are the same, and in this lies one of the primary reasons for using an assembler. After assembly, when the Go-To instruction in Format Example #2 is encountered in the program, a jump to the instruction with the same label will be performed. Instead of an address, we have specified the flow of the program using an English word, BEGIN. In a long program, the use of labels and arguments greatly simplifies programming. And even BASIC does not give you this capability.

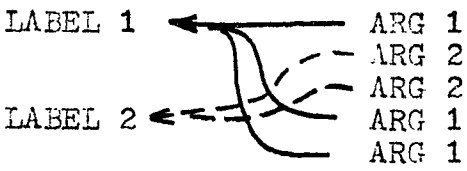
In line #3 of the Format Example, not only is there a label "BEGIN" on the left, but there is a new argument, "VARS" on the right. The rules have all been followed. The instruction is an addressing type (AXXX), and there is a space in position #11 between the instruction and the argument. Look at the sample program at the end of this section. You should be able to find the label VARS in the third column of the source listing. At this position are the seven bytes of data which will be the

initial values for V0-V6 when the program is run. The first line of the program (at the label BEGIN) when assembled will cause the "I" pointer to point to these bytes of data. The F665;GET instruction will load these values into the variables V0-V6. All this is performed without having to know the location in memory of the data. The assembler will match the label and argument together and insert the proper address into the AXXX instruction.

An important rule: For any and all arguments there must be one and only one identical label. Failure to adhere to this will cause an error message to be output and the assembly will halt (see section III,D). The converse of this rule is not true. In other words, there may be labels without accompanying arguments. However such labels would be without purpose unless used to simply identify a section of the program. (Use of more than 73 labels will cause overflow.)

The same argument may be used as many times as needed. For example, in the case of a much-used subroutine call, the same argument will occur many times in the source listing. However, any label must be used only once as it is responsible for identifying the position in memory for that subroutine, or block

of data, or Fire Phasers routine. A chart will help clear this:



If ARG 1 = LABEL 1 and ARG 2 = LABEL 2 then we see an example where two sections of the program are called several times by the corresponding arguments. In the case of LABEL 1, three calls are performed where in the case of LABEL 2, only two calls have been programmed.

The whole discussion on labels and arguments will boil down once you understand the simple principal of using words or symbols to direct the program flow instead of addresses. Take a moment to examine the sample program at the end of this section. (Don't worry about how it operates.) You will see that each label to the left of the instructions occurs once and only once, and that in most cases several occurrences of identical arguments will be found to the right of the instructions. Confirm for yourself that the labels and arguments of the sample program conform to the above rules.

One final restriction, which I purposely waited until now to bring up in order not to confuse you, must also be remembered. The label LINK -- (L...I... N...K...(space)) is reserved for use by the assembler. Section I C will explore the function of the LINK. DO NOT USE THIS AS A LABEL IN ANY OF YOUR PROGRAMS.

This ends the section on creating the source listing for Chip-8 programs. For your reference, below is a generalized table condensing the above rules.

SUMMARY

1) COMMENTS
   a) First character in position #1 = semicolon denotes the rest of the line to be a comment line.
   b) First character following an instruction (in position #11) == semicolon denotes the rest of that line to be a comment.
   c) Blank lines are ignored by the assembler.

2) INSTRUCTIONS
   a) Must begin in position #7 and follow the normal Chip-8 format.

b) Addressing types may take the form NXXX where XXX is intended to be an address to be calculated by the assembler, and where N = the identifying character (A, 2, etc.) for that instruction.

c) Data, bit patterns for display, etc., are treated exactly as are instructions.

3) LABELS

a) Begin with capital letters on position #1, and may extend up to five characters in length total using any combination of the 128 ASCII codes for the remaining four characters.

b) May be followed by a colon, other punctuation or space in position #6 just before an instruction or data for clarity.

c) A maximum of 73 labels may be used.

d) The label LINK is illegal and is reserved for use by the assembler.

4) ARGUMENTS

a) Identical construction to labels.

b) Begin in position #12, preceded by a space in position #11, preceded by a

Chip-8 instruction of the addressing type.

c) For any and all arguments, there must be one and only one identical label.

d) Arguments may be repeated as many times as needed to allow multiple calls to subroutines, etc.

I C. RMKK, LINKING

RMKK

Most assemblers contain pseudo instructions whose only purpose in life is to instruct the assembler to perform a particular operation.

Chip-8 Assembler-3 has one pseudo instruction, the RMKK. It is usually preceded by a label and is never followed by an argument as it is not of the addressing type. (In fact it is not really an instruction at all, and never becomes a part of the final Chip-8 program in assembled form.)

Typing RMKK -- beginning in position #7 of the line as for normal data and instructions -- will cause the assembler to reserve KK bytes of memory at that position in the program. (Hence the name RM = Reserve Memory) The KK is replaced by the number of bytes needed (in

hex), and you may reserve up to 256 bytes (KK=FF) at any place in the source listing. On assembly these bytes will be equal to 00's before running.

The main purpose of the RMKK pseudo instruction is to allow the use of instructions such as the FX33 which requires three bytes of memory for converting the value of VX to a three digit decimal number. "I" must first be set to the address of these "work" bytes before using the FX33. As we are no longer concerned with addresses, the following sample source listing demonstrates how this may be accomplished using labels and arguments instead.

```
BEGIN:60FF;V0=FF
      AXXX DECIM
      F033;=3DD
END  :1XXX END
DECIM:RM03
```

In the above example three bytes will be reserved by the RM03 instruction located at DECIM for the use of the F033 instruction. The label DECIM has its counterpart in the argument DECIM. The assembler will insert the proper address of the three bytes at DECIM:RM03 into the AXXX instruction. The RM03 does not become a part of the program -- only the number of bytes specified are assured to be empty at that position.

In the above short example, BEGIN is a label that in this case has no corresponding argument. I have included it simply to show that this marks the first line of the program, though if I wanted to expand the program and include a Go-To BEGIN, I could accomplish this without having to search my notes to see just where the beginning is. (BEGIN does not have to be the first line of the program.) Notice also the next to last line of the example. The 1XXX Go-To has the same argument and label -- END. When assembled, this instruction's own address will be inserted into it causing a Go-To itself or halt at that location when the program is run. (The example is a complete program by the way, and may be assembled and run after you learn how to operate the assembler.)

Now I have two confessions to make about the RMKK and the above example. First you do not need to type RM to reserve memory -- actually only the first R is needed followed by any character followed by KK. You could very well type RBKK (Reserve Bytes or RQKK (Reserve ...?). Because of the way the assembler is constructed, only FF bytes may be reserved by the RMKK, and the use of RM reminds you not to overstep this limit (as opposed to specifying RKKK where the first K has to equal 0).

Second confession: The RM03 in the above example will actually cause 4, not 3, bytes to be reserved at the location labeled DECIM. This is because the RM03 could have been followed by a Chip-8 instruction (the beginning of a subroutine for instance) and all instructions must begin, as you know, on even-numbered addresses. The assembler takes this into account, and will reserve one additional byte for any odd number specified. Thus FF = 256 bytes, not 255 for those readers who thought they had me trapped earlier in a logical confusion of mistaking zero bytes to be the equivalent of the zero $\underline{\text{th}}$ byte equal to one. You can not, of course, do very much by reserving zero bytes, (in fact, it will cause an error message to be output) just as you will not find very much to play back from a tape on which you recorded 0 (i.e. the first?) pages. Those of you who have tried to record more than 3.75K of memory appreciate just what I mean.

;End confession

LINKING

Linking refers to the process of joining sections of programs together for assembly. Up to 14 such sections may be linked by Chip-8 Assembler-3 before

overflow occurs.

Normally, linking is completely automatic. If your program is too long to fit on the 6 text pages provided by Text Editor-21 (as for example in the sample source listing at the end of this section) you only need to record what you have on tape and then continue typing your program after clearing memory (a function also provided by Text Editor-21 -- see instructions). When these sections are assembled (a procedure discussed in section II), they will be automatically linked together by Chip-8 Assembler-3 in ready-to-run form.

But many times you will want to jump to the next section from the midpoint of a previous one, or you will have a large amount of data in the way to jump over before continuing. For that reason, the addresses of each section's first instruction line are calculated by Chip-8 Assembler-3 and stored in a link table which you may access to join or link sections together.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
						2	X	X	X		L	I	N	K	

When the argument LINK is used in any section of

the source listing, the address of the next section will be inserted into the instruction which precedes (on the same line) the argument LINK. (Remember, LINK is an illegal label, and in fact this is the only time where an argument will not have a corresponding label in the source listing.) The argument LINK follows all the rules for other arguments -- it may be used as many times as necessary. Every time LINK is encountered as an argument, the address of the next section to come will be inserted in that instruction.

Provided the various sections do not have to be in any particular order to run -- a possibility when you have certain initializing routines you use often that you would rather not turn into subroutines -- the sections may be linked without assigning a label to each one.

In the case where one section will be called vs. the next section depending on the outcome of a certain test or key press, a skip instruction could specify a jump (or a subroutine call, or an AXXX set "I" instruction) to either a particular labeled program section of the first instruction of the next section via the link. The first line of such a section may also be labeled with no effect on the link, providing a method of jumping around in memory (and a

way to keep track of what you are doing) that suggests some imaginative programming possibilities.

For most programming, the link will not be used frequently. But after the program has been assembled, the link table containing the address of each joined together section will be recorded on tape (see section III) for viewing with Text Editor-21. Even if the link is not used in the source listing, the link table provides a valuable debugging tool should the program -- knock on silicon -- not work the first time.

CONCLUSION - SECTION I

You now have enough information to enter the sample source listing into your computer using Text Editor-21. The listing is printed in three columns, each column conforming exactly to the 16-character a line format described here. Typewriters and printing presses, however, give different results than will your TV screen so be sure you understand how to enter labels, instructions, arguments, comments and pseudo instructions before beginning. If you find it slow going, it is because you are not yet familiar with the ASCII code. Practice! After a week or so, I was touch-typing at a good two characters a second on

the hex pad -- it's really no more difficult to learn than Morse code.

In the middle of the center column you will see the words "**SEE NOTE." Do not enter these words in your listing. The note is to tell you that you will have reached the end of Text Editor-21's memory buffer and will have to record this section, clear the text buffer and then continue with the next section. When you finish this, record the second section on tape following the first, leaving a 5 second -- not critical -- space or so. You may leave out the comments if you wish to make things go faster, but be careful to include all the labels and arguments in the right places. (Hint- all ASCII numbers begin with 3 followed by the number. 31=1, 32=2, etc. Hex letters A-F all begin with 4 and are numbered sequentially from 41=A to 46=F. That should help.)

I have refrained from describing the new game, and I'm not going to disclose what it is just yet. But here's a small appetizer: I got the idea from a cellophane wrapped toy in a box of Kellogg's Corn Flakes which believe it or not we are able to buy here in Southern Mexico. Also Coca Cola, and Johnson's Baby Shampoo. Donkeys in the dirt paths rub noses with modern technology in our beautiful Sierra Madres,

but the old ladies still come to market with steamed tamales wrapped in corn husks the way they were when the Aztecs built their pyramids on the plateau that is now Mexico City.

That's not much of an appetizer, and I am digressing again. But I want it to be a surprise. Come back when you have loaded the source listing on tape and we'll go on to the next chapter.

-SAMPLE SOURCE LISTING-

```
;GET IT TOGETHER
;5/12/79  T.SWAN

BEGIN:AXXX VARS
      F665;GET

      6D00;VD=00
INIT1:6C10;VC=10
INIT2:2XXX BTSEL
      DCD8;SHOW

;SAVE X-Y
      2XXX PCSEL
      80C0;V0=VC
      81D0;V1=VD
      F155;PUT

      7201;V2+01
      7C18;VC+18
      3C40;SK=40
      1XXX INIT2
      7D10;VD+10
      3D20;SK=20
      1XXX INIT1
      6200;V2=00

START:FE0A;VE=KY

;SET V2 = PIECE

      4E01;NE=01
      6200;V2=00
      4E0C;NE=0C
      6201;V2=01
      4E0A;NE=0A
      6202;V2=02
      4E0F;NE=0F
      6203;V2=03

;ERASE PIECE

      2XXX PCSEL
      F165;GET
      8C00;VC=V0
      8D10;VD=V1
      2XXX BTSEL
      DCD8;ERASE

      4E00;NE=K0
      2XXX ROTAT

;ADJUST XY'S

      4E02;NE=02
      7DFF;VD-01
      4E04;NE=04
      7CFF;VC=01

      4E06;NE=06
      7C01;VC+01

      4E08;NE=08
      7D01;VD+01

;SAVE XY

DISPL:2XXX PCSEL
      80C0;V0=VC
      81D0;V1=VD
      F155;PUT

;REDISPLAY

      2XXX BTSEL
      DCD8;SHOW
      3F01;SK=01
      1XXX START
      6001;V0=01
      F018;TONE
      DCD8;ERASE
      7CFF;VC-01
      7DFF;VD-01
      1XXX DISPL

;END MAIN PROG

;CONTINUED

**SEE NOTE

;GET IT TOGETHER
;5/12/79  (CONT)

;SUBROUTINES

PCSEL:AXXX POSIT
      8020;V0=V2
      8004;V0x2
      F01E;I+V0
      00EE;RET
POSIT:RM08;08 BT


BTSEL:AXXX BITS
      4200;NE=00
      F31E;I+V3
      4201;NE=01
      F41E;I+V4
      4202;NE=02
      F51E;I+V5
      4203;NE=03
      F61E;I+V6
      00EE;RET
ROTAT:8030;V0=V3
      4201;NE=01
      8040;V0=V4
      4202;NE=02
      8050;V0=V5
      4203;NE=03
      8060;V0=V6

      7008;V0+08
      4020;NE=20
      6000;V0=00

      4200;NE=00
      8300;V3=V0
      4201;NE=01
      8400;V4=V0
      4202;NE=02
      8500;V5=V0
      4203;NE=03
      8600;V6=V0
      00EE;RET

;END SUBROUTINES

;DISPLAY
;PATTERNS

BITS :FCFC;"1"
      0C0C
      0000
      0000

      0000;"2"
      F0F0
      C0C0
      C0C0

      0000;"3"
      0000
      3030
      3F3F

      0303;"4"
      0303
      0F0F
      0000

;V0-V6 VALUES

VARS :0000;V0,V1
      0000;V2,V3
      0800;V4,V5
      1800;V6,XX

;END DATA
```

**NOTE-RECORD THIS MUCH (SEE TAPE I/O INSTRUCTIONS FOR TEXT EDITOR-21)
CLEAR MEMORY, THEN CONTINUE

II - ASSEMBLY

II,A. FIRST PASS

The most complicated part of assembly is already completed -- the typing of the source listing. The actual assembly of the source listing is fully automatic and requires only a few easy-to-remember instructions from you.

You now have two sections on tape containing the sample source listing at the end of section I. Each of these sections are 6 pages long, but as Text Editor-21's auto Tape I/O contains its own taping routines, you do not have to specify (or even be aware of) how many pages have been recorded. Likewise, CHIP-8 Assembler-3 contains its own Tape I/O routines, and you do not have to punch in either an address or the number of pages to read or write from tape -- all is automatic.

The first step however, is to load in the Chip-8 Assembler-3 program. <u>This</u> you must do using the normal ROM system monitor. Load four pages beginning @ ML 0000. (If you were running Text Editor-21 before, it will no longer function, of course.) After loading the assembler, flip the run switch up. Nothing will

happen, and the screen will be blank.

Position your tape just before the first of the two sections of sample source listing you have recorded. Get ready to play back the recording, but do not start the tape machine yet.

START ADDRESS

1) The assembler requires an initial instruction for it to know at what address you want it to start assembly. For most Chip-8 programs, 0200 will be the start address though you may specify any start address -- even mid pages -- that you want. (Remember, Chip-8 programs require even-numbered addressing.)

Punch in 0200 on the hex keypad. This address will appear in the upper left corner. (One=page resolution comes as quite a surprise after using the higher 4-page routine, doesn't it?) If you make a mistake, simply continue punching numbers till you have the address right.

After you have entered the start address of 0200, press Key E (Enter) to tell the assembler you are ready to begin. A quick series of beeps will be heard to signal a ready and waiting condition.

IMPORTANT: FROM NOW ON, DO NOT FLIP THE RUN SWITCH DOWN UNTIL YOU ARE TOLD TO DO SO. TAPE I/O IS AUTOMATIC AND THE COMPUTER MUST NOT BE RESET DURING ANY PART OF THE ASSEMBLY PROCEDURE TO FOLLOW.

Whenever you hear the quick series of beeps during the first pass of assembly, you have two choices:

1) Key 1 -- continue (or start) first pass
2) Key 2 -- begin second pass

You are now ready to read in the first of the two sections of the source listing from tape. Press Key 1. The screen will go blank indicating the computer has entered the tape read mode. Start the tape. After the first section has been entered and processed, the display will come back on, and the quick series of beeps will again be heard. Stop the tape before the next section begins to play. (Use "pause" if you have it.)

Once again, press Key 1 and restart the tape, playing in the second section of the source listing which you have recorded. This completes the first pass. DO NOT TOUCH THE RUN SWITCH!

After you become familiar with its operation, you won't have to stop the tape in between sections as long

as each are separated by about 2-3 seconds of leader. Up to 14 sections may be played in by simply pressing Key 1, playing the tape, pressing Key 1, playing the tape, etc.

What the computer has done is to make a list of all the labels in your program calculating the correct address of each using the start address which you fed in at the beginning and taking into consideration RMKK pseudo instructions to reserve memory bytes at specified locations. This list of labels and addresses is called the symbol table and will be available for your reference following assembly. The link table has also been created during the first pass.

The actual computation of the symbol and link table occurs for each section within the time the TV display comes back on and the sound of the beeps signaling "ready." What takes time is the taping, not the assembly itself.

(If you receive any error messages - shown by a number and the letter "E" and accompanied by a long tone -- turn to the Error Messages section III D. This is unlikely but could happen if you were not careful while loading in the source listing.)

II B. SECOND PASS

Back up the tape to the same spot where you began the first pass just before the first of the two sections of source listing recorded there. Press Key 2. The screen will remain on, the beeps will sound, and you once again are presented with two options:

1) Key 2 -- continue (or start) second pass
2) Key F -- begin taping

Every time you hear the quick series of beeps during the second pass, you have the above two options. (The only way to return to the first pass from here is to start over again.)

Playing in the second pass of your source listing is accomplished in the exact same manner as for the first, except that Key 2 now must be used to signal the computer to expect another section. Therefore, press Key 2, the screen goes blank; start the tape. When the display returns, the beeps will sound; press Key 2 again. Continue in this manner until all sections are completed. At the end of the last section, the display will be on. ONCE AGAIN -- DO NOT TOUCH THE RUN SWITCH.

The second pass is now completed. What the assembler has done this time is to match all the arguments in the source listing with the symbol table created during the first pass. It takes the addresses from the table and inserts them in the Chip-8 instructions needing those addresses. An area of memory (three pages long) now contains the completed object program in a ready-to-run form, taking into account RMKK pseudo instructions and linking instructions as previously discussed.

It is vital that you play each section during the second pass in the exact same order as for the first pass. If you receive an error message, it will most likely be a 1E type -- no symbol found in table and you will have to check your source listing and start again. (see III,D).

Provided you have received no errors, you are ready to procede to the next chapter. DO NOT TOUCH THE RUN SWITCH. Leave the computer on and turn to the next section.

III -OUTPUT

III,A. RECORDING

The assembly of your program is now complete and you are ready to record the program, reload and run. Chip-8 Assembler-3 contains a special two-part recording scheme -- please read the next two sections III A and B carefully before proceeding.

1) OUTPUT PARTS 1 AND 2

The display now looks as it did before with the start address (0200 for the sample source listing) in the upper left corner. The first step in the recording process is to tell the computer that both passes of the assembly have been completed. This is done by pressing Key F to signal "Finished." Press Key F -- but press it only one time. The beeps are again heard signaling ready, and the letter F is displayed in the upper right corner to remind you that the next key press (Key F) will begin the tape output.

Position your tape at an empty spot. Probably the best spot for this is immediately after the sections of source listing previously recorded. If

you have a "pause" control, set the tape machine to record, but do not start the tape just yet. (If you do not have a "pause" control, you will have to start recording in whatever way is normal for you.) Make a note of the location number from your tape counter.

Taping is in two parts as mentioned previously. Once taping has begun, DO NOT STOP THE TAPE RECORDER UNTIL AFTER PART 2 HAS BEEN COMPLETED. Between parts, the display will return and you must resist the urge to turn off the recorder in between parts.

Following is the tape output format:

1) Symbol and Link Tables - 6 pages - display off
2) Four-second pause (approximate) - display on
3) Object program - 3 pages - display off

Taping begins as soon as you press Key F for the second time. (You do not have to specify addresses or the number of pages -- all is automatically controlled by Chip-8 Assembler-3.)

First start the tape with your machine in the record mode. Next press Key F.. The familiar high-pitched tone of recording will be heard as the output is sent to tape. Sit back in your chair, and when the display returns, sit on your hands! DO NOT STOP THE TAPE

RECORDER AFTER THE FIRST TAPING SEGMENT!

When the display does return, the assembler's output routine waits four seconds, and then begins another taping output automatically (see the tape output format above.) The display will again go off and you will hear the high-pitched tone indicating output to tape. When the display returns for a second time, you may stop the tape recorder.

NOW YOU MAY FLIP THE RUN SWITCH DOWN.

Assembly and output are complete. Please read the next section, III,B, for loading and running the object program.

III,B. SYMBOL AND LINK TABLES -- THE OBJECT PROGRAM

You now have a tape which is configured as follows. (I use the sample source listing for the example.)

1) Source Listing -- main program
2) Source Listing -- Subs and data
3) Symbol and Link Tables
4) 4 second pause
5) Object program in assembled form

Resisting temptation to try out the new program, (besides you don't have the instructions yet -- no fair peeking ahead) go back to where you have Text Editor-21 on tape, load the program (12(C) pages @ ML 0000 using the ROM system monitor) and run. Return to the point on your tape where the Symbol and Link Tables are recorded (number 3 in the above example). This is at the tape counter number you wrote down before starting the tape output.

Load in the Symbol and Link tables using Text Editor-21's tape read feature. (Without resetting the computer, press Key C, then Key B for tape read; start tape play; stop tape when display returns.) Leave the tape in this position which will be just before the object program (#5 in the above example).

You are now viewing the symbol and link tables which were created during the first pass of assembly. The symbol table comes first. All labels used in the program are listed alphabetically sorted by the first letter along with the address at which they were calculated to exist. These addresses correspond to the assembled object program. If you examine the program (do it later, though, after this section) you will notice that the label BEGIN which is the label for the first line in the source listing has been assigned

the address 0200. At 0200 in the object program, then, you will find the AXXX instruction which we discussed in section I. The assembler has calculated all the addresses, and the symbol table gives you a memory map to your program which you will find valuable in debugging the object code. This means you do not have to reassemble to adjust such things as a forgotten bit in a display pattern, or the fine-tuning of a timing loop. Along with one of the Chip-8 Editors mentioned in the introduction, the symbol table will prove to be quite useful in program development.

Under the symbol table you see a list of addresses followed by many 0000's. This is the link table. The first link address is the address of the first line of the second section of source listing that was assembled. The second address goes with the third section, etc. The last address in the link table (the one just before the 0000's) represents the byte that immediately follows the last byte of the object program. This information, important for debugging, will also come in handy when you wish to assemble only part of a program, then continue in the normal way using the system monitor or your own Chip-8 Editor.

The object program is now ready to load at the start address (0200) using the normal ROM system

monitor. Since the link table tells you where the end of your program lies, you can easily calculate the number of pages you need to load. However, you are always safe if you enter three pages from the start address. (For creating object programs longer than three pages, see section V.)

After loading the sample object program @ 0200, and before running the program, you will also need to load the two-page Chip-8 Interpreter supplied with your VIP manual. If you do not have this recorded separately anywhere, you may load the first two pages (@ 0000) from one of the game programs you have that include the Chip-8 Interpreter on tape. Now you are ready to run the game "Get it Together." Please turn to the game instructions which follow the summary of operations. (I hid the instructions there and did not put them in the table of contents. Don't tell me you peeked?!)

III,C. SUMMARY OF OPERATIONS

For your reference, the operations necessary to assemble source listings are condensed below. It is assumed that you have the source listings stored on tape in the order they are to be assembled.

1) Load Chip-8 Assembler-3 - 4 pages @ 0000

2) Flip up run switch -- screen blank

3) Enter Start Address -- 0200 for most Chip-8 programs -- press Key E to "Enter" this address and continue

4) PASS 1

   a) Press Key 1 -- play one source listing section

   b) Beeper signals ready for next section, or second pass

5) PASS 2

   a) Press Key 2 -- beeper signals ready

   b) Press Key 2 again - play same source listing sections

   c) Beeper signals ready for next section, or begin output

6) TAPING

   a) Press Key F -- beeper signals ready to begin output. "F" is displayed in upper right corner

   b) Advance tape to clear spot

   c) Start record on tape machine

   d) Press Key F again - tape output begins

   e) Automatic Tape Format:

      1) 6 pages - Symbol and Link Tables

      2) 4 second pause

      3) 3 pages - Object Program

7) Flip Run Switch down - load program and interpreter to run

8) View Symbol and Link Tables with Text Editor-21

Game Description -- "Get it Together" evolved from a toy I found in a box of cereal. The toy consists of four pieces of plastic that must be joined together to form a square. Upon flipping the run switch up, you will see the four pieces, one in each corner of the screen (more or less). Each piece may be moved and/or rotated in your attempt to join them together. No two pieces may overlap (though they may "touch") and if you try to do so, the program will find the first available space for the piece and redisplay it there.

Each piece is selected by pushing one of the Keys 1; C; A; F; corresponding to their initial screen positions. If after a while you forget which piece is which, you can find which key goes with which piece by pushing them all -- the pieces blink once each time they are selected.

The Keys 2; 4; 6; 8; move the selected piece in the appropriate direction, same as for most Chip-8 games. Occasionally a piece will disappear off screen, but it may be brought back by continually pressing Key 2. Key 0 will rotate whichever piece you have selected.

I know of two distinct solutions to the game. One is relatively simple, the other isn't quite so obvious. I have not included the solutions here (I think you'll have more fun working them out for yourself), nor have I included flowcharts, program description, etc. I think the source listing is well enough documented for easy following - but if you do have trouble, just write to me in care of ARESCO, and I'll try to help.

III D ERROR MESSAGES

During assembly, you may accidentally overstep the limits of CHIP-8 Assembler-3. If assembly were to continue, the object program would be incorrect, causing bugs which would be difficult to find. CHIP-8 Assembler-3 helps locate bugs before they occur, aiding greatly in creating bug-free object code. If you receive no error messages during assembly, you can ge sure that none of the following conditions exist. If your program still not run, these conditions can be eliminated as possible sources of error, thus narrowing your search for the problem.

All error messages will be displayed in the lower left corner, taking the form "N E", where N = the code for one of the errors listed below. A long error tone also occurs, and assembly is halted - and cannot be continued - after receiving an error message. The section just played in - on either the first or the second pass - contains the error, further narrowing the search for problems to that section.

#

1 E NO SYMBOL IN TABLE -- this will probably be the most common error you encounter. It means you used an argument somewhere, and forgot to include a corresponding label in the source listing. Recheck that section, watching carefully for misspellings which could cause the message to

to be output. Remember, a zero is not the same as the letter "O". If you still have problems, reread the sections on writing labels and arguments into your program.

#

2 E MEMORY OVERFLOW -- Object programs may be up to three pages in length before overflow occurs. Exceeding this limit will result in error message #2. If your start address was 0200, overflow will occur if your program extends beyond 04FF. Most game programs in Chip-8 will fit easily within this range, but see section V,C for assembling programs longer than three pages.

3 E SYMBOL TABLE OVERFLOW -- Up to 73 different labels may be used in your program before the symbol table overflows. (Any number of arguments may be used as long as all have at least one identical label.) This is a difficult condition to fix as it means you must rewrite the source listing using fewer labels. Chances are, however, if you exceed this limit you are using far too many Go-To's in your programming -- 73 labels are quite a lot!

4 E (NOT ENABLED AT THIS TIME -- reserved by the author for later contemplated expansions.)

5 E RESERVE 0C BYTES -- Specifying RM0C (see section I,C for using this pseudo instruction) will result in the "5E" error message. Many times you will not know just how many bytes you intend to reserve while writing the program. In that case the RM0C will reserve the line for you in the source listing. If you forget to go back and specify the correct number, Error "5E" reminds you to do so, thus preventing a major crash when you attempt to store data in an area occupied by a subroutine, etc.

6 E LINK TABLE OVERFLOW -- Up to 14 sections of source listings may be joined together before the link table will overflow. In most cases, short subroutines and data blocks that occupy separate source listing sections may be combined to overcome this error condition. (A one-byte change may disable the link table, but then neither the link argument nor the link table will be valid. See program description. This will allow unlimited section-joining subject to to the other error messages above.)

After receiving any error message, the error must be located and fixed before attempting to reassemble. The above descriptions should help you locate such errors quickly. You may examine the symbol table at this time by recording 6 pages (using the ROM System Monitor) from 0A00. Text Editor-21 may then be used to view the table though it will not be sorted and will not have accompanying addresses. It will also not be arranged in a column and words may be split across rows, but it is available for you to examine. The link table may be viewed using the ROM System Monitor, beginning @ 02E0-02FF in hex form.

Should you make an error during assembly, simply restart the computer and reassemble. Chip-8 Assembler-3 does not alter any of its own instructions making it a candidate also for a permanent home in EPROM. (Text Editor-21 is also PROM-able, though a couple of features such as cursor on/off and reverse video would need revising or they would be disabled in EPROM.)

IV - RUNNING AND DEBUGGING

IV A. USING THE SYMBOL AND LINK TABLES

Much has already been said about the symbol and link tables. This section will suggest some further hints on their use in debugging the object program.

While writing source code, keep a list of all the labels you have used. Before assembly, check all arguments against this list. This simple preventative measure will help your programs to assemble the first time through. Be sure comments are preceded by a semicolon and that only arguments are preceded by a space. If your program does not run properly, view the symbol table using Text Editor-21, writing down the addresses next to the list of labels you made. Also note down the start addresses of each section from the link table. Often, programs may be debugged with this information without having to reassemble from scratch.

You may further identify sections of your program by inserting dummy labels at known locations even though they do not have accompanying arguments. The dummy labels become part of the symbol table during taping, and will further direct your search for a bug.

(This could take the form: DBUG1; DBUG2; etc., which would result in all the dummy labels being output to the sorted symbol table in order of use and more or less in one batch.)

The addresses of each label in any order you want may be "captured" and inserted into the end of object programs by using the following sequence at the end of the last section of the source listing.

```
;DUMMY LABELS
       OXXX  BEGIN  ┐
       OXXX  START  │
       OXXX  DBUG1  │
       OXXX  DBUG2  │
        '     '     ├─ Your labels go here
        '     '     │
        '     '     │
        '     '     │
        '     '     │
       OXXX  DBUGX  ┘
 ;END CAPTURE
```

When assembled, the addresses for each of the labels will be inserted into the OXXX's with matching arguments. The object program then contains its own symbol table for viewing with the ROM System Monitor, or other Chip-8 Editor. (Of course the symbols themselves are not there, only the addresses.) If the Dummy Labels Capture Routine is included on a separate source listing, the link table will indicate where it can be found in memory.

The uses for the symbol and link tables should

grow with experimentation. New uses for them would seem to make good subjects for an article or two in VIPER.

IV B. STORING THE PROGRAM

Tools such as text editors and assemblers tend to eat up tape faster than a wild cassette machine with a burned out bearing under the capstan. In no time at all you will have a lot of information to store and keep track of, creating a potential confusion among your game tapes, utility tapes, source listings, etc. The following section recommends a tape storage system which I have used extensively to organize my cassettes. Only once have I accidentally erased a program, and that was at 2:00 AM, when I had no business programming a computer. The solution to that problem was to go to bed. The back ups built into the system allowed me to restore the tape with no further problems the next morning, and I slept soundly.

As these chapters are intended to describe the workings of CHIP-8 Assembler-3, I will not go into the tape storage system as deeply as I would like. I will instead concentrate on storing source listings and object programs. But first a general look at the system:

All tapes are coded by a single letter and number. The letter stands for a general grouping, or program category to be found in that section. The number indicates a particular tape within that group.

LETTER

A = Utility Programs - Editors, Disassembler, Chip-8 Language, Character designer, etc.

B = Game programs -- final versions

C = Work tapes -- trial runs of programs

D = Subroutine source listing library

E = Program source listings -- symbol tables

It's not a complex system by any means. The only firm rule is that all programs exist in one form or another on at least two tapes allowing re-creation of destroyed programs.

When working with Chip-8 Assembler-3, it will be helpful to organize a separate "C" tape with the following format:

Cassette C-N

1) Disassembler-7 -- to view machine language subs
2) Text Editor-21 -- for creating source listings
3) Chip-8 Assembler-3 -- assembling source listings
4) Chip-8 Editor -- debugging object programs

5) Chip-8 Language -- running Chip-8 programs

6) Chip-8 Language -- modification #1 (your own)

7) Chip-8 Language -- " (2-page)

8) Custom Language

9) Work space -- trial source listings

This tape contains all the tools you need to create source listings and try out Chip-8 object programs. The same work space is used for all new programs, and is intended only as an intermediate storage area for those programs.

After assembly, when all the bugs are worked out, record at least one final, ready-to-run version on the appropriate tape (A or B in my system).

Then using Text Editor-21 as the work horse, transfer all source listings to Tape B along with the symbol table where they will provide a back-up in case the object program is accidentally destroyed. (If the source listing happens to be erased, at least you still have the program.) As the object program is most vulnerable -- it is the one that will be used over and over -- this should provide adequate back-up. A more

elaborate system could be devised on this principal, and I sometimes put programs on three tapes if they are very complex, or save intermediate (I-give-up) programs elsewhere while the idea settles.

I would strongly suggest that the above Tape C contain only duplications of the programs that are tucked safely away elsewhere too. While you could always reload the above tools using the program listing provided, the destruction of the above tape if it is your only copy will mean a long afternoon of (boring) hex pad punching.

Tape D is intended as storage for general often-used subroutines kept in source form, ready to be assembled into your program. Chip-8 Assembler-3's capability to relocate subroutines anywhere in memory means you can concentrate on the main program rather than re-write the same old beeper routine that you included in the last twelve games. Now when you write a general purpose subroutine, it is always available for any of your programs -- you never have to write it again.

It is a good idea to transfer these subroutines when you need them to Work Tape C for assembly, then assembly may procede without changing tapes around. These subs may or may not be stored with the saved source listings on Tape E. It's up to you how much

duplication you want in the system. (See section VI for five starter subroutines to begin your own library.)

Whatever system you use, be sure to keep accurate notes in a separate book containing the tape numbers for all programs. Periodically my wife types these notes for me to clean things up. If you are not married, type them yourself. If you do not own a typewriter, why not marry someone who does? (Last suggestion, I promise.)

V - SPECIAL CASES

V A. LINKING OUT OF ORDER

Autonomous source listing sections may be linked regardless of their order in memory, and will function after assembly unaffected by the order of assembly. (Pass 2 must duplicate the order of Pass 1, however.) By autonomous I mean each section must stand on its own as a unit, carrying with it work bytes, data, etc.

Provided each section is given a distinct label, that section may be called by the main program no matter where it resides. This is the theory behind the subroutine library in the next section. If, for instance, you need a beeper sub and a timer sub, you may

assemble them in any order and they will work just as well. Be careful, though. The program must still be constructed in the usual sense. Consecutive instructions will be executed in memory in a path traveling downwards unless that path is altered by an instruction directing the flow elsewhere. You may use the 1XXX LINK instruction to jump over data blocks at the end of source listing sections rather than have to wait until you are done programming to remember to insert bit patterns, etc. Subroutines can and probably should carry their own work spaces, constants, etc., along with them. This will aid in debugging as you will always know where to find these bytes no matter where the subroutine is assembled.

V.B. MACHINE LANGUAGE SUBROUTINES

Machine language subroutines present a special problem to Chip-8 Assembler-3. If the subroutine is not memory dependent -- in other words if it is completely relocatable -- it may be written into the source listing and assembled along with the rest of the program. It may be labeled and called with the instruction OXXX MLS where the MLS argument here is only my choice for the example -- you may use any label and argument

you want.

However, if the subroutine depends on its location in memory in order to run -- it contains branching instructions for instance -- you will have to use another method to get it into your object program. This is less difficult than it sounds.

The most convenient way to insert machine language subroutines is to write them into an area of memory where you know your program is not going to interfere. Once you know the address, you may call the subs by writing in that address in the normal way when you write the source listing. In other words if the subroutine is at 0600, the instruction 0600 in the source listing will call the subroutine. Do not put an argument after the call, though you may of course insert a comment there preceded by a semicolon.

If you wish, however, to have the subroutine begin at the very end of your program, and don't yet know where that is, you may assemble the program before writing the subroutine. Reserve two bytes for the call wherever you want, using the pseudo instruction RMO2, labeling the position so you can find it in the symbol table. The link table tells you where to start the subroutine which may now be entered in the usual manner using the ROM system monitor or another editor.

Then go to where you reserved the two memory bytes and insert the call to the sub.

Data blocks that are very large may be handled in the same way as it is probably easier to key the data in and record it on tape for loading after assembly. It is possible to enter data, for example, beginning at ML 0000. Record one extra page than you have entered. After your program has been assembled, find the last address in the link table and load the data from there, loading only as many pages as are needed this time. (You may always read and write to tape using the ROM system monitor set to begin anywhere in the middle of a page -- you don't have to always start with the first byte. However, even if you start with 02FE, the monitor will think that those two bytes are a full page. In other words, if you record one page starting @ 02FE, exactly two bytes will be output to tape. You cannot record exactly 256 bytes from ML 02FE to 03FE. You would have to specify 2 pages and record from 02FE to 03FF to include the second page. Experiment with this in order to grasp the idea.)

V C. LARGE PROGRAMS

Break up your sections of source code into groups

that you know will fit in no more than 3 pages of memory for each group. At the end of each group, insert a go-to instruction using the actual address of the next 3-page segment of memory to follow that group. If your first group starts @ 0200, for example, 04FF would be the last byte in that 3-page area (whether it is used or not). Therefore enter a Go-To 0500 in the last line of that group. You will be assembling the first group at 0200 and the second group at 0500, etc., until all groups are assembled. Loading them in together then will form your long program. With care you can avoid gaps in between groups.

But, you say, how does the first group know where to jump to if that jump goes to the middle of a different group? Easy, but pay attention.

1) Load Chip-8 Assembler-3. Using the ROM System Monitor, change the byte at 0034 to 2A.

Everytime you press Key 2, instead of going to pass 2, the assembler will now expect you to enter a new start address -- you have just disabled pass 2.

Does your program contain more than 14 sections of source code? If not, then continue. Otherwise perform

the following step.

2) Change the byte at 02C4 to D5. You have just disabled the link function. If your program used it, it will not now work. (What? You want to eat your cake too? Really!)

Perform a first pass on the first group of your source listings. After this, press Key 2. Now you must enter the address of the next group as described above (0500 probably if the first group started at 0200). Press Key 1 to start the first pass for the second group. Continue until all groups are done. As long as you never exceed the 73 labels limit, you'll have no problems.

What you have done it to create a symbol table that includes all the labels and addresses for all groups.

Flip the run switch down and call up the operating system.

1) Change the byte at 0034 back to 7B -- don't worry about the other change if you made it.
2) In addition, change the byte at 0329 to 0D

You may now flip the run switch back up. The symbol table you have created has been preserved by

the changes you made and you may now proceed with the second pass. First passes must not be made at this point.

Enter an address and press Key E -- any address, it does not matter -- then press Key 2 to select the second pass routines. For each group (the 3=page groups that you calculated previously) perform a normal second pass, recording the output just as explained before. In between groups you will reset the computer, enter a bogus address, then go directly to pass 2.

After all the groups have been assembled, they may be loaded consecutively into memory (group one at 0200, group two at 0500 etc., for example) and the whole program is ready to run. All symbol tables are duplicates of each other. The groups are linked together by the Go-To's you inserted at the ends of each group. The LINK function will work provided you have no more than 14 sections to join. This represents a maximum of 1,344 Chip-8 instructions which would occupy 2,688 bytes of memory. If the link table is disabled, then the size limit is only checked by the 73 labels limit. Both conditions represent a large capability for a 4K system. Owners of additional memory may want to make permanent modifications to enable even larger programs. See the program description

and the program listing.

VI - SUBROUTINE LIBRARY

Five subroutines of a very general nature are presented here as a starting point for building your own subroutine Source Listing Library tuned to your personal software approach. Each of the five routines lists its required input (if any); its output; the variables that are changed by it; and the subroutines it may itself call in order to perform its function. In addition the labels used by the routine are listed and you should avoid using the same labels in your programming if you plan to use any of these routines. The title of the subroutine represents the label or argument with which it may be called in the source listing by a 2XXX instruction followed by that argument.

After you write your own subroutines to add to the library, list the information for each in the same way. When you need to use a particular routine, then, you do not have to delve into the actual code to figure out how it works three months later. The information is right there in your notes and on tape, and the routine may be assembled without further examination. You know what the subroutine needs and what it

will return -- it works -- and beyond that you can trust it to do its job.

It helps to keep certain conventions thoroughout your programming. Here I have specified V0 as the variable most often used to pass a value to a subroutine. V1 and V2 are also often changed during the program run, most often due to the use of the FX55 and FX65 which I have termed the PUT and GET pair in the sample source listings here. VC and VD were chosen for XY coordinates used to display patterns on the screen. No real reason for the choice -- you may doctor the routine any way you wish.

(The actual source listings for each routine follow these descriptions. You may record them as separate source listing or together as one all-purpose listing -- the choice is up to you.)

1) TIMER -- TIMER SUBROUTINE

INPUT: V0 = Value for Timer

OUTPUT: Returns when timer == 00 -- use to delay programs at call point

CHANGES: V0

CALLS: No other subroutines

LABELS USED: TIMER TIME (Call with 2XXX TIMER)

2) BEEPR -- BEEPER SUBROUTINE

INPUT: None

OUTPUT: 7 short quick beeps

CHANGES: V0 V1

CALLS: TIMER

LABELS USED: BEEPR BEEP (Call with 2XXX BEEPR)

3) NUMB3 -- THREE NUMBER DISPLAY SUBROUTINE

INPUT: V0 = value for conversion to 3 digit decimal number

VC VD = XY coordinates of the upper left corner of a 14 x 5 bit block where the three numbers are to appear.

OUTPUT: Displays/Erases (on second call) the converted value in V0 @ VC VD

CHANGES: "I" V0 V1 V2 (VC and VD are reset to their original values)

CALLS: No other subroutines

LABELS USED: NUMB3 C-3DD (Call with 2XXX NUMB3)

(To use this sub let's say you are keeping the player's score in VE. Set VC,VD = the XY coordinates of where you want the score to appear, set V0 = VE (80E0;V0=VE) and call the sub with the instruction and argument, 2XXX NUMB3. To erase the score, just set V0 = VE again and call. As VC VD are reset by the subroutine, they only need to be

set the one time unless you change them for other uses later on.)

4) BORDR -- DRAW PLAYING BORDER SUBROUTINE

INPUT: None

OUTPUT: Draws/Erases (on second call) a one-bit wide border around the display area (one-page resolution

CHANGES: "I" V0 V1

CALLS: No other subroutines

LABELS USED: BORDR TOP SIDE BIT (Call with 2XXX BORDR)

5) WINLO -- "YOU WIN/LOSE" SUBROUTINE

INPUT: VC VD set to the XY coordinates of the upper left corner of a 16 x 11 bit block in which you wish either the words "YOU WIN" or "YOU LOSE" to appear or be erased.

V0 = 01 selects the message "YOU WIN"

V0 = 00 selects the message "YOU LOSE"

OUTPUT: Message displayed or erased (on second call)

CHANGES: "I" V1 (VC VD reset to original values; V0 unchanged)

CALLS: One internal sub - no external subs

LABELS USED: DISP LOSE WINLO WIN YOU (Call with 2XXX WINLO)

I hope you will take advantage of the above format to expand your personal subroutine library. Chip-8 Assembler-3 frees you from having to rewrite your most commonly-used subs by having them stored in source form ready to be assembled along with your program the next time you need them.

```
****************
      1
****************


;TIMER

 ;PASS VALUE - VO

 TIMER:F015;TI=VO
 TIME  :F007;VO=TI
       3000;SK=00
       1XXX TIME
       00EE;RET

 ;END SUBROUTINE


****************
      2
****************


;BEEPR

 ;NO INPUT
 ;VO, V1 CHANGED
 ;CALLS TIMER

 BEEPR:6107;V1-07
 BEEP  :6001;VO=01
       F018;TONE
       6005;VO=05
       2XXX TIMER
       71FF;V1-01
       3100;SK=00
       1XXX BEEP
       00EE;RET

 ;END SUBROUTINE


****************
      3
****************


;NUMB3

 ;DISPLAY
 ;03 NUMBERS SUB

 ;I-CHANGED
 ;VO,V1,V2 USED

 ;DISPLAYED
 ;AT VC,VD
 ;VC,VD-UNCHANGED

 ;PASS VALUE FOR
 ;CONVERSION
 ;IN VO

NUMB3:AXXX C-3DD
      F033
      F265;VO-V2
      F029
      DCD5;VO
      7C05;VC+05
      F129
      DCD5;V1
      7C05;VC+05
      F229
      DCD5;V2
      7CF6;VC-0A
      00EE;RET
C-3DD:RM03

;END SUBROUTINE


****************
      4
****************


;BORDR

 ;VO,V1 USED

BORDR:AXXX BIT
      6000;VO=00
TOP   :6100;V1=00
      D011
      611F;V1=1F
      D011
      7001;VO+01
      3040;SK=40
      1XXX TOP
SIDE  :603F;VO=34
      71FF;V1-01
      D011
      6000;VO=00
      D011
      3101
      1XXX SIDE
      00EE;RET

BIT   :8000; "."


****************
      5
****************


;WINLO

 ;VC,VD=VX,VY
 ;VO=01=WIN
 ;VO=00=LOSE
 ;V1 CHANGED
 ;16x11 BIT BLOCK

WINLO:6105;V1=05
      AXXX YOU
      2XXX DISP
      AXXX WIN
      3001;SK=01
      AXXX LOSE
      2XXX DISP
      7DF4;VD-12
      00EE;RET

;DISPLAY

DISP  :DCD5;SHOW
      F11E;I+V1
      7C08;VC+08
      DCD5;SHOW
      7CF8;VC-08
      7D06;VD+06
      00EE;RET

;BIT PATTERNS

YOU   :8BDA
      7222
      2302
      5252
      52DE

WIN   :8B89
      A9F9
      73A2
      322A
      26A2

LOSE  :8E8A
      8A8A
      EEEE
      88EC
      28EE

;END SUBROUTINE
```

PROGRAM DESCRIPTION

Please refer to the program listing. This description will follow the listing in the order it was programmed, not necessarily in the order it performs. It is assumed that you understand the operation of the assembler, and the use of the standard call and return technique as described in the 1802 manual.

Many subroutines of Chip-8 Assembler-3 are general enough that they could be used without modifying in other programs. For instance the ASCII-HEX conversions would be very useful in a number of possible "utility" programs. For that reason, the more general subroutines are catalogued on page 2 of the listing under the heading, SUB ROUTINE DESCRIPTIONS. In the same manner as the Chip-8 Program Library already discussed, each subroutine is listed with its input; its output; the registers that are changed by it; and any other subs it needs in order to function. (Not all of the subroutines are included -- only the ones of a general all-purpose nature.) From here on, all page numbers refer to those of the program listing.

* * *

Page 8 begins with the initialization. R2 is the stack pointer @ 00FF, R3 becomes the program counter

and the call and return registers R4 and R5 are set up (see your 1802 manual). Page 1 lists the registers and their functions in detail.

At ML 001E, the first subroutine is called. Memory is cleared (set = 00's) from 0A00 to 0FFF to prepare for the symbol table, to clear the display page @ 0F00, and to clear the 3-page area which will eventually hold the object program. This will insure that all unused bytes in the object program will equal 00's, making the use of such debugging devices as the Checksum Program published in VIPER, easier to use. (A patch at 0074 from 0021 sets up the pointers for the link table -- to disable the link table, do not change this call -- an easier way exists and will be explained.)

FIRST PASS CONTROLLER

The First Pass Controller on page 9 to page 11 is the main routine in charge of the first pass. RC is initialized to 8195, the address of the keyboard scan routine in ROM, and Address Entry is called to allow the user to specify the start address for assembly. The main loop begins at 0020 with a call to the beeper subroutine, then waits for a key press to determine if another (or an initial) first pass request has been made with Key 1.

Any other key selects pass 2 (though the instructions specify Key 2 which is easier to remember). As very little can happen to your program if you make an error -- you can always start over -- few failsafe features were needed to assure, for instance that Key 2 was selected. These features would have caused the program to exceed its 1K allotment (to conform with Text Editor-21) and are minor deletions anyway.

Tape read is selected @ 0038 for inputting a 6-page segment of source code to 0400-09FF which as a unit consumes more memory than anything else. From 0040 to 0069, each line of text is tested. If the line begins with a space then the assembler assumes no label exists there and goes on to check for the pseudo instruction RMKK on the same line. If a semicolon is encountered in the first character position, the next line will be tested (comment lines are ignored). If the first character is not a space or a semicolon, then the Symbol Table Manager is called to insert the label in the Symbol Table. (You can see, then, that there are 126 valid first characters for labels -- I'll show you later how to enable punctuation, numbers, etc. as first characters for labels.) Along with all this, and unless the line is a comment line, the character at position #7 (see Format Example, section I,A.) is tested. If this character is

an "R", then the Reserve Memory subroutine is called. If the character is a space, the rest of the line is ignored -- no instruction exists there.

A Carriage Return @ 0067 is called in order to loop through all source code text lines, then when RA.1 goes past the last line of text in memory (RA = 0A00), the current address is stored in the link table as the first address of the next segment. A branch back to 002D signals the user to input another instruction to continue or go to pass 2.

You may change the character that signals comment lines by changing the value @ 0041 to the ASCII code of whatever character you want except spaces. Similarly, if your text editor is not inserting spaces in all unused locations, you will have to change from recognizing ASCII 20's @ 0046 & 0062 to whatever occupies your blank spots. Remember, a control character is not recognized as anything but another character. By creating special symbols using my program, A Character Designer for the VIP, and then using that special character set for Text Editor-21, whole realms of custom-tailored labels could be created. Chip-8 Assembler-3 would recognize all your special characters as valid (subject again to semicolons = 3B; spaces = 20). Such versatility is seldom seen

on video boards costing as much as your computer. It's your character set -- you designed it!

Should you want to locate the text for assembly at a different location, instructions at 003E-0042 set RA to the beginning of the data area. At 006C is the page address that RA.1 will equal when it goes past the last page of data. New parameters could be inserted here to examine text elsewhere im memory. The tape read routine will also need modifying.

(If you make any changes here, please read through the second pass controller -- corresponding changes will have to be made there too.)

SECOND PASS CONTROLLER

The pointer to the link address is first reset allowing the LINK argument to be used. R9 is set to the address of the object code area - 3 pages @ 0C00-0EFF. Once again, the ROM key scan routine allows selection of pass 2, or procede to output. (You can see that pass 2 is easily by-passed if all you want to do is create the symbol table. Also, any key other than 2 will jump to the Output routines -- F is just easier to remember.) Tape read again allows entry of the source code, RA = 0400 to point to the beginning

of that code, and the routine is terminated at 00F0 when RA.1 goes to 0A, past the last byte of text.

In the meantime, comment lines are again ignored (@ 009D is the character ";" 3B). Labels are ignored during the second pass -- their usefulness only extended to the creation of the symbol table during pass 1. Therefore the first character of the instruction field is immediately examined @ 00A0 provided the line is not a comment line.

If the instruction field contains an ASCII space (20) the next line is examined. If it is an "R", reserve memory is called to accomdate the pseudo instruction RMKK. (The M is completely inconsequential -- change it if you like.) At 00B4, the four-character instruction is placed in the object code. At this point the instruction is only an embryo. If it is of the addressing type, it will still be incomplete -- the address has not been inserted yet.

Following any instruction, data, or pseudo instruction, the character in position #11 is examined @ 00B7. If a semicolon exists there, the rest of the line is ignored. This will occur <u>regardless</u> of the instruction type. If a semicolon does not exist after an instruction the program checks if an argument is entered starting at position #12. If a space is in

position #12, then the assembler proceeds -- the embryo instruction is assumed to be complete. If not, several things happen. (To avoid confusion in the instructions I did not mention that you may actually precede arguments with a punctuation symbol as long as it is not a semi-colon. A colon could be used for instance. Anything but a semicolon is OK. To me, a space looks better -- but you have to look at your own programs -- change it if you want.)

After determining that something exists after an instruction, that something is tested at 00C1 to 00D2 to see if it is a LINK argument. You may branch over this section by inserting two bytes 30 D8 at 00C1 and 00C2. Now the link table will be prepared but the LINK instruction will no longer operate. (You gain a few bytes by doing this if that is your motive.)

At 00D8, the symbol table created during pass 1 is searched matching the argument found with the instruction being examined. The address is inserted at 00DB and a test is performed @ 00E0 to see if the 3-page object code limit has been exceeded. If so, assembly halts and an error message is output. If not the sction loops until the user selects Key F at which point the Output Manager is called at 00FB to begin taping. Though that subroutine will halt the program,

00FE contains a 23 DEC R3 stop instruction in the event a later programming change causes the Output Manager to execute a return. (Don't you just hate runaway programs?)

Each subroutine mentioned above is detailed below. The assembly procedure is not difficult to understand if you remember that you specify the starting address at the beginning of pass 1. This address is incremented for each valid instruction line (x 2) and every time a label is encountered, the address is recorded along with that label in the symbol table. On the second pass, the object code is created, and arguments are compared with labels in the symbol table. Addresses are inserted and that's it! All assemblers operate in pretty much the same way.

SUBROUTINES

PUT INSTRUCTION IN MEMORY

This is a simple routine (the best kind) that loops twice, taking a pair of ASCII codes on each loop, converting them to hex bytes and storing them in the object code memory area @ 0C00-0EFF via R9. The ASCII codes make up the Chip-8 instructions in the text area

@ 0400-09FF, therefore this routine can be said to create the embryo object code instructions that were previously mentioned.

SEARCH SYMBOL TABLE

RE.0 is used as an equality flag here to determine if the label in the symbol table equals the argument pointed to by RA. RD points to the appropriate label in the table. (The actual testing is done by the next subroutine) If RD.1 is sensed at 0123 to have gone past the end of the table @ 0BFF, the error message #1 is output -- no symbol was found in the table.

TEST STRING

Refer back to the subroutine descriptions at the beginning of the listing to use this sub in your programming. The number of bytes in the string to be tested is set at 0131. Labels in the assembler are always 5 characters long -- even if one character labels are used, the rest are spaces. To use this subroutine, set the length of the string into RF.0 and chop off the first 3 bytes of the sub -- making sure

you set the stack pointer with an E2 before beginning. Two branches also need to be recalculated @ 013A and 0140 in order to relocate the routine elsewhere in memory.

INSERT ADDRESS

Once the embryo instruction has been formed, if an argument is discovered with it, the symbol table search finds the label to match. This routine takes the address from the table and "ORS" it with the instruction preserving only the first character of the embryo.

If you want to write an interpreter that uses 16-bit addressing, I would assume that you will use 2 bytes for the instruction and 2 bytes for the address. In that case the routine would boil down to a simple transfer from the symbol table to replace the embryo which only has the purpose then of holding a place for the address to go. This would be the only change needed for Chip-8 Assembler-3 to handle such an advanced interpreter design.

SYMBOL TABLE REARRANGER

Long but not very complicated, this routine takes

the symbol table which is in the form of labels in ASCII code along with addresses in hex, and transforms the whole thing to ASCII placing the labels and addresses in neat rows in the text area at 0400 for later output to tape. An insertion sort is performed on all labels which assumes a range of 41 to 5C (A-Z) for the first letter of all labels.

If you want to enable the entire 128 ASCII codes (actually 125 are possible) for first characters of labels, all you need to do is increase the sorting range. Restrictions to this have already been covered in the instructions. Change 0160 to 00 and 016D to 80. The sort now assumes a range of 01-7F which is the entire ASCII set less one. This will take an average of 4.6 times as long to accomplish, but you won't be waiting long -- it's still only a matter of a second or two for most tables. It is not possible nor desirable to use 00 nulls as first characters with this routine. This is because the sorting for any letter terminates upon finding a 00 null character -- the table always ends with a null. This confines the sort to only the symbols in the table thus only taking as long as necessary.

An interesting project would be to write a routine to truly alphabetize the set of labels though I don't

know if there is enough space here to do so. You might first sort by first letters and then resort by groups until no exchanges take place. Other algorithms exist in numerous books and papers.

OUTPUT SYMBOL TO TEXT

The only purpose to this routine is to control RA while passing the sorted label to the text buffer. Five characters are transferred pointed to by RD and a colon is inserted for cosmetic reasons after the label. The address is attached by the next routine and RA is set to the next line by a call to the Carriage Return sub.

OUTPUT ADDRESS TO TEXT

The address in the symbol table pointed to by RD is first converted to ASCII form then stored behind the label already in the text buffer.

SYMBOL/OBJECT CODE TAPING

This routine performs the two part taping control procedures for output after pressing Key F the second

time. As the tape I/O routine is described elsewhere, there's not much to say. This routine only orders the others. At 00BA-00BF a timing loop using the interrupt timer (not the one in ROM) causes the 4-second pause between the output segments. The value for the timer is controlled @ 01DD and is currently set to the maximum time possible. Longer times would be difficult, but not impossible to achieve. For a shorter pause, use a lower value.

CLEAR TEXT PAGES

Sets 0400-09FF equal to ASCII spaces (20 hex). Before the symbol and link tables may be output to the text area, all unused spaces must be erased so that garbage is not seen along with these tables when viewing with Text Editor-21. The byte stored in all locations is itself located at 01D2 if you need to change it to something else. RF does the work of storing.

OUTPUT LINK TABLE

Following the output of the symbol table to the text buffer, this routine first creates a space under the last line of labels. Two link address routines

(discussed later) control the pointer to the link table. Output address to text is used to convert the address to ASCII code, storing the code as a line of text. All addresses are output, resulting in the 0000's seen underneath the last active address in the table.

DISPLAY DIGIT

RE is set to the bit pattern in ROM of the hex digit in RE.0 (which is changed by this routine). R6 points to the byte where the digit is to be displayed and RF is a loop counter to assure only 5 rows of bits are output. Following the display of the digit, R6 points to the next byte over in the display area. This is similar to the display routine in ROM.

INTERRUPT ROUTINE

Identical to the ROM one-page routine with the exception of the random number source R9. Freeing R9 was necessary, however R8 functions in the normal way with R8.1 a decrementing timer (once every 1/60 second) and R8.0 a tone generator, producing a latched Q line for a length of time proportional to its value. Use this one-page interrupt when, as here, you need R9

for other reasons.

RESERVE MEMORY

The KK bytes of the pseudo instruction RMKK are converted from ASCII to hex and put in RF.0 at 024C as a loop count. R9 which is the address counter during the first pass and a pointer to the object code during the second pass is incremented the number of times set in RF.0. Thus the desired number of bytes are skipped or reserved for either pass 1 or pass 2. At 0252, R9 is tested if it is odd or even by shifting the least significant bit into the DF register. If odd, it is incremented by one to make it even. All Chip-8 instructions are then assured of starting on even-numbered addresses provided the start address was selected to be even.

CARRIAGE RETURN

RA is the pointer to the text area containing the source listings and other text (the symbol table for output). On a 16-character line, when RA.0 equals "N0" where N is unimportant, then RA points to the first byte of that line. In other words when the

last 4 bits of the pointer equal zero, then the pointer is at the beginning of a 16 byte segment. This routine increments RA until that condition becomes true.

ASCII HEX CONVERSION/CONVERT ASCII-HEX

These two subs will be handy in any programs operating with ASCII data. The subroutine descriptions on page 2 give the parameters for using these routines. The conversion is simple -- adding 9 to all letters and then stripping off the first 4 bits. (As 32 in ASCII equals the number 2 for instance, the conversion only requires stripping off the 3 for all numbers.) The two routines operate together to produce a full hex byte in RF.1 No checks are made, however, for illegal hex digits (i.e. large than F and smaller than 0).

HEX TO ASCII CONVERSION/CONVERT HEX-ASCII

Similarly, these routines perform conversions between hex values and ASCII codes. The formula for conversion is to add 7 to all numbers, then add 30 (hex) to all letters and numbers. The formula is only good, however, for a nibble at a time (4 bits)

as each 4-bit hex digit will result in an 8-bit ASCII equivalent. To convert a full byte, it must first be separated into 4 bit chunks (handled by the first routine) which are then converted by the second sub. RE.1 and RE.0 hold the finished products upon return. In this case no illegal byte checks are needed. It is not possible to represent hex digits greater than F or less than 0 with 4 bits. (This should be obvious, but it is awfully easy to forget that, when analyzed, we are really dealing with binary numbers all the time. Hex and Octal are products of the mind -- our minds -- and no binary computer has ever seen an "F" or an "A" and wouldn't know what to do with one if it did.)

POINT TO INSTRUCTION

This shorty takes advantage of the fact that all instructions begin at position #7 on any line. Counting from left to right, if the last 4 bits of RA equalling 0 means that RA points to the first character of a line, then if the last 4 bits of RA = 6, RA must point to the 7th byte of that 16-byte section. (If RA.0 = A0, then RA.0 = A6 = the seventh byte counting A0 as the first byte.) This routine strips off the last 4 bits of RA, replacing those bits with a 6. No matter where RA

pointed before, it will point to the 7th byte of the line on return, thus it points to the first character of the instruction field.

ERROR MESSAGE

Before calling this routine, RE.0 is set to the value corresponding to the appropriate error message. (RE.0 = 01 = no symbol in table, for example.) This digit is displayed after first setting R6 to the bottom left area of the display page, with a call to Display Digit. An E is displayed by the same technique (RE.0 = 0e @ 02BB), a long tone sounded, and the program halted at 02C2. The length of the tone may be changed or eliminated by changing the byte at 02C0 to any new value.

(At this point, the typist hiccupped and the next routine got bumped to the last page of the listing. We'll cover it when we get there. But don't forget to include it if you are hand-loading the program.)

CALL AND RETURN ROUTINES

Your 1802 manual covers the use of these routines

well. To call subroutines, you execute a D4 (SEP R4) instruction followed by two bytes equal to the address of the subroutine being called. To return from routines, a single D5 (SEP R5) is executed. R7 (instead of R6 as for Text Editor-21) is used to point to the return address. This change allowed a shorter Tape I/O routine to be written as this uses R6. All routines except these (and the ones in ROM) run in R3 using this method. It provides a means of jumping around memory without regard to page boundaries and given a large enough stack, unlimited nesting capability. Other registers may be used instead of R4, R5 and R7. One disadvantage with the technique is the amount of time required to execute a call and return.

CLEAR MEMORY

Memory from 0FFF down to 0A00 is set to all zeros using RF as the pointer. The process is stopped by the byte at 0329 representing the page in front of the last one to be erased. (09 erases to 0A00, 07 would clear to 0800) This byte may be changed to 0B in order to preserve the symbol table when resetting the computer in between pass 1 and 2. (The reasons for doing so are outlined in section V,C - Special Cases.)

This routine erases the display page at 0F00-0FFF, and also clears the areas holding the symbol table and the eventual memory pages for the object program. All unused bytes of the object program will then be equal to 00's, a useful condition for debugging with the help of a Checksum program (see VIPER, NOV. 1978).

ADDRESS ENTRY

Calls the ROM key scan routine 4 times to input the start address for assembly. This address is shifted into R9 A 4-bit nibble at a time using a double precision shift left by 4 at 0337-0343. The digits are displayed one at a time (a feature which precluded using ROM routines exclusively to get the address). Finally if Key E is pressed, the routine returns. If another key is pressed, the routine repeats with the branch to 032D at 0351.

SIGNAL USER

A simple beeper routine that causes 6 short beeps to signal "done" with whatever the computer was doing. Several beeper routines are undoubtedly possible, but

the rest of the program had my attention and I just whipped this up. It uses both R8.1 & R8.0 as timer and tone generator respectively plus RF.0 and a loop counter of 6. The number of beeps may be changed by inserting a new value at 0355.

SYMBOL TABLE MANAGER

When a label is encountered during the first pass, it is output by this sub to the symbol table @ 0A00-0EFF and a test is made to see if this area of memory has overflowed. If not, the address in R9 is attached to the label and the program continues with RD pointing to the byte following the label (position #6 on the line). The symbol table could be relocated to a larger area of memory allowing more than 73 labels to be used by changing the initial value for RD way back @ 0078 at which point RD is set equal to the first byte of the first page for the table. You will also need to change the byte at 03A2 to reflect the new error condition -- when RD goes past the end of its allotted memory space. Also the Search Symbol Table will have to be adjusted in a similar manner.

Using 3 pages of memory could present a problem, though, as the last symbol won't overflow the table until

the address is added. With 2 or 4 pages for the table, there is always room for the address if there was room for the label, and that is the only time overflow is tested.

You may want to shorten the table, thus gaining one extra page for object code. Change 03A2 to 0B and 0125 to 0B. Also set 007F to 0B. Also change 01C1 to 04 and 01C6 to 0B to enable 4 pages of object code to be written to tape during the second part of output. That's it, you now have a capacity of 4 pages for object code, but you only may use 36 labels maximum. If you are a tidy programmer this should be a highly desirable change. (Important: be sure whatever format you use that the byte following the last table entry is a 00 byte. This is needed when the table is sorted by the Symbol Table Rearranger.)

TAPE READ/WRITE

Again, the subroutine descriptions give good hints on how you may include this routine in your programs. The two-byte start address for either reading or writing is passed to the routine via R7 (which points back to wherever the tape routine was called). At 008F you will see such a call. The two bytes at 0092 and 0093 are the

address of the text buffer into which a six-page segment of source listing will go for assembly. Immediately following this address is a byte C2 at 0094. This byte tells the tape routine to begin a tape read function. If it had been equal to 91 instead of C2, tape write would have been selected. The number of pages to be read or written is specified by the value in RE.0 (This was done to allow the same routine to be used in programs where a user would specify the number of pages to be input or output. That number would then go into RE.0)

The actual tape I/O is handled by the ROM routines at 80C2 or 8091. Except for setting the parameters, this routine serves only to preserve the registers changed by the ROM routines. As the ROM subs return via a D4 instruction, R4 must be used as the program counter before calling with the D3 SEP R3 instruction at 037F. This allows bypassing of the halt in ROM that usually means resetting the computer before continuing. This is not necessary when using this technique.

The greatest advantage of this sub, though it is not exploited by this program, is the ability to record any number of pages (up to 256) starting from anywhere in memory you like. The disadvantage is the amount of room required for housekeeping. If your program will

not use the registers changed, then you may shorten it considerably.

The program counter swaps at 036A and 0380 utilize a technique which is the subject of a Little Loops column in VIPER (perhaps already published by the time you read this.) Though a bit confusing, they permit the entire tape routine to be relocated anywhere in memory. The PC swap routines at 036A-036F and 0380-0385 must not cross page boundaries, but the rest of the sub could at any point. You might want a copy of this routine separate on one of your utility cassettes.

INITIALIZE LINK TABLE

This is a two-part subroutine having two entry points. Entry #1 @ 03B7 first sets the table to 00's. When finally output, then only true addresses will be seen -- no garbage information is allowed in. Entry #2 @ 03C4 skips the erase. This point is called when pass #2 is selected. From 03C5 to the end at 03CD, the byte at 02FF is set to equal FE. This byte will be used to index the table to obtain or insert the next entry.

SET LINK ADDRESS

The link table is indexed by the byte stored at 02FF. The link table grows downwards (as a stack would) in memory from 02FE to 02E0. The byte at 02FF represents the address of the last use of the table and is set into RD which is decremented once to point to the next entry down in the table. Not only is the table created in this manner, but it is also " searched" by indexing RD to the next address needed from the table.

SET LINK TABLE BACK

Following each section of source listing during second passes, the index into the table @ 02FF is decremented by 2 and returned. This allows the sequential use of the table without tying up a dedicated register.

OUTPUT MANAGER

In control following the first press of Key F after assembly is complete, the output manager first calls the two subroutines responsible for outputting the symbol and link tables to the text buffer (which

has already been cleared). The letter F is displayed on the screen, and the beeper sub is called to signal ready. At 03F7-03FB, a loop requires the press of Key F to start taping. Only Key F will begin the process which, once begun, may not be halted. The program is halted at 03FF.

At 03FF is the last byte of the four pages I allotted to Chip-8 Assembler-3. Hitting it this close felt somewhat like making the bull's-eye in darts at the club while your friends look on. You know it was luck, but you intend to pass it off as talent. Why waste the moment.

CREATE LINK TABLE

This routine is out of place in the listing. Its purpose is to transfer the address in R9 to the link table via RE using the index into the table stored at 02FF. This takes place following each completed first pass. The index is altered for the next entry. If the pointer to the table went to 02E0, then the table is full and an error message will be output. One additional section or source listing may be squeezed into the table by changing the byte at 02D5 to DE. I do not recommend this as a permanent change -- use it only

if you have exactly 15 sections to assemble. Should you exceed the 15 sections, the error message will be output as before, but the byte at 02DF will be destroyed, which means the program will crash -- without fail -- on a re-start. You may, of course, reload the program from tape. Please make this change only if absolutely needed.

* * *

You will probably want to read these chapters over several times in order to appreciate the finer details of the operation of Chip-8 Assembler-3. It is, I believe the only programming system of its kind available for the Cosmac VIP and will help you advance and improve your skills in programming. Should you have an opportunity to use a machine language assembler in the future, you will be well acquainted with the basics as they do not operate much differently.

And now that you store comments on tape rather than in the corner mixed in with last week's newspapers, your programs will take on added order and professionalism. Your subroutine library will free you from "housework" to concentrate on new programming concepts. When you design your own custom interpreter, Chip-8 Assembler-3

is there to assemble your programs. Such versatility will be appreciated by both the beginner and the experienced programmer. CHIP-8 Assembler-3 will not become obsolete as your experience grows.

Whatever your tastes in software, I hope you enjoy using CHIP-8 Assembler-3 and will use it hard and often. As always, I wish you the very best of luck with your programming!

REGISTER ASSIGNMENT

R0 - DMA Pointer
R1 - Interrupt Program Counter @ 0217
R2 - Stack Pointer @ 00FF
R3 - Program Counter
R4 - Dedicated Pointer To Call Routine
R5 - Dedicated Pointer To Return Routine
R6 - Display Cursor - otherwise available
R7 - Pointer To Return & Arguments
R8 - R8.0=Tone R8.1=Timer (In Interrupt)
R9 - Address Counter (Pass 1) / Object Listing Pointer (Pass 2)
RA - Pointer to Source Listing / Loop Count In Address Entry Sub
RB - RB.1=Display Address
RC - PC for Keyboard Scan Routine @ 8195 - Dedicated
RD - Symbol Table Pointer / Available for second pass loops, etc.
RE - Utility
RF - Utility

LOCATIONS OF CHIP-8 ASSEMBLER-3 ROUTINES

MAIN PROGRAM

0000-03FF	Initialization / 0024 is First Pass Controller, 007B is second Pass Controller
0400-09FF	"Assembly" Storage of CHIP-8 Program
0A00-0BFF	Symbol Table. Up to 73 symbols allowed / 7 bytes per entry
0C00-0EFF	CHIP-8 Program - Final Form. Up to 3 pages of program.

SUBROUTINES

8195 ROM Keyboard Scan running in RC
8146 ROM Interrupt Routine

0100-0113 Put Instruction in Memory
0114-012E Search Symbol Table
012F-0146 Test String
0147-0155 Insert Address
0156-0187 Symbol Table Rearranger
0189-019D Output Symbol to Text
019E-01B0 Output Address to Text
01B1-01C9 Symbol/Object Code Taping
01CA-01D9 Clear Text Pages
01DA-01ED Output Link Table
01F0-01FF 16-Byte Stack
0200-0216 Display Digit
0217-0241 Interrupt Routine (w/o Rnd # Generation) Entry @ 021A
0242-025E Reserve Memory (First pass)
025F-0265 Carriage Return
0266-026E ASCII to Hex Conversion
026F-0284 Convert ASCII/Hex
0285-0297 Hex to ASCII Conversion
0298-02A5 Convert Hex/ASCII
02A6-02AF Point to Instruction
02B0-02C2 Error Message
02C4-02DF Create Link Table
02E0-02FF Link Table
0300-030F Call Routine Entry @ 0301
0310-031C Return Routine Entry @ 0311
031D-032C Clear Memory
032D-0353 Address Entry
0354-0364 Signal User (Pass 1 or 2 choice) (Pass 2 or End Choice)
0365-0399 Tape Read/Write
039A-03B6 Symbol Table Manager
03B7-03CD Initialize Link Table - 2nd Entry @ 03C4
03CE-03D7 Set Link Address
03D8-03E2 Set Link Table Back
03E3-03FF Output Manager

SUBROUTINE DESCRIPTIONS

Put Instruction in Memory:

Input:
RA points to ASCII Encoded Instruction
R9 points to Free Object Code Slot

Output:
RA points to byte immediately after instruction
R9 is reset - points to the Object Code
Object Code in hex is in memory

Changes:
R6.0 (Loop Counter) RE RA

```
        Calls:
            ASCII to hex conversion

Search Symbol Table:
        Input:
            RA points to label in source listing-(points to same label on retur
        Output:
            RE.0=Match found (though this won't be used)
            RD points to address in Symbol Table/Error #1=No match
        Changes:
            RD as noted-RE for error
        Calls:
            Test String/Error Message

Test String:
        Input:
            RA points to string for comparison (i.e. Label)
            RD points to known string (i.e. Symbol or "link")
        Output:
            RE.0=00 =Equality  RE.0≠00 =Inequality
            RD points to byte following known 5-byte string
            RA reset - points to same label
        Changes:
            RF.0  RE.0  RD
        Calls:
            No sub routines

Insert Address:
        Input:
            R9 points to Object Code needing address - in hex form
            RD points to address in Symbol Tabel
        Output:
            Combines address in table with the Object Code
        Changes:
            RD which will point to 2nd byte of address in table
            R9 which will point to same Object Code slot-is reset not changed
        Calls:
            No sub routines

Symbol Table Rearranger:
        Input:
            None - assembly complete however
        Output:
            RA points to first symbol byte for outputting to text
            Text @ 0400-09FF-Symbol & address list alphabetized by 1st letter
        Changes:
            R9/RA  R9 is set to 0400 & goes down depending # symbols
        Calls:
            Output Symbol to Text
```

```
Output Symbol to Text:
        Input:
            RD points to label in table (R9 initialized by caller)
        Output:
            One line text @ RA - i.e. Shoot:0234
        Changes:
            RF.0/RA (points to next symbol in table)
            R9 (which stores the text lines)
            RF.1/R7 to affect the return address-skipping 7 bytes by caller
        Calls:
            Hex/ASCII conversion (twice)

Symbol/Object Code Taping:
        Input:
            None - assembly complete however
        Output:
            6 pages Symbol Table to tape
            4 second (+) space
            3 pages Object Code (Chip-8 Program) to tape
        Changes:
            RE.0 (# pages)
            R8.1 (Timer)
        Calls:
            Tape Write (twice)

Display Digit:
        Input:
            Hex code for display in RE.0
            Set R6 to destination display address
        Output:
            Displays digit @ R6
        Changes:
            RE;  RF.0;  R6
            R6 left pointing to next position over for another digit
        Calls:
            No sub routines

Reserve Memory (First pass)
        Input:
            RA points to 2-byte ASCII number of bytes for reserving
        Output:
            R9=Next even # address following bytes reserved
        Changes:
            RA;  RE;  RF.0; (RA points to last byte instruction)
        Calls:
            Error #5 - 00 bytes reserved
```

```
Carriage Return:
        Input:
            RA points somewhere on old line
        Output:
            RA points to beginning next line
        Changes:
            RA
        Calls:
            No sub routines

ASCII to Hex Conversion:
        Input:
            ASCII bytes in RE.1 & RE.0 (two codes)
        Output:
            Hex byte equivalent in RF.1
        Changes:
            RE
        Calls:
            Convert ASCII/Hex  (twice)

Convert ASCII/Hex
        Input:
            One ASCII code in RE.1
        Output:
            Hex nibble "ORED" with RF.1 - low 4 bits first shifted left
        Changes:
            RE.1 (if<39)  RF.1 as noted
        Calls:
            No sub routines

Hex to ASCII Conversion:
        Input:
            Hex byte for conversion in RF.1
        Output:
            RE.1 & RE.0 contain the two byte ASCII equivalents
        Changes:
            RE as noted
        Calls:
            Convert Hex/ASCII  (twice)

Convert Hex/ASCII:
        Input:
            RE.0 contains hex digit in form "ON"
        Output:
            ASCII code in RE.0
        Changes:
            RE.0 as noted
        Calls:
            No sub routines
```

```
Point to Instruction:
        Input:
            RA points anywhere on a line
        Output:
            RA points to first byte of instruction (RA.0=N6)
        Changes:
            RA
        Calls:
            No sub routines

Error Message:
        Input:
            Code for error in RE.0/1=no symbol found/2=memory overflow/
                                3=symbol table overflow/4=coding error/
                                5=00 bytes reserved/6=link table overflow
        Output:
            Displays code & E (i.e.  2E)
            Sounds warning tone
            Halts program
        Changes:
            RE.0/R6
        Calls:
            Display digit (twice)

Create Link Table:
        Input:
            None, though R9 is set to linking address of next section
        Output:
            R9 address put in table--link addresses--14 maximum possible
        Changes:
            RF;  RE
        Calls:
            Error #6 - Link Table overflow

Clear Memory:
        Input:
            None
        Output:
            ML's 0A00-0FFF set to 00's
        Changes:
            RF  (which= 09FF on return)
        Calls:
            No sub routines
```

Address Entry:

Input:
Enter 4 digit address on keypad
Press E to continue/other keys cancel & restart

Output:
16 bit address selected in R9

Changes:
RC (8195 on return)/RA.0/R6/RE.0/RF.0/R9 (contains address)

Calls:
Display digit, ROM Key Scan

Signal User (Pass 1 or 2 choice)

Input:
None

Output:
6 short warning beeps

Changes:
RF, R8 (for tone & timer)

Calls:
No sub routines

Tape Read/Write

Input:
Preset RE.0 to # of pages
After the Call address, have ready:
1)Start address - A.1; A.0 - 2 bytes
2)C2=Read/91=Write - indicator byte - 1 byte

Output:
Reads or Writes as many pages to or from memory as indicated

Changes:
R6; R4 (but resets R4)

Calls:
ROM Read/Write via R3

Symbol Table Manager:

Input:
RD points to symbol table @ free location
RA points to label in source listing
R9 = address counter (unchanged!)

Output:
Label & address stored in table - error #3= table overflow

Changes:
RF.0/RA (points to byte <u>before</u> Chip-8 Instruction)
RD (points to next free space in symbol table)
RE.0 <u>only</u> upon error message call

Calls:
Error Message

Initialize Link Table:

Input:
None
Output:
Entry 1 erases & initializes/Entry 2 preserves & initializes
Last byte in table set= FE, (or first available space)
Changes:
RF
Calls:
No sub routines

Set Link Address:

Input:
None
Output:
RD points to next link address in table
Next address marker @ 02FF <u>not</u> changed
Changes:
RD as noted
Calls:
No sub routines

Set Link Table Back:

Input:
None
Output:
Subtracts 2 from last byte link table to reference the <u>next</u> table entry (end)
Changes:
RF
Calls:
No sub routines

INITIALIZATION

```
0000    91    GHI   R1    ;(Last On Card RAM page)
  01    BB    PHI   RB    ;Put in RB.1 = 0F
  02    81    GLO   R1    ;= FF
  03    A2    PLO   R2    ;R2.0 = FF
  04    90    GHI   R0    ;= 00 (R0 is PC here)
  05    B3    PHI   R3    ;R3.1 = 00 (prepare for use as PC)
  06    AD    PLO   RD    ;RD.0 = 00
  07    F8    LDI
  08    0B
  09    A3    PLO   R3    ;R3 = 00
  0A    D3    SEP   R3    ;R3 is program counter
  0B    F8    LDI
```

```
000C    02
  0D    B1    PHI   R1    ;R1.1 = 81
  0E    F8    LDI
  0F    1A
0010    A1    PLO   R1    ;R1 = 021A -interrupt one page routine
  11    F8    LDI
  12    03
  13    B4    PHI   R4    ;R4.1 = 03
  14    B5    PHI   R5    ;R5.1 = 03
  15    F8    LDI
  16    01
  17    B2    PHI   R2    ;R2 = 01FF -stack pointer
  18    A4    PLO   R4    ;R4 = 0301 -Call routine program counter
  19    F8    LDI
  1A    11
  1B    A5    PLO   R5    ;R5 = 0311 -Return routine program counter
  1C    E2    SEX   2     ;X = 2
  1D    69    INP         ;Turn on video
  1E    D4    SEP   R4
  1F    03                ;Call Clear Memory
0020    1D
  21    30                ;Branch to patch @ 0074 to initialize link
  22    74                                                      table
  23    ED    PHI   RD    ;RD = 0A00 -beginning of Symbol Table

                  FIRST PASS CONTROLLER

0024    F8    LDI
  25    81
  26    BC    PHI   RC
  27    F8    LDI         ;Initialize RC = 8195 -Key Scan routine
  28    95
  29    AC    PLO   RC    ;Initialize RC = 8195 -Key Scan routine
  2A    D4    SEP   R4
  2B    03                ;Call Address Entry to set R9
  2C    2D
  2D    D4    SEP   R4    ;Begin Main Loop
  2E    03                ;Call Signal User (to select pass 1 or 2)
  2F    54
0030    DC    SEP   RC    ;Call Key Scan -get instruction
  31    FB    XRI
  32    01                ;Test if Key 1 pressed for first pass
  33    3A    BNZ
  34    7B                ;If not, branch to do second pass @ 007B
  35    F8    LDI
  36    06
  37    AE    PLO   RE    ;RE.0 = # pages to be read in from tape
  38    D4    SEP   R4
  39    03                ;Call Tape Read/Write
```

```
003A  65
 3B   04                ;Start address to input source code
 3C   00                ;  "      "       "    "      "
 3D   C2                ;Indicator byte to select Tape Read
 3E   F8   LDI
 3F   04
0040  BA   PHI  RA      ;RA = 0400 =Beginning Source listing
 41   F8   LDI
 42   00
 43   AA   PLO  RA
 44   0A   LDN  RA      ;Get first character in line
 45   FB   XRI
 46   20                ;Test if = 20 (ASCII space)
 47   32   BZ           ;If so, branch to check for instruction
 48   51                ;(No Label here)
 49   0A   LDN  RA
 4A   FB   XRI          ;Test if = 3B (ASCII ;)
 4B   3B
 4C   32   BZ           ;If so, branch to Carriage Return & Continue
 4D   67                ;(Comment line -- ignore)
 4E   D4   SEP  R4
 4F   03                ;Call Symbol Table Manager
0050  9A
 51   D4   SEP  R4
 52   02                ;Call Point to Instruction
 53   A6
 54   0A   LDN  RA      ;Get first character of instruction
 55   FB   XRI
 56   52                ;= R ?
 57   3A   BNZ          ;If not, branch to test if = 20
 58   60
 59   1A   INC  RA
 5A   1A   INC  RA      ;Point to number bytes following RM
 5B   D4   SEP  R4
 5C   02                ;Call Reserve Memory to increment R9
 5D   42
 5E   30   BN           ;Branch to Carriage Return
 5F   67
0060  0A   LDN  RA      ;Get first character instruction again
 61   FB   XRI
 62   20                ;Test if = 20 (ASCII space)
 63   32   BZ           ;If so, branch to Carriage Return
 64   67                ;(Line is blank -- ignore)
 65   19   INC  R9      ;R9 + 2 for "normal" unlabeled instruction lines
 66   19   INC  R9      ;  "    "       "       "        "        "
 67   D4   SEP  R4
 68   02                ;Call Carriage Return
 69   5F
 6A   9A   GHI  RA
```

```
006B   FB   XRI        ;Test if RA.1 = 0A = Past text page #6
  6C   0A
  6D   3A   BNZ
  6E   44              ;If not, continue
  6F   D4   SEP  R4
0070   02              ;Call Create Link Table Entry
  71   C4
  72   30   BN         ;Then branch to Signal User for next section
  73   2D              ;Or 2nd pass

                        -PATCH-

0074   D4   SEP  R4
  75   03              ;Call Initialize Link Table
  76   B7
  77   F8   LDI        ;Patched instruction - value for RD.1
  78   0A              ;  "          "              "       "
  79   30   BN         ;Branch back from patch @ 0023
  7A   23

                  2ND PASS CONTROLLER

007B   D4   SEP  R4   ;Begin 2nd pass
  7C   03             ;Call Reinitialize Link Table-Entry #2
  7D   C4
  7E   F8   LDI
  7F   0C
0080   B9   PHI  R9   ;Set R9 = First address for creating Object Code
  81   F8   LDI
  82   00
  83   A9   PLO  R9   ;End preinitialization
  84   D4   SEP  R4   ;Begin Main Loop
  85   03             ;Call Signal User (Pass 2 or END)
  86   54
  87   DC   SEP  RC   ;Call Key Scan in ROM
  88   FB   XRI
  89   02
  8A   3A   BNZ        ;If key 2 not pressed, go end
  8B   F8              ;Else continue
  8C   F8   LDI
  8D   06
  8E   AE   PLO  RE   ;RE.0 = # pages to be read
  8F   D4   SEP  R4
0090   03             ;Call Tape Read/Write
  91   65
  92   04             ;Start address to input source code
  93   00             ;  "      "         "    "     "
  94   C2             ;Indicator byte selects Tape Read
  95   F8   LDI
```

```
0096  04
  97  BA    PHI  RA    ;Set RA = first byte Source Listing
  98  F8    LDI
  99  00
  9A  AA    PLO  RA
  9B  0A    LDN  RA    ;Get first character of line;begin continue
  9C  FB    XRI                                        2nd pass
  9D  3B                ;Test if = ";"
  9E  32    BZ          ;If so, branch to Carriage Return @ 00EB
  9F  EB                ;(Comment lines-ignored) do not increment R9-
00A0  D4    SEP  R4       Object Pointer--Do not test for overflow
  A1  02                ;Call Point to Instruction
  A2  A6
  A3  0A    LDN  RA    ;Get first character instruction
  A4  FB    XRI
  A5  20                ;= 20 (ASCII space)?
  A6  32    BZ          ;If so, branch to Carriage Return @ 00EB
  A7  EB                ;(Space lines-ignored)
  A8  0A    LDN  RA
  A9  FB    XRI
  AA  52                ;= "R" for Reserve Memory bytes
  AB  3A    BNZ         ;If not, continue
  AC  B4
  AD  1A    INC  RA
  AE  1A    INC  RA    ;Point to # bytes after the "RM"
  AF  D4    SEP  R4
00B0  02                ;Call Reserve Memory - R9 incremented
  B1  42                ;To reserve bytes needed
  B2  30    BR          ;Branch to Carriage Return
  B3  EB                ;(No instruction to output)
  B4  D4    SEP  R4
  B5  01                ;Call Put Instruction in Memory
  B6  00
  B7  4A    LDA  RA    ;Get byte just after instruction (RA+1)
  B8  FB    XRI
  B9  3B                ;= ";"
  BA  32    BZ          ;If so, branch to set R9 to next Object
  BB  DE                ;Code slot & Carriage Return (Ignore comments)
  BC  0A    LDN  RA    ;Else get first byte of a possible label
  BD  FB    XRI
  BE  20                ;= "space"?
  BF  32    BZ          ;If so, branch to set R9 to next Object
00C0  DE                ;Code slot & Carriage Return (Ignore spaces)
  C1  93    GHI  R3    ;(Get current PC page address (=00))
  C2  BD    PHI  RD    ; →RD.1
  C3  F8    LDI
  C4  CB                ;Load address byte string ("Link") @ 00CB
  C5  AD    PLO  RD    ;RD points to "link" string stored @ 00CB
  C6  D4    SEP  R4
```

```
00C7   01                    ;Call Test String to check for Linking
  C8   2F                    ;Label
  C9   8D    GLO   RD        ;RD.0 points to byte after string
  CA   A3    PLO   R3        ;Putting in R3.0 will cause a skip next N bytes
  CB   4C    "L"
  CC   49    "I"             Data
  CD   4E    "N"
  CE   4B    "K"
  CF   20    " "             ;(space) (Routine test for 5 bytes)
00D0   8E    GLO   RE        ;Test the equality flag (RE.0)
  D1   3A    BNZ             ;Branch if ≠ 00to Search Table-(String not a
  D2   D8                                                      linker)
  D3   D4    SEP   R4
  D4   03                    ;Else call Set Link Address (RD points to address)
  D5   CE                    ;Setting up conditions for inserting address
  D6   30    BN              ;Branch to Insert Address, skipping the label
  D7   DB                    ;Search
  D8   D4    SEP   R4
  D9   01                    ;Call Search Symbol Table
  DA   14                    ;(RD points to address)
  DB   D4    SEP   R4
  DC   01                    ;Call Insert Address (R9 points to same
  DD   47                                          Object Code slot)
  DE   19    INC   R9
  DF   19    INC   R9        ;R9 + 2 =Next Object Code slot
00E0   99    GHI   R9        ;Begin Carriage Return & continue
  E1   FB    XRI             ;Test if R9 went past end of 3-page
  E2   0F                    ;Object Code Memory area
  E3   3A    BNZ             ;If not, continue
  E4   EB
  E5   F8    LDI
  E6   02                    ;Else load error #2-Memory Overflow
  E7   AE    PLO   RE        ;Into RE.0
  E8   D4    SEP   R4
  E9   02                    ;And Call Error Message halting program
  EA   B0
  EB   D4    SEP   R4
  EC   02                    ;Call Carriage Return
  ED   5F
  EE   9A    GHI   RA
  EF   FB    XRI
00F0   0A                    ;Test if RA.1 = 0A = Past page #6
  F1   3A    BNZ
  F2   9B                    ;If not, continue with 2nd pass
  F3   D4    SEP   R4
  F4   03                    ;Else call Set Link Table Back
  F5   D8                    ;To prepare for next section
  F6   30    BN              ;Branch to Signal User-"Ready" for end
  F7   84                    ;Or next section
```

```
00F8    D4    SEP   R4     ;Begin end
  F9    03                 ;Call Initialize Link Table (for
  FA    C4                 ;Outputting to text) 2nd entry
  FB    D4    SEP   R4
  FC    03                 ;Call Output Manager-(Halts program)
  FD    E3
  FE    23    DEC   R3     ;Halt (Just in case of a false return or
  FF    00    Filler       ;Later programming change)

                  PUT INSTRUCTION IN MEMORY

0100    F8    LDI
  01    02
  02    A6    PLO   R6     ;Set R6.0 = Loop Count of 2
  03    4A    LDA   RA     ;Get first of character pair
  04    BE    PHI   RE     ;Put in RE.1 for conversion
  05    4A    LDA   RA     ;Get 2nd of character pair
  06    AE    PLO   RE     ;Put in RE.0 for conversion
  07    D4    SEP   R4
  08    02                 ;Call ASCII to Hex Conversion
  09    66
  0A    9F    GHI   RF     ;Get the converted byte
  0B    59    STR   R9     ;Store as Object Code
  0C    19    INC   R9     ;Increment the pointer
  0D    26    DEC   R6     ;Decrement the Loop Counter
  0E    86    GLO   R6
  0F    3A    BNZ          ;Do the second pair
0110    03
  11    29    DEC   R9
  12    29    DEC   R9     ;Reset R9 to point to Object Code just stored
  13    D5    SEP   R5     ;Return

                  SEARCH SYMBOL TABLE

0114    F8    LDI          ;Begin search new table
  15    0A
  16    BD    PHI   RD     ;Set Rd = address Symbol Table @ 0A00
  17    F8    LDI
  18    00
  19    AD    PLO   RD
  1A    D4    SEP   R4     ;Begin continue search
  1B    01                 ;Call Test String (RA is reset)
  1C    2F
  1D    8E    GLO   RE
  1E    3A    BNZ          ;If RE.0 ≠ 00, continue
  1F    21
0120    D5    SEP   R5     ;Else return (RE.0=00) RD points to address
  21    1D    INC   RD     ;Increment RD to next table entry
  22    1D    INC   RD
```

```
0123   9D   GHI  RD   ;Test if RD.1 went past table end @ 0BFF
 24    FB   XRI
 25    0C
 26    3A   BNZ       ;If not, continue search
 27    1A
 28    F8   LDI
 29    01
 2A    AE   PLO  RE   ;Else load error #1 - no symbol found
 2B    D4   SEP  R4
 2C    02             ;And call Error Message - halt program
 2D    B0
 2E    D5   SEP  R5   ;Return (In the event error routines changed)

                      TEST STRING

012F   E2   SEX  2    ;X = 2
 30    F8   LDI
 31    05
 32    AF   PLO  RF   ;RF.0 = Loop Count of 5
 33    F8   LDI
 34    00
 35    AE   PLO  RE   ;RE.0 = Equality flag (00=) (≠00≠)
 36    4A   LDA  RA   ;Get character for comparison (Label)
 37    52   STR  R2   ;Push
 38    4D   LDA  RD   ;Get character from known string
 39    F3   XOR       ;Compare the two bytes
 3A    32   BZ        ;If equal, continue checking
 3B    3D
 3C    1E   INC  RE   ;Else flag the inequality
 3D    2F   DEC  RF   ;Decrement Loop Count
 3E    8F   GLO  RF
 3F    3A   BNZ       ;Loop for 5 bytes
0140   36
 41    2A   DEC  RA   ;Reset RA to point to same string
 42    2A   DEC  RA   ;  "   "       "       "      "
 43    2A   DEC  RA   ;  "   "       "       "      "
 44    2A   DEC  RA   ;  "   "       "       "      "
 45    2A   DEC  RA   ;  "   "       "       "      "
 46    D5   SEP  R5   ;Return  (RD points to byte after string)

                      INSERT ADDRESS

0147   09   LDN  R9   ;Get high part Chip-8 Instruction
 48    FA   ANI       ;"AND" with "F0" to strip last 4 bits assuring
 49    F0             ;"N-0" form
 4A    E2   SEX  2    ;X = 2
 4B    52   STR  R2   ;Push
 4C    4D   LDA  RD   ;Get high part address in table
 4D    FA   ANI
```

```
014E   0F                ;Strip first 4 bits to assure "0-N" form
  4F   F1   OR            ;"OR" with top of stack to combine byte
0150   59   STR  R9       ;Store back via R9
  51   19   INC  R9       ;R9 + 1 -2nd half Chip-8 Instruction
  52   0D   LDN  RD       ;Get low part address in table
  53   59   STR  R9       ;Store as 2nd half Chip-8 Instruction
  54   29   DEC  R9       ;Reset to point to same Object Code
  55   D5   SEP  R5       ;Return

                  SYMBOL TABLE REARRANGER

0156   D4   SEP  R4
  57   01                ;Call Clear Text
  58   CA
  59   F8   LDI
  5A   04
  5B   BA   PHI  RA
  5C   F8   LDI
  5D   00
  5E   AA   PLO  RA       ;RA = 0400 Text
  5F   F8   LDI
0160   40
  61   52   STR  R2       ;Push ASCII"@" (one less than "A")
  62   F8   LDI
  63   0A
  64   BD   PHI  RD
  65   F8   LDI
  66   00
  67   AD   PLO  RD       ;RD = 0A00 Table
  68   02   LDN  R2
  69   FC   ADI
  6A   01
  6B   52   STR  R2       ;ASCII on stack + 01
  6C   FB   XRI
  6D   5B
  6E   3A   BNZ
  6F   71
0170   D5   SEP  R5       ;Return after ASCII = 5B "["
  71   0D   LDN  RD
  72   32   BZ
  73   62                ;Check next letter after table searched
  74   E2   SEX  2       ;X = 2
  75   F3   XOR          ;Else compare with stack
  76   3A   BNZ
  77   7F                ;Branch to check next in table if ≠
  78   22   DEC  R2       ;Preserve stack
  79   D4   SEP  R4
  7A   01                ;Call Output Symbol to Text
  7B   89
```

```
017C   12    INC   R2    ;Reset stack
  7D   30    BN
  7E   86                ;Skip the increments
  7F   1D    INC   RD    ;
0180   1D    INC   RD    ;
  81   1D    INC   RD    ;
  82   1D    INC   RD    ; Next Entry
  83   1D    INC   RD    ;
  84   1D    INC   RD    ;
  85   1D    INC   RD    ;
  86   30    BN          ;Continue
  87   71

                    OUTPUT SYMBOL TO TEXT

0189   F8    LDI
  8A   05
  8B   AF    PLO   RF    ;RF.0 = Loop of 5 (5 bytes in each label)
  8C   4D    LDA   RD    ;Get a character
  8D   5A    STR   RA    ;Store in text
  8E   1A    INC   RA    ;RA + 1
  8F   2F    DEC   RF    ;Loop - 01
0190   8F    GLO   RF
  91   3A    BNZ         ;Loop till 5 character stored
  92   8C
  93   F8    LDI
  94   3A                ;Load ";"
  95   5A    STR   RA    ;Store in text
  96   1A    INC   RA   ;RA + 1
  97   D4    SEP   R4
  98   01                ;Call Output Address to Text
  99   9E
  9A   D4    SEP   R4
  9B   02                ;Call Carriage Return
  9C   5F
  9D   D5    SEP   R5    ;Return

                    OUTPUT ADDRESS TO TEXT

019E   F8    LDI
  9F   02
01A0   A6    PLO   R6    ;RF.0 = Loop of 02
  A1   4D    LDA   RD    ;Get half address
  A2   BF    PHI   RF
  A3   D4    SEP   R4
  A4   02                ;Call Hex to ASCII Conversion
  A5   85
  A6   9E    GHI   RE    ;Store in text
  A7   5A    STR   RA    ;  "       "
```

```
01A8    1A    INC   RA    ;Store in text
  A9    8E    GLO   RE    ;  "         "
  AA    5A    STR   RA    ;  "         "
  AB    1A    INC   RA    ;  "         "
  AC    26    DEC   R6    ;  "         "
  AD    86    GLO   R6
  AE    3A    BNZ
  AF    A1                ;Loop till done
01B0    D5    SEP   R5    ;Return
```

SYMBOL/OBJECT CODE TAPING

```
01B1    F8    LDI
  B2    06
  B3    AE    PLO   RE    ;RE.0 = # pages to write
  B4    D4    SEP   R4
  B5    03                ;Call Tape Write/(Read)
  B6    65
  B7    04                ;Address of text -- Symbol Table
  B8    00                ;  "         "       "     "
  B9    91                ;Signal Write operation
  BA    F8    LDI
  BB    FF
  BC    B8    PHI   R8    ;Set timer in Interrupt
  BD    98    GHI   R8
  BE    3A    BNZ         ;Wait 4 seconds between tapings
  BF    BD
01C0    F8    LDI
  C1    03
  C2    AE    PLO   RE    ;RE.0 = # pages - Object Program
  C3    D4    SEP   R4
  C4    03                ;Call Tape Write/(Read)
  C5    65
  C6    0C                ;Address of Object Code (Chip-8 Program)
  C7    00                ;  "         "       "     "       "
  C8    91                ;Signal Tape Write operation
  C9    D5    SEP   R5    ;Return
```

CLEAR TEXT PAGES

```
01CA    F8    LDI
  CB    09
  CC    BF    PHI   RF
  CD    F8    LDI
  CE    FF
  CF    AF    PLO   RF    ;RF = Last byte text
01D0    EF    SEX   F     ;X = F
  D1    F8    LDI
  D2    00                ;Set text area to "spaces" @ 0400-09FF
```

```
01D3   73   STXD       ;Set text area to "spaces" @ 0400-09FF
  D4   9F   GHI  RF    ;  "    "    "        "        "      "
  D5   FB   XRI        ;  "    "    "        "        "      "
  D6   03              ;  "    "    "        "        "      "
  D7   3A   BNZ        ;  "    "    "        "        "      "
  D8   D1
  D9   D5   SEP  R5    ;Return

                  OUTPUT LINK TABLE

01DA   D4   SEP  R4
  DB   02             ;Call Carriage Return
  DC   5F
  DD   D4   SEP  R4
  DE   03             ;Call Set Link Address
  DF   CE
01E0   8D   GLO  RD
  E1   FB   XRI        ;Test if RD.0 past top Link Table
  E2   DF              ;(15 addresses printed)
  E3   3A   BNZ
  E4   E6             ;If not, continue
  E5   D5   SEP  R5    ;Else return
  E6   D4   SEP  R4
  E7   01             ;Call Output Address to Text
  E8   9E
  E9   D4   SEP  R4
  EA   03             ;Call Set Link Table Back (for next address)
  EB   D8
  EC   30   BN
  ED   DA             ;Continue
  EE   00   Filler
  EF   00   Filler

                    DISPLAY DIGIT

0200   F8   LDI        ;RE.0 contains hex digit for display
  01   81
  02   BE   PHI  RE    ;RE = 810H where 0H= Hex digit
  03   0E   LDN  RE    ;Load address of bit pattern from ROM table
  04   AE   PLO  RE    ;Put in RE.0 to reference the bits
  05   F8   LDI
  06   05
  07   AF   PLO  RF    ;RF.0 = Loop Counter of 05
  08   4E   LDA  RE    ;Get bit pattern
  09   56   STR  R6    ;Store in display via R6 (set by caller)
  0A   86   GLO  R6
  0B   FC   ADI        ;Add 08 to R6 for next
  0C   08
  0D   A6   PLO  R6    ;Row of bits
```

```
020E   2F    DEC   RF    ;Decrement Loop Count
  0F   8F    GLO   RF
0210   3A    BNZ         ;Loop until done with 5 rows
  11   08
  12   86    GLO   R6
  13   FF    SMI         ;Subtract 2F hex from R6.0 to point to
  14   27
  15   A6    PLO   R6    ;Next byte over for successive displays
  16   D5    SEP   R5    ;Return
```

INTERRUPT ROUTINE

```
0217  7A  42  70  22  78  22  52  C4
021F  E2  F8  00  A0  9B  B0  E2  E2
0227  80  E2  E2  20  A0  E2  20  A0
022F  E2  20  A0  3C  27  98  32  3B
0237  AB  2B  8B  B8  88  32  17  7B
023F  28  30  18
```

RESERVE MEMORY

```
0242   4A    LDA   RA    ;Get first ASCII digit
  43   BE    PHI   RE    ;→RE.1
  44   0A    LDN   RA    ;Get second ASCII digit
  45   AE    PLO   RE    ;→RE.0 to pass to sub
  46   D4    SEP   R4
  47   02                ;Call ASCII to Hex Conversion
  48   66                ;(Answer in RF.1)
  49   9F    GHI   RF    ;Get the converted number
  4A   32    BZ          ;If = 00, branch to error
  4B   58
  4C   AF    PLO   RF    ;Else put in RF.0 for Loop Count
  4D   19    INC   R9    ;R9 + 1
  4E   2F    DEC   RF    ;Loop - 01
  4F   8F    GLO   RF
0250   3A    BNZ         ;Loop, incrementing R9 x number bytes required
  51   4D
  52   89    GLO   R9    ;Test if R9 is even or odd
  53   F6    SHR         ;By shifting right
  54   3B    BNF         ;(DF=0=Even/=1=Odd)
  55   57                ;If even, branch to exit
  56   19    INC   R9    ;Else increment R9 to make even
  57   D5    SEP   R5    ;Return
  58   F8    LDI
  59   05
  5A   AE    PLO   RE    ;Put #5 error message (Reserved 00 bytes)
  5B   D4    SEP   R4
  5C   02                ;Call Error Message - halt program
  5D   B0
  5E   D5    SEP   R5    ;Return (In case of later change in Error Routine)
```

```
                        CARRIAGE RETURN

025F   1A    INC  RA    ;RA-Source pointer- +01
  60   8A    GLO  RA
  61   FA    ANI
  62   0F               ;Strip first 4 bits from RA.0
  63   3A    BNZ
  64   5F               ;Continue to increment till last 4 bits = 00
  65   D5    SEP  R5    ;Then return

                    ASCII TO HEX CONVERSION

0266   D4    SEP  R4
  67   02               ;Call Convert ASCII/Hex
  68   6F               ;(Byte in RE.1 → RF.1 low 4 bits)
  69   8E    GLO  RE
  6A   BE    PHI  RE    ;Get the next byte for conversion
  6B   D4    SEP  R4
  6C   02               ;Call Convert ASCII/Hex
  6D   6F
  6E   D5    SEP  R5    ;Return

                      CONVERT ASCII/HEX

026F   9E    GHI  RE    ;Get ASCII passed by caller in RE.0
  70   FD    SDI        ;Subtract 39-ASCII
  71   39               ;(If negative, ASCII >39)
  72   33    BPZ        ;Branch if positive (ASCII is a number)
  73   78
  74   9E    GHI  RE
  75   FC    ADI        ;Add 09 to ASCII letters
  76   09
  77   BE    PHI  RE    ;Store intermediate result in RE.0
  78   9E    GHI  RE    ;Get ASCII (either +9 or not depending)
  79   FA    ANI
  7A   0F               ;Strip off first 4 bits
  7B   E2    SEX  2     ;X = 2
  7C   52    STR  R2    ;Push for combining with answer
  7D   9F    GHI  RF
  7E   FE    SHL        ;Shift RF.1 (answer) left x 04
  7F   FE    SHL
0280   FE    SHL        ;To make room for next nibble
  81   FE    SHL
  82   F1    OR         ;"OR" with byte on stack to combine
  83   BF    PHI  RF    ;Put in RF.1
  84   D5    SEP  R5    ;Return
```

HEX TO ASCII CONVERSION

```
0285   9F   GHI  RF   ;Get the hex byte
  86   F6   SHR       ;Shift right x 04 for MSD
  87   F6   SHR
  88   F6   SHR
  89   F6   SHR
  8A   AE   PLO  RE   ;Put in RE.0 to pass to sub
  8B   D4   SEP  R4
  8C   02             ;Call Convert Hex/ASCII
  8D   98             ;(Answer in RE.0)
  8E   8E   GLO  RE   ;Put the result in RE.1
  8F   BE   PHI  RE
0290   9F   GHI  RF   ;Get same hex byte
  91   FA   ANI
  92   0F             ;"AND" with 0F for LSD
  93   AE   PLO  RE   ;Put in RE.0 to pass to sub
  94   D4   SEP  R4
  95   02             ;Call Convert Hex/ASCII
  96   98
  97   D5   SEP  R5   ;Return - two byte answer in RE.1 & RE.0
```

CONVERT HEX/ASCII

```
0298   8E   GLO  RE   ;Get digit passed by caller
  99   FD   SDI
  9A   09             ;09 - value (if negative, value > 09)
  9B   33   BPZ
  9C   A1             ;Branch if positive (a number)
  9D   8E   GLO  RE
  9E   FC   ADI       ;Add 07 to all letter values
  9F   07
02A0   AE   PLO  RE
  A1   8E   GLO  RE
  A2   FC   ADI       ;Always add 30 to complete conversion
  A3   30
  A4   AE   PLO  RE   ;Put result in RE.0
  A5   D5   SEP  R5   ;Return
```

POINT TO INSTRUCTION

```
02A6   E2   SEX  2    ;X = 2
  A7   8A   GLO  RA
  A8   FA   ANI       ;Strip last 4 bits from RA.0
  A9   F0
  AA   52   STR  R2   ;Push
  AB   F8   LDI
  AC   06             ;Or stack with 06 creating a byte N6 that
  AD   F1   OR        ;Will point to the instruction field
  AE   AA   PLO  RA   ;Put in RA.0
  AF   D5   SEP  R5   ;Return
```

```
                                ERROR MESSAGE

02B0    9B      GHI   RB
  B1    B6      PHI   R6      ;Set R6 = last row on page
  B2    F8      LDI
  B3    D8                    ;For error display
  B4    A6      PLO   R6
  B5    D4      SEP   R4
  B6    02                    ;Call Display Digit to display
  B7    00                    ;Error number in RE.0
  B8    16      INC   R6      ;R6.+ 1 to create space
  B9    F8      LDI
  BA    0E
  BB    AE      PLO   RE      ;RE.0 = 0E
  BC    D4      SEP   R4
  BD    02                    ;Call Display Digit ("E" for error)
  BE    00
  BF    F8      LDI
02C0    80
  C1    A8      PLO   R8      ;Tone on
  C2    23      DEC   R3      ;Halt program
* See INSERT

                                CALL ROUTINE

0300    D3      SEP   R3      ;Exit/Call sub with R3 as PC
  01    E2      SEX   2       ;X = 2; Entry point
  02    97      GHI   R7
  03    73      STXD          ;Push R7.1  (Save old return address)
  04    87      GLO   R7
  05    73      STXD          ;Push R7.0  (Save old return address)
  06    93      GHI   R3      ;Save the return address
  07    B7      PHI   R7      ;  "          "       "
  08    83      GLO   R3      ;  "          "       "
  09    A7      PLO   R7      ;  "          "       "
  0A    47      LDA   R7      ;Load sub routine address into R3
  0B    B3      PHI   R3      ;  "    "    "          "        " "
  0C    47      LDA   R7      ;  "    "    "          "        " "
  0D    A3      PLO   R3      ;  "    "    "          "        " "
  0E    30      BR            ;Branch to exit
  0F    00

                                RETURN ROUTINE

0310    D3      SEP   R3      ;Exit to main (Calling) routine
  11    97      GHI   R7      ;Entry point
  12    B3      PHI   R3
  13    87      GLO   R7      ;R7→R3 - load the return address
  14    A3      PLO   R3
  15    E2      SEX   2       ;X = 2
  16    60      IRX           ;Point to saved R7
  17    72      LDXA          ;Pop R7.0
```

```
0318   A7   PLO  R7    ;Restore R7.0
 19    F0   LDX        ;Pop R7.1
 1A    B7   PHI  R7    ;Restore R7.1
 1B    30   BN         ;Branch to exit
 1C    10

                       CLEAR MEMORY

031D   F8   LDI
 1E    0F
 1F    BF   PHI  RF    ;Set RF = 0FFF  Last byte to clear
0320   F8   LDI
 21    FF
 22    AF   PLO  RF
 23    EF   SEX  F     ;X = F
 24    F8   LDI
 25    00
 26    73   STXD       ;Store 00 via X (RF) & decrement
 27    9F   GHI  RF
 28    FB   XRI
 29    09
 2A    3A   BNZ        ;Loop until 00's stored to 0A00
 2B    24
 2C    D5   SEP  R5    ;Return

                       ADDRESS ENTRY

032D   F8   LDI
 2E    04
 2F    AA   PLO  RA    ;RA.0 = Loop Counter of 04
0330   9B   GHI  RB
 31    B6   PHI  R6
 32    F8   LDI
 33    00
 34    A6   PLO  R6    ;R6 = First display byte
 35    DC   SEP  RC    ;Call Key Scan - Get one entry-branches here
 36    AE   PLO  RE    ;Put in RE.0 for display
 37    F8   LDI
 38    04
 39    AF   PLO  RF    ;RF.0 = Loop Count of 04
 3A    89   GLO  R9
 3B    FE   SHL        ;Shift R9 left x 4
 3C    A9   PLO  R9
 3D    99   GHI  R9    ;For double precision
 3E    7E   SHLL
 3F    B9   PHI  R9    ;Prepare for next
0340   2F   DEC  RF
 41    8F   GLO  RF    ;Nibble of 4
 42    3A   BNZ
```

```
0343  3A
 44   89    GLO  R9    ;Then "OR" top of stack (Keypress)
 45   F1    OR
 46   A9    PLO  R9    ;And put in R9.0
 47   D4    SEP  R4
 48   02               ;Call Display Digit (In RE.0)
 49   00
 4A   2A    DEC  RA    ;Loop Count -01
 4B   8A    GLO  RA
 4C   3A    BNZ        ;Loop to get successive
 4D   35               ;Keypresses (4) until RA.0 = 00
 4E   DC    SEP  RC    ;Wait for instruction - Call Key Scan
 4F   FB    XRI
0350  0E               ;Test if key E pressed
 51   3A    BNZ        ;If not, go get new address
 52   2D
 53   D5    SEP  R5    ;Else return

                    SIGNAL USER

0354  F8    LDI
 55   06
 56   AF    PLO  RF    ;RF.0 = Loop Count of 6
 57   F8    LDI
 58   01
 59   A8    PLO  R8    ;Sound tone
 5A   F8    LDI
 5B   07
 5C   B8    PHI  R8    ;Start timer
 5D   98    GHI  R8
 5E   3A    BNZ        ;Wait for space between beeps
 5F   5D
0360  2F    DEC  RF
 61   8F    GLO  RF
 62   3A    BNZ        ;Loop for 6 beeps
 63   57
 64   D5    SEP  R5    ;Return

                  TAPE READ/WRITE

0365  E2    SEX  2     ;X = 2
 66   47    LDA  R7    ;Get start address from caller
 67   B6    PHI  R6
 68   47    LDA  R7
 69   A6    PLO  R6    ;Put in R6
```

```
036A   83   GLO  R3   ;Swap PC's  R4=PC & must later be reset
 6B   A4   PLO  R4   ;  "    "      "  "      "      "        "
 6C   24   DEC  R4   ;  "    "      "  "      "      "        "
 6D   93   GHI  R3   ;  "    "      "  "      "      "        "
 6E   B4   PHI  R4   ;  "    "      "  "      "      "        "
 6F   D4   SEP  R4   ;  "    "      "  "      "      "        "
0370   22   DEC  R2   ;To counter increment next (x=2)
 71   61   INP       ;Turn off video (R(x)+1)
 72   F8   LDI
 73   80
 74   B3   PHI  R3   ;R3.1 = 80
 75   47   LDA  R7   ;Get Read=C2/Write=91 byte from caller
 76   A3   PLO  R3   ;Put in R3 to address proper routine
 77   97   GHI  R7
 78   73   STXD       ;Push R7.1
 79   87   GLO  R7
 7A   73   STXD       ;Push R7.0          Save old R7 R9 which will
 7B   99   GHI  R9                        change (RC.0 will also
 7C   73   STXD       ;Push R9.1          change & must be reset
 7D   89   GLO  R9                        to 95 (8195)
 7E   73   STXD       ;Push R9.0
 7F   D3   SEP  R3   ;Call ROM Read/Write
0380   84   GLO  R4   ;Swap PC's back  R3=PC again
 81   A3   PLO  R3   ;  "     "      "      "       "
 82   23   DEC  R3   ;  "     "      "      "       "
 83   94   GHI  R4   ;  "     "      "      "       "
 84   B3   PHI  R3   ;  "     "      "      "       "
 85   D3   SEP  R3   ;  "     "      "      "       "
 86   E2   SEX  2    ;X = 2
 87   60   IRX       ;Point to saved data
 88   72   LDXA      ;Pop R9.0
 89   A9   PLO  R9   ;Restore R9.0
 8A   72   LDXA      ;Pop R9.1
 8B   B9   PHI  R9   ;Restore R9.1
 8C   72   LDXA      ;Pop R7.0
 8D   A7   PLO  R7   ;Restore R7.0
 8E   F0   LDX       ;Pop R7.1
 8F   B7   PHI  R7   ;Restore R7.1
0390   F8   LDI
 91   03
 92   B4   PHI  R4   ;Reset R4 = 0313 - Call routine PC
 93   F8   LDI
 94   01
 95   A4   PLO  R4   ;Reset R4 = 0313 - Call routine PC
 96   F8   LDI
 97   95             ;Reset RC = 8195 - Key Scan Routine
 98   AC   PLO  RC
 99   D5   SEP  R5   ;Return
```

SYMBOL TABLE MANAGER

```
039A  F8  LDI
 9B  05
 9C  AF  PLO  RF   ;RF = Loop Count of 5
 9D  4A  LDA  RA   ;Get one byte of symbol from source listing
 9E  5D  STR  RD   ;Store in Symbol Table
 9F  1D  INC  RD   ;R8 + 1 - next table position
03A0 9D  GHI  RD
 A1  FB  XRI       ;Test if R8.1 went past
 A2  0C            ;End of table @ 0BFF
 A3  32  BZ        ;If so, branch to Error & halt
 A4  B0
 A5  2F  DEC  RF   ;Decrement Loop counter
 A6  8F  GLO  RF
 A7  3A  BNZ       ;Loop until done with symbol
 A8  9D
 A9  99  GHI  R9   ;Place address in R9 after
 AA  5D  STR  RD
 AB  1D  INC  RD   ;5-byte symbol (no need to test
 AC  89  GLO  R9
 AD  5D  STR  RD   ;Overflow -- if there was room for the
 AE  1D  INC  RD   ;Symbol; There is room for the address)
 AF  D5  SEP  R5   ;Return with RD pointing to next position
03B0 F8  LDI
 B1  03            ;Set RE.0 = 03
 B2  AE  PLO  RE   ;For Symbol Table overflow
 B3  D4  SEP  R4
 B4  02            ;Call Error Routine
 B5  B0            ;No return -- Program halted
 B6  D5  SEP  R5   ;Return (only in case of later changes to Error
                                                              Sub)
```

INITIALIZE LINK TABLE

```
03B7  F8  LDI       ;Entry #1 Initialize & erase
 B8  02
 B9  BF  PHI  RF    ;RF = 02E0 - top of Link Table
 BA  F8  LDI
 BB  E0
 BC  AF  PLO  RF
 BD  F8  LDI
 BE  00             ;Store 00 to erase table
 BF  5F  STR  RF
03C0 1F  INC  RF
 C1  8F  GLO  RF
 C2  3A  BNZ
 C3  BD
 C4  F8  LDI        ;Entry #2 - Initialize & preserve
 C5  02
```

```
03C6   BF   PHI  RF
  C7   F8   LDI
  C8   FF
  C9   AF   PLO  RF    ;Store FE; next available address @ 02FF
  CA   F8   LDI
  CB   FE
  CC   5F   STR  RF
  CD   D5   SEP  R5

                    SET LINK ADDRESS

03CE   F8   LDI
  CF   02
  D0   BD   PHI  RD    ;RD.1 = 02 = Page address Link Table
  D1   F8   LDI
  D2   FF
  D3   AD   PLO  RD    ;RD = 02FF = Storage for address next table
  D4   0D   LDN  RD    ;Get address next table entry              entry
  D5   AD   PLO  RD    ;Put in RD.0
  D6   2D   DEC  RD    ;Then decrement to point to the 2-byte link
  D7   D5   SEP  R5    ;Return                                   address

                  SET LINK TABLE BACK

03D8   F8   LDI
  D9   02
  DA   BF   PHI  RF    ;RF = 02FF = Last byte Link Table
  DB   F8   LDI        ;  "    "       "      "    "    "
  DC   FF              ;  "    "       "      "    "    "
  DD   AF   PLO  RF    ;  "    "       "      "    "    "
  DE   0F   LDN  RF
  DF   FF   SMI        ;Subtract two to contents to point to
03E0   02              ;Next table entry end
  E1   5F   STR  RF    ;Then return new value to table
  E2   D5   SEP  R5    ;Return

                    OUTPUT MANAGER

03E3   D4   SEP  R4
  E4   01              ;Call Symbol Table Rearranger
  E5   56
  E6   D4   SEP  R4
  E7   01              ;Call Output Link Table
  E8   DA
  E9   9B   GHI  RB
  EA   B6   PHI  R6
  EB   F8   LDI
  EC   07
  ED   A6   PLO  R6    ;R6 is destination address for Display Digit
```

```
03EE   F8   LDI
  EF   0F
03F0   AE   PLO  RE    ;RE.0 = 0F for display
  F1   D4   SEP  R4
  F2   02              ;Call Display Digit
  F3   00
  F4   D4   SEP  R4
  F5   03              ;Call Signal User to begin taping
  F6   54
  F7   DC   SEP  RC   ;Call Key Scan
  F8   FB   XRI
  F9   0F
  FA   3A   BNZ        ;Continue only on key F instruction to
  FB   F7              ;Start taping
  FC   D4   SEP  R4
  FD   01              ;Call Symbol/Object Code Taping
  FE   B1              ;(2-part!)
  FF   23   DEC  R3    ;End-halt program
```

CREATE LINK TABLE

```
02C4   F8   LDI
  C5   02
  C6   BF   PHI  RF    ;Set RF = Address Link Table page
  C7   BE   PHI  RE    ; "  RE = "       "    "     "
  C8   F8   LDI
  C9   FF
  CA   AF   PLO  RF    ;RF = next slot storage address
  CB   0F   LDN  RF    ;Get address next available space
  CC   AE   PLO  RE    ;Put in RE.0
  CD   EE   SEX  E     ;X = E
  CE   89   GLO  R9
  CF   73   STXD       ;Push R9.0
02D0   99   GHI  R9                Store the link address for
  D1   73   STXD       ;Push R9.1  the next program section
  D2   8E   GLO  RE
  D3   5F   STR  RF    ;Store RE.0→RF (next available space address)
  D4   FB   XRI
  D5   E0              ;Test for overflow (RE.0 = E0)
  D6   32   BZ
  D7   D9              ;Branch to Error on overflow
  D8   D5   SEP  R5    ;Return
  D9   F8   LDI
  DA   06
  DB   AE   PLO  RE
  DC   D4   SEP  R4
  DD   02              ;Call Error #6 - Link Table overflow -
  DE   B0              ;Halt program
  DF   D5   SEP  R5    ;Return (In case of later change in Error Rou'e)
```

KEYING IN A KEYBOARD

For the last two weeks, I've been ruining my eyesight on keyboard ads -- I'm one of those who encircle all the little numbers on the reader's service cards in the backs of magazines. My files are overflowing. When in the States, I've looked at the Cherry, the Risk, the kits and the bargain bin good luck models. Luckily, before writing any checks, I looked in the mailbox. Rick Simpson had thoughtfully sent a prototype of RCA's new keyboard the final version of which is just now available. Back in my Mexican hideaway, among my friends the donkeys and my more distant acquaintances the iguanas and the occasional scorpian, I couldn't wait to plug it in and give it a try. Fantastic! And it only took a few minutes to get it running.

At the promised retail price of under $50, the keyboard is a hands down bargain. There are disadvantages. A shift lock would have been nice. Also, the keys have the same action as the keys on the VIP hexpad. Unlike a more conventional style board, the keys do not travel down when pressed (as do the keys on an electric typewriter for instance). Touch typists will find the action slow. But if you are thinking of writing your next best seller on your VIP you may want to first consider reviving the art

of pen and paper. (Actually I prefer a pencil.) It's no secret that the VIP is not very well suited to text processing. And for the price of a computer that is, you could have quite a few boxes of pencils.

But that doesn't mean we Vipers have to limit ourselves to a life of hex and binary. Text applications for the Cosmac will continue to increase as more utility programs such as Text Editor-21 become available. As for the tedious job of entering ASCII codes using the hexpad, there is no question that the new keyboard is what we've all been waiting for. Here then are the promised modifications for hooking Text Editor-21 up with RCA's new Cosmac keyboard.

HAVE IT YOUR WAY

The following changes to Text Editor-21 will work with any keyboard that uses the EF-4 flag line to indicate that a key has been pressed. Should you already own a keyboard wired to EF-3, either modification below will work equally well if you are careful to change from EF-4 instructions to those that test EF-3 instead. Remember that EF-3 is also shared by the hexpad, and you should avoid conflicts of interest (i.e. touching a keyboard key and a hexpad key at the

same time). If you have the RCA keyboard, simply plug 'er in. It will not interfere with the normal operation of your computer. (The connector goes in the right side of the VIP, by the way.)

Text Editor 21 has two function classifications which will remain as separate groups after you style the program to your keyboard. I will refer to these as:

#1 - One Key Functions -- those that were originally selected by a single hexpad key press.

#2 - Escape Functions -- those that were selected by first pressing Key C on the hexpad, then the key(s) for the desired function.

Modification #1 will allow only the first group, the One Key Functions, to be activated directly from the keyboard. In order to use these One Key Functions, hold the CONTROL key down on the keyboard at the same time as pressing the key for the function you want. Many formulas have been invented for grouping these functions logically on a typewriter-style key arrangement not designed for such things. I'll add mine to the list, but if you want to change which keys do what, please do. It would be presumptuous to assume my choices to be the best -- it's your keyboard, you should be able to

decide how you want it to operate. (I'll tell you how in a minute.)

My system is based on the first letter of the function wanted. First press the CONTROL key down then hold it down while you press a key from the following list to call up these operations. The CONTROL is in parenthesis to remind you to press that key first.

(CONTROL) U - Cursor Up
(CONTROL) D - Cursor Down
(CONTROL) L - Cursor Left
(CONTROL) R - Cursor Right

Scrolling is a little different. First press and hold the CONTROL key down as before, then press:

(CONTROL) Key 0 (zero!) - Scroll up
(CONTROL) Key O (letter!) - Scroll down

These two are easy to remember as the number 0 is above (scroll up) the letter O (scroll down). You will probably find it more convenient to use the special LINE FEED key to scroll up and the BACKSPACE key to move the cursor left. Similarly, the RETURN key performs a carriage return with auto scroll at the

end of a page. You do not have to press the CONTROL key down to use any of the specially-marked keys. To execute LINE FEED for example, simply touch that key.

In order to select the second bank of functions, the Escape Functions, you only have to press the ESCAPE key on the keyboard. A tone will be heard (from the VIP speaker) indicating that the next keypress will select one of the Escape Functions. If you are using Modification #1, the next keypress after the tone is heard must be on the VIP hexpad. The same numbers are used for these functions (Tape Read, Cursor On/Off, etc.) as described in the instructions to Text Editor 21. (See the second half of the function list beginning with (C)-0 to (C)-F.)

The ESCAPE key on the <u>keyboard</u> simply replaces what Key C used to do on the <u>hexpad</u>. To return control to the keyboard without performing a function, press the hexpad's Key 0. Return to the keyboard is automatic after any function has been performed so you may easily continue typing.

To summarize the operation of Modification #1:

1) Hold the CONTROL key down <u>while</u> pressing the indicated key for all One Key type functions.

2) Press the ESCAPE key to select all Escape Functions. Then press the hexpad key corresponding to the function you want. Return to the keyboard is automatic, but you may press Key 0 on the hexpad to return control in the event you pressed the ESCAPE key by mistake.

If you press a wrong key while holding the CONTROL key down, nothing serious can happen. (Remember, the "bad guy" functions in particular the Erase Text Buffer instruction, are over there on the hexpad; you have to press ESCAPE first to get to them.) Many letters, however, will produce the same ASCII codes as others with the CONTROL key held down, and you will find that some of the keyboard keys will do the same jobs. At the end of Modification #2, I have included a list of what keys are connected in this way. (This is further explained there.) One way to customize your keyboard would be to consult the list making note of which keys match others, then use those instead if they are more to your taste. This is not a very flexible method, but if the right key works, why not use it! That way, you won't need to modify the modifications.

If you would like more extensive revisions, you may also change the Jump Table (as explained later) to

switch keys around, and by consulting the program details of Text Editor 21, you may easily change which hexpad key does what also. But mixing up functions between the keyboard and hexpad would require extensive and possibly undesirable revisions of the program.

Modification #2 transfers both the One Key Functions and the Escape Functions to the keyboard -- the hexpad will no longer be used. An exception is Reverse Field Video which was cut to make room for the revisions. However, the reason I chose to delete the Reverse Field function was to allow the use of the same keyboard keys that were originally programmed for the hexpad. If you have become familiar with those numbers and letters, and if you don't care to have Reverse Field Video, this modification should be right for you.

If you prefer to retain Reverse Field Video, however, you only need to figure out what new keys you want to select each of the Escape Functions. (This is explained following the listing for Modification #2.) Or simply stay with Modification #1. The choice is yours! Text Editor 21 will function as well with as without a keyboard.

Whatever you choose to do, the operation of your

keyboard under the control of Modification #2 is nearly identical to the operation under Modification #1. (If you have not read that section, please do before continuing.) To select One Key Functions, hold down the CONTROL key, then press the desired key as explained before.

To select the bank of Escape Functions, first press the ESCAPE key. A tone will be heard indicating, again, that the next key pressed will select an Escape Function. Using the same list of numbers that were originally intended for the hexpad, press the key on your keyboard that corresponds to the number or letter of the function you want. You do <u>not</u> have to hold down the CONTROL key to select an Escape Function after pressing the ESCAPE key. Just press the right key and the function will be performed. Return to normal typing mode is automatic just as it is for Modification #1.

If after pressing the ESCAPE key you change your mind and do not wish to select any function, pressing the ESCAPE key a second time will return control to normal operation. So will pressing the number 0 just as it did on the hexpad under Modification #1. Be careful to press <u>number</u> 0 and not <u>letter</u> O to return to normal typing mode, though. (Realizing the possible

confusion here, I've allowed for this in the modification and pressing either key zero, letter O or ESCAPE will not matter. But if you change things later, remember that these keys may end up selecting different things.)

Why have I included both modifications? Because some of you may have removed the hexpad from the computer or perhaps you have a homebrew 1802 system. Aside from the routines in ROM, Text Editor 21 could be adapted to those systems admittedly with some remodeling. Others won't want the VIP and the keyboard in the same place. Maybe you'll want a remote system with your computer in one room and your keyboard somewhere else.

A good argument for keeping certain functions on the hexpad is to prevent their accidental selection. When you call Erase to End of Page, there's no stopping the process once it has started. Erase Text Buffer is even more deadly. Unless you are very good at remembering which keys do what, Modification #1 will probably be the best choice for you.

Whichever system you choose, the following changes allow you to Have it Your Way with a custom Text Editor <u>and</u> a full keyboard to boot. Happy typing!

FUNCTION #22

The changes to your copy of Text Editor 21 listed below include a new function not programmed for the original version. The DELETE key calls this new function which will delete any character at the cursor position without moving the cursor over one space to the right. (The SPACE BAR will also delete characters, but then the cursor moves over to the next position.)

This Delete function is particularly useful when you want to erase the last character of a line. Normally the cursor would do an automatic Carriage Return after entering a space, but by using the DELETE key, the cursor will stay where it is. Of course you may always change characters by simply positioning the cursor under the character to be changed, then typing the new character.

One problem that arose with the enabling of the DELETE key was that it is affected by the SHIFT key. For that reason, the DELETE key would only work while the keyboard was in the Upper and Lower Case mode. While holding the SHIFT key down, or while the keyboard is set for Upper Case only, the second half of the DELETE key, the underline, becomes active.

As underlining is not a function of Text Editor 21,

I chose to eliminate it altogether. Therefore the DELETE key will now function as if it were a special key unaffected by either the SHIFT key or the keyboard mode selected. Simply position the cursor under the character you want to delete and push the key.

Those of you with some machine language experience will be able to shorten the DELETE key test routine at 00B3-00C4 although it is not necessary to do so. If you make this change, however, the DELETE key will only function when the keyboard is in the Upper and Lower Case mode.

As the DELETE key outputs the ASCII code 7F, you must be certain that the bit pattern corresponding to that code is blank. This is also true for the code for a space -- 20. Remember that the keyboard itself does not create the characters you see. These must be programmed and stored in memory using the Character Designer program in Pips for Vips. Of course Text Editor 21 comes preprogrammed with a full character set.

You may change which keys produce which characters by using the Character Designer to alter the bit patterns for the new characters you want. If you do not know the ASCII code for a certain key you only need to position the cursor at the top left hand corner of the first

text page and press the key you are unsure of. The ASCII code will be stored at 0400 which you may then examine using the ROM system monitor in the normal way.

Since writing Text Editor 21, I have included a divided by sign (÷) and the symbol for not equals (≠). In my character set, you may want to replace a little-used character with an underline and in that way avoid having to make the change suggested above. Another idea would be to replace the entire lower case set with graphics symbols which could then be typed directly from your keyboard.

One caution. The ASCII "characters" 20 and 7F should be the only ones to produce a space on the screen. In all other character locations, even if they are not used, you should record a dot or other symbol (using the Character Designer program) in order to insure that all blank spaces are equal to ASCII spaces with the hex code 20. (The DELETE key -- ASCII 7F -- is programmed to store a 20 hex number in memory not a 7F.) Future programs will make use of this feature and will not operate if anything other than the ASCII 20 is used to create a space on the screen. But don't be afraid to experiment, just be aware of this caution.

One final note. The ASCII codes 00-1F are now recognized by Text Editor-21 as control characters

only, and will not now be printed on the screen. This is a normal set up for many text editors and keyboard combinations. If you have graphics symbols stored in these locations, they should now be transferred to any new location from 21-7E hex using the Character Designer. I had included the symbols for the four playing card suits with the original Text Editor 21, and these will now be inaccessable at their present locations after you make the modifications.

MODIFICATION #1

After loading in Text Editor 21 from the Pips for Vips tape, make the following changes using the VIP system monitor. It is a good idea to keep at least one copy of the original Text Editor 21 program in case you wish to make other modifications or in the unfortunate event your keyboard develops a bug and needs repair. That way your programming will not be put out of commission while you wait to hear from the factory. You can always go back to hexpad typing -- temporarily.

When you are finished with the modifications below, record 12 pages (Key C on the hexpad) marking the new program as Text Editor 21-1. If you decide to

use Modification #2, name it Text Editor 21-2. That way, should the programs be mentioned later in the pages of VIPER or in a future publication, we'll all know which version is being discussed.

MODIFICATION #1 HALF 'N HALF

```
0028   F8   LDI        ;Set RC=8195 = Key Scan subroutine in ROM
  29   81              ;   "      "          "        "
  2A   BC   PHI   RC   ;   "      "          "        "
  2B   F8   LDI        ;   "      "          "        "
  2C   95              ;   "      "          "        "
  2D   AC   PLO   RC   ;   "      "          "        "
  2E   3F   BN4        ;Test EF-4.  Branch to 002B if no key is
  2F   2B              ;being pressed.  Use above 3 lines to
                        debounce signal.
0030   6B   INP        ;Get ASCII code at Input port
  31   B8   PHI   R8   ;Put in R8.1 (Also on stack)
  32   30   BR         ;Branch to 00B3 -- Delete Key Test
  33   B3
  34   F8   LDI        ;Load 1F into D register
  35   1F
  36   F7   SM         ;Subtract D - M(R(X))
  37   3B   BNF        ;If result is negative, R8.1 >1F therefore
  38   3E               branch to 003E skipping next sub
  39   D4   SEP   R4   ;Call Funtion Decode (R8.1≤1F)
  3A   02
  3B   F7
  3C   30   BR         ;Upon return, branch to 0044 skipping
  3D   44               next 2 subroutines
  3E   D4   SEP   R4   ;Call Next Character
  3F   03

0040   9F
  41   D4   SEP   R4   ;Call Cursor Right
  42   02
  43   C6
  44   37   B4         ;Test EF-4 looping here until key
  45   44               is released
  46   30   BR         ;Branch to 0028 for next key press
  47   28

0048-004F  Available for future expansion
```

MODIFICATION #1 DELETE KEY TEST

```
00B3   F8   LDI        ;RC.1 preset = 81 for Taping Routines
  B4   81              ;(needed for Modification #2)
  B5   BC   PHI   RC   ;
  B6   F0   LDX        ;Pop ASCII code from stack
  B7   FB   XRI        ;Test if equal to 7F (Delete)
  B8   7F
  B9   32   BZ         ;If equal, branch to next part
  BA   C0
```

```
00BB   F0    LDX          ;Else Pop same code from stack
  BC   FB    XRI          ;Test if equal to 5F (allowing delete to
  BD   5F                  ;Function in both upper and upper/lower
                            case modes
  BE   3A    BNZ          ;Branch if ≠ to 0034 or to 0030
  BF   34/30               ;Modification #1=34/Modification #2=30

00C0   F8    LDI          ;Load the D register with ASCII space
  C1   20                   code (20)
  C2   B8    PHI    R8    ;Put in R8.1
  C3   D4    SEP    R4    ;Call next character to delete at
  C4   03                  ;    present cursor position (Cursor
  C5   9F                      will not move)
  C6   30    BR           ;Branch to 0044 or to 004♦
  C7   44/4♦               ;Modification #1=44/Modification #2=4♦
```

MODIFICATION #1 JUMP TABLE

```
0200   10    Scroll up
  01   FF    -None-
  02   C6    Cursor right
  03   FF    -None-
  04   5F    Cursor down
  05   7F    Cursor up
  06   FF    -None-
  07   FF    -None-
  08   DE    Backspace
  09   FF    -None-
  0A   10    Line feed
  0B   F0    Escape-select hexpad functions
  0C   DE    Cursor left
  0D   98    Carriage return
  0E   FF    -None-
  0F   44    Scroll back
```

MODIFICATION #1 FUNCTION DECODE

```
02F7   98    GHI    R8   ;Get ASCII code in R8.1
  F8   FA    ANI
  F9   0F                ;Strip first 4 bits
  FA   AE    PLO    RE   ;And place in RE.0 to reference Jump Table
  FB   93    GHI    R3   ;Get page address of table (from the
                            program counter)
  FC   BE    PHI    RE   ;And place in RE.1
  FD   0E    LDN    RE   ;Load byte addressed by RE
  FE   A3    PLO    R3   ;Put in R3.0 to jump to function
  FF   D5    SEP    R5   ;Return for disabled functions
```

(END MODIFICATION #1)

MODIFICATION #2

As explained for Modification #1, load in Text Editor-21 from the Pips for Vips tape, make the following changes, then save the new program as Text Editor 21-2 for full keyboard control

In addition to the following changes, also enter the three listings from Modification #1 titled "Delete Key Test," "Jump Table" and "Function Decode." These three sections are the same for both modifications. (The following section is similar but not the same for both modifications.)

MODIFICATION #2 FULL KEYBOARD CONTROL

```
0028   E2   NOP       ;No operation lines for Signal Debounce
  29   E2   NOP       ;     "        "       "      "
  2A   3F   BN4       ;Test EF-4 - Branch to 0028 if no key
  2B   28                    is being pressed
  2C   6B   INP       ;Get ASCII code at Input port
  2D   B8   PHI   R8  ;Put in R8.1 (Also on stack)
  2E   30   BR        ;Branch to 00B3 -- Delete Key Test
  2F   B3
0030   F8   LDI       ;Load 1F into D register
  31   1F
  32   F7   SM        ;Subtract 1F - M(R(X))
  33   3B   BNF       ;If result is negative, R8.1 is GT 1F
  34   3A                 branch to 003A
  35   D4   SEP   R4  ;Else call Function Decode (R8.1 is LTE 2F)
  36   02
  37   F7
  38   30   BR        ;Branch to 0040 skipping next 2 sub calls
  39   40
  3A   D4   SEP   R4  ;Call next character (ASCII code in R8.1)
  3B   03
  3C   9F
```

```
003D    D4    SEP    R4    ;Call Cursor Right
  3E    02
  3F    C6

0040    37    B4          ;Test EF-4 looping here until key
  41    40                     is released
  42    30    BR          ;Branch to 0028 for next key press
  43    28
```

(PART OF ESCAPE FUNCTION DECODING)

```
0044    E2    NOP         ;No operation lines for Signal Debounce
  45    E2    NOP         ;     "         "       "      "
  46    3F    BN4         ;Test EF-4 for key pressed signal
  47    44
  48    6B    INP         ;Input byte at Input port
  49    FA    ANI         ;Strip first 3 bits
  4A    1F
  4B    AF    PLO    RF   ;Put in RF.0 to reference Jump Table @ 0100
  4C    D4    SEP    R4   ;Call Escape Function Select
  4D    01
  4E    FA
  4F    D5    SEP    R5   ;Return
```

MODIFICATION #2 TAPE READ/WRITE

```
006B    06          ;Data byte to distinguish tape read
                         from write
```

MODIFICATION #2 JUMP TABLE

```
0100    FF
  01    CF    Key A - Erase Text Buffer
  02    E2    Key B - Tape Read
  03    F5    Key C - Cursor homed
  04    77    Key D - Erase to End of Line
  05    98    Key E - Erase to End of Page
  06    E2    Key F - Tape Write
  07    FF    -None-
  08    FF    -None-
```

```
0109    FF    -None-
  0A    FF    -None-
  0B    FF    (Second Escape key press returns to normal mode)
  0C    FF    -None-
  0D    FF    -None-
  0E    FF    -None-
  0F    FF    -None-

0110    FF    Key 0 - Return to normal typing mode
  11    37    Key 1 - Page Back
  12    28    Key 2 - Page Forward
  13    F0    Key 3 - Select Page "N"
  14    E6    Key 4 - Cursor On/Off
  15    FF    Key 5 - No Function (old Reverse Video)
  16    44    Key 6 - Insert Line
  17    6C    Key 7 - Delete Line
  18    FF    Key 8 - None
  19    FF    Key 9 - None
  1A    FF    -None-
  1B    FF    -None-
  1C    FF    -None-
  1D    FF    -None-
  1E    FF    -None-
  1F    FF    -None-
```

MODIFICATION #2 ESCAPE FUNCTION SELECT

```
01FA    93    GHI   R3   ;Get page address of Jump Table
  FB    BF    PHI   RF   ;Put in RF.1
  FC    0F    LDN   RF   ;Load function address from table
  FD    A3    PLO   R3   ;Put in R3.0 to jump to function
  FE    E2    NOP        ;Filler
  FF    D5    SEP   R5   ;Return (for disabled functions)
```

MODIFICATION #2 CALL TO ESCAPE DECODING

```
02F0    D4    SEP   R4   ;Call warning tone to signal
  F1    02                     ESCAPE key pressed
  F2    3B
  F3    D4    SEP   R4   ;Call Escape Decoding routine
  F4    00
  F5    44
  F6    D5    SEP   R5   ;Return
```

MODIFICATION #2 SHOW PAGE "N"

```
03D3   37   B4          ;Test EF-4 looping here until Key 3
  D4   D3                    is released
  D5   E2   NOP         ;No operation lines for Signal Debounce
  D6   E2   NOP
  D7   3F   BN4         ;Test EF-4 looping here until Key is pressed
  D8   D5
  D9   6B   INP         ;Get byte at Input port
  DA   FA   ANI
  DB   07               ;Strip all but last 3 bits
  DC   32   BZ          ;Branch to exit ignoring command
  DD   E7                    to Show Page "0"
  DE   52   STR   R2    ;Push page selected onto stack
  DF   FD   SDI         ;Subtract 6 - M(R(X))

03E0  06
  E1  3B    BM          ;Branch if byte >6
  E2  E7
  E3  F0    LDX         ;Pop byte off stack
  E4   FC   ADI         ;Add 3 to equal proper memory page
  E5   03
  E6   B9   PHI   R9    ;Put in R9.1
  E7   D5   SEP   R5    ;Return
```

(END MODIFICATION #2)

KEY CHART

ASCII codes of all keys with CONTROL key held down*

ASCII		KEYS	ASCII		KEYS
00	@	SPACE BAR	10	0	P
01	A		11	1	Q
02	B		12	2	R
03	C		13	3	S
04	D		14	4	T
05	E		15	5	U
06	F		16	6	V
07	G		17	7	W
08	H	BKSP*	18	8	X
09	I		19	9	Y
0A	J	LINE FEED*	1A	:	Z
0B	K		1B	ESCAPE* { ;	
0C	L		1C	\	
0D	M	RETURN*	1D	}	
0E	N	PERIOD	1E	~	
0F	O	/	1F	DELETE*	

1) MODIFICATION #1 - Read all the way across the table entries to find which keys produce the same results as others. The keys at ASCII 00, for example, are tied to the keys that produce ASCII 10. Sixteen distinct groups are possible.

2) MODIFICATION #2 - Only keys producing the exact same ASCII codes are tied together. Therefore read the columns of the table -- not all the way across. Thirty-two groups are possible.

*(Special keys -- LINE FEED, BACKSPACE, DELETE, etc., do not require the use of the CONTROL key.)

MODIFYING THE MODIFICATIONS

They say programmers never know when to stop programming. Well I say there's no harm in that. Half the fun of computing, perhaps more, is building a library of software that meets your individual needs and tastes. Attempting some of the suggestions here for customizing Text Editor 21 may help you to understand how such a program works, too.

Changing the choice of keys that select the text editing functions is a fairly easy process, but you must exercise some care. Please refer to the Key Chart printed before this section. You will see two columns of ASCII codes and the groups of keys which produce those codes while the CONTROL key is held down at the same time. (Special keys -- RETURN, etc. -- do not require the simultaneous use of the CONTROL key). The codes in the

Key Chart do not necessarily correspond to the actual ASCII codes produced for particular keys. For the purposes of Text Editor 21 and the modifications here, however, the codes are correct and are the ones you need to refer to if you decide to tinker with the program. And by all means, tinker. It's a good way to gain valuable programming experience!

The ASCII codes directly correspond to the addresses of the entries in the Jump Tables calling the text processing functions. If you turn to the Jump Table for Modification #1 for example, you will see that Line Feed is located at 020A. Turning back to the Key Chart, you see that the LINE FEED key produces the ASCII code 0A when pressed. But, so does Key J when the CONTROL key is pressed at the same time. If you are running either modification of Text Editor 21, try pressing CONTROL J. This will cause a line feed to occur and the Key Chart confirms that the Key J in conjunction with the CONTROL key is connected in this way to LINE FEED. They produce the identical ASCII codes. Therefore they could not be programmed to select different functions -- whatever one does, so will the other.

Looking directly across from LINE FEED in the Key Chart, you will see the ASCII code 1A then the symbol for a colon and the letter Z. Try pressing CONTROL Z.

That too produces a LINE FEED.

For all One Key Functions, then, the keys listed horizontally across each listing in the Key Chart will produce the same result. This applies to both MODIFICATIONS #1 and #2. If you decide to change keys around, you are bound to these groups and cannot separate the keys in them.

To make a change, simply look up the ASCII code in the Key Chart of whatever key you wish to enable. Let's say you want Cursor Right to be selected by Key N. Key N produces the ASCII code 0E when the CONTROL key is also held down as indicated in the Key Chart. Turning to the Jump Table for Modification #1, find the present position of Cursor Right. This is at 0202. The byte at that address is C6. Placing C6 at 020E (the 0E is the same as Key N's ASCII code) will enable Key N to move the cursor right as long as the CONTROL key is also held down. That's all you have to do to switch any of the key groups around.

You don't have to change the original byte at 0202. If you leave it to equal C6, then <u>both</u> Keys N <u>and</u> R will move the cursor right. You may enable as many keys to do as many of the same functions by inserting the appropriate bytes in the Jump Table, provided you don't attempt to mix horizontal groups from the Key Chart as I

have said. (And provided you don't run out of room in the Jump Table.)

Should you want to disable any function, replace the byte in the Jump Table with an FF. Following our example, you would change the byte at 0202 to FF. Now Key R will no longer move the cursor right, only Key N has been assigned the job (plus the other keys in its group).

But what is the C6 byte in the first place? It is not an instruction, but the address of the subroutine performing the function. That subroutine will always be on the same page as the Jump Table. Then, as the Jump Table starts at 0200, if you look at location 02C6 you should find -- what do you know -- the subroutine for cursor right!

To remodify the Modification #2 follow the same procedure for all One Key Functions as just described. (Remember the Jump Table for Modification #1 is the same for Modification #2 -- but only for the One Key type functions.)

If you want to change the keys that control the Escape Functions, you must change the larger Jump Table at 0100-011F. This is done in the same manner just described for the smaller Jump Table at 0200-020F using the same Key Chart to figure out the keys.

The difference is that the key groups for all

Escape Functions do not extend all the way across the rows of the Key Chart. While the keys that produce the same ASCII codes are still tied together, there are now 32 groups instead of 16, allowing a greater choice.

For example look at the top row of the Key Chart. To modify the Jump Table at 0100-011F (for Modification #2) the chart tells you that the "@" key and the SPACE BAR both produce the ASCII code 00. If you program a change for the "@" key, the SPACE BAR would perform the same function. (It may still function normally as a space bar. This is only after pressing the ESCAPE key.) Reading across the table to ASCII 10, you see that the letter O and P keys are tied together in the same way. For the Jump Table at 0100, the groups of keys at ASCII 00 are separate from the groups at ASCII 10 directly across the way. For the Modification #1 Jump Table at 0200, these two groups are tied together.

Earlier I said it was possible if you are using Modification #2 to enable the Reverse Field Video feature though you will need to figure out a new set of keys for each function using the Key Chart. If you decide to do this, you must first shorten the Jump Table at 0100-011F the bottom half of which has taken the place of the subroutine that controls the Reverse Field Video function.

Working from a copy of the original Text Editor 21 program supplied with Pips for Vips, enter all the changes from Modification #2 exactly as described except for the Jump Table at 0100-011F.

Next change the byte at 004A to 0F. This causes the groups in the Key Chart to be linked in the same way as for the Jump Table in Modification #1. You now have 16 distinct groups of keys instead of 32 to assign the job of controlling the various Escape Functions.

Using the following Byte Table, insert the correct values for whatever function you wish to enable at the address in the Jump Table corresponding with the ASCII code you have looked up in the Key Chart. For example if you want Home Cursor to be activated by the SPACE BAR (after having first pressed the ESCAPE key) you would look up the SPACE BAR's ASCII code which is 00. Then at 0100 in the Jump Table for Modification #2 you would enter the byte from the Byte Table that goes with the Home Cursor function.

Important: All unused memory spaces in both Jump Tables must be set equal to FF. If this is not done, serious bugs in the performance of Text Editor 21 will be likely to show up.

The bytes for Tape Read and Tape Write are the

same. This is not a printing error. After selecting which keys you want to control the two taping operations, enter the E2 byte in the appropriate positions in the Jump Table just as described. In addition, note the second half of the address in the Jump Table at which you entered the E2 byte for Tape Write. This will be a hex number from 00-0F (00-1F if you are not enabling the Reverse Field Video.)

Enter this same hex number at 006B which is in the Tape Read/Write subroutine. This allows the subroutine to know which function you have selected.

The two "User-Defined" keys on your keyboard (if you have the new RCA keyboard) may also be enabled to control any of Text Editor 21's functions. You may even double up their operation using them to control two One Key Functions, then after pressing the ESCAPE key, two additional Escape Functions. Not having the encoding circuitry or instructions for my prototype board, I am not able to present the details for doing this. It would require the ability to cause these keys to output a known ASCII code. You may add the keys at the proper places in the Key Chart then procede with the process of inserting them in the Jump Tables as previously described. I assume RCA will include instructions on the use of these two keys.

BYTE TABLE

BYTE TO BE ENTERED IN JUMP TABLE @ 0100	FUNCTION THAT WILL BE PERFORMED
CF	Erase Text Buffer
E2	Tape Read
E2	Tape Write
F5	Home Cursor
77	Erase to End of Line
98	Erase to End of Page
37	Page Backward
28	Page Forward
F0	Select Page "N"
E6	Cursor On/Off
44	Insert Line
6C	Delete Line
17*	Reverse Field Video*
FF	Return to Typing Mode (Also all unused memory locations in table)

*** Use this byte only if you are shortening the Jump Table at 0100 to 16 bytes as described in the text above.**

CONCLUSION

The procedure described here -- that of using a Jump Table to select subroutines in memory -- is a famous old friend of the 1802 microprocessor. Your CHIP-8 interpreter uses this technique as one of its basic building blocks. Even if you decide to keep everything as is -- in which case I am flattered by the compliment -- you may want to try some experiments using the suggestions outlined here. In your future programming, a Jump Table may be just the thing to order the selection of a large number of subroutines that would otherwise be unwieldy and difficult to control.

After you've used your keyboard with the Text Editor 21 for awhile, please feel free to let us know if it meets your expectations. I'm sure ARESCO would love to hear from you, and I make every effort to answer mail addressed to me personally. Please enclose a self-addressed, stamped envelope so I can get back to you as soon as possible.

No - for that Index of VIPER articles you've been meaning to store on tape, your new keyboard's ready and waitin'. Have fun - and good luck with your programming.

GET READY FOR EVEN MORE PIPS FOR VIPS

Due to the enthusiastic response to PIPs and PIPs Volume II, Tom Swan has prepared Volume III of PIPs. Just as in Volumes I and II, you'll find the same detailed documentation, the same carefully commented source code, and the same thoughtful explanations of how you can modify the programs to meet your own desires. And all of the information is delivered in the unique and readable style that is the hallmark of all of Tom's work.

Volume III is devoted to just two programs -- two of the best games we have ever seen on the VIP; VIP-FLOP and VIPoker. VIPoker lets you sit in on a game of five-card draw poker with three computerized opponents (Rick, Terry, and Tom!). Each player has a different strategy; one bluffs a lot, one hardly ever, and one now and then. Naturally, you don't know which player has which style (it changes each time you re-run the program); but each will apply their strategy consistently throughout a game. The program follows all the rules of draw poker, and you can examine the program to make sure the computer isn't peeking at your hand!

As for VIP-FLOP, here's what Tom has to say in his introduction:

"Othello*, the box game similar to your Cosmac VIP-FLOP, has been a popular game on computers for several reasons. For one, it is not terribly difficult to program although the method of figuring the computer's moves duplicates chess, checkers and other games, using a look-ahead feature to make up the computer's mind. Also, the complexity of manipulating the pieces during play -- there is a good chance that players will make errors in flopping poker chips on a board -- make this game an excellent choice for a television display. The computer handles all these chores so the board is always "right", something a box game cannot do.

*Registered Trademark of CBS Corporation

VIP-FLOP is supplied with three versions. You can play against the computer or with a friend. The computer will even play itself, a feature that can be used to demonstrate the game to a beginner or as a way to test new evaluation routines that you may write and insert. Nine levels of play are possible and the computer will require from one or two seconds to as long as 15 minutes to figure its next move! At the highest level, the program looks eight moves ahead and is a very tough customer to beat. When you play with your Cosmac, you may let the computer go first; and if you are stumped during any part of the game, you may ask the computer to recommend a move for you!"

So there you have it...two more of Tom's finest. As with the two previous volumes, we offer a pre-publication special, the book and a tape of the programs for only $14.95. After Jan. 15,1980 the price is $19.95 with tape, or $14.95 without the tape. Order now, and sit back and wait for your opponents to arrive.

Please ship me PIPS FOR VIPS Volume III. I enclose $14.95 for manual and tape (price good until January 15, 1980). After January 15, the price is $19.95.

NAME(please print or type)_____

ADDRESS(Street, not P.O. Box)_____

CITY_____STATE_____ZIP_____

Please charge my MC/VISA/BAC/#_____

Master Charge interbank #_____Exp. date_____

Requred credit card signature_____

**No facilities are available for billing. COD delivery to street address only, not to post office box.

ARESCO
P.O. Box 1142
Columbia, MD 21044
THE PAPER - VIPER - RAINBOW - SOURCE