RCA

RCA COSMAC VIP User's Guide

An Introduction to the VIP and CHIP-8 Programming

VIP-320

Suggested Price $5.00

RCA COSMAC VIP
User's Guide

An Introduction to the VIP and CHIP-8 Programming

RCA COSMAC VIP MARKETING
New Holland Avenue
Lancaster, PA 17604

Printed in USA/12-78

Contents

Contents (cont'd)

Section I
Getting Started

Open It Up

"Well," you ask yourself in bewilderment, "Now that I have this machine, what am I going to do with it?" The assortment of cables and books you just unpacked can look formidable at first. But take heart. The fact that you're reading this manual first shows that you have every intention of doing things right, and it won't be long before all the parts and pieces begin to make sense.

Before you begin, however, it might be a good idea to be sure you have all the items you'll need to get the VIP up and running. If you delay in accumulating things, you may find that you have to stop in the middle of something interesting to go and get a vital part.

You'll need:

A. A table, desk, or other firm, flat surface that won't have to be cleared off for dinner, homework, or any of the other necessities of life.

B. The assembled VIP unit you just unpacked, and all the manuals and books that came with it.

C. A cassette recorder with an 8-ohm earphone or an external speaker jack and a microphone input jack.

D. A video monitor. RCA makes a nice little monitor for use with the VIP, and you can get one from your dealer. You can use your family TV, with an appropriate RF modulator, if you like. Most VIP owners prefer to have a separate video monitor so their families can use the TV simultaneously with the VIP's operation.

Setting It Up

First things being first, locate the video cable on the VIP, and screw it on to the "IN" connector on the back of the video monitor. Fig. 1 shows which of the VIP cables is the video cable, as well as showing the rest of the VIP parts. You'll want to refer to this diagram frequently.

If you use a TV receiver instead of a video monitor, you'll need an RF modulator, a device used to adapt VIP's video signal so it can be recognized by an ordinary TV. There are a number of modulator kits available, and you can buy one from your dealer or at almost any computer retail store. You'll have to assemble the modulator yourself, but they come with detailed instructions. If they don't, ask your computer retailer.

The two cassette recorder connectors on the VIP are labeled "IN" and "OUT". Plug "IN" to the "EAR" jack on your recorder, and plug "OUT" to the "MIC" jack. Be sure all the cables are firmly attached, and be sure the toggle switch (the RUN/RESET switch) on the VIP is in the RESET position.

Now you can plug the VIP, the video monitor, and the cassette recorder into a nearby wall outlet. If the VIP whistles at you, push the toggle switch to the RESET position. Turn on the video monitor or TV, and you're ready to begin.

Fig. 1

What Is All This?

Before trying to make your VIP do something interesting, it might make life a lot easier for you if you understand some of the things you're going to see and hear while the VIP is running.

The "POWER" light is glowing red. It should be lit all the time that the VIP is plugged in.

The "Q" light isn't on at the moment because you aren't pressing any keys and the VIP is in RESET mode. Switch to RUN while holding down key "C". Then press another key, and watch the "Q" light. It should be lit while you hear the tone. Both the light and the tone will be on when you press a key, and both should be off when you release the key. The "Q" line is the direct communication line from the heart of the VIP. It will light up every time the microprocessor signals the tape or the tone circuitry. When you press a key, the microprocessor signals the **tone** - and you hear it. When you're saving a program on cassette tape, the microprocessor signals the tape circuitry as well as the tone. The light is an LED (Light-Emitting Diode) which in this instance serves the purpose of an indicator. The LED and tone tell you when the VIP is receiving data from the keyboard or sending data to the tape. The "TAPE" light will be on when you're loading a program from a cassette tape into VIP memory. Otherwise, you can ignore it.

The "TOGGLE" switch is used to RUN programs or to RESET VIP's program counter (the program counter is explained on page 20). When we use the term "reset the VIP", it refers to the action of switching to the RESET position. When we say "initialize the VIP", we mean "hold down key "C" while switching to RUN".

There are four CABLES from the VIP to the outside world.

1. Cassette recorder "IN"
2. Cassette recorder "OUT"
3. Video
4. Power cord, with a power supply box halfway down the length of the cord.

At the back edge of the VIP are two CONNECTORS. The one on the left will hold extra things you can buy to provide your VIP with more memory or to permit VIP to display color graphics, or other options. The connector on the right is for additional communication (INPUT/OUTPUT, or I/O) with the outside world.

The eight little slots on the VIP cover provide ventilation for all the internal components to prevent overheating.

If you remove the nut on the shaft of the toggle switch, the VIP COVER snaps up and off by pressing with your thumbs on the top while lifting the cover at the grooved or indented spaces on the sides. If you remove the cover and peer inside, you'll find a speaker to the left. The other parts are RAM, ROM, the CDP1802 microprocessor itself, capacitors, resistors, and other electronic stuff that makes your VIP the neat little unit it is.

Now initialize the VIP (hold down key "C" while switching the toggle to RUN). You should see the Q-light go on and hear a tone. Check out the video monitor. If you are using a video monitor instead of a TV, there may be a switch on the back of it, marked "75 ohms" in one position and "HI-Z" in the other. Try both positions, if you like, but you'll probably get a better picture if you set the switch to "HI-Z". There should be an array of lit spots on the screen, some of which are flickering. If the brightness or contrast controls are turned all the way down, you may not have any picture at all, so adjust these controls until you're satisfied with the picture. Most VIP users prefer a very black background with very white square spots, but you should set the controls to the brightness and contrast that is easiest on your eyes.

Now press the "0" key four times. Four zeros appear at the bottom of the screen, beginning at the extreme left. These are called "address" digits. The other two digits which appear at the right are called "data" digits, and they represent the "contents" of the memory location whose address is displayed in the address digits. The meaning of these two data digits is explained later, page 12, under **Hex Notation**. 0000 is the first location in memory, and the other two digits are the data residing in location 0000. Next, press key "A".

The "A" option permits you to examine the contents of any memory location in the VIP, one address at a time. You can press any key now, and the address digits will change from 0000 to 0001, and the data digits will show the data in that address. By all means, keep looking at new addresses and their contents until you're tired of it. Then press the toggle switch back to RESET while we look at some of the other VIP options.

Notice the letters MR on the VIP cover under the "A" key. MR stands for MEMORY READ. Under the "0" key, again on the cover, are the letters MW, for MEMORY WRITE. This option is used when you want to enter VIP programs from the keyboard into memory. TR, under the "B" key, is for TAPE READ, and TW under the "F" key is for TAPE WRITE. These options are used for saving and loading programs from a cassette tape.

When you initialize the VIP (i.e., hold down key "C" while switching the toggle to RUN), you must press the next five keys in this sequence, or VIP will refuse to cooperate:

First, four keys for an ADDRESS in memory.

Then, one key for OPTION selection. The available options are MR, MW, TR, and TW. To

select any of the options, you press the key immediately above the abbreviation of the option selected. Press "0" to select MW, "A" to select MR, "B" to select TR, and "F" to select TW.

Try doing it differently. Initialize the VIP and press any four keys. VIP will think they are address selections. Then press any other key besides "A", "0", "B", or "F". VIP screeches, the screen goes blank, and VIP goes off hiding and sulking somewhere. It won't come back, either. You have to reset and then initialize again. Very picky, these computers! Fortunately, VIP is designed to overlook many kinds of errors. You won't lose your program when you reset the VIP, for example. VIP will hold a program in memory until you replace it by loading in another program or until the power is turned off.

Now your VIP is up and running. You may want to fetch a few cassette tapes, so that when you've learned how to enter a program, you can save it on cassette and not have to type in the code a second time.

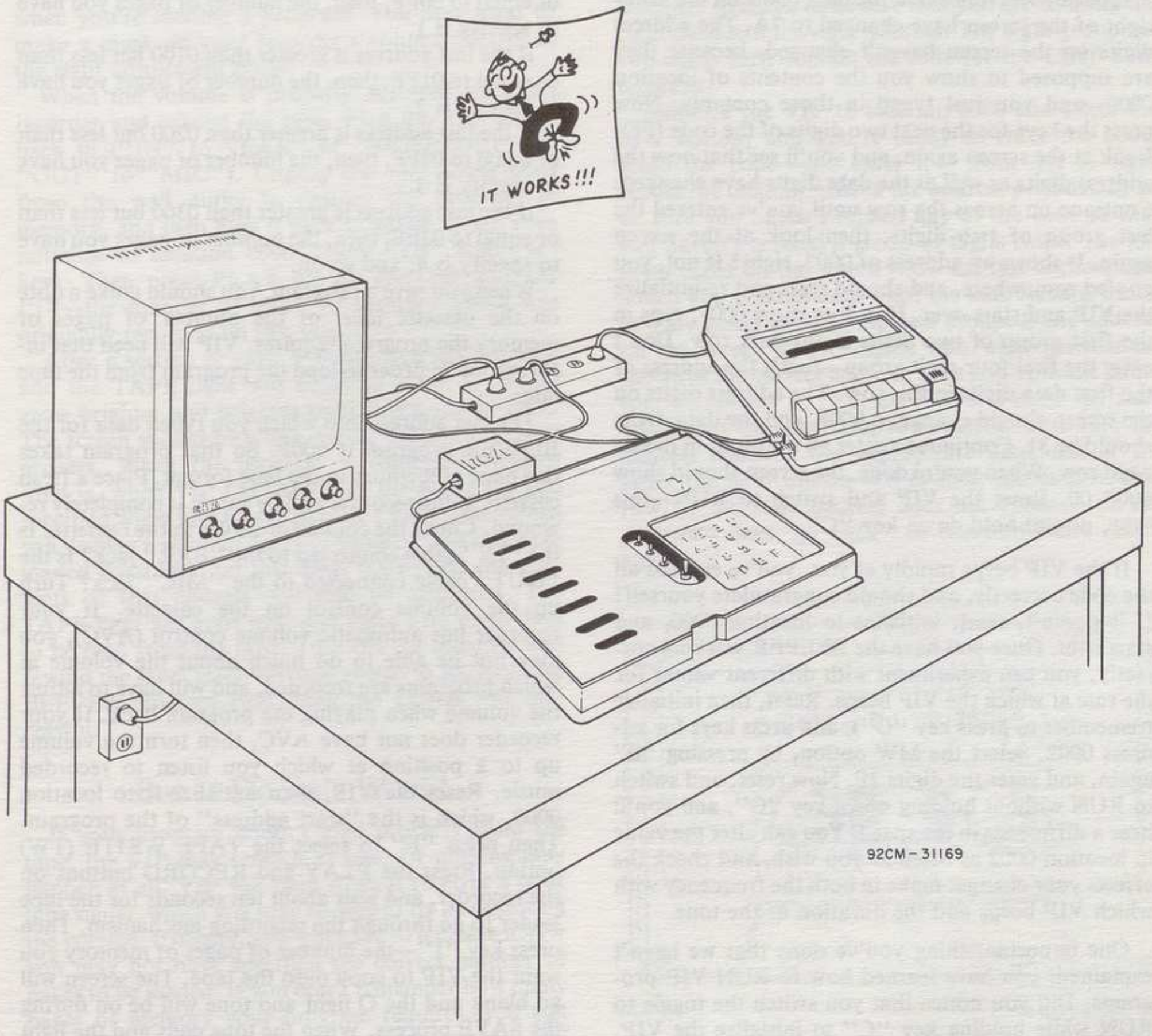


Fig. 2

OKAY - Now What?

Now it's time to load a program. Pick up the **VIP Instruction Manual**, and refer to page 29. At the bottom of the page is the "BEEPER" program. The two rows of digits represent the addresses and "code" you need to store in VIP memory to make the BEEPER work. At the left of each row is a 4-digit address. The address is followed by eight groups of two digits each. These groups contain the program code needed to run the BEEPER. The addresses are shown so you'll be able to check to be sure you have entered all the code in the right places.

Initialize the VIP, enter the first address (0000), and select the MW option (by pressing "0" again). Now you're ready to enter the code. First, press key 7, then key A. Notice that the data digits on the lower right of the screen have changed to 7A. The address digits on the screen haven't changed, because they are supposed to show you the contents of location 0000—and you just typed in those contents. Now press the keys for the next two digits of the code (F8). Look at the screen again, and you'll see that now the address digits as well as the data digits have changed. Continue on across the row until you've entered the last group of two digits, then look at the screen again. It shows an address of 0007, right? If not, you goofed somewhere, and should reset and re-initialize the VIP and start over. If it does show 0007, type in the first group of two digits in the next row. Don't enter the first four-digit group—that's the address of the first data digits in this row. The address digits on the screen should change to 0008, and the data digits should be 31. Continue to enter all the digits in the second row. When you're done, the screen should show 000F 00. Reset the VIP and switch to RUN—this time, do **not** hold down key "C".

If the VIP beeps rapidly at you, you've entered all the code correctly, and should congratulate yourself! If it doesn't, reset, initialize to location 0000, and start over. Once you have the BEEPER working correctly, you can experiment with different values for the rate at which the VIP beeps. Reset, then initialize (remember to press key "C"), and press keys for address 0002. Select the MW option, by pressing "0" again, and enter the digits 2F. Now reset, and switch to RUN without holding down key "C", and you'll hear a difference in the speed. You can alter the value in location 0002 as often as you wish, and check the effects your changes make in both the frequency with which VIP beeps and the duration of the tone.

One important thing you've done that we havn't explained: you have learned how to RUN VIP programs. Did you notice that you switch the toggle to RUN while holding key "C" to initialize the VIP, and that you switch the toggle to RUN without holding key "C" to run a program?

Saving Programs

Once you get the BEEPER running smoothly, you may want to save it on cassette tape. You have to tell the VIP how many addresses of data to copy onto the tape before it starts to save the program.

VIP needs to know how much data to transfer in order to be certain it copies all of your program onto tape. You tell VIP how large your program is by specifying the number of pages of memory the program occupies. You can figure out how many pages of memory are needed by looking at the last address into which you entered program code. (If you're interested, there are 256 addresses on one page of memory.)

If the last address is greater than 0000 but less than or equal to 00FF, then, the number of pages you have to specify is 1.

If the last address is greater than 0100 but less than or equal to 01FF, then, the number of pages you have to specify is 2.

If the last address is greater than 0200 but less than or equal to 02FF, then, the number of pages you have to specify is 3.

If the last address is greater than 0300 but less than or equal to 03FF, then, the number of pages you have to specify is 4; and so on.

When you save a program, you should make a note on the cassette label of the number of pages of memory the program requires. VIP will need that information in order to load the program from the tape later.

The last address into which you typed data for the BEEPER program is 000F. So that program takes one page of memory in the tape format. Place a fresh cassette in the recorder and be sure it is completely rewound. Check the connector cables to the cassette. Is the "IN" cable connected to the "EAR" jack? Is the "OUT" cable connected to the "MIC" jack? Turn up the volume control on the cassette. If your recorder has automatic volume control (AVC), you may not be able to do much about the volume at which programs are recorded, and will have to adjust the volume when playing the program back. If your recorder does not have AVC, then turn the volume up to a position at which you listen to recorded music. Reset the VIP, then initialize it to location 0000, which is the "start address" of the program. Then press "F" to select the TAPE WRITE (TW) option. Press the PLAY and RECORD buttons on the recorder, and wait about ten seconds for the tape leader to go through the recording mechanism. Then press key "1"—the number of pages of memory you want the VIP to copy onto the tape. The screen will go blank and the Q light and tone will be on during the SAVE process. When the tone ends and the light goes out, stop the recorder and rewind the tape. The screen will light up with a random array of spots and

the address 00FF. It doesn't show the actual last address of the BEEPER program; it shows the last address actually transferred to tape. Now LABEL THE TAPE. One of the most useful labels would be "Beeper...1 page".

Loading From Cassette

Having saved the BEEPER program, let's try to load it back into memory from the cassette.

Before we do that, however, remove the VIP cables from the recorder and press PLAY. If you've rewound the tape, you should hear the data. The noise should be unpleasant and raspy and loud. Adjust the volume until the noise is too loud for comfort, but not excruciating. If your recorder has AVC, you'll have to do all the adjustment to the volume when you're loading a program. You may want to make a mark on your recorder's volume control at the right adjustment point.

When the volume is properly adjusted, stop the recorder and rewind the tape. Plug the VIP cables back in ("IN" to "EAR" or "MONITOR" and "OUT" to "MIC"). Unplug the VIP power cord from the wall outlet to destroy the program in memory, and then plug it back in again. Reset, then initialize to location 0000. Press "TR" (the "B" key). Then press PLAY on the recorder and key "1"—the number of pages of memory which was originally recorded on this tape.

After a second or two, VIP will find the program and the TAPE light will begin to glow. It should grow brighter and brighter until it is a steady light. The screen should be blank during the entire LOAD process. When the program is completely loaded (that is, when VIP has loaded all the addresses you specified when you told it to load 1 page), the screen will display a random array of squares. The last page loaded from tape (in this case, 00FF) and the contents of that address will also be displayed. The TAPE light has gone out, and there's no more data to be read from the tape. Turn off the recorder and rewind the tape; reset VIP and switch to RUN. If the BEEPER program runs, then the LOAD was successful.

But It Doesn't Work!

If your program doesn't load properly from the tape, the VIP may screech at you. Or nothing may happen at all. VIP will beep when it receives a poor tape signal, which you may be able to fix by adjusting the volume. (Usually, you will have to increase it.) Or you may not have specified the correct number of pages to be loaded. If you don't remember how many pages of memory your program requires, you can type "F"—VIP will fill its memory, and it will holler, but your program will be loaded.

If nothing at all happens, it is probably because the cassette cables were not connected to the proper jacks on the recorder. Check them and reposition them if necessary, and try again.

Entering Programs

Now you're ready to do some serious program entry. Refer back to the **VIP Instruction Manual**. In Appendix C, on page 35, you'll find the code for the CHIP-8 interpreter. The interpreter is the program that translates the CHIP-8 program instructions into machine code so the instructions can be executed. This program must be entered into VIP memory with every CHIP-8 program—or the program won't work. Notice that there are two columns of figures in the manual. At the left of each column is a group of four digits—the address digits. You use these numbers to check to be sure that you are entering the right data into the right address. The eight groups of two digits each contain the code for the CHIP-8 interpreter.

Initialize the VIP to location 0000 and select the MW option, and you're ready to enter the code. Read the code—and enter it— from left to right across the column. Check the address periodically to be sure you haven't missed any of the code or that you haven't entered a group of digits twice. The address digits on the screen show the last address into which you entered data—not the address into which you are going to enter the next group. If the address on the screen doesn't agree with what the manual shows as the correct address, then you've made an error. To correct it, reset the VIP and enter the start address of the preceding row. Check to be sure the address contains the appropriate data, and if so, select the MW option and re-enter the data in that row. If the start address does not contain the appropriate data, go back one more row and check again.

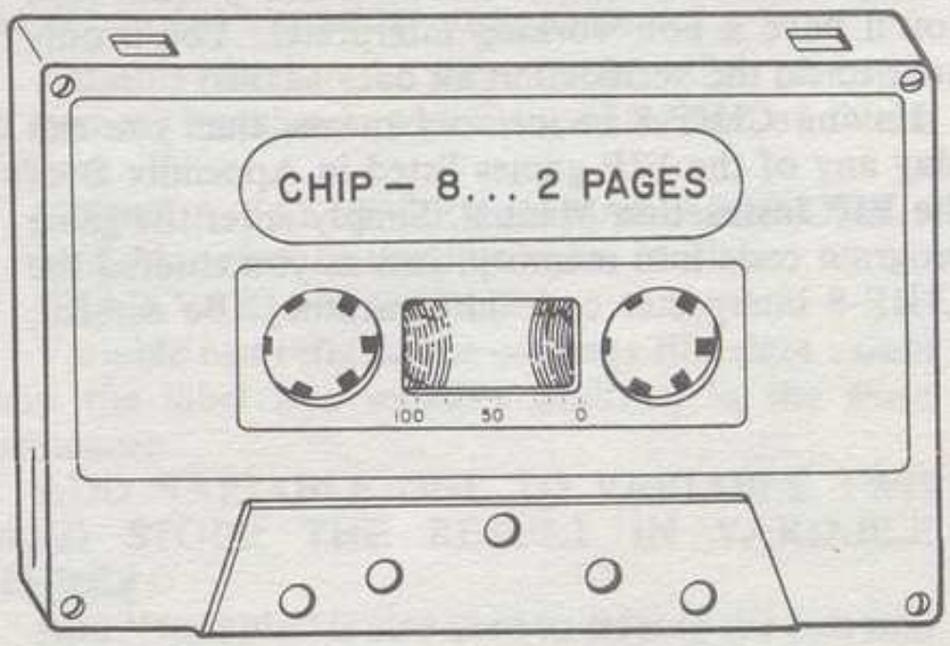


Fig. 3

When you've entered all the code on page 35, SAVE it! Remember to check the last address of the program (it's 01FF). CHIP-8 takes two pages of memory. After the SAVE is complete, label the tape. Be sure to write the number of pages required by the program on the cassette label. See Fig. 3.

Verifying Data Entry

No one—except possibly Superman or Wonder Woman—can always enter data accurately. It is helpful to be able to verify the contents of each location in memory in order to be sure the data you think you put there really is there. Now the MR (MEMORY READ) option will be invaluable.

Reset the VIP and initialize to location 0000. Select the MR option by pressing key "A", and refer back to Appendix C again. One step at a time, check each location in memory to be sure it contains the correct data. Press any key on the keyboard (after selecting the MR option) and the addresses will increment. If there is an error in the code:

1. Write down the address at which the data is incorrect.
2. Write down the correct data from the manual.
3. Reset the VIP, then initialize it to the address you wrote down.
4. Press MW (the "0" key).
5. Type the correct data.
6. Reset again, and initialize to the same address.
7. Press MR (the "A" key)
8. Verify that the correct data is now in memory. If it is, continue to verify the code. If it isn't, begin again at step one.

If you had to make any corrections—even one—save the entire program again on your cassette recorder. If you don't, your corrections will not be on tape and next time you load CHIP-8 from tape you'll have a non-working interpreter. You'll only have to do the verification all over again.

Having CHIP-8 in memory means that you can play any of the VIP games listed in Appendix D of the **VIP Instruction Manual**. Simply enter the game program code into memory, just as you entered the CHIP-8 interpreter code into memory. Be careful, though, to begin each program at location 0200 instead of at 0000. Never enter a game program into locations 0000-01FF. That's where the CHIP-8 interpreter resides. If you enter any game code into that area of memory, the CHIP-8 games won't work. They need the interpreter to translate the CHIP-8 code into machine language so the instructions can be executed.

After entering a game program, save it on cassette tape and then verify it. Save it again if you have to make any corrections. Start the save with address 0000, so you'll have CHIP-8 on each cassette with the program code. This step will save you from having to load two tapes for every game you want to play. Again, be sure to tell the VIP how many pages of memory to copy to tape, and be sure to write the number of pages on the cassette label with the game title. Look at the last address into which you type the game code, and be certain you specify the page that address is in.

By this time, you have learned how to enter programs from the keyboard, how to save programs on a cassette tape, and how to load programs from tape once they are saved. You know how to adjust the volume control on your recorder, and what to do if the program fails to load properly from tape.

Now you can enter each of the VIP programs in the **VIP Instruction Manual** (in Appendix D). You can also enter programs you or other people have written. You can buy programs from people who write and sell them on cassette tape. And, following the instructions given with each of the games, you and your family can spend many pleasant hours playing fascinating computer games with your VIP.

Some Special Warnings

1. Chip-8 programs won't work unless you have loaded the CHIP-8 interpreter into locations 0000 through 01FF.
2. Program bugs (errors) have been known to destroy the interpreter. If you suspect that this has happened, reload the interpreter (the cassette labeled CHIP-8...2 pages).
3. Since CHIP-8 is always loaded into locations 0000 through 01FF, your programs should begin at location 0200. If you try to enter a program into CHIP-8's memory area, your CHIP-8 programs won't work.

Section II
Some Background Information

Want To Do It Yourself?

The game programs already designed for the VIP will keep you and your family occupied for quite a while, but the fun starts when you begin to develop some of your own ideas into games and graphic displays. Somehow, your own work will seem much more interesting and more fun than almost any one else's work.

To write a game program for the VIP, you have to learn either CHIP-8 (VIP's user-oriented language) or CDP1802 machine language, or both, since the VIP doesn't speak English. Fortunately, CHIP-8 isn't very complicated, and once you can speak CHIP-8 fluently, machine language will be easier to learn.

CHIP-8 is an interpretive, hexadecimal language which you can learn fairly quickly to help you communicate with the VIP. "Interpretive" means that the VIP interprets each CHIP-8 instruction when the instruction is encountered in a program. The CHIP-8 instruction may tell VIP which of its many machine language subroutines to execute. **Subroutines** are sections of code which are executed repeatedly by a program. They can be thought of as blocks of instructions which are used in common by several different parts of a program. Some CHIP-8 instructions tell the VIP which values to use in the execution of another instruction. These values are referred to as "data". Other CHIP-8 instructions tell the VIP where in memory the data is stored.

Each CHIP-8 instruction consists of four hexadecimal digits. The first digit is the actual instruction and tells the VIP which of the machine language subroutines to use. The rest of the digits in a CHIP-8 instruction give the VIP all the necessary information about the instruction, such as the values to use or the location where the values are stored.

The VIP uses a **program counter** to keep track of each instruction as it is executed. When a CHIP-8 instruction is encountered, the interpreter calls an appropriate machine language subroutine and increments the program counter to "point" to the next CHIP-8 instruction in the program. For example, the program counter will contain an address like 0200 when a CHIP-8 program is first run. When the interpreter finds the instruction and decodes it, the program counter is incremented to 0202. The machine code does its work, then tells the interpreter the results. Now VIP is ready for the next CHIP-8 instruction and looks at the program counter to find out where that instruction is. The interpreter finds the instruction in location 0202, decodes it, increments the program counter, and waits for the machine language subroutine to finish. And so on, until the program is finished.

When you reset the VIP, you reset the program counter to "point' to the first program instruction in your game. When you switch to RUN, the interpreter looks at the program counter and fetches the instruction, and the game begins. Since CHIP-8 is always loaded into locations 0000 through 01FF, the CHIP-8 program counter is always reset to point to location 0200.

Variables

One of the hardest things for the beginning computerist to unlearn is the narrow English definition of the word "variable". In English, the word means "changeable; able to vary". This definition is only one of the uses to which the word "variable" is applied in the jargon of computer fans. The other ways in which the word variable is used is as a name, as a data value, and as the contents and label in the same sentence. Let's consider some examples.

Variable refers to the name (label) which identifies a **memory location** in which a specific item of data is stored:

STORE A VALUE OF 4 IN VARIABLE ONE.

Variable refers to the data value itself:

READ VARIABLE FOUR.

Variable can refer to the contents (the data value) and the label (the memory address) in the same sentence:

ADD VARIABLE ONE TO VARIABLE TWO AND STORE THE RESULT IN VARIABLE THREE.

And "variable" is also used to refer to the fact that the data has the attribute of changeability and is able to vary:

IF VARIABLE ONE EQUALS 10, THEN PRINT VARIABLE ONE.

Because CHIP-8 uses 16 different variables, each of which has a unique name (well, numbers 0 through F), and because the variables are constantly in use and frequently referred to, the word "variable" is abbreviated and represented by the capital letter V. V0 means "variable zero" and is pronounced "vee zero". VF represents variable F and is pronounced "vee eff".

Let's look at some diagrammatic examples (Fig. 4):

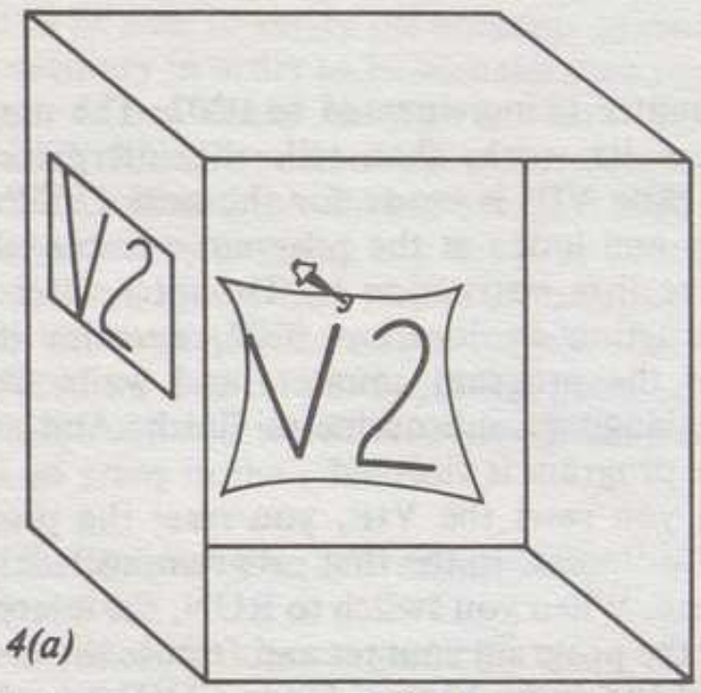


Fig. 4(a)

This box represents a specific location in memory. It can be referred to as a variable, as "STORE A VALUE OF 4 IN V2". The label identifying the location is also referred to as a variable, and is used as such to help you remember where you stored your data.

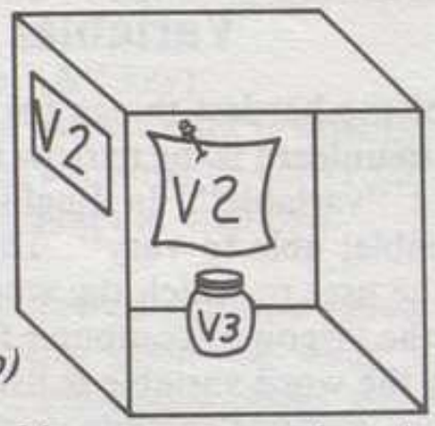


Fig. 4(b)

The data value stored in any memory location is also referred to as a variable, as in "STORE V3 IN V2".

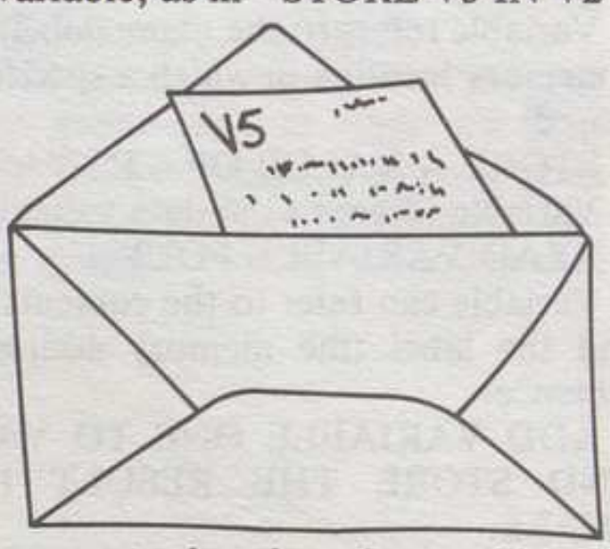


Fig. 4(c)

92CM-31166

This envelope represents the place from which the computer is supposed to get data from the outside world, like from the keyboard. The information contained in the envelope is also referred to as a variable as in "READ V5".

When the computer reads or gets data from the outside world, or transfers data from one location to another in memory, the data value in the new memory location is destroyed, and the new data value takes its place. This ability to change values in a location is also referred to as variable. And the fact that you can add to the value of any data stored, and thereby change the value of the data, is also called "variable". In short, there are many uses to which the word "variable" applies, and if you are temporarily confused, you aren't alone. But as you use variables you'll learn rather quickly which definition applies.

One of the important things to remember is that the CHIP-8 variables can only store positive numbers between 0 and 255. If you try to store negative numbers, the results may look incomprehensible and seem to bear no relationship whatever to the number you thought you were storing. We'll talk about using negative numbers later (on page 45) when we discuss subtraction.

Hex Notation

The VIP keyboard, with its 16 keys, allows you to use hexadecimal notation when programming your game or graphic display. Hex is used in most machine languages because binary is too tedious and clumsy (a number like 15 is written 1111 in binary), machines don't do decimal arithmetic very gracefully, octal doesn't permit an even number of digits, and base 256 is rather unwieldy. Hexadecimal, on the other hand, means "base 16"—numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. If you're planning to learn CHIP-8, it will be helpful to know the relationship between hex numbers and decimal numbers. Fig. 5 has all the hex values from 0 to FF, and all the corresponding decimal values from 0 to 255. However, here's a short list to get started with:

HEX NUMBER	DECIMAL NUMBER
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

You can see in the list that 0 through F requires 16 digits, as does 0 through 15 in decimal. Zero is really

the first number in both counting systems, and it is the first number for a computer. Because VIP uses one "byte" of memory (one location, or address) for each number, FF is the largest value that can be stored in one variable. FF is equivalent to 255 in decimal, and most games and graphic displays will require numbers far smaller than that. With values as low as 255, a lot of people learn to relate hex values to decimal values by simply multiplying the high-order digit (the leftmost digit) by 16, and then adding the value of the low-order digit (the rightmost digit) to the result. For example, if you are looking at the hex number A4, you would multiply A times 16 (A in hex = 10 in decimal) to get 160. Then add 4 to the product to obtain the decimal equivalent of A4: 164. If the hex value were 5B, you would multiply 5 times 16 (80) and add the low-order digit B (11) to get the decimal value 91.

While it isn't necessary to know how to do hex arithmetic in order to use CHIP-8, you should thoroughly understand the relationship between hex and decimal values. To use the conversion table in Fig. 5, follow the right or left hand column down until you find the first hex digit (the high-order digit) in your number. Then follow the row until you come to the intersection of the second hex digit (the low-order digit). The number in the square is the decimal equivalent of the hex value you're looking up. On the other hand, if you know the decimal value and want to convert to hex, find the decimal value in a box in the table and follow the column up to the top to find the rightmost, low-order digit. Then follow the row to the right or left to find the leftmost, high-order digit. Locate decimal 163 in the table; the hex equivalent is A3. And the decimal equivalent of EC is 236. What is the hex value for decimal 79? And the decimal value of hex 2B?

The Display Screen

The VIP has a separate section of memory reserved for the display screen. This area contains all the data which is currently being displayed, and you can alter the display by typing data into it. In the standard VIP, the display area of memory begins at location 0700. Initialize VIP to this address, and type FF after selecting the MW option. A bar should appear on the screen. Type any hex value into this location and watch as the data appears on the screen, even when a program is not running. If you type 00 into location 0700 after typing in FF, the bar will disappear.

The VIP can display all sorts of graphics on the screen because it sees the screen as a grid 64 squares across and 32 squares down. Each square can be lit and erased to give the illusion that something is moving across the screen. VIP can "remember" which square is lit because it has a separate section of its memory devoted to keeping track of the display. When a square is lit, the corresponding address in this separate section of memory contains a "1"; and when the square is not lit, the corresponding address contains a "0". In this manual, when we wish to show a lit square, we'll use shading (since the paper is

Second Hex Digit

First Hex Digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Fig. 5

white). When the shaded spot is displayed on the screen, however, it will show up as a white spot (since the screen is black).

The CHIP-8 language specifies all the screen positions in an X and Y coordinate system, where X represents the horizontal (from left to right) coordinate and Y represents the vertical (top to bottom) coordinate. Since CHIP-8 uses hex instead of decimal values, the rightmost X coordinate is 3F (decimal 63) and the lowest Y coordinate is 1F (decimal 31). Any two variables can be used to store the coordinates, but V1 is usually reserved for the X value and V2 is used for storing the Y value.

Fig. 6 shows the X and Y coordinates of the four corners of the screen. Notice the notation (00,01) and (00,1F). The first value in the parentheses is always the X, or horizontal value, and the second number is the Y coordinate (vertical).

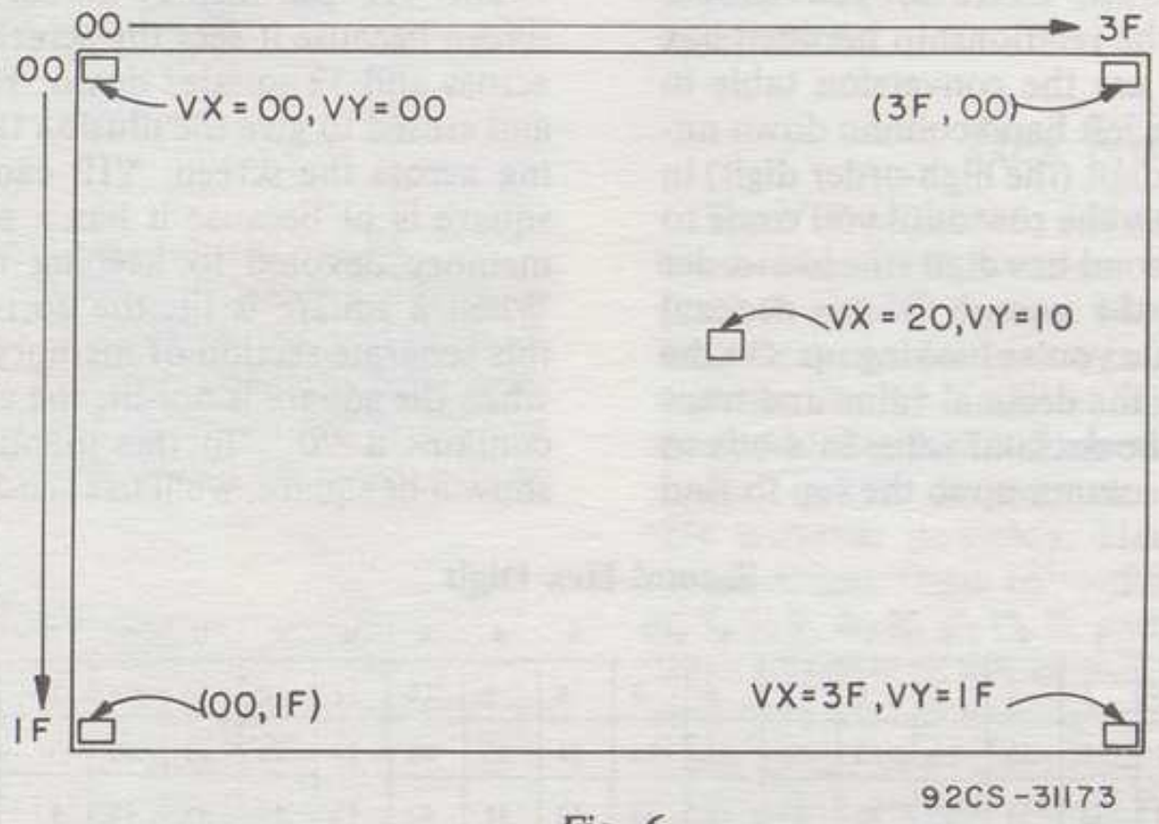


Fig. 6

Bits, Bytes, and Nybbles

Fig. 7 represents a "byte". To a computer, a "byte" is as significant as a "word" is to a people. Each byte is made up of eight bits. The word "bit" is an acronym for BInary digiT. Binary means "two", and in the binary arithmetic system, there are only two digits—zero and one. The bits are turned on or off electronically, and that's a hard thing to tell a computer about. It's made much easier by the fact that VIP understands that if a bit contains a "1", it is "ON", and if a bit contains a "0", it is "OFF". The

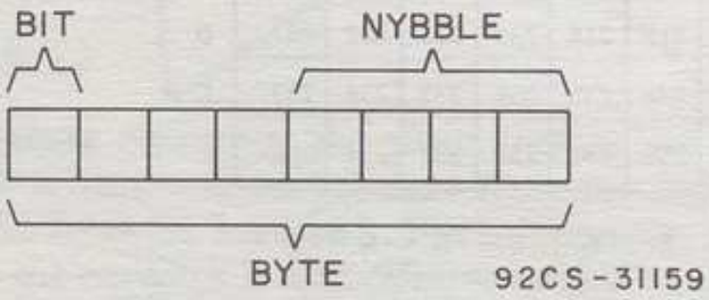


Fig. 7

VIP, like other computers, uses a pattern of off/on bits to represent data or instructions. All the graphics displayed on your TV screen (when it is connected to your VIP) or on your video monitor, are created when CHIP-8 tells the VIP operating system which bits are on and which are off, at any location on the screen. The bits are separated into two groups of four bits each, called "nybbles", and it takes two nybbles to make a byte.

Each bit in a nybble is numbered, from **right to left**, in a manner designed to permit 16 unique combinations of "off" and "on". Each combination corresponds to one hexadecimal digit, 0 through F. Fig. 8 shows all the combinations, and the hex value represented. Look closely at the shaded squares, which represent bits "on", and see how the sum of the "bit numbers" at the top of each column correspond to the hex value alongside the figure.

This may well be the last time you ever see a single nybble. Most computers, including the VIP, require that data used to generate instructions or graphic displays be at least one byte wide. If you design a pattern that requires only half-a-byte (a nybble), you must tell the computer that the other half of the byte is to be turned "off".

In each bit of the hex pattern, there is either a one or a zero. The bit is either 'on" (contains a 1) or it isn't (and contains a 0). It can't, for example, be half-on and half-off. When you enter a hex value into VIP memory, the VIP translates that hex value into its binary equivalent, in order to determine which bits in the pattern are to be turned on and which bits are to be turned off. If a bit in your pattern is to be turned on (we shade that bit in this manual, although it will appear as a lit spot on the screen), add the "bit number" (at the top of the columns in Fig. 8) to the values of the bit numbers for all the other "on"

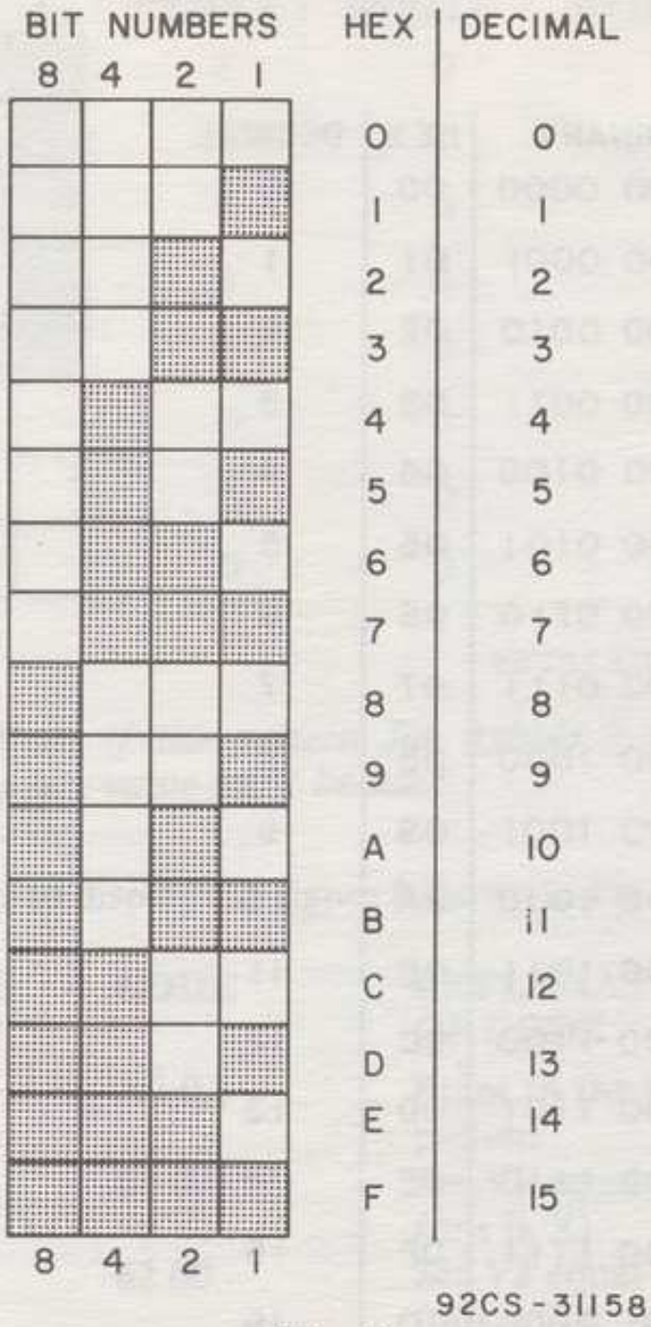


Fig. 8

(shaded) bits, to obtain the hex value. If, for example, your pattern is a bar, 4 bits wide, your pattern might look like Fig. 9.The value of nybble 1 is 3 (2+1=3) and the value of nybble 2 is C (8+4=C—12 in decimal). You don't add the values of the two nybbles together to get the byte value 3C.

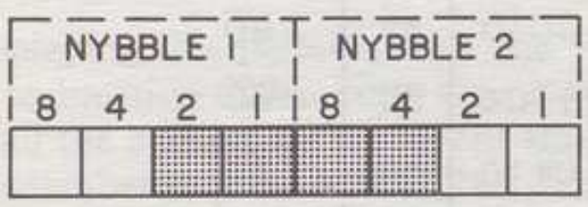


Fig. 9

The value of nybble 1 is referred to as the "high-order" digit of the byte, and the value of nybble 2 is referred to as the "low-order" digit. In this example, the high-order digit is 3, the low order digit is C, and the value of the byte is 3C.

You won't be able to "see" the binary equivalent of 3C when the VIP translates it into binary. When the value is stored in binary, it looks like a string of ones and zeros, each digit corresponding to a shaded or unshaded (on or off) bit in your pattern. The binary value can be "seen" to be equivalent to the hex value when you compare the representation of your bar pattern in Fig. 10 with the previous representation in Fig. 9.

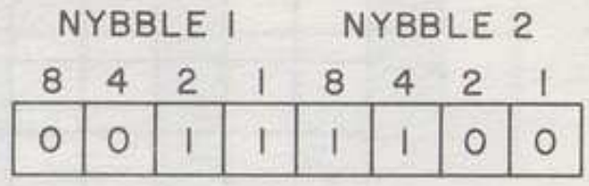


Fig. 10

Since it takes less space to write 0011 1100 than it takes to draw shaded boxes, this manual will usually use the binary "bits" to show the hex pattern. You can see an example of this usage now, if you turn to page 45 and look at the sentence beginning "V4 contains 0000 1001 (hex 09)". When you reach that section of the manual, you may want to refer back to this page if you need help remembering the relationship between hex and binary; but it won't be long before you recognize the "1"s as "shaded" spots and the "0"s as "unshaded" spots. Again, remember that when your pattern is displayed on the screen, it will appear as white-on-black, not as black-on-white the way it appears in the manual.

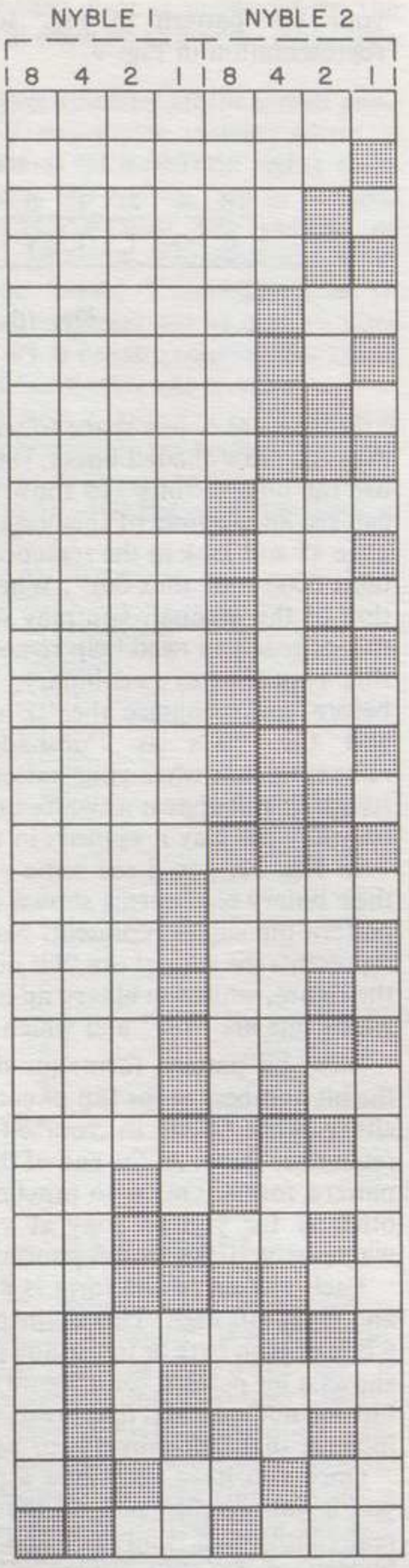
In Fig. 11, you'll see some of the hex values and their binary equivalents shown next to the hex or bit pattern the values represent. Note that the **hex** value represents the sum of the "bit numbers" at the top of the figure, while the **binary** numbers actually tell you which bits are "on" and which bits are "off".

Most bit pattern forms (or drawings) don't have the bit numbers at the top of each column. You must either write them in yourself or (in due time!) remember them. At the end of this section are two bit pattern forms. One is to practice with now, and the other is for you to copy at your local library or wherever you can find a photocopy machine.

Each section of the form is 8 bits wide (one byte) and eight bits high. This shape permits you to use all 8 bits of each byte or to use any part of a byte. Fig. 12 shows a bit pattern for a figure 8. Notice that all the bits are not used and that the hex values for each byte indicate that half-a-byte is to be turned off.

Once you have generated a design of your own, you'll want to display it on the screen to see what it really looks like. You may find it useful to store the bit patterns for your display in VIP's screen memory (beginning at location 0700) before writing a program to display the patterns. You'll be able to see the results of your efforts immediately, which will make it easier for you to make any corrections which may be needed.

We will assume now that you have the VIP connected to a screen of some sort, that you know how to power up the VIP, how to reset it so you can write to memory, and how to enter data from the



BINARY	HEX	DECIMAL
0000 0000	00	0
0000 0001	01	1
0000 0010	02	2
0000 0011	03	3
0000 0100	04	4
0000 0101	05	5
0000 0110	06	6
0000 0111	07	7
0000 1000	08	8
0000 1001	09	9
0000 1010	0A	10
0000 1011	0B	11
0000 1100	0C	12
0000 1101	0D	13
0000 1110	0E	14
0000 1111	0F	15
0001 0000	10	16
0001 0001	11	17
0001 0010	12	18
0001 0011	13	19
0001 0100	14	20
0001 1110	1E	30
0010 1000	28	40
0011 0010	32	50
0011 1100	3C	60
0100 0110	46	70
0101 0000	50	80
0101 1010	5A	90
0110 0100	64	100
1100 1000	C8	200

92CM-31165

Fig. 11

keyboard. We also assume you're willing to take the necessary CHIP-8 code on faith, just to get your pattern on the screen!

Let's suppose we want to show a pattern in the upper left hand corner of the screen, where X=0 and Y=0. We'll assign X to variable 1 (V1) and Y to variable 2 (V2). Then we'll store the hex values for the pattern beginning in locations 020A. Remember that this is a CHIP-8 program, and it won't work unless you first load the CHIP-8 interpreter into locations 0000 through 01FF.

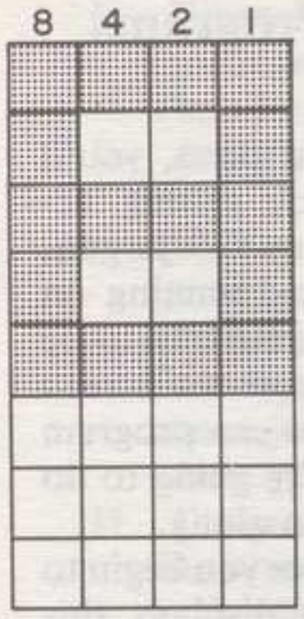


Fig. 12—Even if the pattern for nybble 2 is not shown, the hex value must be used.

The code to display the figure 8 shown in Fig. 12 is:

ADDRESS	CODE	EXPLANATION OF CODE
0200	A2 0A	Point to the bit pattern
0202	61 00	Set V1 equal to 0 (this is X)
0204	62 00	Set V2 equal to 0 (this is Y)
0206	D1 25	Display the pattern on the screen
0208	12 08	Stop here
020A	F0 90	The pattern for the figure 8
020C	F0 90	begins in location 020A and
020E	F0 00	each row of code is for two bytes

After displaying this pattern, reset the VIP, then initialize it to location 020A. Now type in any digits you like until the address on the screen shows 020F. (Did you remember to press MW?) Run the program again, and see how the display is affected by the data you entered. Suppose you wanted to display a letter H. The code for the bit pattern for H is 90 90 F0 90 90 00 00 00. Initialize the VIP to location 020A, enter the code for "H", and then run the program again.

Just for the practice, here are two hex pattern forms in Fig. 13. One contains the pattern, and you can figure out the hex values to enter as code to display the pattern. The other pattern is blank, but the hex values are given. Try to draw the pattern before displaying it on the screen.

You can use the same program code you used to display the pattern for the letter H to display these patterns.

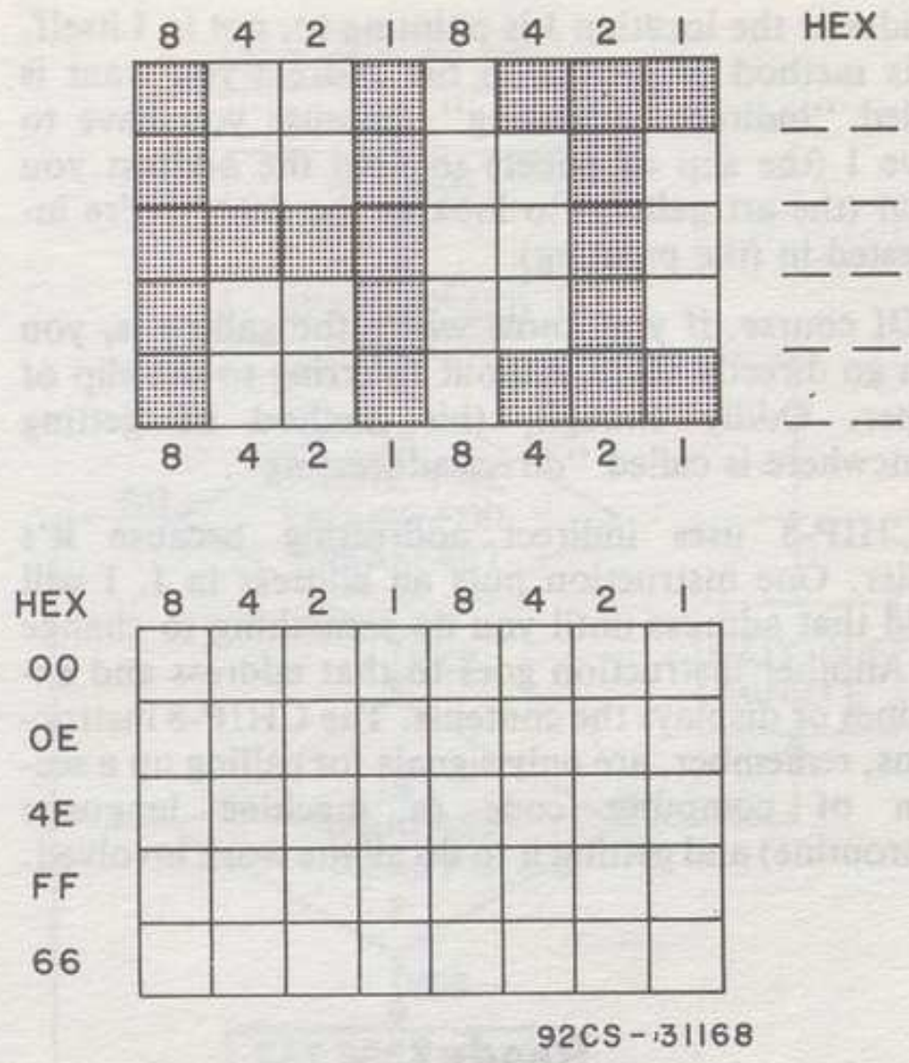


Fig. 13

All About "I"

The CDP1802 microprocessor in your VIP has a group of internal registers which can be thought of as machine language variables. In the **VIP Instruction Manual**, the registers are abbreviated R, so you'll have no trouble distinguishing them from the CHIP-8 variables (V). CHIP-8 uses one of these registers as a "pointer" to any location in memory that you might select. The selected memory location will usually contain data to be displayed on the screen during execution of a CHIP-8 program. This manual, and the **VIP Instruction Manual**, refers to this pointer as "I".

Suppose you want to examine a painting which is hanging in the West City Art Gallery, but you don't remember the address of the gallery. Fortunately, you have a slip of paper on which you have written the address, so if you're familiar with the city, you can go directly to the gallery.

Once you've arrived at the gallery, however, you still can not examine the painting until you have gone inside the building.

I is analogous to the slip of paper on which an address is written. Just as the paper can have any address in the world written on it (depending on where you want to go), so I can have any address in VIP memory stored in it.

The painting in the gallery is what you're really interested in, of course, not the address. And, when making use of I, the information you'll be using

resides at the location I is pointing to, not in I itself. This method of getting to the address you want is called "indirect addressing", because you have to have I (the slip of paper) to hold the address you want (the art gallery) to look at the data you're interested in (the painting).

Of course, if you know where the gallery is, you can go directly there without referring to the slip of paper. Oddly enough, this method of getting somewhere is called "direct addressing".

CHIP-8 uses indirect addressing because it's easier. One instruction puts an address in I. I will hold that address until you do something to change it. Another instruction goes to that address and examines or displays the contents. The CHIP-8 instructions, remember, are only signals for calling up a section of computer code (a machine language subroutine) and getting it to do all the work involved.

Ready?

Now it's time to think about how to start learning to write CHIP-8 programs. All the information you've received so far is background stuff—things you needed to know before learning any computer language, and especially before learning CHIP-8.

Before going on to the next section, which describes the CHIP-8 instructions, be sure you thoroughly understand the concepts to which you've been introduced in this section. If you find yourself confused later on, it will be because you didn't fully grasp an idea presented earlier in the text. If any of the following terms are not clear to you, go back now and refresh your memory. Once you understand the vocabulary and the concepts represented thereby, you'll find the going much easier.

How To Write CHIP-8 Programs

Now that you've mastered the hard parts, you'll soon be digging into CHIP-8 itself, getting acquainted with a new language, learning VIP jargon, and becoming eager to try some programming on your own. Your first few programs are likely to be uncomplicated ones; moving dots across the screen or blinking specific squares. These you can program "on the fly"—figuring out what you're going to do and typing in the code for it as you go along.

This is fine for short programs. But as you begin to plan exotic games or dramatic graphic displays, this "on the fly" method of programming will cause you more trouble and frustration than you can imagine! You'll find yourself having to go back through your code time and again to find out what value you stored in which variable, and then have to hunt through the program to find out what you did with the data. And worse: when you decide at a later date to add a neat new feature to the program, you will have forgotten how it works, and will no doubt end up introducing "bugs" (errors) instead of the added attraction you planned.

So what will you do once you have a reasonable grasp of CHIP-8 and have written some small programs and are ready to take on your first "big" project?

Well, since computers are methodical beasts, you can understand that they get along best with methodical people. This doesn't mean programming will be any less fun. Indeed, programming is far more enjoyable when you aren't spending all your time tracking down bugs in a poorly planned program!

Planning a computer program can be likened to planning a speech. When you are asked to speak at a meeting of some sort, you prepare the speech carefully to be certain you make all the important points clearly. First you decide on the topic about which you will speak. Then you prepare a rough outline, and define the key words which will be the main points. You might even write a short paragraph about each key word, to refresh your memory while you're speaking. Then you write up the finished product, and rehearse it carefully. You may try on different expressions or tones of voice. Finally, you go to the meeting and present your speech, assured that you have done all your homework and that nothing will go wrong.

You will use approximately the same procedure when planning a computer program:

A. Define the task you want the program to perform.
B. Draw a flowchart of the program operation.
C. Define the key variables you'll use in your program.
D. Write and test any special-purpose routines you plan to use.

E. When the program is finished, test it thoroughly, using the data you expect to use. Then test it with unexpected data.
F. Add final documentation—note anything unusual or complex. The coding forms provided at the end of this section provide a space for your comments.
G. Fit all your notes together—coding forms, flowcharts, anything you've written down about your program. File everything away where you'll be able to find it later!
H. Congratulate yourself on a job well done. Show your program off to someone now—not back when you were still getting rid of the bugs.
I. Look at programs written by other people, read the VIPER and other computer-oriented publications. If you see a neat way to do something, file it away for use in some future programming effort.

Let's look at each of these steps and see how they'll help you plan your program.

A. **Define the task.** Now is the time to carefully think through the entire program. What will the program do if the user presses a wrong key? What graphics do you plan to use? Will the user understand what the program wants him to do? After this step, you'll have a pretty good idea what a program RUN will look like.

B. **Draw a flowchart.** A flowchart is a "road map" of your program. It shows how each step is related to the other steps; which procedure is performed when; and in what order. People have dreamed up scores of special symbols to use for flowcharts, but only two are really necessary: a box to indicate some operation or procedure, and a diamond to represent a decision made by the computer.

Let's look at Fig. 14, a flowchart for a familiar situation. Look at each step carefully. Notice that the decision boxes (the diamonds) have two possible exit paths while the procedure boxes (the rectangles) have only one possible exit path. What would happen if you tried putting the last step (GO TO WORK) before the first step (GET OUT OF BED)? You wouldn't get a very good start in the morning, would you?

The flow chart helped you plan the order in which you would do the things you need to get done.

A computer is like a very small child in that it needs to be told exactly what to do in any situation. You might be able to tell a seven-year-old child "empty the trash" with some degree of assurance that he'll be able to accomplish the task successfully. But suppose the child is only three years old? Then you might have to say "Pick up the waste basket and carry it to the garage. Take the lid off the garbage can and empty the waste basket into the can. Then put the lid back on the can and bring the waste basket back to the kitchen." Even at the age of three a child

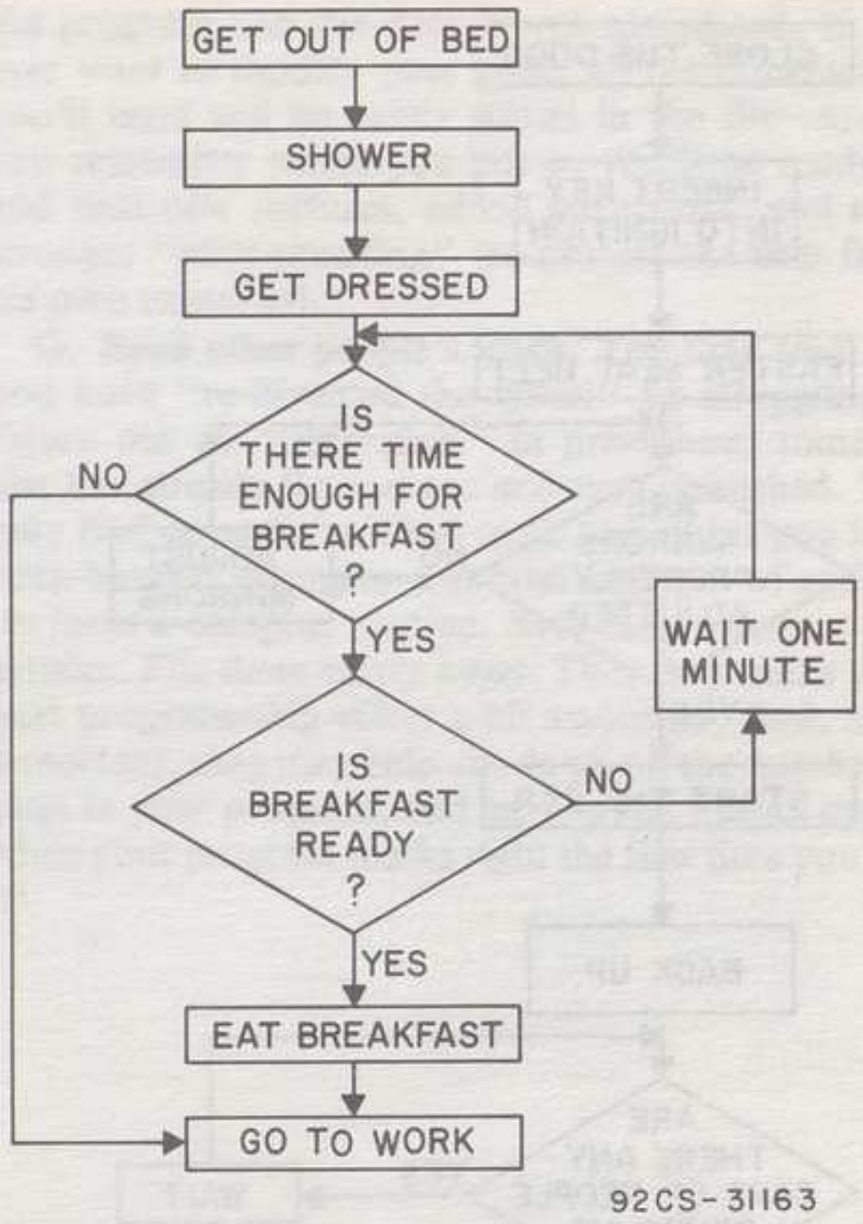


Fig. 14

will know whether he has to put the waste basket down before he can remove the garbage can lid, whether there are doors between the kitchen and the garage that he'll have to open, and so on. A computer wouldn't know, however, and it is up to you to give very explicit instructions.

Try to draw a flowchart for all the things you should do when you get in the car. Start with "CLOSE THE CAR DOOR" and end with "BACK OUT OF THE DRIVEWAY." Fig. 15 is an example. What steps have been left out?

Fig. 16 is a flowchart for a program to light a square on the screen. The square is to be moved up if the user presses the "2" key, down if the user presses "8", to the right if the user presses "6" and to the left if the user presses "4".

Look closely at the flowchart. Will the symbol "blink" even if the "0" key is pressed? What feature has been added that wasn't mentioned? Can you see where a GO TO or a JUMP instruction would have to go to?

Notice that each diamond contains only two exits: one for "yes" and one for "no". Since a computer can only make "yes"or "no" decisions, you should

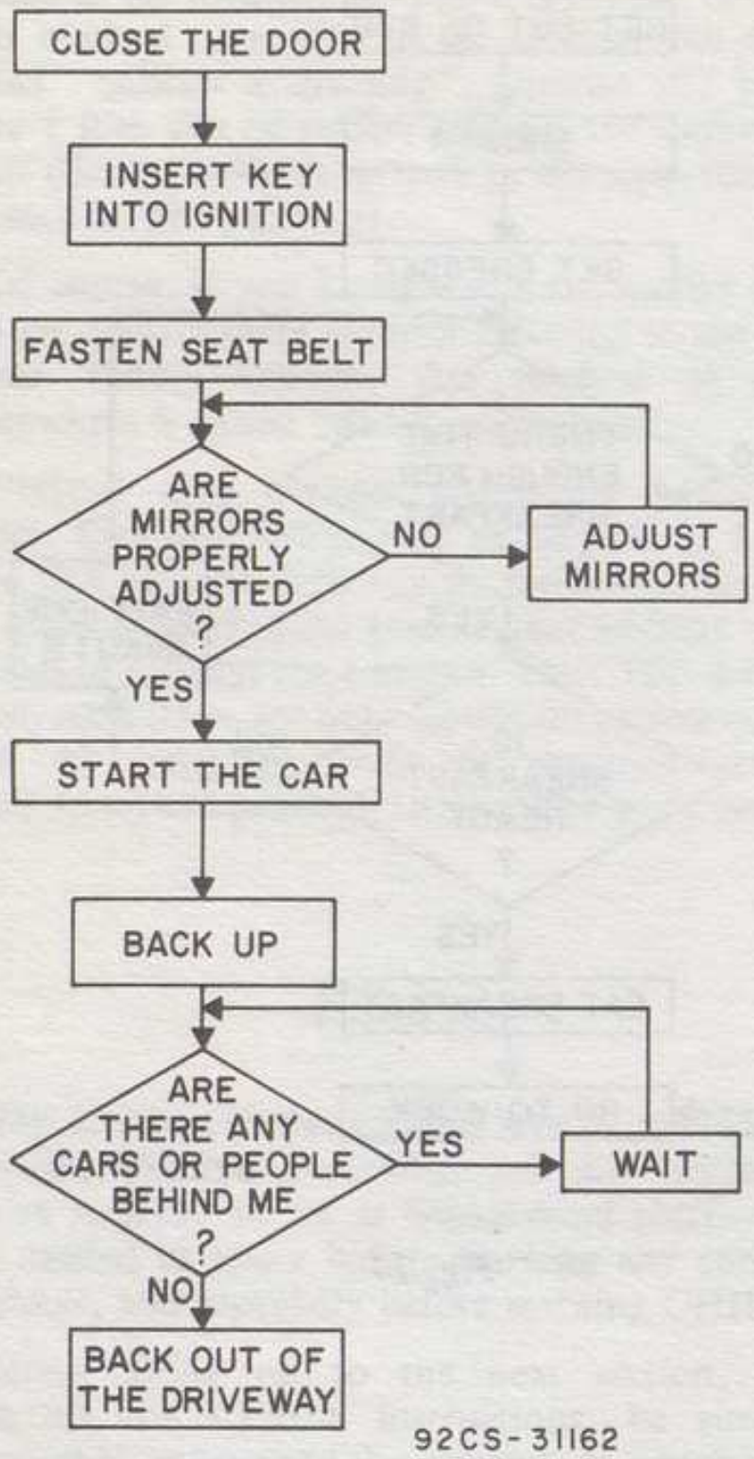
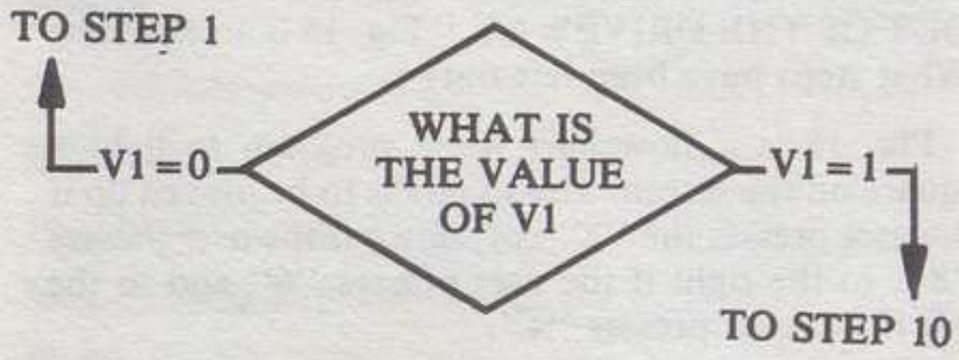


Fig. 15

plan for that when designing your program. This also avoids the problem of a decision like

"IF V1 = 0 GO TO STEP 1 AND IF V1 = 1 GO TO STEP 10"



DON'T DO THIS!! We haven't told the computer what to do if V1 is equal to something besides 0 or 1!

C. **Define the variables.** Decide what data each variable will contain. One variable should be used exclusively for the X coordinate of your display, and one should be reserved for the Y coordinate. What other variables will you need ?

D. **Test special features first.** It's easier to find a bug if you have some idea where to start looking for it. There's nothing more frustrating than entering an entire program—then finding out something doesn't work—and not knowing where to look for the problem. It's even more frustrating if your bug destroys the CHIP-8 interpreter by trying to move data into the CHIP-8 memory area—and you have to re-load CHIP-8 again—and again—and...

Routines to display special graphic patterns or to test for keyboard responses are good examples of program segments which should be written and tested before they are incorporated into a larger program. If you have flowcharted your program, you can

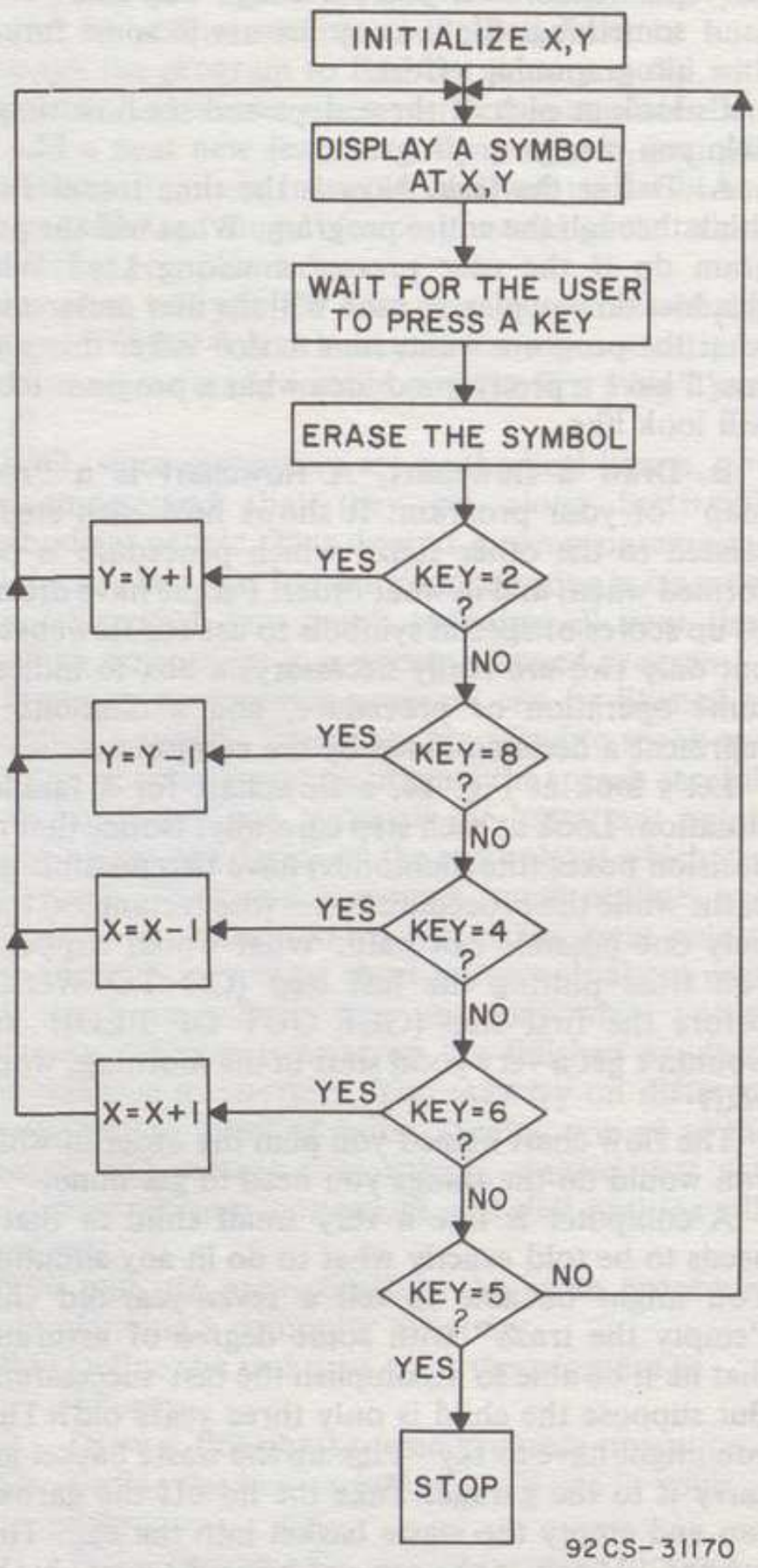


Fig. 16

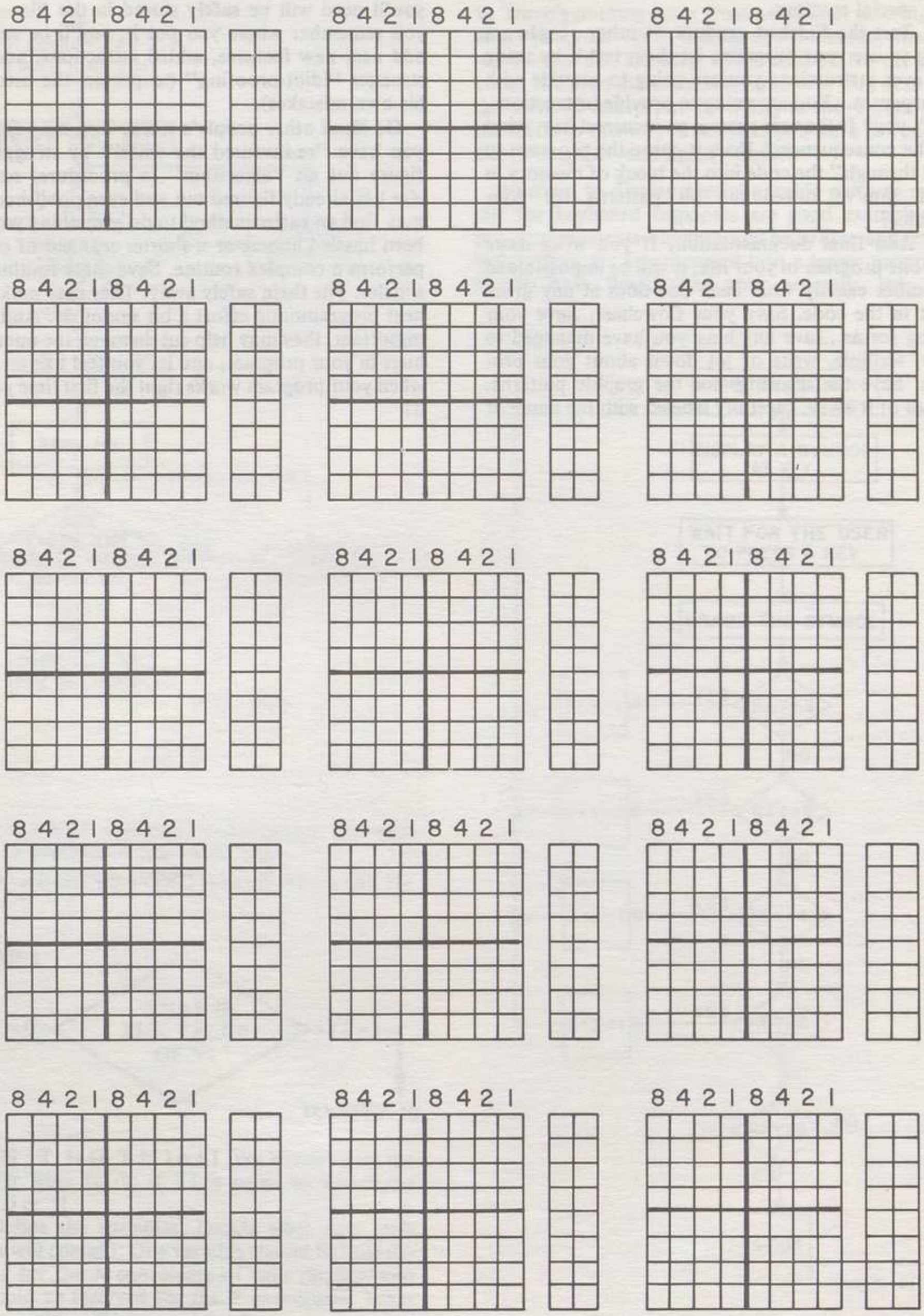
assign blocks of memory for each step in the program and use the pre-assigned addresses when you write your special routines.

E. **Test the finished product.** If other people are going to use your program, let them test it by using whatever instructions you are going to provide with the program...You are going to provide instructions, aren't you? If the user presses an incorrect key, what are the consequences? Does it cause the program to "fall through" the code into the block of memory in which you've stored the bit patterns for your graphics?

F. **Add final documentation.** If you write more than one program in your life, it will be impossible to remember exactly what each one does at any given point in the code. Save your flowchart...save your coding forms...save anything you have managed to draw, scribble, write or jot down about your program. Save the drawings for the graphic patterns. File all of it away, carefully labeled with the name of the program and the date it was completed. If you ever want to modify your code, all the information you'll need will be safely stored in the file—and if you remember where you put it, you'll be ready to add neat new features, added attractions, and even stronger "idiot-proofing" (to protect the user from his own mistakes).

G. **Read other people's work.** You may find that you have "re-invented the wheel" by struggling to figure out an "algorithm" (a procedure) someone else has already figured out and even published. You may find an easier method to do something you have been hassled about, or a shorter segment of code to perform a complex routine. Save these routines and articles. File them safely away. They may make your next programming effort a bit smoother. And, most important, they may help cut down on the number of bugs in your program, and let you feel like an expert when your program works right the first time you run it!

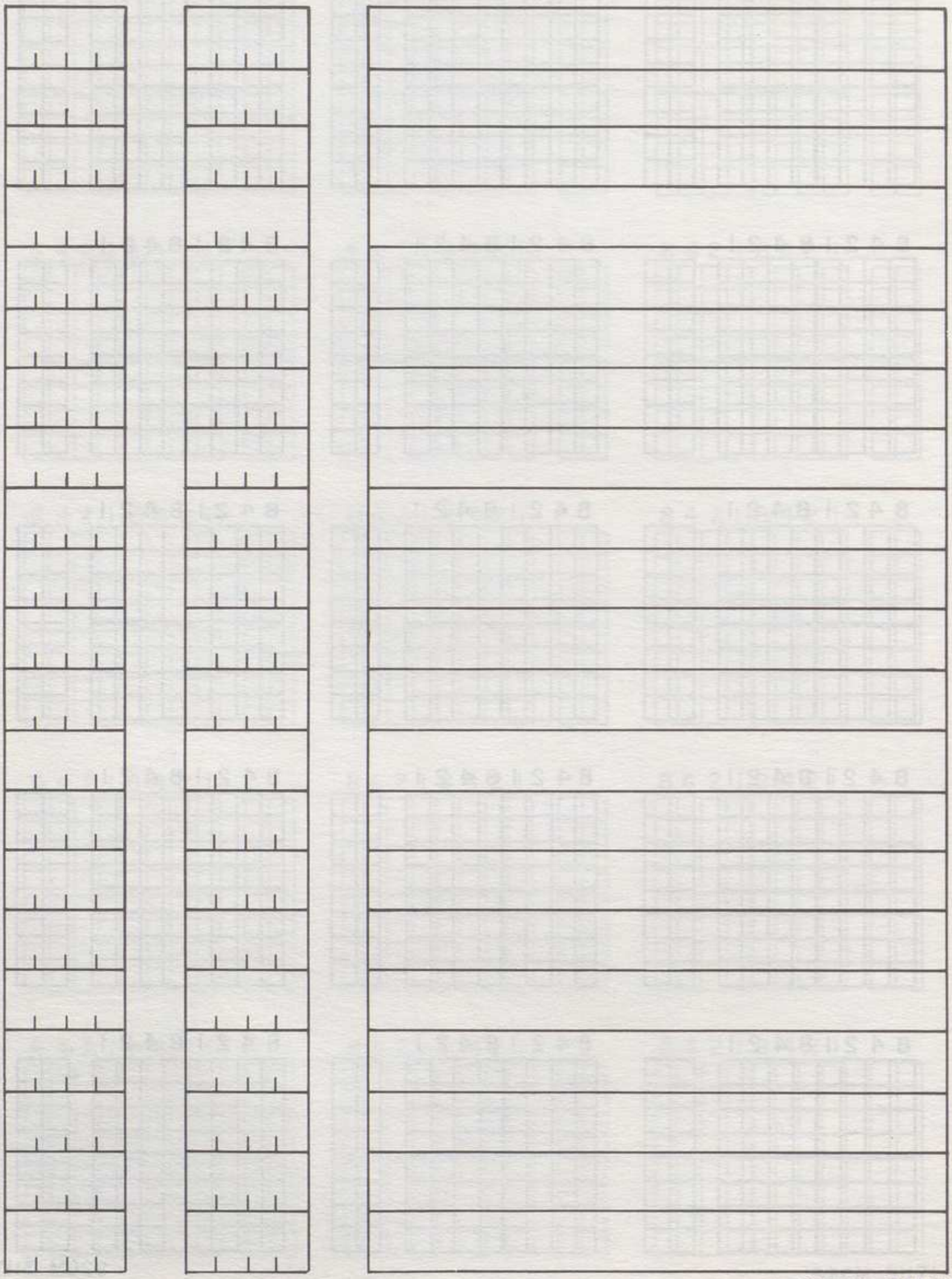
Bit Pattern Forms



92CM-31171

Bit Pattern Forms

CHIP-8 Coding Form



92CS-31172

Section III
The CHIP-8 Instruction Set

Introduction

CHIP-8 has 31 instructions, each of which is composed of four hex digits. The first digit is a command code and helps determine which machine language subroutine is supposed to be executed. The other three hex digits contain information (such as variable names, constant values, or memory addresses) which is needed by the interpreter in order for the instruction to be properly executed.

In the following descriptions of the CHIP-8 instructions, certain letters of the alphabet are used to indicate the nature of the specified digit; whether it is a variable name, a hex constant, or a memory address.

X and Y are used to represent variable names. In using an instruction like 6XKK, for instance, the X would be replaced with a variable name such as V3. Since the "V" isn't actually placed in the instruction, the "3" would be the digit actually replacing the X: 63KK.

KK is used to represent a 2-digit hexadecimal constant value. The value is determined by the user (in this case, that means you!). In the 6XKK instruction, the KK would be replaced by a numeric value such as 01. Since two digits are required to replace KK, the "01" would be the digits actually replacing the KK: 6X01.

Following the example, if X were replaced by "3", and KK were replaced by "01", the 6XKK instruction would be 6301. The "6" is the command code, the "3" is the name of the variable, and the "01" is the hex constant.

MMM is used to represent three digits of a 4-digit memory address. Since VIP RAM addresses all begin with 0, the leading 0 is not used or needed in the instructions. In the AMMM instruction, for example, the MMM would be replaced with the last three digits of an address, such as 020A. Since the leading zero is not used, the four digits of this example instruction would be A20A.

N is used to represent the number of vertical bits to be used in the display of a specific pattern. In the preceding section of this manual, you were shown how to draw bit patterns on the forms. These patterns were usually 8 × 8 bit squares, since the VIP requires that a pattern be one byte wide (and 8 bits = 1 byte). The N indicates how many bits high the pattern is, and can be replaced by any number from 1 to F (F in hex = 15 in decimal). In an instruction like DXYN, for example, the X and Y would be replaced by variable names, and the N would indicate the number of bits (vertically) required by the pattern. It may be of interest to know that it takes exactly as many bytes of code to store a pattern in memory as the number of bits high the pattern requires for display. So that if it takes 5 bytes of code to store a pattern (such as the pattern used as an example in the preceding section of the manual), that pattern will require 5 bits (vertically) to display, and N will equal 5. Thus, the DXYN instruction (assuming X is replaced by V1 and Y is replaced by V2 for this example) will become D125.

M(I) represents the memory address which is being pointed to by, or stored in, I. M(I) is pronounced "M of I" or "contents of memory pointed to by I" or "memory addressed by I".

The first few CHIP-8 instructions we'll look at will help you understand the CHIP-8 program code on page 17 of the preceding section. These instructions are introduced in the order in which they appear in the code. Then we'll go on to examine the rest of the CHIP-8 instructions. Once you're familiar with even five or six of them, you'll be able to make your VIP play all sorts of exciting games and display a great variety of exotic graphics.

Note that the instructions are written here in 4-digit groups: 6XKK, 1MMM, etc. When the actual program code is shown, however, the instruction is broken into two 2-digit groups: 6X KK, 1M MM. This separation is done specifically to emphasize the fact that each instruction requires two bytes of memory. The group of four digits at the left of each example of program code represents the address into which the first byte of the instruction is entered. Looking at the code on page 17, you'll see that the address is incremented by two (0200, 0202, 0204, etc.). The odd-numbered addresses contain the second byte of a CHIP-8 instruction. Also, note that CHIP-8 does not use familiar arithmetic operations like +, −, ÷, ×, =, >, <. They are not used because the CHIP-8 instruction code calls a subroutine in which these operations are incorporated. So, the only information you need to supply is the CHIP-8 instruction which calls the appropriate interpreter subroutine, and that subroutine knows which operation to perform.

It is important for you to try all the program examples and little exercises presented here. They aren't difficult, and some of them may seem incredibly dull or even repetitive. Their purpose is to help you gain familiarity with the routine of entering code, making changes to your programs, and verifying the work you do, as well as teaching you how to use the CHIP-8 instructions. The more you practice, the more comfortable you'll feel when the time comes to show off your new toy and its tricks.

AMMM

WHAT IS IT?

AMMM Set I=0MMM

WHAT DOES IT DO?

Stores memory address 0MMM in memory pointer, I.

WHEN IS IT USED?

When your program first begins to run, the memory pointer I has random hex digits stored in it. Before transferring data from memory to either the display screen or to another location in memory, you must first store the address of the first byte of that data in the memory pointer. The CHIP-8 interpreter (and the machine language subroutines) cannot automatically know where in memory you may have stored the data for a bit pattern you want to display on the screen, or where you may have stored data you want to transfer to another location in memory. The AMMM instruction places an address in the memory pointer; that address is the location in memory where the first byte of your data is stored. No matter how often you use the data, the address stored in I stays in I until you change it. Of course, while the program is running, you can change the address stored in I by using the AMMM command again. Then, I will point to an address containing other data you want displayed or transferred.

HOW IS IT USED?

Since VIP memory addresses all begin with "0", it isn't necessary to use the leading zero in the instructions. If you have data stored at address 020A, simply change the MMM in the AMMM instruction to 20A; AMMM becomes A20A. You have now "set" I to point to address 020A. Even if you have data stored at 020B, 020C, 020D, etc., I will only point to the first byte of data—other instructions tell CHIP-8 how many bytes of data to use. If you want to store data in locations 0300 through 0340, you should set I=0300 by replacing MMM in the AMMM instruction with 300; AMMM becomes A300.

6XKK

WHAT IS IT?

6XKK Set VX=KK

WHAT DOES IT DO?

Assigns the value of KK to the variable specified by X.

WHEN IS IT USED?

When a program first begins execution, each variable has a random value in it, and that random value will probably be useless to you. In fact, it may cause your program to do unexpected and/or undesirable things on the screen. Therefore, you will want to assign a value to each of the variables you plan to use in the program—before you use them. You can assign that value with the 6XKK instruction. After execution of the 6XKK instruction, the initial random value stored in VX has been destroyed and replaced by the value you specified with KK. You can use this instruction anywhere in a program; to assign new values, to reset variables to an earlier value, etc. You can store things like the X and Y screen co-ordinates where you want a graphic displayed, or store the values of keys that the program user will be expected to press, or store a value which will be used to determine how long a tone will sound or how fast a character will "blink" on the screen.

HOW IS IT USED?

Suppose you have decided to display a pattern on the screen at X-coordinate 01 and Y-coordinate 05. The 6XKK instruction lets you replace the X with a variable name and the KK with a value. Let's store the X-coordinate in V1. Then replace the X in 6XKK with "1", and replace the KK in 6XKK with "01". Thus, 6XKK becomes 6101. If you decide to store the Y-coordinate in V2, use "2" to replace the X, and 05 to replace KK; 6XKK becomes 6205. Note that you must use two digits to replace KK. You can use 6XKK as many times as you like in a program, with each of the 16 CHIP-8 variables and values ranging from 00 to FF. (FF in hex=256 in decimal.)

Just for drill, let's assign a value of 2A to each of the CHIP-8 variables:

60 2A	Set V0=2A
61 2A	Set V1=2A
62 2A	Set V2=2A
63 2A	Set V3=2A
64 2A	Set V4=2A
65 2A	Set V5=2A
66 2A	Set V6=2A
67 2A	Set V7=2A
68 2A	Set V8=2A
69 2A	Set V9=2A
6A 2A	Set VA=2A
6B 2A	Set VB=2A

CXKK (cont'd)

6C 2A	Set VC=2A
6D 2A	4Set VD=2A
6E 2A	Set VE=2A
6F 2A	Set VF=2A

Now, let's put a sequence of numbers into V7:

67 01	Set V7=1
67 02	Set V7=2
67 03	Set V7=3
67 04	Set V7=4
67 05	Set V7=5
67 06	Set V7=6
67 07	Set V7=7
67 08	Set V7=8
67 09	Set V7=9
67 0A	Set V7=A
67 0B	Set V7=B
67 0C	Set V7=C
67 0D	Set V7=D
67 0E	Set V7=E
67 0F	Set V7=F

DXYN

WHAT IS IT?

DXYN	Display N bytes of a bit pattern at coordinates X, Y on the screen.

WHAT DOES IT DO?

Displays a pattern stored in memory. The pattern is displayed at coordinates (X,Y) on the screen, and is N bits high.

WHEN IS IT USED?

After storing an X coordinate in VX and a Y coordinate in VY, determine how many vertical bits exist in your pattern and replace the N in DXYN with that number. The memory pointer I must previously have been set to the address of the first byte of the pattern to be displayed. If you wish to display a pattern which is eight bits high, it will take eight bytes of memory to store the data for that pattern.

Replace N in DXYN with the number of bytes in the pattern; in Fig. 17, 8. Store the X coordinate for your display in V1, and store the Y coordinate in V2. Then replace the X in DXYN with "1" (V1) and the Y in DXYN with "2" (V2). So DXYN becomes D128. (X=variable in which X coordinate is stored. Y=variable in which Y coordinate is stored. N=number of vertical bits required to display the character). Or, suppose you have a pattern five bits high, which can then be stored in five bytes of memory. If an X coordinate is stored in V1 and a Y coordinate is stored in V2, replace X with 1, Y with 2, and N with 5, so DXYN becomes D125. If the display is to be shown in locations X=5 and Y=10 on the

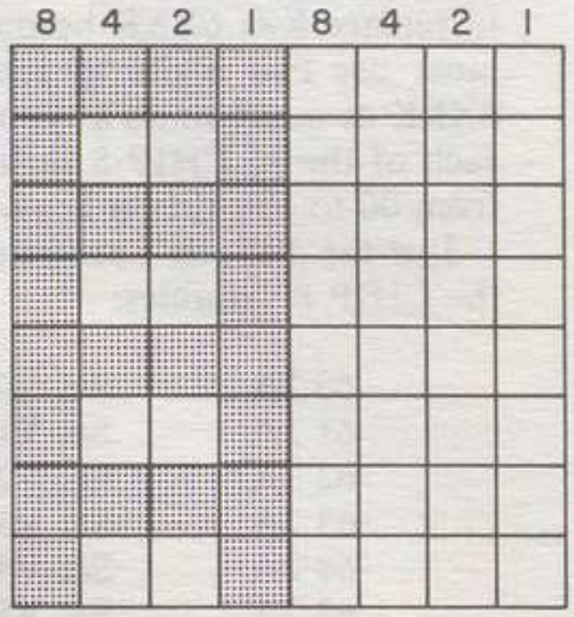


Fig. 17—This pattern is 8 bits or 1 byte high. It is also 8 bits or 1 byte wide.

DXYN (cont'd)

screen, the values 5 and 10 must be stored in variables before any pattern can be displayed at that position usually by using the 6XKK instruction. (In this text, the X coordinate will be stored in V1 and the Y coordinate will be stored in V2.) The memory pointer I must be set to point to the first byte of pattern data, using the AMMM instruction, and the X and Y coordinates must be assigned to variables using the 6XKK instruction.

You may use any of the 16 CHIP-8 variables to store X and Y, but V0 and VF are special purpose variables and may be altered inadvertently during program execution.

DXYN is the only CHIP-8 instruction which will display anything on the screen. When you build a bit pattern for a graphic character, that pattern must always be one byte wide. However, the pattern may be as many as 15 bits high. If you notice when you create the pattern, each row of bits (1 byte) is represented by 2 hex digits. These digits represent the data used to create the pattern. The number of bits used in the pattern must not exceed F (F in hex = 15 in decimal), and should not be confused with the data digits. Only one digit can replace the N in the DXYN instruction.

Suppose, for instance, you wanted to display only 3 bytes of the pattern in Fig. 17. Then, N in DXYN would be 3, and your pattern would look like Fig. 18.

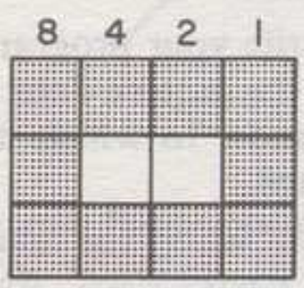


Fig. 18

1MMM

WHAT IS IT?

1MMM	Go to 0MMM or Jump to 0MMM.

WHAT DOES IT DO?

Causes the address specified by 0MMM to be stored in the program counter so that the instruction at location 0MMM is the next instruction executed. All program steps between the 1MMM instruction and the instruction at location 0MMM are skipped.

WHEN IS IT USED?

a)To effectively end program execution.

b)To return to the beginning of the program, effectively restarting program execution.

c)To transfer control of the program to an out-of-sequence address.

d)To provide "no-operation" instructions.

a)If you don't specify an "end" to your program, CHIP-8 won't know when to stop, and will "fall through"—possibly into data intended for a bit pattern. This can have disastrous effects, even "destroying" CHIP-8—so the interpreter would have to be reloaded from tape or re-entered from the keyboard—because the interpreter will think the pattern data is instruction code.

b)If you wish to start the program over from the beginning, for example, in a game situation, you can use the 1MMM instruction to jump back to the beginning of the program. All the variables will be reset to their initial values, and scoring will begin again at zero.

c)If you wish to branch to another section of code under specific circumstances, the 1MMM instruction will jump to the selected address, skipping all the steps between the 1MMM instruction and the instruction at the selected address.

d)Especially when coding a long and complicated program, it's often a good idea to leave "blank" spaces in the address sequence—just in case you have to go back and fill in an instruction you forgot. 1MMM provides enough space for one instruction; and if you want to leave room for more, 1MMM makes that possible, too. Later, if the space is not used, anything stored in the "saved" or skipped space will be ignored.

HOW IS IT USED?

a)To "end" the program at address 0208, replace the MMM in the 1MMM instruction with "208". Note that again, as in the AMMM instruction, the leading zero of the specified address is not used. Thus, 1MMM becomes 1208 and means "stop"—if you are entering the instruction into location 0208.

b)To "restart" the program, replace the MMM in the 1MMM instruction with the address containing

1MMM (cont'd)

the first instruction in the program. Since CHIP-8 programs should always begin at address 0200, the 1MMM instruction becomes 1200.

c)If you are currently writing an instruction in address 0206 and wish to transfer control to the instruction at location 020E, the 1MMM instruction will become 120E. All the instructions between location 0206 and location 020E will be skipped.

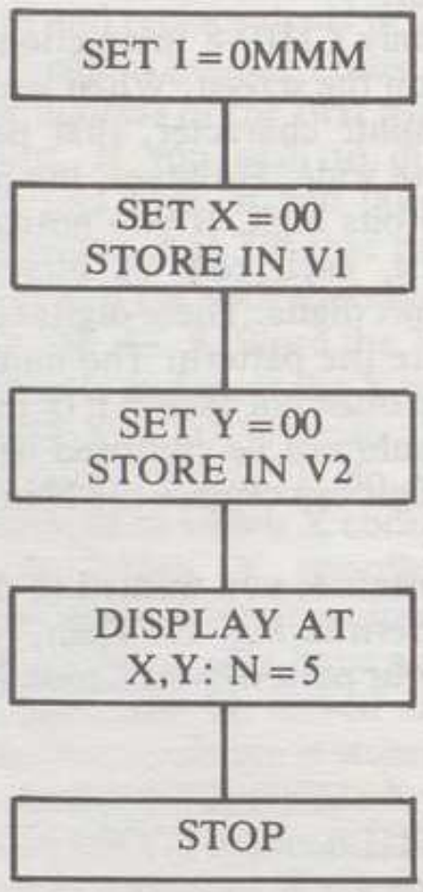
d)If you wish to save space for a possibly overlooked instruction, write the 1MMM instruction to jump to the next address in sequence. For example, if you are writing an instruction in location 0202, and you want to save space for one instruction, 1MMM becomes 1204. If you want to save space for two instructions, 1MMM becomes 1206. Notice that only the even numbered bytes are referenced. Each CHIP-8 instruction requires two bytes of code; two bytes of memory. The 1MMM instruction occupies two bytes, for example. If you are currently writing code at address 0200, the first byte of the 1MMM instruction occupies location 0200 and the second byte of the instruction occupies location 0201.

The instruction becomes easier to understand when you realize that the "1" in 1MMM is the instruction code and all "unconditional" branching instructions in CHIP-8 will have a "1" as the first digit unless the instruction is a call to a subroutine. This feature makes it possible for you to let your program "flow" jump all over memory. (There's no good reason to do that, but the 1MMM instruction does make it possible.)

In section II, page 17, you were introduced to the following CHIP-8 code:

Address	Code	Comment	
0200	A2 0A	Set I=020A	(This is the AMMM instruction)
0202	61 00	Set V1=0	(This is the 6XKK instruction)
0204	62 00	Set V2=0	(The 6XKK instruction again)
0206	D1 25	Display a 5-bit-high pattern at X,Y on the screen	(This is the DXYN instruction)
0208	12 08	Stop	(This is the 1MMM instruction,used to stop execution of the program)
020A	F0 90		(This is the first byte of data for a bit pattern. The address is stored in I in the instruction at location 0200)
020C	F0 90		
020E	F0 00		

Let's look at a flowchart of this program:



When you first begin to write your program, you may not know exactly where the pattern data will be stored. Usually you won't know to which address you should point I at this stage.

You do know where you want the character displayed on the screen: at X=0, Y=0. And you know you'll have to store those values in some variables. It is a good idea to note the variable names in the flowchart.

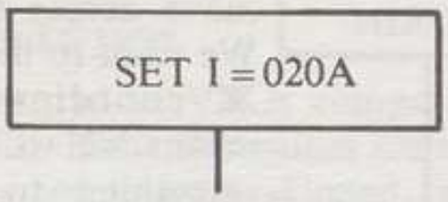
The value of N may not be known at this point if you haven't drawn your pattern yet. Since, in this case, we do know that N=5, we make a note of that in the flowchart.

Since that's all we want the program to do, we want the program to stop here. If it doesn't stop, it may "fall through" into the pattern code and could destroy all our work—and the CHIP-8 interpreter, as well!

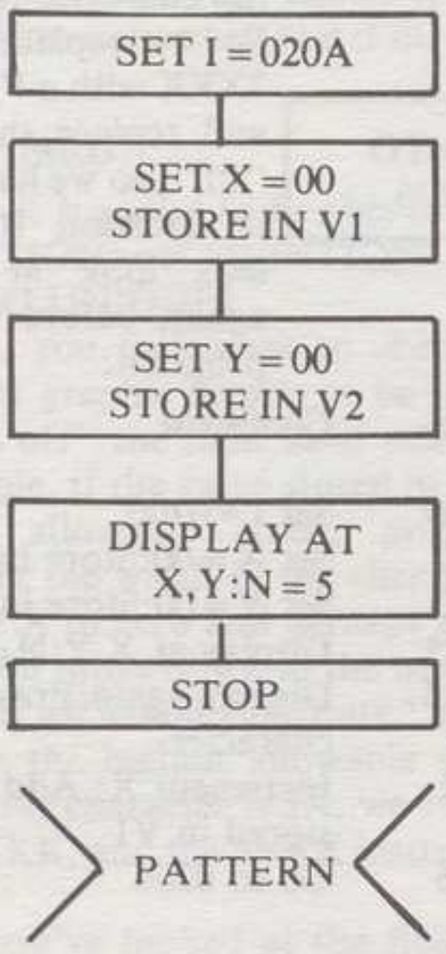
Since the CHIP-8 interpreter occupies memory through locations 01FF, our first instruction should be at location 0200. Since our program is 5 steps

IMMM (cont'd)

long, and since each step requires two bytes of memory, we can calculate that our bit pattern can be stored at 020A. Now we can go back and amend step one of the flowchart:



Let's look at a completed flowchart—after all the addresses and instructions have been worked out:



If you can figure out what's going on here, you're ready to move on to new information. If not, perhaps this summary of these four instructions will help:

Instruction or Command Digit	Remaining Digits	Meaning
A (SET I)	MMM	Set I = 0MMM
6 (SET VARIABLE)	XKK	Set VX = KK
D (DISPLAY)	XYN	Display pattern at X,Y on screen. Pattern is N bits high
1 (GO TO)	MMM	Go to 0MMM or Jump to 0MMM

Review for a while, try the code, create your own program using these instructions. Even now, knowing only four of the 31 CHIP 8 instructions, you can make your VIP do something!

So What Can We Do Now?

The VIP has a very interesting feature you haven't been told about yet; any pattern displayed over itself will be erased. If it is displayed over itself more than once, it will appear to blink! (This description is a gross oversimplification of what's really going on, but for our purposes now, this is what happens on the screen.)

To accomplish this feat, we have to change our program. At location 0208, where we have the 1MMM instruction 1208, we have to enter a different 1MMM instruction. Can you see where we have to "jump" to in order to get the pattern to be displayed over and over again? Right! We have to jump back one instruction, to 0206. So enter the new 1MMM instruction in location 0208. Remember how to do this? Initialize the VIP by pressing key "C" as you switch the toggle to **reset** and back to **run**. Type the address you want (in this case, it's 0208), select the Memory Write Option (key 0), and enter the code for the instruction: 12 06. All done. Now reset the VIP and switch to RUN—and surprise! The pattern doesn't blink—it vibrates! CHIP-8 is pretty quick, isn't it? Later we'll see how to slow down the vibration and get a true blink, but for now, let's look at more instructions and learn how to move the pattern across and down the screen.

7XKK

WHAT IS IT?

7XKK Add KK to VX.

WHAT DOES IT DO?

Adds the hex value specified by KK to the value stored in VX. The old value of VX is destroyed and the new sum is stored in VX.

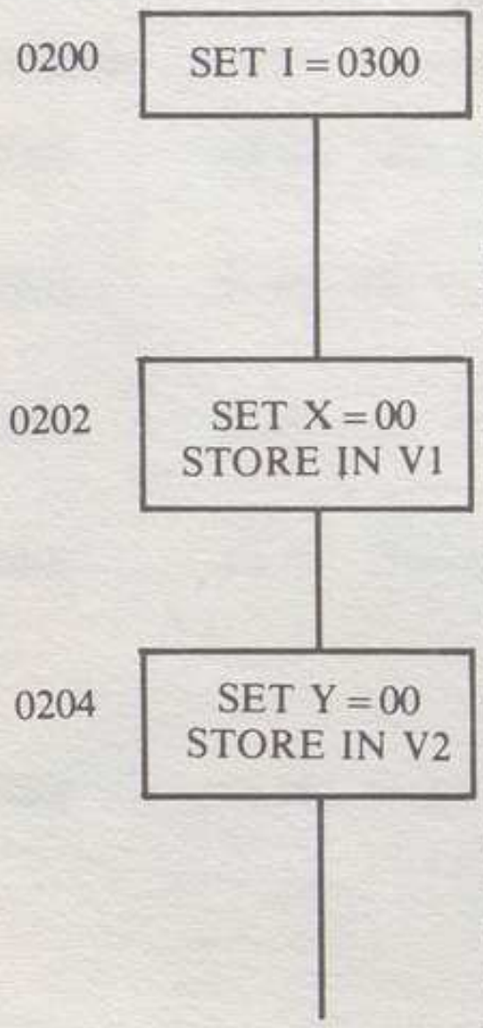
WHEN IS IT USED?

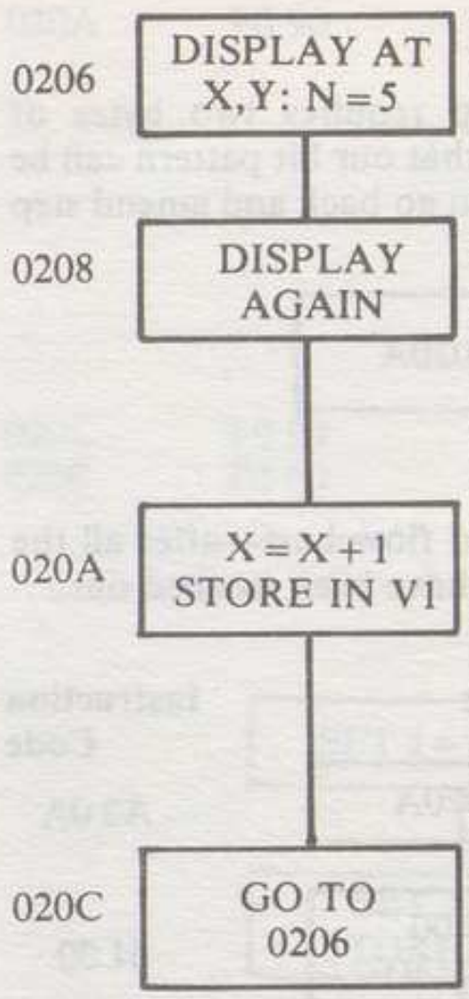
7XKK is used when you wish to add a specific hex value to a stored value. For example, when keeping game scores, or moving a pattern across the screen a specified number of bits, it is 7XKK which increments the score or the screen coordinate. Remember, however, that the value stored in VX prior to execution of 7XKK no longer exists. The sum of the old value and KK is now stored in VX.

HOW IS IT USED?

First, you must have a value stored in one of the variables. For example, suppose you have a value of 01 stored in V1. Second, you must determine which hex value (from 00 to FF) you wish to add to the value stored in V1. We'll use a hex value of 05 for this example. So replace the X in 7XKK with "1" (V1) and replace the KK with "05". And 7XKK becomes 7105.

An example that would be fun to work with is our existing program. We can use the 7XKK instruction to move the pattern across and down the screen. First, of course, we must draw a flowchart:





Address	Chart	Comment
0200	SET I=0300	The first thing to think about is that our program is going to take more than five steps. At least, that's a real possibility. So it might be a good idea to move our pattern data. If we move it all the way to location 0300, we'll have room to add more instructions as we learn them. Do this first. Once that is accomplished, we have to change the AMMM instruction in location 0200 to reflect the new data address so the memory pointer will be properly set. Now, if we compare the two flowcharts, we can see that the 1MMM instruction that was in location 0208 has to be moved to 020C, so we can put in the DXYN instruction which will cause the "erase" effect (so we won't leave a "trail" of the character behind as we move across the screen). We want to increment the X coordinate, which means we will be adding something to the value stored in V1. If we plan to move the character one bit at a time across the screen, we'll want to add 01 to the X coordinate each time the character is displayed. So we replace the X in 7XKK with a "1" (for V1) and replace the KK with "01", so we have the 7101 in location 020A. Now let's look at the code again, before we run the program:
0202	SET X=00 STORE IN V1	
0204	SET Y=00 STORE IN V2	
0206	DISPLAY AT X,Y: N=5	
0208	DISPLAY AGAIN	
020A	X=X+1 STORE IN V1	
020C	GO TO 0206	

Address	Code	Comment
0200	A3 00	Set I=0300
0202	61 00	Set X=00:Store in V1
0204	62 00	Set Y=00:Store in V2
0206	D1 25	Display at X,Y:N=5
0208	D1 25	Display again:Erase the character
020A	71 01	Increment X: Add 1 to value stored in V1
020C	12 06	Go to 0206
•		
•		
0300	F0 90	First byte of pattern data
0302	F0 90	
0304	F0 00	

Now run the program. What do you think would happen if you added a larger number to X? Like 5, perhaps, or 10 (remember that 10=A in hex). What would be the effect if you add to Y instead of to X? Or if you add to both X and Y? Try all these variations, and make up some of your own. You may be surprised by the effect the various values of KK will have on the apparent speed of the character as it moves across the screen.

Don't skip any of these exercises! The more you practice using the instructions, the easier it will be for you to understand the effect they'll have in a program you design and write yourself. Even though you may know what will happen, do try to code and run the example programs and do the exercises. Sometimes you may be surprised—and better now than when you're trying to design a complex, exotic game.

3XKK

WHAT IS IT?

3XKK — Skip next instruction if VX=KK.

WHAT DOES IT DO?

The value stored in VX is compared to a value specified by KK. If the two values are equal, the next instruction in sequence is skipped, and execution continues with the following instruction.

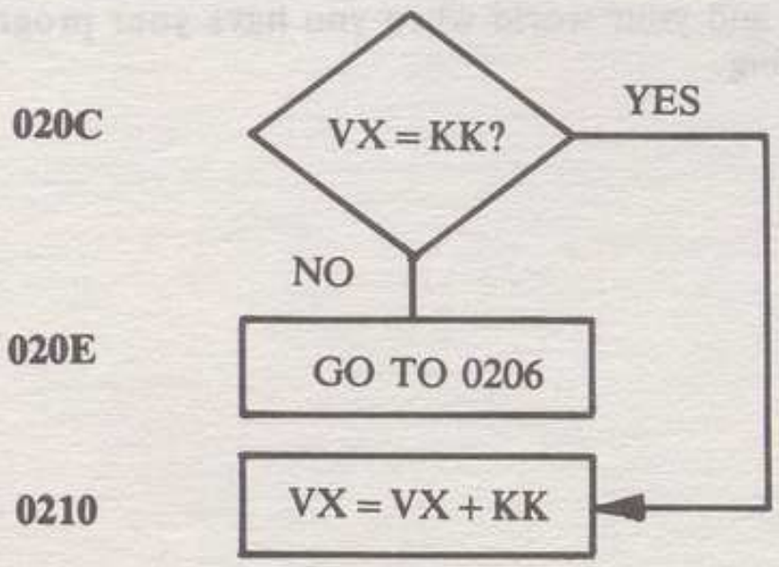
WHEN IS IT USED?

Use 3XKK when you wish to test VX for a specific value and (a) branch to an out-of-sequence set of instructions if the values are equal, or (b) execute a single instruction only when and if the two values are equal.

HOW IS IT USED?

Sometimes it is useful to be able to say, in effect, IF (CONDITION)=(CONDITION), THEN (DO SOMETHING UNIQUE)
For example, you may want to check the X coordinate of your graphic display to be sure the display doesn't "run off" the right hand side of the screen. In this example, if the value stored in VX is equal to the highest allowable screen position (making allowances for the width of the character), you may want to reset X to zero and perhaps increment Y so the display will move down to the next lower line of the screen. So we would substitute "1" for the X in 3XKK. Since the highest allowable screen position available to our character is 3A, we substitute "3A" for KK in 3XKK, and the 3XKK instruction becomes 313A

By now, you've looked at the flowchart for our program so often, you probably know it by heart. So, instead of drawing the flowchart for the whole program, we'll just draw the segment for the part of the program we'll be changing. Compare the two flowcharts and see if you can figure out why we added this segment at location 020C instead of somewhere else in the program.



In this example if V1 is not equal to the value 3A, the 1MMM instruction at location 020E is executed. The program jumps back to 0206 and continues to do the same thing it has been doing all along: displaying, erasing, adding to X, displaying, erasing, adding to X, etc.. But if the value stored in V1 is equal to 3A, the 1MMM instruction at 020E is skipped, and if there is an instruction at location 0212, it is the one which will be executed next. Our flowchart doesn't show an instruction at 0212. Can you see what has to be done next? Shall we increment Y, so the character will be displayed on the next lower screen line? And what has to happen to X? It must be reset to zero, or it will run off the right side of the screen and wrap around to the left. We must add new code and extend the length of the program. The new code begins at 020C.

Address	Code	Comment
020C	31 3A	Is VX=3A?
020E	12 06	No: go to 0206
0210	72 06	Yes: add 6 to 7; store in V2
0212	61 00	Reset X to zero and store in V1
0214	12 06	Now go to 0206

Run the program. After a while, the character will "run off" the bottom of the screen, and appear again at the top left-hand corner. But it writes over whatever is there (although, if we've done this right, there shouldn't be anything there). So we must add yet another segment of code to our program to keep the character from wrapping around to the top of the screen.

Knowing that the lowest Y coordinate on the screen is 1F, we know that if Y=1F our character has no place to be displayed. Remember, the X and Y coordinates are the position for the first byte of the data—and that means there are 4 bits that won't get displayed. If Y=1B, however, all 5 bits of the character will be shown. So we want to stop the display when Y=1B. Again, we can change the program and make use of the 3XKK instructions:

Address	Code	Comment
0214	32 1B	Does Y=1B?
0216	12 06	No: Go to 0206
0218	12 18	Yes: Stop

Try running this additional code with the rest of the program. You are able now to successfully move the character from left to right across the screen, and from top to bottom. Experiment with different values for KK in the 3XKK instruction, until you're fairly comfortable with the hex values for the X and Y coordinates on the screen. You can make the character move as far as you like and make it stop wherever you please. It would be nice to be able to

3XKK (cont'd)

move the character from right to left and from bottom to top, as well, but another little detail must be explored first.

Reason tells us that if we add to X to move the character right across the screen then we must subtract from X to move the character left; and if we add to Y to move the character down the screen, we must subtract from Y to move the character from the bottom to the top of the screen. Adding large hex numbers to the stored values has the effect of subtracting small hex numbers. Why? Let's look at it in decimal terms. Suppose 99 is the largest number you can use, and that you are restricted to two digits regardless of what numbers you add together. You've never learned how to subtract, so you're stuck with adding. How, then, would you get to 8 if you start with 9?

By adding 99 and 9 together!

```
  09
+ 99
----
 108
```

But you can't use the leading "1", since you have only two digits to work with. So your answer, 8, is all that's left to you. To "subtract" 4 from 9, then, you would count backwards from 99: 99, 98, 97, 96—and then add 96 to 9:

```
  09
+ 96
----
 105
```

See how it works?

If you aren't really familiar yet with hex digits, it may help to either look up the value in the table on page 13 or to remember that since F(hex) = 15(decimal), you can count backwards from F; F, E, D, C, B, A is equivalent to 15, 14, 13, 12, 11, 10.

F − 5 = A is equivalent to 15 − 5 = 10

"Subtracting" in hex, then, works the same way.

```
  09          09
+ FF        + FC
----        ----
 108         105
```

Because the VIP can only store two digits in each variable, the leading "1" is "invisible". (The "1" is stored somewhere else—in V0—and we'll see later just how useful that information can be!)

What we want to do now is move the character up to the top of the screen. If, at location 0204, you set Y = 1B, the character would begin its motion on the bottom line of the screen. To get the character to move up rather than down, you have to change the instruction at address 1210 from 7206 to 72F9. Try it and see. But remember—the 3XKK instruction at location 0214 is a check to see if Y = 1B. Perhaps it would be a good idea to have it checked to see if Y = 0 (the new instruction is 3200). If you want to move the character to the left, you must change the 6XKK instruction at location 0202 to read 613A, to start the character on the right-hand side of the screen. Then change the 7XKK instruction at location 020A to read 71FE. What other instruction needs to be changed? Look at the code and flowchart carefully and try to determine which instruction will "mess you up" if it isn't changed.

Right. The 3XKK instructon at location 020C. What will happen if you don't change it? If you don't know, try running the program before making the change. (But make the changes at locations 0202 and 020A first.) Then change the 3XKK instruction to read 3100 and run it again.

Now you can move your character anywhere you like, in any direction, on the screen. Now would be a good time to try to build a "frame" around the outside edges of the screen, or try to move a character clockwise around the screen—or counter-clockwise. Practice using these instructions until you can create the illusion of a bouncing ball. Once you accomplish all that, show off a bit to your family and friends; you deserve praise, applause, and admiration.

Remember—be patient and forgiving! One of the disconcerting aspects of programming is that if the program doesn't work, you can't blame the computer—or your kids or your spouse. If **your** program doesn't work, **you** did something wrong. It's frustrating and maddening. However, one of the greatest satisfactions in a programmer's life is "Hey Ma! It works!" So keep at it until you've mastered it—you'll be terribly pleased with yourself, your VIP, and your world when you have **your** program working.

FX0A

WHAT IS IT?

FX0A Wait for keypress.

WHAT DOES IT DO?

Causes the program to pause and wait for a key to be pressed. When the key is pressed, the Q-tone sounds, and the value indicated by the key is stored in the variable indicated by X.

WHEN IS IT USED?

Any time it is desirable for the program to halt and wait for the user to react or respond, FX0A is the instruction required. It doesn't matter which key is pressed; as soon as the keypress is detected, the Q-tone sounds and program execution continues. The value indicated by the key that was pressed is stored in the variable indicated by X in the FX0A instruction.

HOW IS IT USED?

First, decide which variable will hold the key value. It's a good idea to assign a value of zero to this variable before you use it (In fact, it's a good idea to "initialize" all the variables you intend to use in a program before you use them. In the first place, it gives you space in the "comments" part of the coding form to make a note as to the use to which the variable will be put during the program. But it isn't necessary. It's just a good practice.) Since we'll want to find a way to incorporate this instruction in our existing program, we'll assign the key value to variable 3 (V3). Then, by replacing the X in FX0A with "3", so that FX0A becomes F30A, we can use it in our program. Recall that the last instruction in the existing program is a 1MMM instruction in location 0218, used to stop the program. By adding a few more lines of code, we can make the program start all over from step one, at location 0200, or we can set I to point to a different address and select a new bit pattern for display.

Let's enter data for a new pattern, beginning at location 0306:

Address	Code	Comment
0306	F0 80	
0308	E0 80	This is the code for the letter "E"
030A	F0 00	

Now let's add our new instruction, FX0A to the program, reset I, and see what happens:

Address	Code	Comment
0218	F3 0A	Wait for key to be pressed
021A	A3 06	Set I=0306
021C	12 02	Go to 0202

This instruction gives you a good deal of flexibility; for you can stop and wait for the user to do something during the program. This will make the user of your program feel much more comfortable playing whatever game you have designed. He'll feel that he is participating, rather than just spectating, which is an important feature of most computer games.

It would be even more fun if the key press had an effect on what was going on during program execution. And CHIP-8 permits that, with the use of the EXA1 and the EX9E instructions—coming up next.

EXA1

WHAT IS IT?

EXA1 — Skip next instruction if VX ≠ key pressed.

WHAT DOES IT DO?

The low-order digit of the value stored in VX is compared to the value indicated by the key. If the correct key has been pressed, the value in VX will equal the keyed value. If the correct key has not been pressed, the next instruction in sequence is skipped, and the instruction following it is executed.

WHEN IS IT USED?

This instruction is used to make a program more "interactive"; that is, to allow user participation. If the user presses the correct key, the program skips the instruction immediately following the EXA1 instruction and goes on to the next instruction in sequence. This capability is useful when the user is being expected to determine, by way of a specific key, the direction in which a displayed character is to be moved. If he doesn't press the correct key, the character won't move in the correct direction—or whatever trap you (the wily programmer) set up for people who don't press correct keys!

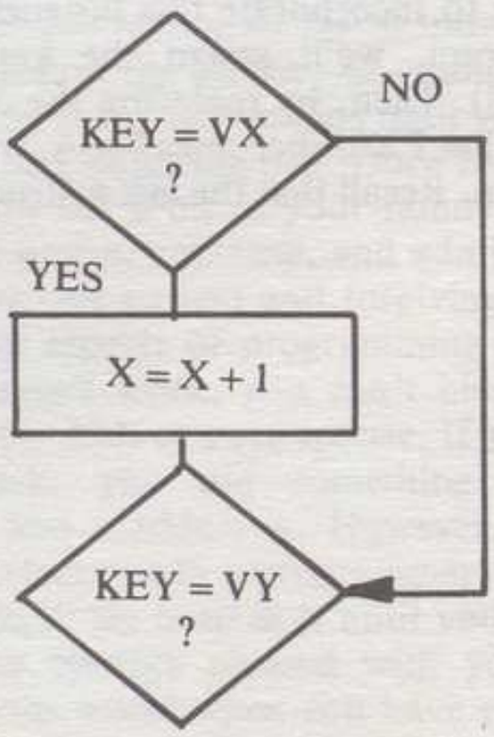
HOW IS IT USED?

Since we've worked our existing program to death, we'll begin a new one—especially since this instruction helps make things a bit more exciting. If you'll look back to Fig. 16 on page 20 in Section II, you'll find a flowchart which describes the program as well as the next few we'll explore.

To make a long story short, the code for the program is:

Address	Code	Comment
0200	61 00	Set X = 00: Store in V1
0202	62 00	Set Y = 00: Store in V2
0204	63 00	V3 = 00: Keypress will be stored here
0206	A3 00	Set I = 0300
0208	64 02	V4 = 02: To move character up
020A	65 04	V5 = 04: To move character left
020C	66 06	V6 = 06: To move character right
020E	68 08	V8 = 08: To move character down
0210	69 05	V9 = 05: To stop program
0212	D1 24	Display character at X,Y: N = 4
0214	F3 0A	Wait for user to press a key
0216	D1 24	Erase the display
0218	E4 A1	Key = 2?
021A	72 FF	Yes: move up
021C	E5 A1	Key = 4?
021E	71 FF	Yes: move left
0220	E6 A1	Key = 6?
0222	71 01	Yes: move right
0224	E8 A1	Key = 8?
0226	72 01	Yes: move down
0228	E9 A1	Key = 5?
022A	12 2A	Yes: Stop here
022C	12 10	No: Go to 0210

The EXA1 instruction does not wait for a key to be pressed. It is to be used after the key has been detected, and is used to compare the value of the key with a value stored in one of the 16 CHIP-8 variables. In this example program, values have been stored in five variables which are to be compared with the key. If the values are not equal, the program skips the instruction which immediately follows the EXA1 instruction and executes the next instruction. A flowchart of the sequence of instruction may prove helpful:



As you can see, the "skipped" instruction doesn't get executed if the key doesn't equal the stored value.

Conditional Branching

You were introduced to the concept of conditional branching with the 3XKK and the EXA1 instructions. The "formula" is:

IF (CONDITION) = (CONDITION), THEN (DO SOMETHING UNIQUE)
or
IF (CONDITION) ≠ (CONDITION), THEN (DO SOMETHING UNIQUE)

Since the concept is the same for all the instructions in the "conditional branching" classification, all of those instructions will be explained here.

3XKK	Skip the next instruction if VX = KK
4XKK	Skip the next instruction if VX ≠ KK
5XY0	Skip the next instruction if VX = VY (VX and VY are variable names)
9XY0	Skip the next instruction if VX ≠ VY
EX9E	Skip the next instruction if VX = value indicated by a key press
EXA1	Skip the next instruction if VX ≠ value indicated by a key press

In each case, the program will execute the step-after-next in sequence if the specified condition is met. To fit one of the examples into our "formula" at the top of the page, values must have already been assigned to the necessary variables or constants. With the 5XY0 instruction, for example, the formula would read

IF (value stored in VX) = (value stored in VY), THEN (Skip the next instruction).
And the EX9E formula would read
IF (value stored in VX) = (value indicated by key press), THEN (Skip the next instruction).

Need a Rest? Let's Have Fun

We've been working hard for quite a while—let's look now at some "just fer fun" instructions. There are several CHIP-8 instructions which make game-playing (and programming) just plain fun; these are the ones you've been waiting for. We can make the VIP "squeal"; we can display the contents of any of the CHIP-8 variables, and we can convert from hex to decimal without even trying! We can copy the value from one variable into another, we can set the timer and give our poor, unsuspecting user a real start when the VIP suddenly hollers for no apparent reason.

From now on, though, we're going to move a little faster. You know how to substitute a variable number for X in any instruction which has an X in it. You know how to replace KK with a hex value, and you know that MMM in an instruction represents an address. In short, you've got all the basics down pat by now, and there's no need to slow you down by being repetitive.

So let's look at some fun things for a while. We'll get back to the tedious stuff in good time.

FX18

WHAT IS IT?

FX18 SET tone timer = VX.

WHAT DOES IT DO?

Load the tone timer with the value stored in VX. The timer is decremented 60 times each second (at the same rate at which the display is refreshed, incidentally).

WHEN IS IT USED?

You can control the duration of the VIP's tone with the FX18 instruction. The maximum tone duration from a single instruction is a little more than 4 seconds.

HOW IS IT USED?

First, store a value from 00 to FF in one of the variables. Then replace the X in FX18 with the variable name. For example,

Address	Code	Comment
0200	63 FF	Set V3 = FF (for a good, long tone)
0202	F3 18	Load the tone timer from V3
0204	12 04	Stop

Experiment with different tone durations—use different values in V3 to obtain different tone durations. You could even automate that process by "subtracting" from V3 and going to 0202 instead of stopping at 0204.

0200	63 FF	Set V3 = FF
0202	F3 18	Load the tone timer from V3
0204	73 10	"Subtract" from V3 to change the tone duration
0206	12 02	Go to 0202 to load the tone timer again

Of course, you could start with V3 set to 01 and increment it by 01 each time you load the tone timer. Try a few experiments of your own.

FX29

WHAT IS IT?

FX29 "Fetch the least significant digit from VX".

WHAT DOES IT DO?

Sets the memory pointer to point to the address of the location containing the first byte of the 5-byte display pattern list for the least significant digit of the variable indicated by X.

WHEN IS IT USED?

FX29 is used to display the contents of a variable. The address pointer is set with the intent of displaying the value of the hex digit in the variable indicated. Only the digit derived from the 4 least significant bits of the byte is "fetched", so the value displayed will be between 0 and F.

HOW IS IT USED?

The FX29 instruction makes it easier to display all the hex digits. You don't have to store a bit pattern in memory because the pattern list is already there for each of the variables. You don't have to use the AMMM instruction to set I to point to the pattern list; the FX29 instruction does that for you, too. Simply store a value in any of the variables, "fetch" it with the FX29 instruction, and then display it. Remember that the digit will be 5 bits high, so N = 5 in your DXYN instruction:

Address	Code	Comment
0200	61 00	Set X = 00: Store in V1
0202	62 00	Set Y = 00: Store in V2
0204	63 00	Set V3 to any value (we used 00 so we can add to it)
0206	F3 29	"Fetch" the value from V3
0208	D1 25	Display it
020A	73 01	Add 1 to V3
020C	71 06	Add 6 to X
020E	12 06	Go to 0206 and "fetch" the value again

You'll find FX29 to be one of the most frequently used instructions in CHIP-8! It is used when you're keeping scores in a game, for example. You can convert a hex value to a decimal value; then the FX29 instruction must be used to "fetch" each digit in the score for display.

FX65

WHAT IS IT?

FX65 — Transfer data from memory into variables V0 through VX.

WHAT DOES IT DO?

FX65 transfers bytes of data from memory, beginning with the address pointed to by I, and stores that data in variables V0 through VX. The address pointed to by I after execution of the FX65 instruction will have been incremented by X+1.

WHEN IS IT USED?

The FX65 instruction is used to transfer data from RAM memory into CHIP-8 variables. You may transfer data from up to 16 bytes of memory into variables V0 through VF. Only the number of bytes indicated by the number of variables reserved will be transferred. If, for example, you replace X in FX65 with 0, only one byte will be transferred, and it will be stored in V0. If you replace X with 5, six bytes of data will be transferred, and they will be stored in V0, V1, V2, V3, V4, and V5. (0 is the **first** value, or number, remember?)

The memory pointer will be incremented when the FX65 instruction is used. If you have transferred six bytes of data, the memory address stored in I will be increased by six.

BEFORE EXECUTION: Set I=0300

F5 33 Transfers six bytes of data (remember, 0 is the first number and counts as "1" when transferring information)

AFTER EXECUTION: I=0306

The "X" in the instruction was replaced by 5; I is incremented by X+1, so now I=0306

FX33

WHAT IS IT?

FX33 — Set M(I)=3-digit decimal equivalent of hex value stored in VX.

WHAT DOES IT DO?

Converts the value of VX to decimal form and stores each digit in separate consecutive locations in memory, beginning with the address indicated by I.

WHEN IS IT USED?

Before you can use the FX33 instruction, you must set I (via the AMMM instruction) to point to a vacant location in memory. Then FX33 can be used to convert the value in VX to a 3-digit decimal number. The high-order decimal digit is stored in the address pointed to by I. The second digit is stored in the location immediately following the address pointed to by I, and the low-order decimal digit is stored in the address indicated by I+2.

The FX33 instruction is useful in scorekeeping situations and is used in conjunction with the FX65 instruction. I is changed during execution of FX33. An FX29 instruction (to fetch the pattern for the least significant digit of each byte) must be used to prepare each decimal digit for display.

The program code under the FX65 instruction shows one of the ways to use FX33. First assign a value to a variable, then assign an address to be pointed to by I. Select the screen positions for the display, then use FX33 to convert the value stored in the variable to a decimal number. Remember, when assigning an address in which to store the three digits, that three bytes are needed; one for each of the decimal digits, even if one or more of these digits=0.

Hex to Decimal Conversion

CHIP-8 permits you to convert any hex value to a 3-digit decimal value and to display the decimal digits on the screen. This capability is important for score-keeping operations during game situations. In the example program below, note that V1 and V2 are not used to store the X and Y coordinates of the display. This change, and the unfamiliar instructions, will be discussed and explained. First, read the program code and comments—then we'll explore them in some depth.

Address	Code	Comment
0200	65 C3	Set V5 = C3
0202	A3 00	Set I = 0300
0204	F5 33	Convert V5 to 3 decimal digits. Store the first digit in 0300, the second digit in 0301, and the third digit in 0302.
0206	F2 65	Transfer the data from 0300-0302 to V0-V2: The digit in 0300 is stored in V0, the digit stored in 0301 is transferred to V1, and the digit in 0302 is transferred to V2
0208	63 18	Set X = 18: Store in V3
020A	64 10	Set Y = 10: Store in V4
020C	F0 29	Fetch the pattern for the least significant digit of V0
020E	D3 45	Display the first decimal digit at X,Y: N = 5
0210	73 06	X = X + 6: Shift horizontal coordinate
0212	F1 29	Fetch the pattern for the least significant digit of V1
0214	D3 45	Display the second decimal digit at X,Y: N = 5
0216	73 06	X = X + 6: Shift right again
0218	F2 29	Fetch the pattern for the least significant digit of V2
021A	D3 45	Display the third decimal digit at X,Y: N = 5

Enter this code, verify it, and run the program. (Did you add one more instruction so the program would know where to stop?) The unfamiliar instructions FX33 and FX65 are the instructions necessary to convert from hex to decimal values. X and Y cannot be stored in V1 and V2 because the FX33 and FX65 instructions use these variables to do the conversion.

CXKK

WHAT IS IT?

CXKK — Let VX = Random byte (KK = mask).

WHAT DOES IT DO?

CXKK generates a random byte and stores the value in VX. KK is used as a "mask". This means that each bit in the random byte is compared to the corresponding bit in KK. If the bit in KK contains a 0, the matching bit in the random number is set to 0. If the bit in KK contains a 1, the matching bit in the random number is not changed.

WHEN IS IT USED?

The range KK can be used to provide a specific number of alternatives from which the random number resulting from a CXKK instruction selects one.

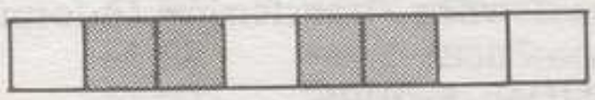
HOW IS IT USED?

Let's look at how is works, since it is necessary to understand what's happening in order to effectively use CXKK. In this example, KK is arbitrarily set to 5B. In a hex (bit) pattern, 5B will look like

0101 1011

Remember, 0 means "off"—the unshaded bits in the pattern, and 1 means "on"—the shaded bits.

Using CXKK, a random number is generated—again, we'll arbitrarily select a value: 6C. In a hex pattern, 6C looks like



0110 1100

Each bit is compared with the corresponding bit in KK, and if the KK bit is set to 0, the random number bit is changed to 0. If the KK bit is set to 1, there is no change in the corresponding random number bit.

The first bit in KK in our example is 0, so the first bit in the random number is set to 0, regardless of what it earlier may have contained. The second bit in KK contains a 1, so there is no change made to the second bit of the random byte. The third bit in KK is set to 0, so the 1 in the third bit of the random byte is set to 0. The fourth bit in KK is set to 1, so no change is made to the fourth bit of the random byte. Therefore, the first four bits of the value stored in VX will be 0100. Can you work out the hex pattern for the second four digits?

In the following example, the random number is stored in V5. The third instruction in the sequence will be skipped if the value stored in V5 is 0. As the range of KK increases, the probability of executing the third instruction in the sequence decreases.

CXKK (cont'd)

Code	Comment
C5KK	Set a random byte into V5
4500	Skip next instruction if V5 = 00
1AAA	If V5 ≠ 00, go to address 0AAA
1BBB	If V5 = 00, go to address 0BBB

Remember that AAA in this example, as well as BBB, represents any address in memory to which you want to jump, depending on the value of the byte stored in V5.

The following table may help you learn to calculate the odds of the third instruction in our example being executed. The last column in the table shows the possibilities of the final value stored in VX after execution of the CXKK instruction. Note that if KK = FF, the resulting number will not have any bits changed and the byte stored in VX will be the same as the random byte originally generated by CXKK.

If KK is equal to:	%probability of going to 0AAA	0BBB	Bit mask possibilities	
01	50%	50%	0000 0001	0 or 1
03	25%	75%	0000 0011	0 to 3
07	12.5%	87.5%	0000 0111	0 to 7
0F	6.25%	93.75%	0000 1111	0 to F
FF	0.392%	99.608%	1111 1111	0 to FF

Now let's look at some of the bit patterns not shown in the table, and see how they would work as a mask. Let's look at the hex patten for 6, to begin with—in case you wanted to simulate a dice game.

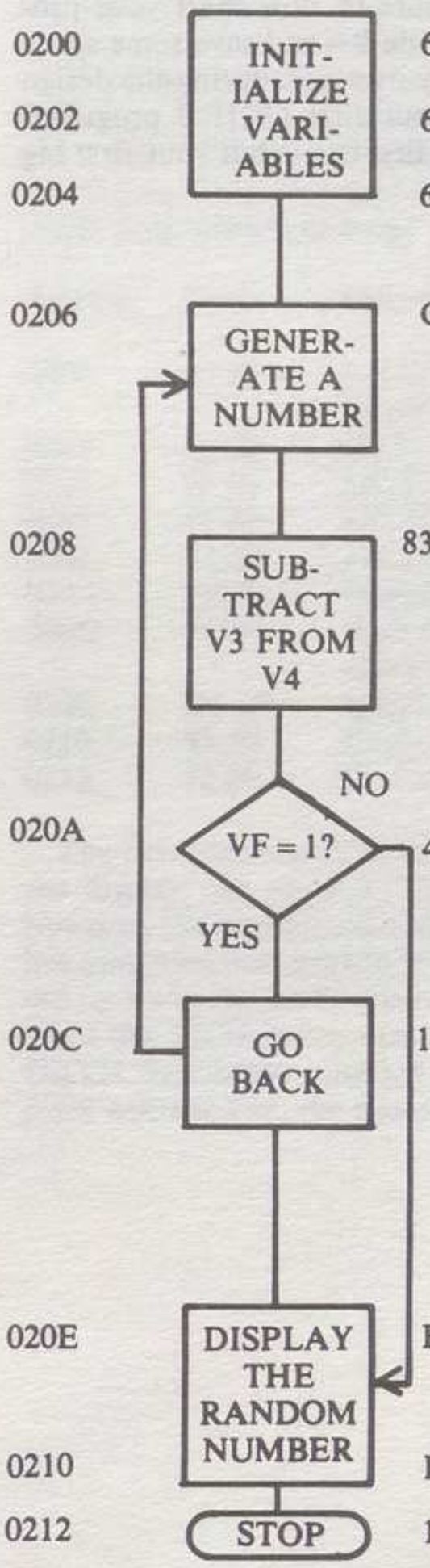
The bit pattern for 06 is 0000 0110. What random numbers would rsult from this mask? Remember, every 0 in the mask causes a corresponding 0 in the random number. The 1 in the mask leaves the corresponding bit in the random number unchanged. In this case, then, the only values available are 0, 2, 4, and 6. You would never get 1, 3, or 5 because the 1-bit in the mask (the leftmost bit) is 0, which will cause the corresponding 1-bit in the random number to be set to 0.

Work out what random values could result if you selected bit masks of 0E, 26, AA, or 39 for KK values.

0E	0000 1110
26	0010 0110
AA	1010 1010
39	0011 1001

Now let's try to work out a real program using CXKK.

Suppose you want to generate a random number between 0 and 8. What value would you give to KK? Before you answer, look at the bit pattern for 08: 0000 1000. What values are possible? Remember—any bit in KK which is set to 0 causes the corresponding bit in the random number to be set to 0. In this case, the only values available are 0 and 8. Is this what you expected? Try assigning a value of F. The hex pattern is 0000 1111. But now the values available are all the values between 0 and F— and the only values you want are between 0 and 8. Here's where we make use of the fact that when you implement the 8XY5 instruction, VF is set to 1 if the value of VY is greater than the value of VX. First, let's look at a flow chart.



Address	Code	Comment
0200	61 00	Set X = 00: Store in V1.
0202	62 00	Set Y = 00: Store in V2.
0204	63 08	Set V3 = 08: This is the highest number desired.
0206	C4 0F	Generate a random number between 0 and F.
0208	83 45	Subtract highest value in range from random number.
020A	4F 01	Is VF set?
020C	12 06	The random number was greater than the highest number we wanted. Go get a new random number.
020E	F4 29	Fetch the random number.
0210	D1 25	And display it—it's okay.
0212	12 12	Stop here.

6XKK (cont'd)

By now, you've looked at enough flowcharts so you don't need a detailed explanation about what's going on. You might try expanding the program, though, and design a whole game around the CXKK instruction. Try a dice game—just generate two numbers between 1 and 6, and display them. Perhaps you might add the two numbers together, using the 8XY4 instruction, then convert the sum to decimal and keep score. Yahtzee, a favorite dice game from way back, can be successfully played on the VIP—so can Craps or a simple "guessing" game in which the user tries to guess the sum after each roll, before the value is displayed. Be sure to flow-chart your program before trying to code it—and save some space for instructions you may overlook during the design process. Will this be your first CHIP-8 program? Well, possibly not your first ever—but your first **big** project.

FX15

WHAT IS IT?

FX15 SET Time = VX.

WHAT DOES IT DO?

Load the timer with the value stored in the variable indicated by X. Once the timer is loaded, it is decremented 60 times a second until it reaches 0. The timer stops when it reaches 0.

WHEN IS IT USED?

After assigning a value to a variable, the contents of that variable can be loaded into a timer. This capability is useful when you want to time the user's response in a game situation; or when you restrict the amount of time during which the game progresses. For example, assign a value of FF to variable 9, then use the FX15 instruction to load the timer. The X in FX15 is replaced with the 9 and the instruction becomes F915.

Address	Code	Comment
0200	61 00	Set X = 00
0202	62 00	Set Y = 00
0204	69 FF	Let V9 = FF
0206	F9 15	Load timer from V9
0208	F5 07	Check the time
020A	45 00	Time = 00?
020C	12 0C	Yes: Stop
020E	D1 25	Erase old pattern
0210	F5 29	No: Fetch the pattern for V5
0212	D1 25	Display it at X, Y: N = 5
0214	12 08	Go to 0208 to check the time again

Notice the instruction in 0208. The FX07 instruction loads the variable V5 with the value in the timer, so you can examine it and thus "check the time". This program will simply display the value in the timer on the screen. The timer is decremented about 60 times a second, so about 4 seconds will elapse between the time you set the timer to FF and the time it reaches 00.

FX07

WHAT IS IT?

FX07 Let VX = current timer value.

WHAT DOES IT DO?

Load the variable indicated by X with the current value of the timer. The contents of the timer may be loaded into any variable to permit the user to "check" the time by examining the contents of the variable into which a timer value was stored.

WHEN IS IT USED?

FX07 is used to determine when a delay or a "time allowed" period has ended; especially in game situations. The program on the previous page shows how the FX07 instruction can be implemented. First the timer must be loaded from one of the CHIP-8 variables. The same variable can be used to store the current timer value when the FX07 instruction is used to check the time. You can see how this will work by making two changes to the program: In location 0204, change the 6XKK instruction to read 65 FF; and in location 0206, change the FX15 instruction to read F5 15. Then run the program as usual.

Each time the FX07 instruction is encountered, the value of the timer is loaded into the variable specified by X. You can check the time as often as you like during a program. It is useful to reserve one of the variables to serve as a "clock" variable in any game requiring a timer.

8XY0

WHAT IS IT?

8XY0 Let VX = VY.

WHAT DOES IT DO?

Copy the value of the variable indicated by X into the variable indicated by Y.

WHEN IS IT USED?

8XY0 is used to transfer the contents of one variable into another variable. First, assign values to at least one of the variables to be used; then use 8XY0 to copy that value into another variable. The old value stored in X will be destroyed and replaced by the value transferred from Y. For example, if a value of B8 is stored in V5, and you want to copy that value into V3, replace the X with the "3" (V3) and the Y with "5" (V5). Remember that you copy **into** the first variable expressed **from** the second. An example program might help:

Address	Code	Comment
0200	65 FF	Let V5 = FF: An arbitrary number
0202	61 00	Set X = 00: Store in V1
0204	62 00	Set Y = 00: Store in V2
0206	63 00	Let V3 = 00
0208	F3 29	Fetch the pattern for V3
020A	D1 25	Display it at X, Y: N = 5
020C	F5 0A	Wait for a key; store key value in V5
020E	D1 25	Erase old pattern
0210	83 50	Copy V5 into V3
0212	12 08	Go to 0208

The first time the D125 instruction is encountered, the display will show a "0". After the first time, however, the display will show the value of whatever key you press and store in V5. That value is transferred to V3 in the 8XY0 instruction at location 020C. Then the FX39 instruction fetches the pattern, the DXYN instruction displays it, and then after you press another key, the process starts over again.

Back to Work

Now you know enough about CHIP-8 to make your VIP do just about anything you like. You can sound the VIP tone, you can keep time, you can keep score—in decimal—and you can display the contents of any of the CHIP-8 variables. You can make your program interactive by waiting for the user's response and executing instructions according to the value of the key pressed by that user. You can copy data from any location in memory into any of the CHIP-8 variables, and you can transfer data from any of the variables into any other variable or into any location in VIP RAM memory. You can generate a random number, you know how to alter the sequence of execution of the program steps. You have learned quite a lot, in fact!

But there are more goodies to come: Arithmetic instructions, for example, so you can add the values of variables together or subtract one from another. Instructions to permit logical operations such as AND and OR; and instructions to permit branching to either CHIP-8 or machine language subroutines. You'll learn how to increment the address in the memory pointer, and then we'll show you how to decipher existing CHIP-8 code so you can figure out what another programmer may have had in mind when he designed his program.

Most of the still-unfamiliar instructions require only a few words of explanation. You're probably familiar enough with the concepts of addition and subtraction, for example, that a detailed explanation will bore you. So we'll move right along.

8XY4

WHAT IS IT?

8XY4 VX = VX + VY.

WHAT DOES IT DO?

Add the contents of the variable indicated by X to the contents of the variable indicated by Y and store the result in VX. If the sum is greater than FF, VF will be set to 01. If the sum is equal to or less than FF, VF will be set to 00.

WHEN IS IT USED?

8XY4 is used to add the contents of two variables together. If a value of 09 is stored in V4, for example, and a value of E1 is stored in V5, the sum, after performing the 8XY4 instruction, is EA and is stored in V4. (This assumes you have replaced the X in 8XY4 with "4" and replaced the Y in 8XY4 with "5".)

V4 contains 0000 1001 (hex 09)
V5 contains 1110 0001 (hex E1)
After 8454, V4 contains 1110 1010 (hex EA)

and V5 is unchanged. If the sum had been greater than FF, variable VF would be set to 01. If you aren't certain that a sum will be less than FF, it is a good idea to check the value of VF (with the 3KXX or 4KXX instruction) to find out, if it will be important to further progress in the program.

8XY5

WHAT IS IT?

8XY5 VX = VX − VY.

WHAT DOES IT DO?

Subtract the contents of the variable indicated by Y from the contents of the variable indicated by X, and store the difference in VX. If the contents of VX are equal to or greater than the contents of VY, VF is set to 01. If the contents of VX are less than the contents of VY, VF is set to 00.

WHEN IS IT USED?

8XY5 is used to subtract the contents of one variable from the contents of another. The difference is stored in the first variable to be expressed in the 8XY5 instruction. The order of appearance is important in that the contents of VF are affected. The new value of VF will depend on whether VX is greater or less than VY. For example, if a value of 09 is stored in V4 (and V4 is replacing X in 8XY5), and if a value of E1 is stored in V5 (and V5 is replacing Y in 8XY5), VF will equal 00.

V4 contains 0000 1001 (hex 09)
V5 contains 1110 0001 (hex E1)
After execution of 8455,
V4 contains 1101 1000 (hex D8)
V5 is unchanged
VF contains 00

VF can be thought of as a 'flag' to let you know which of the two variables contains the larger number. If VF is set to 0, then the variable indicated by Y in 8XY5 is greater than the variable indicated by X. CHIP-8 doesn't have any direct instruction to permit you to compare variables or values on a 'greater-than' or 'less-than' basis. VF can be tested and the appropriate action taken can depend on whether or not VF = 1.

For example, suppose you store a value of 5 in variable 3 and a value of 7 in variable 4. Then you execute an 8XYN, substituting 3 for X and 4 for Y. In this example, the value in Y is greater than the value in X (7 is greater than 5), so VF is set to 0. By testing the value of VF, you can determine which action to take next. Be careful, however, If, instead of 8345, you execute 8435, then the variable indicated by X will contain the 7 and the variable indicated by Y will contain the 5. After execution, VF will be set to 1.

V3 = 5	V3 = 5
V4 = 7	V4 = 7
After 8345, VF = 0	After 8435, VF = 1

FX55

WHAT IS IT?

FX55 Transfer contents of variables V0 through VX to memory.

WHAT DOES IT DO?

Transfers the contents of 1 to 16 variables to consecutive memory locations, beginning with the location indicated by the address stored in I. The variables saved will always include V0 through the variable indicated by VX.

WHEN IS IT USED?

Since a few game applications require more than 16 variables, this instruction (together with the FX65 instruction) can be used to store and recall any number of values and thus create the effect of having more than 16 variables at your disposal. The memory pointer will be incremented by X + 1, just as it is with the FX65 instruction, when the FX55 instruction is executed. This instruction is the reverse of the FX65.

HOW IS IT USED?

You may have stored data in memory which you will want to use in the execution of the program. The most common reason is that your program needs more than 16 variables, so you have to store the data until it is needed, and transfer it into the variables as needed. For example, if you have values of 04, AF, and B2 stored in locations 0400, 0401, and 0402 and you want to use this data in variables, you would use the FX55 instruction to get the data from memory into V0, V1, and V2. First, set I = 0400, so the interpreter will know where the data is stored. Then replace the X in FX55 with "2" (because you want to transfer data to variables V0 through V2). After the instruction is executed, V0 will contain the value 04, V1 will contain AF, and V2 will contain B2. The address pointer will be pointing to location 0403 (X + 1) after the FX55 instruction is executed for this example.

00E0

WHAT IS IT?

00E0 Clear Screen.

WHAT DOES IT DO?

Erases the screen by setting every bit to 0.

WHEN IS IT USED?

Whenever you wish to completely clear the screen.

HOW IS IT USED?

You can use this instruction anywhere in a program. No variable names or constant values are required. When your program begins execution, the CHIP-8 interpreter automatically clears the screen for you. During program execution, however, especially when a game is over and you are maintaining cumulative scores, it is useful to clear the screen before beginning the second round.

Address	Code	Comment
0200	61 00	Set X=00: Store in V1
0202	62 00	Set Y=00: Store in V2
0204	63 FF	Set V3=FF so you'll have something to look at
0206	F3 29	Fetch the value from V3
0208	D1 25	Display it
020A	F3 0A	You have to press a key to go on
020C	00 E0	Clear the screen
020E	12 00	Do it all again so you can see what happened

Remember that the 00E0 instruction just clears the active display page; the screen. It doesn't clear memory or disrupt any of the variables. It just clears the screen.

8XY1

WHAT IS IT?

8XY1 Let VX=VX or VY.

WHAT DOES IT DO?

Logically OR the contents of the variable indicated by X with the contents of the variable indicated by Y, and store the result in VX.

WHEN IS IT USED?

The contents of VX are examined and each bit is compared to the corresponding bit in VY. A logical OR operation is performed, and the resulting byte is stored in VX. 8XY1 is used to selectively add bits into an existing hex pattern.

HOW IS IT USED?

If a value of 07 is stored in V4, for example, and a value of 31 is stored in V5, execution of the 8XY1 instruction (substituting "4" for the X and "5" for Y) produces a value of 37. This value is produced because a bit in the resulting byte is set to 1 if either bit (in VX or in the corresponding bit in VY) is equal to 1. A logical OR differs from an addition in that the OR operates on individual bits while the addition operates on the whole byte as a single number.

The resulting byte is stored in VX, destroying the previous value. It is not necessary for both bits (one in VX and the corresponding bit in VY) to be equal to 1 in order for the resulting bit to be set to 1. Only one of the two bits being examined and ORed must =1 for the resulting bit to =1.

8XY2

WHAT IS IT?

8XY2 Let VX = VX and VY.

WHAT DOES IT DO?

Logically AND the contents of the variable indicated by X with the contents of the variable indicated by Y, and store the result in VX.

WHEN IS IT USED?

The logical AND operation (and the logical OR operation) are rarely used in CHIP-8 except in advanced game applications and many control applications. The logical operations permit the user to selectively turn on the bits within a byte (using OR) and to selectively turn them off again (using AND).

If a value of 09 is stored in V4, for example, and a value of E1 is stored in V5, the 8XY2 instruction produces a value of 01. This value is produced because a bit in the resulting byte is set to zero if either of the corresponding bits in VX and VY are set to zero. A bit in the resulting byte is also set to 0 if both corresponding bits in VX and VY are set to 0. In effect, then, the corresponding bits in each value must be 1 for the resulting bit to be set to 1. The resulting byte is stored in VX and the old value is destroyed.

FX1E

WHAT IS IT?

FX1E Let I = I + VX.

WHAT DOES IT DO?

Add the value stored in the variable indicated by X to the address stored in I. The value of X is unchanged.

WHEN IS IT USED?

This instruction is used to vary the address pointed to by I in accordance with other parameters which may change during execution of a program. For example, you may have several patterns to be displayed on the screen. The FX1E instruction will increment I so you don't have to remember the addresses of each of the patterns while you're coding your program. Not only that, but you can use the FX1E instruction in a subroutine after having assigned a specific value to VX or having the value in VX generated at random by the CXKK instruction.

The best example of the use of FX1E is in the VIP SPOOKY SPOT program in the **VIP Instruction Manual**, on page 48. The pattern data begins at location 0270. A random number is generated in location 0224 and is used to determine whether the "answer" will be yes or no. Enter the program and run it on your VIP—not only is it fun, but it's valuable as a learning tool, too.

Subroutines

A subroutine is a segment of program code that is used repeatedly during execution of the program. Many programmers use subroutines for their display routines, since they use the same code over and over for each display. Because it may take eight or ten bytes to store all the code needed for the display routine, and it only takes one byte to "call" that routine when you need it, you can see the time and memory space are conserved by the use of subroutines.

CHIP-8 permits you to call subroutines in machine language as well as in CHIP-8. This gives you added flexibility (or will, once you have learned machine code). The instruction needed to call a CHIP-8 subroutine is 2MMM. This instruction calls a subroutine at location 0MMM, and can be used at any time as often as you like in a program. When the code in the subroutine has been executed, the 00EE instruction will return execution to the CHIP-8 instruction immediately following the "call" instruction, in the main program. Every CHIP-8 subroutine must end with a 00EE instruction so the CHIP-8 interpreter will know where the end of the subroutine is.

Subroutines may be called from within another subroutine (a practice known as "nesting"). Up to 12 levels of subroutines may be in use before conflicting with the CHIP-8 memory map. Subroutine 1 can call subroutine 2, which in turn can call subroutine 3. Subroutine 3 can call subroutine 4, which can call subroutine 5, and so on. But every level of subroutine must terminate with the 00EE instruction to insure proper return to the right place.

You can call a machine language subroutine from a CHIP-8 program by simply typing in the address at which the subroutine resides: 0MMM. This capability is useful when you want to perform operations in CDP1802 machine code which are not available in CHIP-8—it gives your programs more flexibility. As with CHIP-8 subroutines, you can "nest" machine language subroutines. If control is to be returned to CHIP-8 at the instruction immediately following the source of the call, the machine language instruction D4 must be used. (See the **VIP instruction Manual** for a description of the machine language code and the D4 instruction.)

There are 20 programs in the **VIP Instruction Manual**, and many of them use subroutine calls. The VIP SPOOKY SPOT program uses both CHIP-8 and CDP1802 machine language subroutines, and you might pick up some good techniques in subroutine use by examining the code in detail. If you're willing to spend the time on it, try flowcharting the VIP SPOOKY SPOT program—it will give you practice and teach you a lot about how the subroutines and branching instructions work.

Where Do We Go From Here?

Now you're ready to go to the **VIP Instruction Manual** and look at the comments there about the CHIP-8 instruction set. The whole list of instructions is printed on page 14—and there are some valuable remarks about using the instructions.

RCA has a number of add-on peripherals available for the VIP, such as a color board, a sound board, etc., and you needn't be an expert to use them. CHIP-8 can be modified to include color instructions and sound instructions—in fact, CHIP-8C is provided with the color board so all you have to do is type in the changes to CHIP-8 and start writing your programs in eight colors!

If you're interested in becoming an expert VIPper, the machine language code is detailed in the **VIP Instruction Manual**, beginning on page 106. And you can turn to the accompanying manual, the **User Manual for the CDP1802 COSMAC Microprocessor** for programming techniques and lots of other lower level information.

The important thing is for you to start doing something with your VIP. Don't worry about "blowing your program up"—the worst you can do to the VIP, without drowning it or beating it with a hammer or throwing it out a window, is to mess up CHIP-8. And all you have to do to fix that is to reload it from tape or from the keyboard.

If you write a really interesting game, or develop someone else's game so it's more interesting, drop us a line. RCA is looking for programs to provide for VIP users—and who knows? You could be receiving payment for the fun you dream up!