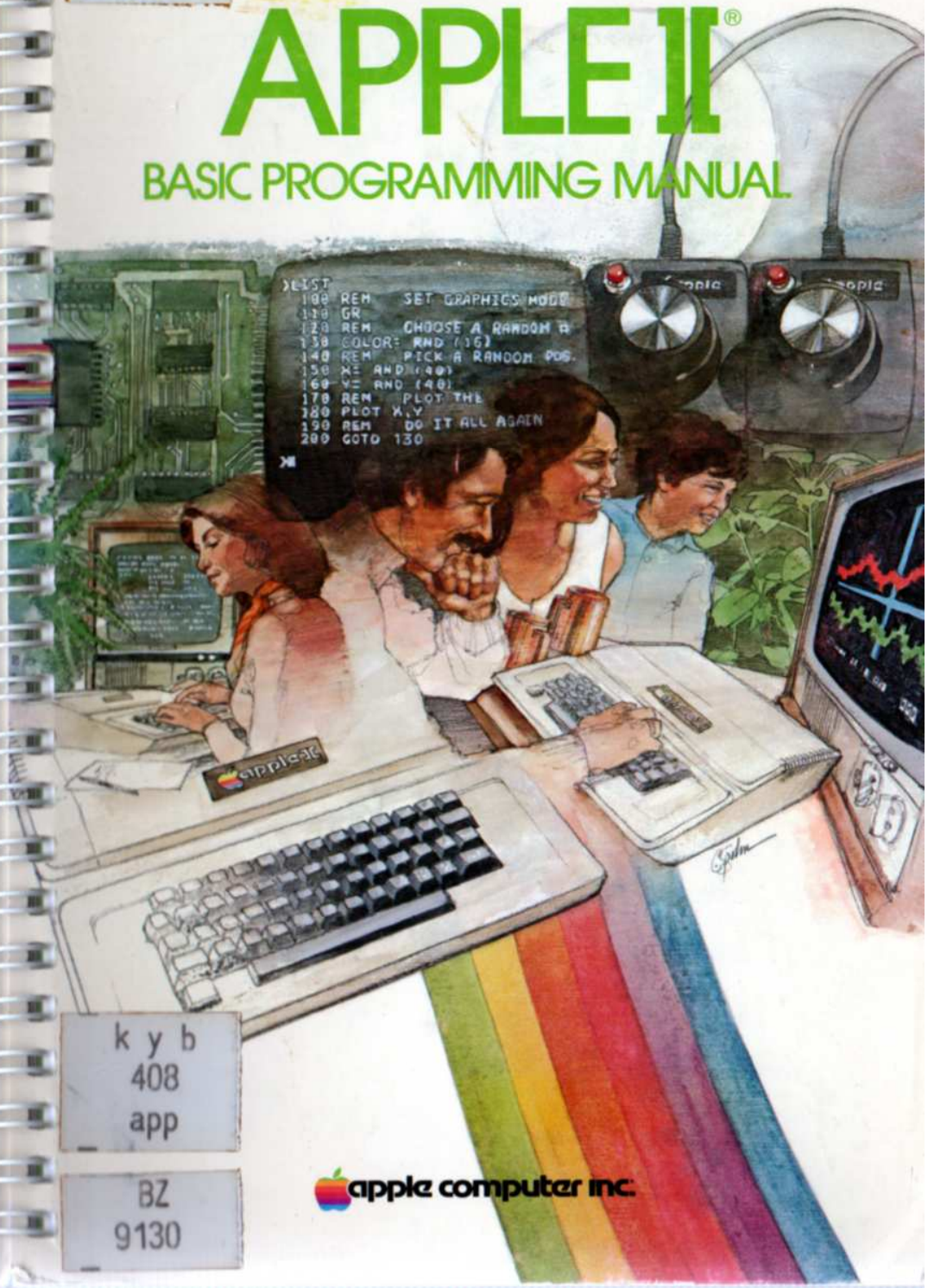
APPLE II®

BASIC PROGRAMMING MANUAL



apple computer inc.
10260 Bandley Drive
Cupertino, California 95014

apple computer inc.

Published by
APPLE COMPUTER, INC.
10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

Written by Jef Raskin

 Reorder Apple Product #A2L0005X

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

CHAPTER 2

BEGINNING BASIC

CHAPTER 3

ELEMENTARY PROGRAMMING

CHAPTER 4

STRINGS, ARRAYS AND SUBROUTINES

APPENDICES

INDEX

CHAPTER 1
INTRODUCTION

AN APPLE TODAY

keeps the doldrums away. This manual will show you how to plug in your APPLE II (easy) and be a guide as you learn to program it (also easy). If you are an Old Hand at programming,you will find some new features and conveniences in APPLE BASIC that make programming a lot more fun. If you are a Newcomer to programming, you will also find many features and conveniences in APPLE BASIC that make programming a lot of fun. But, if you are a Newcomer, be warned that programming, though not difficult, can only be learned by *doing*. More will be said on this topic later, but remember—this is a book to be used, not merely perused.

If you purchased your APPLE II from an authorized APPLE dealer, they will be willing to let you set your APPLE II up in their shop, and make sure you know how to set it up at home. If you received it as a gift or through the mail, it is not difficult to hook up—it is as easy as setting up a stereo system and no technical knowledge is needed at all.

If you have not already done so, please take a few minutes to complete and mail your **Owner/Warranty Registration Card**.This Registration Card will register your APPLE II with the factory, give you membership in the **APPLE SOFTWARE BANK**, and include you in the list of APPLE II owners. If you don't send us this card you will not receive any newsletters, information about new accessories for your APPLE II, nor any of the other information that is frequently mailed to APPLE II owners. So please mail in the completed card.

WHAT YOU WILL NEED

This manual was in the accessory box. This box should also contain:

1. The power cord (the cord that plugs into the outlet on the wall).
2. A set (2) of controllers (the boxes with knobs).
3. A cable to connect the APPLE to a tape recorder. This cable has two plugs on each end.
4. Some cassette tapes. These tapes contain programs for the APPLE.

In addition to the APPLE II itself and the contents of the accessory box, you will need these two thing *(neither are supplied)*:

1. A Cassette recorder. If you do not own one, we recommend the Panasonic RQ309 (under $40).
2. You will need *one* of the following items:
   a. A color TV monitor and a cable that has a phono plug (also called a male RCA-type connector) at one end and something to match the monitor at the other end. The dealer that sells you the monitor can supply the cable.

or

   b. An ordinary home color TV and an "RF Modulator" with the connecting cables. The RF Modulator changes the signal put out by the APPLE II so that it matches what your TV expects. A number of Modulators are available. There is one made especially for the APPLE II called the SUPERMOD II. Your computer dealer probably sold you one, or, if not, it can be ordered from—

M&R Enterprises
P.O. Box 61011
Sunnyvale, CA 94088

The Modulator has instructions on how to hook it up. Your TV's ability to receive normal programs will not be diminished (or enhanced) by having the APPLE II hooked up to it.

HOOKING UP THE TV

If you have a color (or black and white) *monitor*, just connect the appropriate cable from the jack marked "**VIDEO OUT**" (on the rear of the APPLE II) to the input of the monitor.

If you have an ordinary TV, you will have to install an RF modulator. Open the top of the APPLE II by pulling straight up on the back of the lid using both hands, one on each side. Then install the modulator following the directions that come with the modulator.

PLUGGING IN THE CONTROLLERS

With the lid open, plug the controllers' rather delicate plug into the **GAME I/O** socket located in the right-rear corner (front view) of the APPLE II board. Be very careful and make sure that all the pins go into the socket. The white dot should be toward the *front* of the computer.

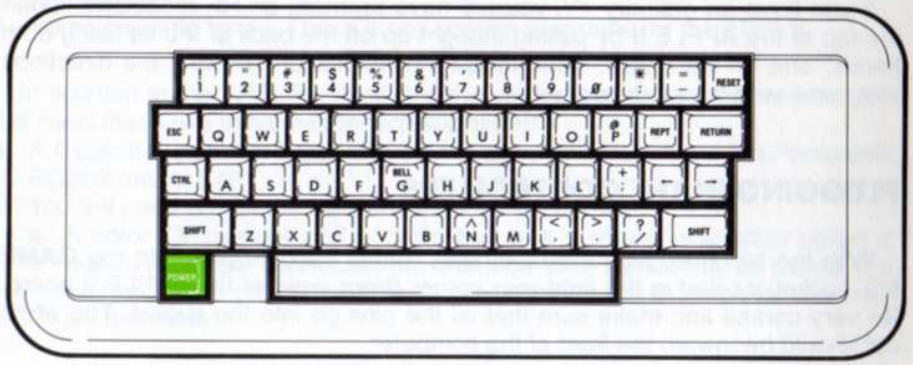
THE CASSETTE RECORDER

Use the supplied cable (the one with two plugs on each end) to connect the APPLE II to your cassette tape recorder. Connect one black plug to the *MIC* or *MICROPHONE* jack on the recorder, and the other black plug (on the opposite end of the cable) to the jack on the back of the computer marked "**CASSETTE OUT**". Connect the grey plug on the recorder end to the *EAR* or *EARPHONE* or *MON* or *MONITOR* jack on the recorder (different brands use different words) and the grey plug on the computer end to the jack marked "**CASSETTE IN**". "OUT" means "out of the computer" and "IN" means "into the computer." Now the cassette recorder is hooked up.

Now close the top of the APPLE. Plug the APPLE end of the power cord into the APPLE (on the rear of the APPLE, next to the Power switch), and the other end into a three-prong grounded outlet. Now the APPLE II is completely set up and you have only to turn the page to begin exploring the fascinating world of personal computing.

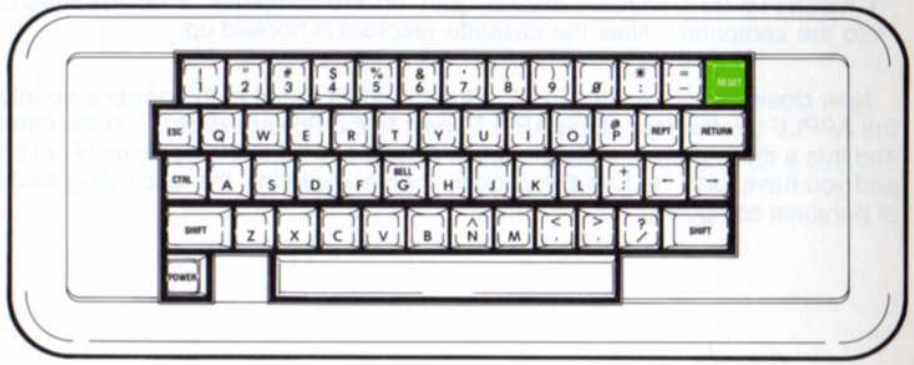
THE APPLE KEYBOARD

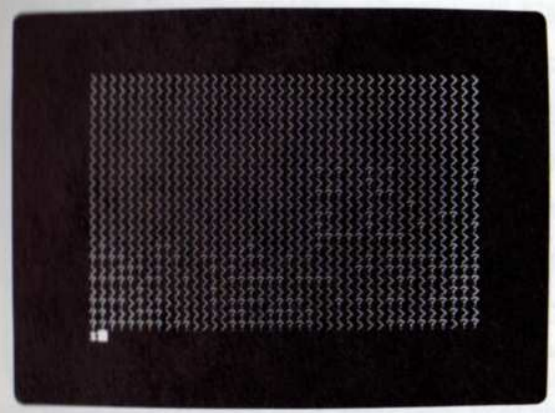
The first thing to do, now that all the connections have been made, is to turn the APPLE on. The switch is on the back of the computer. Push it into the upward position. You will be rewarded by the light at the bottom of the keyboard marked "POWER" coming on. This light is not a key, and cannot be depressed.



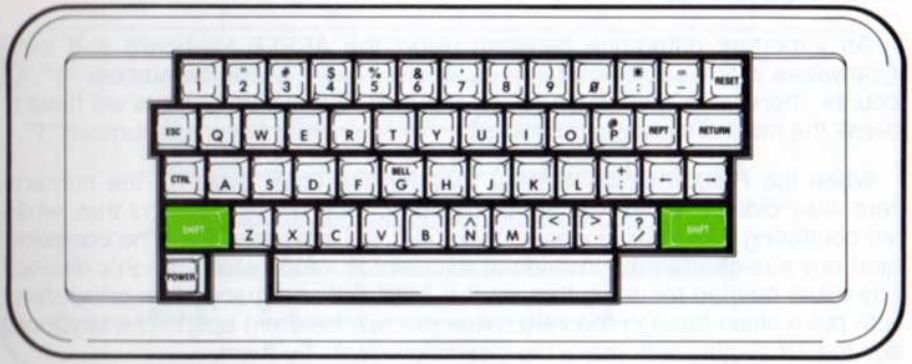
Don't be concerned with what appears (or doesn't appear) on the TV screen at this point. So that you will be able to hear the computer, turn the **TV's** volume control all the way down. The TV's speaker is not used.

Whenever you turn the APPLE on, you have to press the RESET key located in the upper right corner of the keyboard. Try it now. The APPLE will (if everything is OK) go "beep" when you *release* the RESET key. The screen should show an asterisk (*) in its lower left hand corner, with a blinking square to the right of the asterisk. The blinking square is called the **cursor**. At this point, don't worry about the rest of the screen, nor about what colors are showing, if any.





Study the keyboard. If you are familiar with standard typewriters, you will find a few differences between the APPLE keyboard and a typewriter keyboard. First, there are no lower case letters. You can get only capital letters on the APPLE II. This is all you need for programming.



Using the diagram, locate the two SHIFT keys on the keyboard. The reason the keyboard has the SHIFT keys is to allow for nearly twice as many characters with the same number of keys. A keyboard with a separate key for each character would be very large, making it hard to find any desired key.

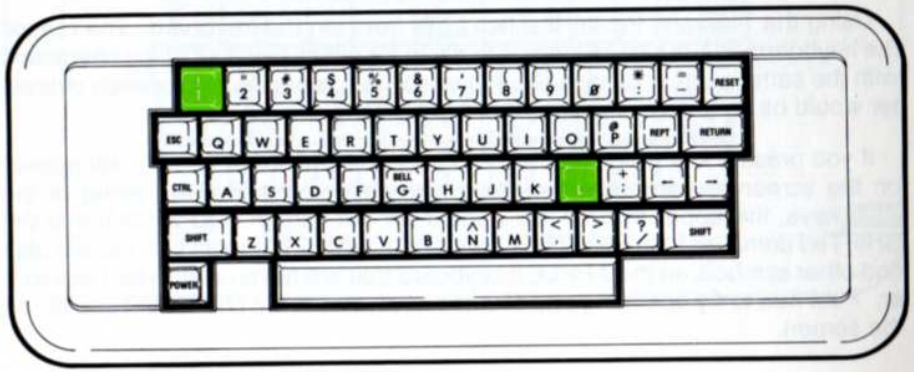
If you press a key which has two symbols on it, the lower symbol will appear on the screen. If you press the same key while you hold down either of the SHIFT keys, the upper symbol will appear on the screen. You will find that the SHIFTed coma and the SHIFTed period are < and > respectively. You will also find other symbols on the APPLE II keyboard that are not on a standard typewriter. Feel free to try operating any of these keys. Watch the characters appear on the screen.

If there is no upper symbol on a key, then holding the SHIFT while the key is pressed has no effect. There are two exceptions: the M key and the G key.

The SHIFTed M key gives a right hand square bracket (]) . The G key has the word "BELL" above the "G". But SHIFT G does not put a bell on the screen, it just puts a "G" there. The meaning of the word "BELL" on the G key will be explained later.

An important difference between *using* the APPLE keyboard and most typewriters is that you cannot employ a lower case "L" for the number "1". Of course, there is no lower case "L" on the APPLE, but some typists will have to break the habit of reaching for the letter "L" when they mean the number "1".

When the Hindu mathematicians invented the open circle for the numeral zero, they didn't use the Roman alphabet. So they chose a symbol that, while not conflicting with **their** alphabet, looks just like our letter "O". The computer (and any straight-thinking individual) will want to keep zeros and oh's distinct. The usual method for doing this, on the APPLE II and many other computers, is to put a slash through the zero. Now you can tell them apart. The keyboard and the TV display both make the distinction clear. Try them.



After a bit of typing, the screen tends to get full of stuff. To clear the screen, you need to use the key marked ESC. ESC stands for the word "ESCape." Press the ESC key, and then type an "at" sign (@) which is obtained by holding down either SHIFT key and pressing the key marked P. You have to operate three keys to clear the screen. First press the ESC, then, *while holding down* the SHIFT, press the P key. Instant gratification: the contents of the screen promptly disappear.

KEYBOARD NOTATION

At this point we will introduce a simple notation.

As you have seen, when a key is to be pressed, such as the key for the letter "H", that key's symbol will be shown H .To express pressing several keys in succession, we will simply list the keys in the order to be pressed: H E L L O .

On occasion, you will need to hold down one key while pressing another key. For example, to type a dollar-sign ($) you must *hold down* the SHIFT key *while* you press the 4 key. Whenever this dual action is required, we will show the symbols for both keys, one above the other.

SHIFT

S
4

The above key is to be held down *while* the bottom key is pressed. Here's how to clear the screen using the new notation:

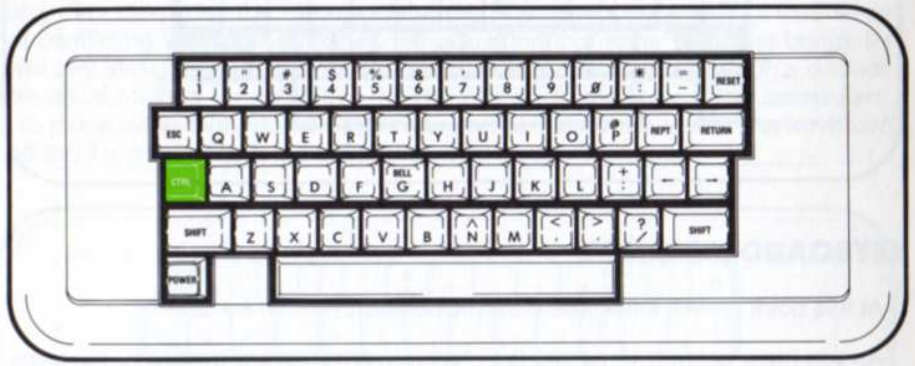
SHIFT

ESC | @ P | RETURN

Try it.

CONTROL, AND OTHER UNSAVORY CHARACTERS

When you press the 5 key, the numeral 5 appears on the TV screen. I'm sure you believe this is true, but try it anyway. If you *hold* the SHIFT key down *while* pressing the 5 key, a percent sign (%) should appear on the screen. Does it? The SHIFT key permits some of the keys of the keyboard to have two different functions. Several of the keys also have a *third* function. The third function is obtained by holding the CTRL key down *while* other keys are pressed. " CTRL" stands for the word "**ConTRoL**." Instead of putting new characters on the screen when you use the CTRL key, the computer responds by performing certain actions. **Control characters never appear on the screen.**



Hold the CTRL key down and press G

CTRL

BELL
G

It doesn't go "ding," but it does go "beep." Whenever the computer wishes to call your attention to something, it will sound the beeper. Control G is called "BELL" for historical reasons. The present keyboard design is based on that of the Teletype, and on that venerable machine, Control G rings a real bell.

Now type

CTRL

RESET | B | RETURN

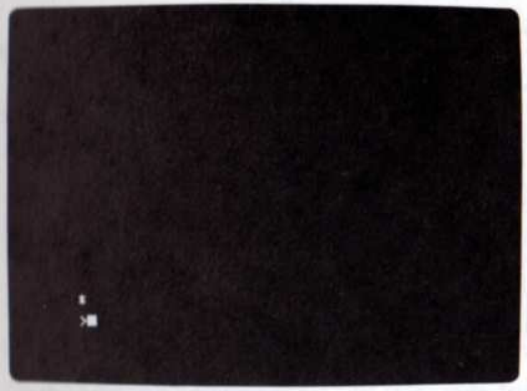
By our conventions, this means to press the RESET key , hold down the CTRL key while pressing the B , and then press the RETURN key . As you can see, our new notation is easier to read than the written instructions.

When you press

CTRL

RESET | B | RETURN

a right-pointing arrowhead (>) should appear at the bottom of the screen. The blinking square (called the **cursor**, remember?) will be to its right. If this doesn't happen the first time, try again.



Another key that is not usually found on typewriters is the

REPT

which stands for "**REP**eaT." Holding down the REPT key while you press

any other key just makes that key's character appear repeatedly on the screen. Experiment with it. If you happen to press

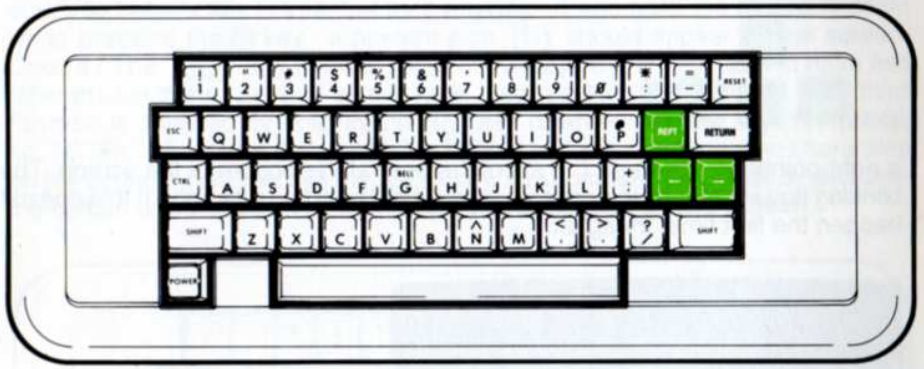
RETURN

you will sometimes get a "beep" and the message

```
*** SYNTAX ERR
```

will appear on the screen. For the time being, ignore this message.

The only keys left unmentioned are the right-and left-pointing arrows on the keyboard. They move the cursor to the right and the left. They will be explained more fully later. Test out these keys and any others you can find. There is nothing you can do by typing at the keyboard that can cause any damage to the computer. Unless you type with a hammer. So feel free to experiment. With your fingers.



GETTING INTO BASIC

To put the APPLE II into a mood to be receptive, press the keys

CTRL + RESET, then B, then RETURN

which not only gets you the **prompt** character (the right-pointing arrowhead: >) and the blinking square **cursor**, but also puts the APPLE II into the **BASIC** computer language. (More about what this means later.) For practice, turn the computer off; then turn it back on, and get it back into BASIC. If you don't get the arrowhead the first time, just try it again. Notice that you don't have to type in either the prompt character or the cursor. They are both generated by the computer for your use.

Now that you are "in BASIC" or have BASIC "up" (as they say), you are ready to set the volume control on the tape recorder.

SETTING THE TAPE RECORDER

When you play a tape recorder, it is usually with the intent of making sounds that you can hear. If it is too soft, you miss some of the words or music. If it is too loud, it is annoying. When you play the tape recorder into the APPLE, it is with the intent of putting the tape's information into the computer. If the volume setting is too soft, the APPLE will miss some of the information, and it will complain by giving an error message. If the volume setting is too loud, the APPLE will also complain.

To find the right volume setting, you will use a trial-and-error method. You will play a tape softly to the computer and see if the information got in OK. If it doesn't work, you will try the tape again, a little louder this time. If *that* doesn't work, you will make it a little louder still. Eventually the volume will be just right for the APPLE and it will say so.

Put the computer in BASIC and clear the screen for action:

CTRL + RESET, then B, then RETURN, then ESC, then SHIFT + P, then RETURN

Place the tape marked "COLOR DEMOS" into your recorder. For *each* position of the volume control you are going to do the following:

1. Rewind the tape to the beginning.
2. Start the tape playing.
3. Type:

L O A D RETURN

When you do this, the cursor will disappear. It may take up to 15 seconds before something happens. There are these possibilities:

a. The message `*** SYNTAX ERR` appears.
b. Nothing at all happens.
c. The message `ERR` appears (with or without a beep).
d. The message `*** MEM FULL ERR` or `ERR*** MEM FULL ERR` appears (with or without a beep).
e. The computer goes "beep" and nothing appears.

In case **a.**, do not reset the volume control, but go back to step **1.** where you rewind the tape.

In cases **b.** and **c.**, make sure you waited for 15 seconds before giving up. If there is no prompt character or cursor, and the APPLE does not respond to its keyboard, put the computer into BASIC again:

CTRL + RESET, then B, then RETURN

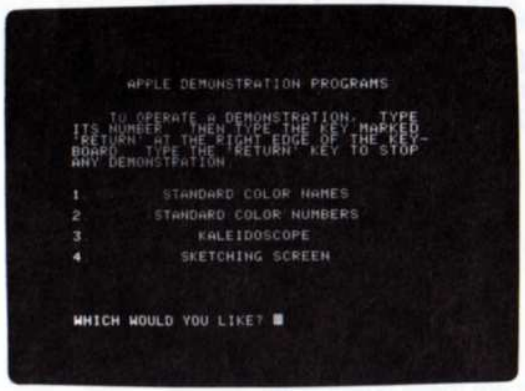
Set the volume control a bit higher and go back to step **1.**

In case **d.**, set the volume control a bit higher and go back to step **1.**

In case **e.**, you are on the right track. When you hear the beep, wait another fifteen seconds. Either you will get an error message (case **c.** or **d.**), or the prompt character (>) and the blinking cursor will reappear. If they do reappear, stop and rewind the tape. Then type

R U ^N RETURN

The screen should look like this:



Computerniks call this list of numbered descriptions a "menu." It works like a menu at a roadside cafe. If you want scrambled eggs with hash brown potatoes, toast, jelly and coffee you can just say,"I'll have a number 5." Try selecting one of the color demonstrations by typing its number (followed by a RETURN, of course). When you are viewing one of the demos, just press the space bar to

get back to the "menu."

A HELPFUL HINT

What is it that the computer finds so interesting about these tapes? Listen to one of them. It's not music to your ears. Yet you can recognize some of the sounds the computer listens for. The information starts with a steady tone. Then there is a short "blip" followed by more of the steady tone. The tone is at 1000 cycles per second. This pitch is just below the C two octaves above middle C. After the tone comes a burst of sound rather reminiscent of a rainstorm.

When you are used to the sound of a good tape, you can quickly check a tape by ear to see if it is a computer tape or not. If you can tell what the tape contains by listening to it, you are a mutant, and will go far in the computer world.

STOPPING THE COMPUTER

To stop the computer, type

CTRL + C, then RETURN

This will cause the prompt character and blinking cursor to appear. The prompt character tells you that it is OK to proceed with typing information to the computer. That is why it is called the **prompt** character: it "prompts" you to type something.

Once the computer is stopped, it may be started again by typing

R U ^N

(and, of course, a RETURN, but you hardly need to be told that anymore. In fact, you won't be, from now on.)

Use

CTRL + C

to stop the computer, and

RUN

to start it again. Try this a few times.

WHAT TO DO IF YOU HIT RESET BY ACCIDENT

Sometimes when you reach to press the RETURN key you may accidentally strike the nearby RESET key — or you may hit RESET for some other reason. To get back into BASIC after hitting the RESET key, type

CTRL

C

This will get you the prompt character (>) back, and you will not have lost any information you may have read in from the cassette tape. You could get the prompt character back by typing

CTRL

B

but this would cause any information stored in the computer to be lost. When you try these features, remember that we are no longer mentioning the required RETURN except for an occasional reminder.

THE USUAL PROCEDURE FOR LOADING TAPES

1. Make sure the computer is in BASIC
2. Rewind the tape
3. Start the tape playing
4. Type LOAD
   After you hit RETURN the cursor will disappear. Nothing happens from 5 to 20 seconds, and then the APPLE beeps. This means that the tape's information has started to go into the computer. After some more time (depending on how much information was on the tape, but usually less than a few minutes) the APPLE beeps again and the prompt character and the cursor reappear.
5. Stop the tape recorder and rewind the tape. The information has been transferred, and you are finished with the tape recorder for the time being.
6. Type

RUN

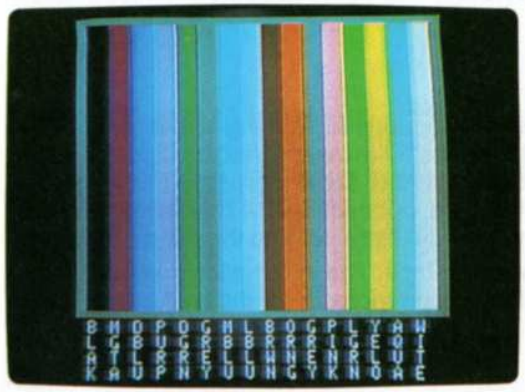
and your program will begin to execute.

Computerniks use many different words to describe the process of taking information from a tape and putting the information into the computer. The computer is said to "read" (pronounced "reed") the tape. The information on the tape is said to be "entered" or "read" (pronounced "red") into the computer. The act of reading a tape is also called "loading" a tape into the computer and the information on the tape is said to be "loaded into" the computer. All these expressions are ways of saying the same thing.

SETTING THE TV COLOR

If the "menu" is not on your TV screen, follow the USUAL PROCEDURE for loading the tape marked "COLOR DEMOS." One of the items on the menu is called COLOR NAMES. We will use this DEMO to set the TV color. Type in the number of the COLOR NAMES DEMO, 1 , and press RETURN . A number of bars of light (perhaps in color) will appear. Under each bar is a four letter abbreviation of a color name. The full names are:

0 BLACK	8 BROWN
1 MAGENTA (a slightly bluish red)	9 ORANGE
2 DARK BLUE	10 GREY
3 PURPLE (a light purple, lavender)	11 PINK
4 DARK GREEN	12 GREEN
5 GREY	13 YELLOW
6 MEDIUM BLUE	14 AQUA
7 LIGHT BLUE	15 WHITE



If you have a black-and-white television, adjust the brightness and contrast until you are pleased. Of course, if the picture is flipping over, stop it the way you would for any TV show. If you have a color set, a bit more work is necessary.

Remember that this color business is quite subjective, and that you can do whatever you want with the color. The following instructions will give the picture that *we* like, using the standard colors. But it's *your* eyes you've got to please. Besides, the optimum settings will vary with different amounts of room light as well.

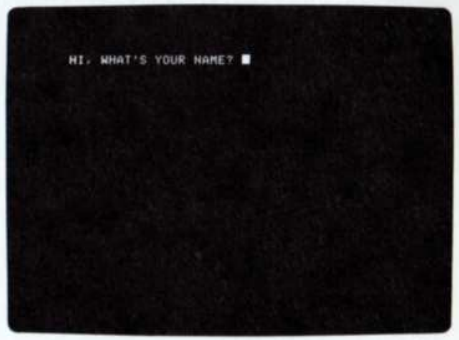
Turn off any Automatic Color switch. On some sets it is marked "AUTO COLOR" or simply "AUTO." Turn the TV set volume control all the way down (but don't turn the set off). Four controls are now important: Picture, Brightness, Color and Hue. Some sets have a knob marked "Contrast" rather than "Picture," but it does the same thing. Turn the Picture control to its dimmest position, and then turn down the Brightness until the background *just* goes completely dark. Turn the Color control to the middle of its range. Now turn up the Picture control to make things brighter. Do not make it so bright that the colors "spill" off the edges of the bars too much.

Now adjust the Color knob. At one extreme, all color is lost and the picture is black and white. This setting is handy when you are just showing text on the screen. Adjust the Color control until the colors are intense but not "blooming" or spilling into one another. Lastly, adjust the Hue knob until all the colors agree with their names. Purple, Pink and Yellow are especially sensitive indicators. Also, make sure that the three Blues are distinct.

When the TV set's colors are OK, hit the space bar and the menu will reappear. Now try DEMO 2, which shows the color bars with their code numbers. Also try the other demonstrations. You'll never believe how talented your TV is until you replace the local stations with your APPLE II.

PLAYING BREAKOUT

Put the tape labeled "BREAKOUT" into your recorder. Use the USUAL PROCEDURE for getting the tape loaded, of course. The screen will look like the photo on the left when you **RUN** the program. After the game announces itself, the screen suddenly changes to look like the photo on the right.



When asked, type your name, and then hit RETURN, as usual. We will type, for example, (as it appears on the screen):

```
MR. APPLESEED
```

The APPLE will respond with a question:

```
STANDARD COLORS, MR. APPLESEED?
```

Before answering this earthshaking question, we should mention a few things that can go wrong. If you put in a name that is just too long for the poor game program to handle, the computer will say

```
*** STR OVFL ERR
STOPPED AT 10
```

This stands for "**STR**ing **OV**er**FL**ow **ERR**or," which is just the computer's way of saying, "Enough, already!" Don't be alarmed, just type

```
RUN
```

If you accidentally hit the RESET key instead of the RETURN key (it can happen), the screen will light up. Don't panic. You know what to do. Hit

CTRL

C

of course. Don't forget that we are no longer mentioning RETURN every time it is necessary.

Try deliberately making some errors, such as giving the computer a name that's too long, or "accidentally" hitting the RESET key , so that you can get some confidence in your ability to recover from errors.

Meanwhile, back at the BREAKOUT program, MR. APPLESEED had been asked if he wanted the standard colors. This time around he does. So type

```
YES
```

and be ready with the game controller.

"Which controller?" you ask. Try them both. One of them will make the paddle (the blue rectangle at the left of the screen) move up and down. The idea is to bounce the ball off the paddle. You lose the ball if it hits the left edge of the playing area. You get one point for hitting bricks in the first row, two for bricks in the next and so on.

When you have run out of balls, or have won the game (by getting a score of 720) you will be asked the question

```
SAME COLORS?
```

To play again using the same colors, just type

```
YES
```

Of course, you are free to say

```
NO
```

if you wish, and see what happens. But we'll let you figure that one out. Have fun.

CHAPTER 2
BEGINNING BASIC

BEGINNING BASIC

As you know, you get BASIC by typing

If you are already in BASIC, of course, you needn't bother.

Now that you have the prompt character (>) and the blinking cursor on the screen, you are ready to begin using the BASIC language. Type

```
PRINT "HELLO"
```

and the computer will print the word "HELLO" on the next line. If it didn't, ask yourself this question: "Did I forget the RETURN ?" If you make a mistake, such as omitting one of the quotes or misspelling the word "PRINT", you will get this error message:

```
*** SYNTAX ERR
```

If you forget *both* quotes, the computer will print a zero (you can tell it's a zero by the slash):

```
0
```

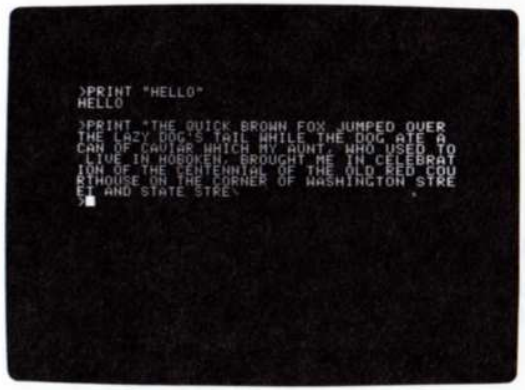
The **statement**

```
PRINT "HELLO"
```

is an instruction to the computer telling it to display on the screen all the characters between the quotes—in this case a word of greeting. You can place any message you wish between the quotes. However, if you try to **PRINT** something that is much longer than 100 characters, you may get the message

```
*** TOO LONG ERR
```

If you type much beyond 240 characters, the computer will start to beep, then give you a backward slash and let you start over again.



Now try the statement

```
PRINT "150"
```

The computer obediently prints the number 150 on the next line, as expected.

But type

```
PRINT 150
```

and the computer again prints the number, without any fuss or error message about the missing quotation marks. In fact, the APPLE II will let you PRINT any integer between 32767 and −32767 without enclosing it in quotes.

Without further study, the APPLE II can be used as a simple-minded desk calculator. This calculator only operates with integers (numbers such as 67 or 935 or −72, but not 3.14 or 56.9). There is a program on cassette tape to handle numbers with decimal points (AppleSoft Floating Point BASIC). This program is for APPLE II systems having 16K or more memory and is covered in a separate manual.

Try this on your APPLE:

```
PRINT 3+4
```


The answer, 7, appears on the next line. The APPLE can do six different elementary arithmetic operations:

1. **ADDITION.** Indicated by the usual plus sign (+)
2. **SUBTRACTION.** Use the conventional minus sign (−)
3. **MULTIPLICATION.** This is more difficult. Many people use an "X" to represent multiplication. This could be confused with the letter "X." Some people use a dot (.), but this could be confused with a period or a decimal point. So the APPLE uses an asterisk (*). To find 7 times 8 (in case you don't remember), just type

```
PRINT 7*8
```

and have your memory jogged.

4. **DIVISION.** As is customary, use a slash (/). To divide 63 by 7, type

```
PRINT 63/7
```

and the correct answer will appear.

Try dividing 3 by 2. The correct answer is one and one-half. But the computer stubbornly insists that the answer is one! Try it. This is because the computer only gives you the *number of times* the divisor goes into the dividend. To get the remainder (remember the remainder, from grade school?), you have to use the next arithmetic operation.

5. **MOD.** Say you wanted to divide 13 by 5. You know to type

```
PRINT 13/5
```

This will give you an answer of 2. Try it. Two times five, however, is only 10, not 13. There is a remainder of 3. If you type

```
PRINT 13 MOD 5
```

the computer will print the remainder of 3. Thus the expression "**13 MOD 5**" means "find the remainder upon dividing 13 by 5". "**MOD**", by the way, stands for the mathematical term "MODulo," and a mathematician would say "13 modulo 5 is 3." But all computer nuts just say "MOD".

You may wonder if you need to skip spaces, as in **13 MOD 5**. Try it and find out. **Very often it will be faster for you to try something out on the computer than to look it up in the manual.** Besides, the computer is always right, and the manual could be wrong.

Another thing we should point out is that you can use a number of arithmetic operations on the same line. For example, it is legal to say

```
PRINT 3+5+9+4
```

The exact rules governing such usage will be given later, but you can experiment with it now if you wish.

6. EXPONENTIATION. It is often handy to multiply a number by itself a given number of times. Instead of bothering to write

```
PRINT 4*4*4*4*4
```

you can substitute the shorthand

```
PRINT 4^5
```

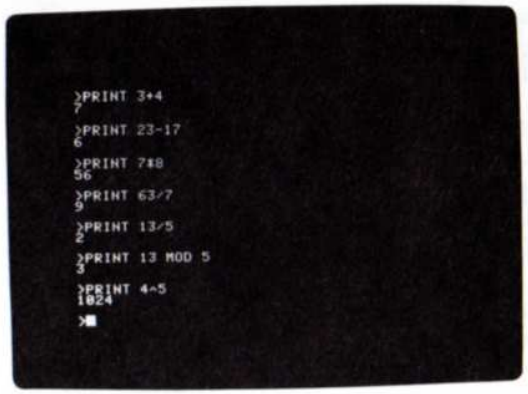
The upward pointing arrow is typed:

SHIFT

^
N

In normal mathematical notation, this would be written with a superscript five, like this: 4^5

If you are not familiar with exponentiation, don't worry. It isn't needed very often.



WHAT'S SPECIAL ABOUT 32767?

Whenever you do something that the APPLE II doesn't like or understand, it gives you an error message. While it is rather abrupt and curt about it (Beep! You goofed!), the APPLE *is* trying to be helpful. The APPLE II has a rather limited range of numbers that it can handle in calculations. The largest number is 32767, and the smallest number is −32767. It can use −32768 internally, but the smallest answer it can *PRINT* is−32767. From now on, we will forget that −32768 exists on the APPLE. Any attempt to calculate a number outside of the range −32767 to +32767 will give you this message:

```
*** >32767 ERR
```

Some neat ways of getting this error message are:

```
PRINT 3/0
PRINT 6^6
PRINT 56789
```

These statements will get you error messages because: division by zero is a no-no in mathematics, 6 to the 6th power (6^6) is 6*6*6*6*6*6* or 46656 which is larger than 32767, and 56789 is larger than 32767, too. Getting this error message is not a disaster. Just fix whatever is wrong and carry on.

It is possible, through programming, to handle numbers of any size on the APPLE II. However, the techniques for doing so are outside the scope of this manual.

MORE ABOUT RETURN

So far, you have been hitting RETURN after every line, like a zombie. We thought we might tell you why this button gets so overworked. The reason is simple: without the RETURN , the computer does not know when you have completed the instruction. For example, you might start typing

```
PRINT 4+5
```

If the computer immediately jumped in and printed a 9, you might be upset because you had planned to type

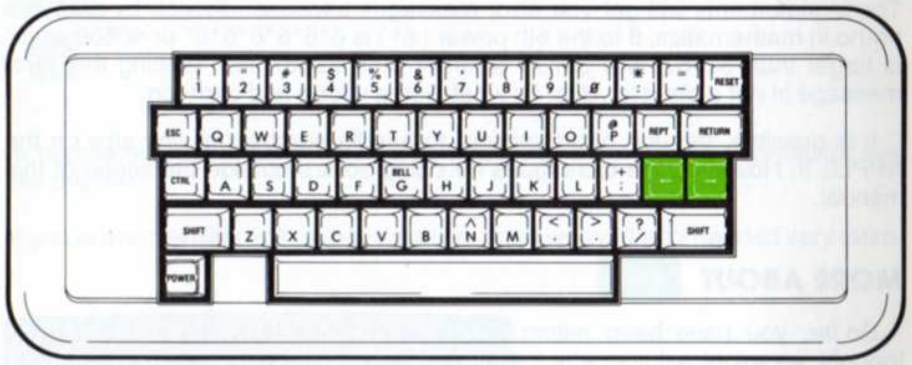
```
PRINT 4+5+346
```

which would have given a different answer entirely. Since the computer can't tell when you have finished typing an instruction, you must tell the computer. You do this by pressing the RETURN key. Since you *always* have to do this after typing an instruction, we have (as you know) stopped mentioning RETURN after every instruction. Hitting RETURN should be a habit by now , if you have been doing all the examples.

We really hope you *have* been trying all the examples. Learning to program is very much like learning to ride a bicycle, play the piano, or throw a baseball. You can read all the books in the world on the subject of bicycle riding, and be a great "paper expert." But all this book-learning is of little help when you actually get on a bicycle for the first time. Once you have learned to ride *through experience* (which can be a bit painful), you can go almost anywhere. The same is true of programming. You can read this manual and think you understand it. But you won't be able to program. Only if you *do each example*, as it is given, will you learn to program. That's the truth.

EASY EDITING FEATURES, or: WHAT TO DO BEFORE YOU HIT RETURN

No one is a perfect typist. We make mysteaks (Oops. See what I mean?). The APPLE II has several features that aid in correcting errors, thereby saving you the effort of retyping a whole line for each goof. This is where the left-and right-pointing arrows on the keyboard come in.



The **left-pointing arrow** is rather like the backspace key on a typewriter. A few experiments will make this clear. Type (exactly as shown) the statement:

```
PRINT  COMPUTER"
```

and, as usual, press the RETURN key. The computer will reply

```
*** SYNTAX ERR
```

because of the missing quote. Now if we had typed

```
PRINT "COMPUTER"
```

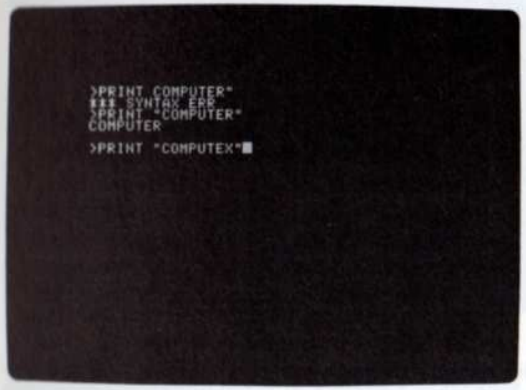
the computer would have responded with

```
COMPUTER
```


Don't believe this manual . Try it. Now, *without pressing* RETURN, type the "mistaken" instruction:

```
PRINT "COMPUTEX"
```

Since you haven't pressed RETURN, nothing has happened yet. As shown in the photograph, the cursor is sitting to the right of the last quote. (Sorry, we can't make the photo blink)



To change

"COMPUTEX"

into

"COMPUTER"

we can use the left-pointing arrow key. Notice that each time you press this key, the blinking cursor moves back (to the left) one space. We will call this key the **backspace** from now on. "Backspace" is also a verb. So backspace the cursor back to the "X". Type an R . As you see, the "R" replaces the "X". *Now* press RETURN

You got

```
*** SYNTAX ERR
```

from the computer? That is because you backspaced *over* the quote. Any character that is backspaced over is *not* sent to the computer when you press RETURN . One solution would be to correct the X by backspacing to it, and then type

SHIFT

R " 2 RETURN

Try it.

It works! Now type this error (don't press RETURN yet):

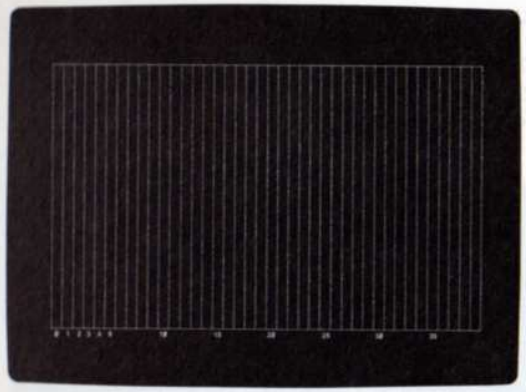
```
PRINT "COMFUTER"
```

Backspace to the incorrect "F." Type "P." This leaves you in the condition shown in the next photo.

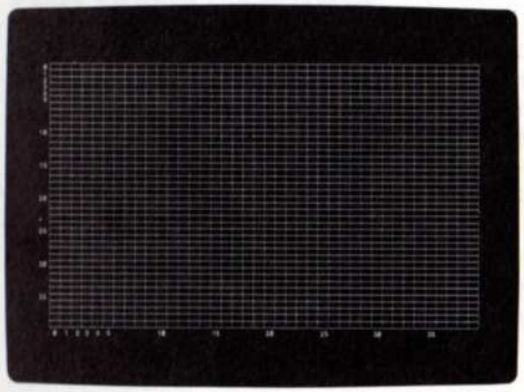
To complete the word, we could (as we did before) retype all the characters backspaced over. There is, however, an easier way. When you press the **right-pointing arrow**, the cursor moves to the right. As the cursor moves to the right across a character, it has the same effect as if that character had been retyped. To complete the correction, then, merely press the right arrow five more times, and then press RETURN. Does it all work? The use of the left and right cursor-moving keys will save you a lot of time. Make a point of using them a number of times on your own "mistakes," so that these keys become familiar.

PUTTING COLORS ON THE SCREEN

To put color graphics on the screen,we need a way to decribe which color out of the sixteen available colors we want and *where* we want it. To specify where a color goes, we divide the screen into forty columns, numbered zero through thirty-nine. The zero column is at the left, and the numbers increase to the right. You may wonder why the numbers didn't go from one through forty,instead of zero through thirty-nine. As you get more experience programming, you will find that the choice we have made is somewhat handier, even though it may not seem that way at first.



The screen is also divided into forty rows, again numbered zero to thirty-nine starting at the top of the screen and moving downwards. These rows cut across the columns, partitioning each column into 40 "bricks" numbered zero (the top brick) through thirty-nine (the bottom one). Those who like formal terminology will recognize that this is merely a system of rectangular Cartesian co-ordinates. Those who don't like fancy talk can just think in terms of columns of bricks.



For the purposes of using the screen colorfully, type the following instruction:

```
GR
```

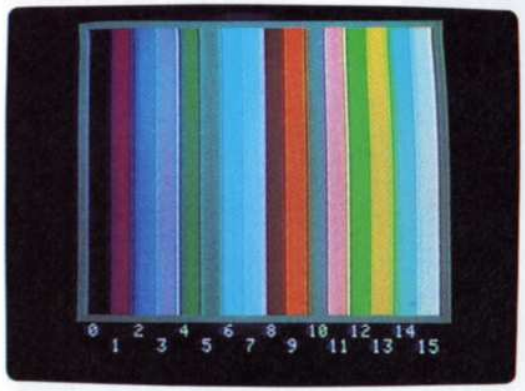
You remembered the RETURN, no doubt. When you use this command the screen wipes itself clean, leaving only four lines for text at the bottom.The "**GR**" stands for **GR**aphics. To get back to things as they were (before you typed **GR**) you use the command

```
TEXT
```

When you type this command the screen will suddenly change to a lot of "at" signs (@). This is normal. Try typing the **TEXT** instruction, and then getting back

to graphics by typing the **GR** instruction.

Before you can place a dot of color on the screen, you must tell the computer what color you want it to be. There are sixteen colors available. You have seen them before. Each one has a *number* from zero to fifteen, as shown in COLOR DEMO 2.



Suppose you want to put a green dot somewhere. You first type

```
COLOR=12
```

This means that any dot (or spot or brick) of color that you place will be green. In fact, *until otherwise instructed*, everything the computer puts on the screen will be green. Except, of course, for the small area reserved at the bottom of the screen for your instructions. To put a spot of color in the upper left-hand corner of the screen (zeroth column, top or zeroth brick), you type

```
PLOT 0,0
```

To put a spot of the same color in the upper *right*-hand corner, you must specify the 39th column, zeroth brick. So type

```
PLOT 39,0
```

Notice that you always give the **column first**. Let's put an orange brick at the lower left-hand corner. First, change the color. Type (Remember—you should really be doing these exercises, not just thinking about them. So, put out your fingers and type):

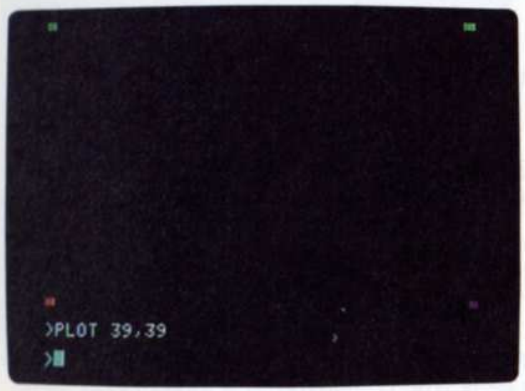
```
COLOR=9
```

Nothing happens on the upper, graphic portion, of the screen (even if you *did* remember to hit RETURN). But the computer remembers that when you next **PLOT** something, it will be in orange, not in green. Now that we have chosen the color, we can put a dot in the lower left-hand corner. That's the zeroth column, and the thirty-ninth brick.


```
PLOT 0,39
```

Did it work? Did you forget to type RETURN ? Is orange your favorite color?

Now put a magenta dot in the lower right-hand corner. Figure it out for yourself.



PLOT ERROR MESSAGES

There are two error messages that can easily turn up when you are using the **PLOT** statement. You already know that if you typed

```
PLAT
```

or

```
PLOP
```

instead of

```
PLOT
```

you would get the message

```
*** SYNTAX ERR
```

The first new error message occurs when you write a number higher than those permitted for coordinates in a **PLOT** command. Type

```
PLOT 13,85
```

and you get the message

```
*** RANGE ERR
```

This message means that you have tried to plot a point out of range and off the screen. The highest numbers you can use in a **PLOT** statement are 39 for the first coordinate, and 47 for the second. Use of numbers over 39 for the second coordinate, as in a statement such as

```
PLOT 20,45
```

will just give you peculiar characters in the text area at the bottom of the screen.

If you try to use *negative* values in a **PLOT** command, you get a somewhat surprising error message. The statement

```
PLOT 22,-8
```

will give the seemingly illogical message

```
*** >255 ERR
```

Don't worry, it is just the APPLE trying to tell you that you used a negative number in a **PLOT**. Other mistakes can also give you these two messages, but those will be discussed later.

DRAWING LINES

Suppose you want to draw a light blue horizontal line from the fifth column to the ninth column at the 14th brick level. You could type

```
COLOR=7
PLOT 5,14
PLOT 6,14
PLOT 7,14
PLOT 8,14
PLOT 9,14
```

Notice that the joints between adjacent bricks do not show, and it looks like a continuous line. However, there is an easier way to do horizontal lines. There had better be. Suppose you want to draw a light green line across the middle of the screen. Using the long way, it would take forty typed statements:

```
COLOR=12
PLOT 0,20
PLOT 1,20
PLOT 2,20
```

and so on, until

```
PLOT 39,20
```

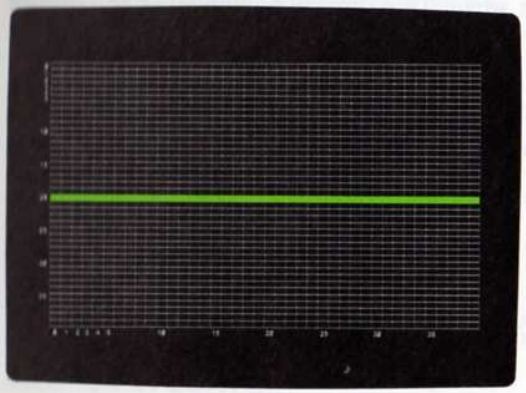
The easier way is this: Just type

```
COLOR=12
```



```
HLIN 0,39 AT 20
```

Press the RETURN key, and there you have it: an instant **H**orizontal **LIN**e from column 0 to column 39 at the 20th brick level.



NOTE: The grid shown on the screen is for illustrative purposes only, and does not appear on your screen.

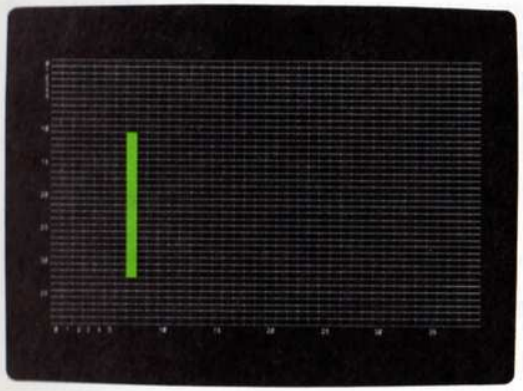
Now try to place a purple line from the 19th to the 28th column at the 18th level. Try a few others. Doing about 6 different horizontal lines should give you the hang of it.

Notice that when you put a colored dot or line at the same location as an existing dot or line, the new color takes over, and the old color disappears. To clear the screen of all graphics at once, use the **GR** command.

There is a provision for automatic vertical lines similar to that for horizontal lines. To draw a light green vertical line from the 11th to the 32nd row at column 7, we type:

```
COLOR=12
VLIN 11,32 AT 7
```

Try this statement.



NOTE: The grid shown on the screen is for illustrative purposes only, and does not appear on your screen.

Practice making several more vertical lines by changing the numbers for the rows and column. You can test your proficiency with both horizontal and vertical lines by drawing a magenta border around the screen in five statements. Then put a green cross on the screen. Try drawing some lines with **COLOR = 0**. Play with **PLOT**, **HLIN** and **VLIN** for a while. This manual's usefulness to you will self-destruct in five seconds if you don't experiment with these commands. **Pfffsssss.**

THE GAME CONTROLS

Grab the control that you used in playing BREAKOUT. With the other hand type

```
PRINT PDL(0)
```

and a number should appear. Move the control a bit. Now type

```
PRINT PDL(0)
```

again. Experiment with moving the control and typing

```
PRINT PDL(0)
```

If the number never changes, you've got the wrong control. What are the highest and lowest numbers you can get? What is the smallest change you can make?

You can discover the position of the other control by PRINTing **PDL(1)**. The abbreviation **"PDL"** comes from the word "PADDLE" since these controls are most often used to control "paddles" in games. As we shall see, there are many other uses for these controls.

PIGEONHOLES AND MORE CALCULATOR ABILITIES

On many simple calculators you can save a number for later reference or use. To do this, you put the number into a special place in the calculator—a place we shall call, for now, a pigeonhole. Usually this is done by pressing a key marked "M" for "Memory." On the APPLE II you can do the same thing. For instance, to save the value 77, you type

```
M=77
```

The value, 77, is *not* printed, just stored in the pigeonhole called M. If you now type

```
PRINT M
```

the computer will print the value of M. Try typing the two statements.

Now type


```
M=324
```

and PRINT the value of M. It is 324, right? What happened to the 77? It is gone forever. The pigeonhole can hold only *one* value at a time. When you put a new value in M, the old value is erased.

Type

```
PRINT "M"
```

What happens? There is a big difference between

M

and

"M"

It is just like the difference between these two statements in English:

MICE HAVE FOUR FEET.

"MICE" HAS FOUR LETTERS.

In one case we are referring to little furry things with long tails. In the other case we are referring to the word itself. This is how quotes are used in computerese. When we say

```
PRINT "M"
```

we mean to print the letter itself. When we say

```
PRINT M
```

we mean to print what the letter *stands for*. You would never confuse the *name* of someone you love with the actual person that name stands for.

You can store the result of a computation in a pigeonhole. For example:

```
M=4+5
```

You can see that the answer has been stored by PRINTing the value of M.

You can also use the value of M in further computations. For example, try this on your APPLE:

```
M=5*6
PRINT M+2
```

Is the answer what you expected? Try some other calculations using M.

A simple calculator has one pigeonhole. Computers have hundreds or thousands of pigeonholes. The formal term for pigeonholes is **variables**. But this term is somewhat misleading since pigeonholes don't behave like "vari-

ables" in mathematics. They are much simpler. *Each one is merely a place where one value is stored.* But we will defer to common usage. Just forget the math you've learned. In the APPLE II all variables have the value of zero until you put something into them. To reset all the pigeonholes to zero you type

Note that the instruction

does *not* reset the variables.

A pigeonhole, or variable, can have almost any name that you like, so long as it starts with a letter. For example:

```
SUM=56+34+1523+8
GAMEPOINTS=45
PLAYER2=9
```

Some names are not allowed because they include a word that has a special meaning to the APPLE II. These are known as **reserved words**. One of these words is "COLOR." Thus a variable's name must not have the word "COLOR" in it. Try typing

```
THISCOLOR=6
```

or

```
COLORFUL=9
```

All you get for your pains is an error message. Whenever a variable name gives you the `*** SYNTAX ERR` error message, it means that you have unwittingly included a reserved word in the name. Don't worry. Just choose another name. Names must also be less than about one hundred characters long. Not much of a limitation. When you are choosing names, make them reflect the use to which they are being put. This will make them easier to remember.

Here is a useful trick. Let's say that you had some value in the variable PRICE, and you wanted to increase this value by 5. One way you could do this would be to PRINT the value of PRICE, then add 5 to that value, and finally store the resulting value back in PRICE. For instance:

```
PRICE=28
PRINT PRICE
PRINT 28+5
PRICE=33
```

But see how much easier it is to type

```
PRICE=PRICE+5
```

Try these statements:

```
PRICE=2
PRINT PRICE
PRICE=PRICE+3
PRINT PRICE
PRICE=PRICE*6
PRINT PRICE
PRICE=PRICE/10
PRINT PRICE
```

At the end of this sequence of statements, you will probably have the value 3. Is this correct? Is this what you expected? Try this sequence:

```
PLAYER=55
OPPONENT=11
QUOTIENT=PLAYER/OPPONENT
PRINT QUOTIENT
```

First think what answer you expect, then see if you are right. If you are not, find out why. Lastly, try these statements:

```
HELLO=128
PRINT "HELLO"
HELLO=HELLO/2
PRINT "HELLO"
HELLO=HELLO/2
PRINT HELLO
```

What did you expect? What did you get?

SIMULATING A PAIR OF DICE

Try this on you APPLE:

```
PRINT RND(6)
```

Now, try it about 10 more times. Really. Just try it, then read on.

Each time you typed that instruction into the computer it PRINTed a number. It is not likely that you can predict what number will come up next. If you can, you are either a computer or have ESP. The numbers that were PRINTed were all less than six and none were less than zero. The reason that they are, for most practical purposes, unpredictable, is that you were using the RaN-Dom function. If LIMIT is a positive number, then

RND(LIMIT)

is an integer from zero through LIMIT-1. Thus RND(6) gives 0 or 1 or 2 or 3 or 4 or 5. Similarly the statement

```
PRINT RND(3)
```

would display either a 0 or a 1 or a 2.

To simulate a pair of dice we need two random numbers, each between 1 and 6, inclusive. Since RND(6) gives a number between zero and five, inclusive, it is pretty obvious that

RND(6)+1

gives a random number between 1 and 6, inclusive. The statements

```
PRINT RND(6)+1
PRINT RND(6)+1
```

will simulate the throw of a pair of dice. In the same vein, try the following statements

```
REDDICE=RND(6)+1
WHITEDICE=RND(6)+1
PRINT REDDICE
PRINT WHITEDICE
```

RND is a *function*. A function, in BASIC, is something that takes one or more numbers and then performs some operation on them to yield a single value. The numbers that the function uses are called its *arguments* and are always put in parentheses after the function name. RND is a function that has one argument. The number the function finds is said to be *returned* to the program. The RND function *returns* a random number between zero and one less than the argument, inclusive. This is only true, by the way, if the argument is positive. If you make the argument to the RND function negative, RND does something a bit different. You can find out what it does with just a few experiments.

PRECEDENCE or: WHO'S ON FIRST?

At certain old-fashioned banquets, the people were served their food according to a strict plan: first the guest of honor, then the female guests (in order of the rank of their husbands), then the male guests (in order of rank), and finally the host. No matter where they were seated, the waiter went among them choosing the appropriate persons to be served next. We could say there was a certain *precedence* among the diners. In a simple calculation like

```
PRINT 4+8/2
```

you can't tell whether the answer should be 6 or 8, until you know in which order (or precedence) to carry out the arithmetic. If you add the 4 to the 8, you get 12. If you then divide 12 by 2, you get 6. That's one possible answer. However, if you add 4 to eight-divided-by-two, you have 4 plus 4, or 8. This is another possible answer. Eight is the answer your APPLE II will give. Here's how the APPLE chooses the order in which to do arithmetic:

1. When the minus sign is used to indicate a negative number, for example

−3+2

the APPLE will *first* apply the minus sign to its appropriate number or variable. Thus −3+2 evaluates to −1. If the APPLE did the addition first, −3+2 would evaluate to −5. But it doesn't. Another example is

```
BRIAN=6
PRINT -BRIAN+10
```

The answer is 4. (Notice, though, that in the expression 5−3 the minus sign is indicating *subtraction*, *not* a negative number.)

2. After applying all minus signs, the APPLE then does exponentiations. The expression

4+3^2

is evaluated by squaring three (three times three is nine), and then adding four, for a grand total of 13. When there are a number of exponentiations, they are done from left to right, so that

2^3^2

has the value 64, and not 512.

3. After all exponentiations have been calculated, all multiplications and divisions and MODs are done, from left to right. Arithmetic operators of *equal* precedence are always evaluated from left to right. Multiplication (*), division (/), and MOD have equal precedence.

4. Lastly, all additions and subtractions are done, from left to right. Addition (+) and subtraction (−) have equal precedence.

Let's summarize the APPLE's order of precedence for carrying out mathematical operations:

First: − (minus signs used to indicate negative numbers)

Second: ^ (exponentiations, from left to right)

Third: **MOD** * / (MOD, multiplications and divisions, from left to right)

Fourth: + − (additions and subtractions, from left to right)

Below, you will find some arithmetic expressions to evaluate. With each one, first do it in your head (or with the help of a hand-held calculator, or pencil and paper), and *then* try it on the APPLE. If your own answer is different from the APPLE's answer, try to find out why. We will give only the expressions here. You will have to put a PRINT in front of each one to get its value from the computer.

Unless you have a lot of experience with the way computers evaluate expressions, you should actually *do* these examples. Don't do them all at once and then check with the computer. Do an example by hand and then do it on the computer. *Then* go on to the next one. And so on.

```
3+2
4+6-2+1
8*4
4^2+1
5-4/2
4/2-2
6*-2+6/3+8
4+-2
2^2^3+1
2*2*3+1
2*2+1*3
2*2*1+3
8/2/2/1
8*2/2+3*2^2*1
20/2*5
```


No answers are given in this book. Your APPLE will give you the correct answers.

HOW TO AVOID PRECEDENCE

Suppose you want to divide 12 by four-plus-two. If you write

```
12/4+2
```

you will get 12-divided-by-four, with two added on. But this is not what you wanted. To accomplish what you wanted in the first place, you can write

```
12/(4+2)
```

The parentheses modify the precedence. The rule the computer follows is simple: do what is in parentheses first. If there are parentheses within parentheses, do the *innermost* parentheses first. Here is an example:

```
12/(3+(1+2)^2)
```

In this case, doing the innermost parentheses, you first add 1+2. Now the expression is, effectively,

```
12/(3+3^2)
```

But you know that 3+3^2 is 3+9 or 12. So the expression has now been simplified to 12/12, which is one.

In a case like (9+4)*(1+2), where there is more than one set of parentheses, but they are not "nested" one inside the other, you just work from left to right. This expression becomes 13*3, or 39.

Here are some more expressions to evaluate. Again, if you are not familiar with computers, the few minutes you spend actually working these expressions out and trying them on the APPLE will be very valuable. You will be well repaid for your efforts by being able to use the computer more effectively. Incidentally, these rules for precedence and parentheses hold good for most computer systems anywhere in the world, not just the APPLE II.

```
7 MOD (3+2)
44/(2+2)
(44/2)+2
3+(-2*2)
(3+-2)*2
100/(200/(1*(9-5)))
32/(1+(7/3)+(5/4))
```

(Remember how division works? No remainders!)

CHAPTER 3
ELEMENTARY PROGRAMMING

DEFERRED EXECUTION

No, this section is not on last minute reprieves for condemned criminals. Up to now, when you typed

```
PRINT 3+4
```

and hit RETURN, the computer would do what you told it to do, *immediately*. When a computer performs according to the statement you have given it, it is said to *execute* that statement. Thus, you have been using the computer to do **immediate execution** of each statement you have typed on the APPLE's keyboard.

You are about to learn how to store statements for execution at a later time **(deferred execution)**. To make sure that the computer is cleared, type

```
NEW
```

Like everything else you have seen, **NEW** has to be followed by a RETURN. You tell the computer to **store** a statement by typing a number before the statement when you type it in. For example, if you type

```
100 PRINT 3+4
```

nothing seems to happen, even if you press RETURN. The APPLE II has *stored* the statement. To see that it has stored the statement, you type the instruction

```
LIST
```

Try it. Unless you mistyped something (and probably got a
`*** SYNTAX ERR` for your effort),

```
100 PRINT 3+4
```

appears on the screen. Now type the statement

```
RUN
```

and the answer

```
7
```

appears on the screen. You also get a message saying

```
*** NO END ERR
```

For the time being, just ignore this message, which will appear many times. This message indicates that the computer has finished executing (or attempting to execute) your stored statement. Pretend that it says "***DONE."

Typing **RUN** caused your stored statement to be executed, but the computer has not forgotten the statement. You can **RUN** the same statement as many times as you like. Try it.

What's more, the computer does not forget the stored statement when you clear the screen. Clear the screen as follows:

and type

```
LIST
```

The computer has not forgotten the stored statement. Clear the screen and type

```
RUN
```

The computer faithfully executes the stored statement. Type

```
NEW
```

and then

```
LIST
```

and see what happens. Typing NEW has caused the stored statement to be lost permanently. Type

```
RUN
```

and you get the message

```
*** NO END ERR
```

but nothing else. That is because your old statement has been erased by the NEW statement .

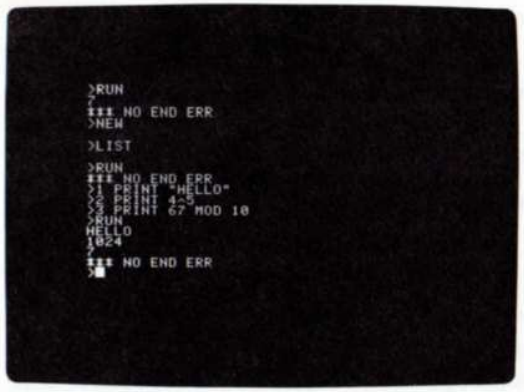
It is possible to store many statements by giving each of them a different number. Try typing this:

```
1 PRINT "HELLO"
2 PRINT 4^5
3 PRINT 67 MOD 10
```

Nothing much has happened so far. But now type

```
RUN
```

and watch the answers appear.



The numbers that we put in front of statements in order to tell the computer to store them are called **line numbers**. The computer stores and executes statements *in order of increasing line number*. To see this in action, erase the statements you stored by typing

```
NEW
```

and then type these statements:

```
1 PRINT "P"
0 PRINT "A"
3 PRINT "E"
2 PRINT "L"
```

Notice that zero is an allowed line number. The highest line number that you can use is 32767. Now RUN these instructions. The results should look like this:

```
A
P
L
E
```

To see what has happened inside the computer, type

```
LIST
```

Notice that you do not have to LIST a set of instructions before you RUN them. It is, however, a good idea to do so.

A set of instructions that is executed when you type RUN is called a **program**. You have just typed and executed a computer program.

The program was meant to print

```
A
P
P
L
E
```

but, it seems, a PRINT statement was left out. How can you add it in? Only by retyping the statements with line numbers 2 and 3, as statements 3 and 4 and adding a new line number 2. To make the corrections type this:

```
2 PRINT "P"
3 PRINT "L"
4 PRINT "E"
```

To see what has happened, LIST the program.

Notice that in whatever order statements are entered, the APPLE II stores them with their line numbers in numerically ascending order. Now RUN this program.

It was a bother to have to retype those statements in order to merely add one in the middle. It is therefore good programming practice to leave some line number room between lines, and before the first line. Type

```
NEW
```

to eliminate that program and put in this one:

```
100 PRINT "C"
110 PRINT "T"
```

When you RUN this program it doesn't quite print the word "CAT" vertically. But *now* you can go back and type

```
105 PRINT "A"
```

LIST and RUN this program. From now on this book will start all programs at a reasonably high line number (100 or more) and increment them by 10.

ELEMENTARY EDITING

Earlier, you discovered that the instruction

```
PRINT PDL(0)
```

would print a number corresponding to the present position of one of the game controls. It took quite a number of PRINTs to discover very much about the control. Now that you can write programs, life is much easier. Clear the computer with a

```
NEW
```

and type

```
100 PRINT PDL(0)
```

Now, each time you type RUN this short program is executed and you get to see the position of the game control. If you are getting tired of the `*** NO END ERR` you can get rid of it by putting in the instruction

```
900 END
```

It doesn't matter what line number the **END** statement has, as long as the line number chosen makes it the *last* statement executed in a program. You have a choice of either putting in the **END** statement, or getting the error message. The program will run essentially the same with or without the **END** statement at the end.

For doing something more than once, the stored program is already saving you some work. Before, you had to retype a whole statement or group of statements. Now, you merely retype

```
RUN
```

Deferred execution confers another advantage. You can modify part of a program and leave the rest the same, without having to retype the whole thing. For example:

```
NEW
200 POSITION=PDL(0)
210 PRINT POSITION
220 PRINT "MOVE THE GAME CONTROL"
230 PRINT "TO A NEW POSITION"
240 END
```

RUN this program a few times, changing the game control's setting between RUNs. Check to see if the program responds to both game controls. It should work for only one of them. You might take this opportunity to mark this control with the number zero.

This same program can be used, with a slight change, to look at the other game control. List the program the way it is now, then type

```
200 POSITION=PDL(1)
```


When you type a statement with the same line number as one that already exists in a program, the new line *replaces* the old one. LIST the program to see how it has changed. RUN it a few times to see what happens. Move the *other* game control between RUNs. Does this program respond to both controls? Mark a number one on the control to which this program responds.

Modifying a program in this way is one example of **editing** a program. You will learn other ways to edit programs later in this book.

As you have seen, there are several commands that help you deal with whole programs. They are

```
NEW
```

which erases programs,

```
LIST
```

which displays programs, and

```
RUN
```

which executes programs, beginning with the statement having the *lowest* line number. It is also possible to start execution elsewhere, and to LIST only part of a program. These abilities will be covered later.

ELEMENTARY AEROBATICS

At this point you are beginning to fly, so this section will discuss loops.

The best way to see how the **PDL** function works—and to understand program "loops"—is to use a statement we haven't discussed, until now. It's very simple. Type the following lines (after clearing any old programs that might be around):

```
10 PRINT PDL(0)
20 GOTO 10
```

Line 10 of this program PRINTs the number representing the current value of the game control. Line 20 does just what it seems to say: it causes program execution to go to line 10. What happens then? The program PRINTs the current value of the game control. Then it executes line 20, which says to do line 10 over again, and so on. Forever. This is a **LOOP**. A Loop is a program structure that exists when the program includes a command to return to a statement executed previously. RUN the program. Play with the game control. In the next section, we will tell you how to stop this program. Meanwhile, admire the fact that—if you typed RUN when instructed to do so, three sentences back—your APPLE has executed the statement PRINT PDL(0) a few hundred times already. *Now* the power of a stored program begins to increase significantly over what you can do by hand. Your abilities with the computer will increase dramatically in the next few sections, now that a good groundwork has been established.

SOME MORE THINGS THAT MAKE LIFE EASIER

But first, you are probably wondering how to stop the paddle program. You have already noticed how the numbers ripple up the screen as you move the paddle. This is because the numbers are printed at the bottom of the screen and as each new number is printed, all the rest of them are moved up one line. This is called **"scrolling"** and you've been seeing it all along, but at a much slower rate. To stop the running program, press

CTRL

C

It is OK to press RETURN after the CTRL C, but in this case it is not necessary. (This is an exception to the cardinal rule that you need a RETURN after typing any instruction to the computer.)

Although you need not press RETURN when using CTRL C to stop a program, you may reach for it anyway and accidentally press the RESET key. To get back to BASIC, you use CTRL C *with* a RETURN. If you press CTRL B and RETURN, you will also get back to BASIC, *but your program will be lost*. Therefore CTRL B is only used to get into BASIC when you first turn on your APPLE II.

When you stop a program with CTRL C, you can resume its execution by typing the instruction

```
CON
```

which stands for "**CON**tinue."

Try it. Now try this program:

```
100 X=PDL(0)
110 PRINT "GAME CONTROL ZERO IS"
120 PRINT X
130 Y=PDL(1)
140 PRINT "AND CONTROL ONE IS"
150 PRINT Y
160 END
```

Earlier we said that when you type RUN, the program starts executing at the lowest numbered line. True. However, if you want to start RUNning at some other line, such as line 120, you simply type


```
RUN 120
```

You can specify line numbers in the LIST statement, as well. If you type

```
LIST 130
```

the APPLE will LIST line 130 (if there is one, of course). If you type

```
LIST 110,130
```

the APPLE will LIST all the lines of your program starting at line 110 and continuing through line 130. This feature is not available with the RUN instruction.

To erase line 100 (assuming there is a line 100 in your program) you can type

```
100
```
RETURN

You could also have used the **DEL**ete instruction, as in

```
DEL 100
```

The advantages of the **DEL**ete are not apparent until we reveal to you that whole sections of programs can be erased with instructions like

```
DEL 120,140
```

which deletes every statement whose line number is 120 or greater, but less than or equal to 140. Try these commands, and LIST the program to see what they do to it. The ability to DELete blocks of line-numbered statements will be handy when you are writing large programs.

THE MOVING CURSOR HAVING WRIT CAN ERASE OR COPY ANY OF IT

When the leftand right-pointing arrows on the keyboard are pressed, they move the cursor. But they also either erase or retype characters, as you have just seen. It is possible to move the cursor *without* affecting anything at all (except the cursor position). You do this by using **pure cursor moves**. Each pure cursor move requires that *two* keys be pressed, in sequence. To move the cursor up, for example, type

ESC

and then type

D

Do this a few times. Each time you type

ESC D

the cursor should move up one line. When the cursor reaches the top, and you try to go up further, the cursor stays there since it can't get any higher. As you

will see, this is the most useful pure cursor move. To move the cursor down, type ESC C
When the cursor reaches the bottom, and you try to go down further, the cursor stays put, but the rest of the screen moves up! To make a pure cursor move to the right (*without* retyping the characters the cursor is moving over) use ESC A
To make a pure cursor move to the left (*without* erasing the characters moved over), use
ESC B
These last two pure cursor moves, when seen on the screen, *appear* the same as the rightand left-pointing arrow moves. But the *effect* is different, as you will discover when you LIST the results.

When the cursor reaches the right edge in a pure cursor move, and you try to make it move to the right some more, it appears one line lower down on the *left* edge. When you try to go beyond the left edge, the cursor sneakily appears on the *right* edge, one line higher.

These pure cursor moves do have an application, so they are not so pure after all. For example, if you typed this statement:

```
130 PRINT "THE QUALITY OF MERCY IS NOT
    STRAINED
```

and mistakenly pressed RETURN before typing the final quote, you would get the message

```
*** SYNTAX ERR
```

However, to make the correction you can use this trick to effectively retype the entire statement: Type

ESC D

three times to move the cursor up to the beginning of the incorrect statement. Then use the right-pointing arrow to effectively retype the entire line. This time, when you come to the end of the line, press the quote SHIFT " 2 before pressing RETURN. Use LIST to see that the line is properly corrected. The computer does have mercy on poor typists.

When a portion of a line that is somewhere on the screen has to be retyped, the pure cursor moves and the right-and left-pointing arrows can be used to speed the retyping. A few minutes of playing with this feature now will save you much work later.

A WORD ABOUT LEARNING BASIC

Many times there are questions you can ask about the BASIC language that are not answered directly in this book. For instance, in the statement

```
PRINT "HELLO"
```

do you have to put a space after the word "PRINT"? Rather than give you the answer, we recommend that you simply turn to your APPLE and try it both ways. Usually a simple experiment will answer your question and, since you have taken the time to try it yourself, you will remember it far better than if you had merely read it.

AN ACCIDENT ABOUT TO HAPPEN

Notice that you can delete a line by typing its line number and pressing the RETURN. This is a favorite way of introducing errors into your program. Say that you wanted to eliminate line 1100 from your program and you slip and type

```
110
```
RETURN

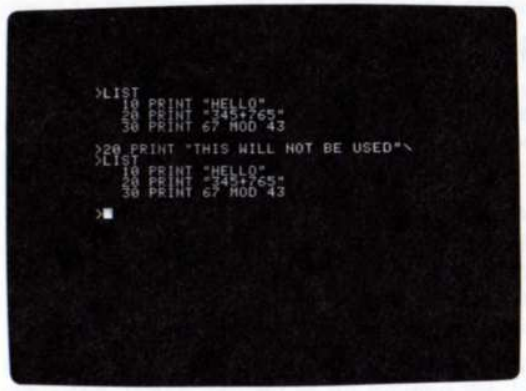
Congratulations, you have just wiped out line 110. This happens. Or you are about to fix up line 450 so you type

```
450
```

and think about it and decide not to change the line after all. *Don't press* RETURN. Either backspace over the line number, or use the special "wipe out this line" command

CTRL X

Using CTRL X places a backslash at the end of the line, and it will be as if you never typed it at all.



THE TRUTH

The APPLE can distinguish between what is true and what is false. Since this is more than most of us can do, a few words of explanation are in order. The symbol ">" means **greater than**. The **assertion**

6>2

(which is read "six is greater than two") is certainly true. The APPLE II uses the number 1 to indicate truth. If you type

```
PRINT 6>2
```

the computer will reply with a one. The assertion

55>78

is false. The APPLE II uses the number 0 to indicate falsehood. If you type

```
PRINT 55>78
```

the computer will reply with a zero.

The symbol "<" means **less than**, and you can make assertions using it as well. Here is the full set of symbols used in making assertions:

> **greater than**

< **less than**

= **equal to**

>= **greater than or equal to**

<= **less than or equal to**

not equal to

To type the symbols for "greater than or equal to" and "less than or equal to" on your APPLE II keyboard, you must first type either a "<" or a ">" and then type an "=."

Think about and then test to see which of these assertions are true, and which are false.

```
5#5
6>2
8>8
8<=8
9534=4359
5<8
45>=-4
-8<-7
-2>=-5
9#-9
```

Assertions can include variables and expressions as well as numbers.

```
PRINT (45*6)#(45+6)
```

will print the value 1 since 270 is not equal to 51 (remember that 1 means the assertion is true).

So the APPLE can tell truth from falsehood in simple assertions about numbers. An assertion such as

ABLE > BAKER

may be true or false, depending on the value of the two variables, ABLE and BAKER. If

ABLE = 5

and

BAKER = 9

then

ABLE > BAKER

is false. But if

ABLE = −8

and

BAKER = −15

then

ABLE > BAKER

is true.

Assertions have the numerical values of zero or one. They can be used in arithmetic expressions instead of ones and zeros. For example,

```
PRINT 3+(4>2)
```

will print the value 4. The statement

```
T=4#3
```

gives T the value 1, since 4 does not equal three, and thus

4 # 3

has the value 1. The statement

```
HOT=67=19
```

looks very confusing at first, but it is easily understood. Since 67 does not equal 19, the assertion is false and has the value zero. The value of 0 is given to the variable "HOT."

As we have seen, the APPLE uses 1 to mean true, and 0 to mean false. If something is not true, it is false. If something is not false, it is true. This may not always be the case in real life, but it is *always* the case with computers. Try this on the APPLE


```
PRINT NOT 1
```

and then try

```
PRINT NOT 0
```

The computer agrees: not true is false and not false is true. Of course, you can use expressions instead of ones and zeros. For example

```
PRINT NOT (45>3)
```

The sentence

TRIANGLES HAVE THREE SIDES.

is true. And the sentence

THIS BOOK IS IN ENGLISH.

is true. Consider the sentence

TRIANGLES HAVE THREE SIDES **AND** *THIS BOOK IS IN ENGLISH.*

Is this sentence true or false? It is true. Consider the sentence

TRIANGLES HAVE EIGHT SIDES **AND** *THIS BOOK IS IN ENGLISH.*

This sentence, as a whole, is false. Lastly, consider the sentence

TRIANGLES HAVE EIGHT SIDES **AND** *THIS BOOK IS IN SWAHILI.*

This sentence is also false. In general, when you combine two sentences, or assertions, by joining them with the word "AND," you find that:

a. The new sentence is true if *both* original sentences were true.

b. The new sentence is false if *at least one* of the original sentences were false.

The APPLE II knows how to determine whether an assertion containing the connecting word **AND** is true or false. Test your computer with the following instructions; try to predict each answer:

```
PRINT 1 AND 1
PRINT 1 AND 0
PRINT 0 AND 1
PRINT 0 AND 0
PRINT (3>2) AND 0
PRINT (NOT 0) AND (3*3=9)
PRINT (4#5) AND (4=5)
```

Is this sentence true or false?

A TRIANGLE HAS THREE SIDES **OR** *THIS BOOK IS IN LATIN.*

It's true. A triangle does have three sides, even if this book isn't in Latin, so the sentence *as a whole* is true. Quod erat demonstrandum. In general, when you combine two sentences by joining them with the word "OR", you find that:

a. The new sentence is true if *one or both* of the original sentences were true.

b. The new sentence is false if *both* of the original sentences were false.

The APPLE II can also determine if an assertion containing **OR** is true or false. Try each of these on your APPLE—*after* figuring out what the answer *should* be.

```
PRINT 1 OR 1
PRINT 1 OR 0
PRINT 0 OR 1
PRINT 0 OR 0
PRINT (4#5) OR (4=5)
PRINT 1 OR (0 AND 1)
PRINT ((3>4) OR (54<337)) AND (NOT 0)
```

AND, **OR**, and **NOT** will become very useful in the next section.

You have already found that in the statement

```
PRINT 1 OR 0
```

the computer regards 1 as *true* and 0 as *false*. Now try this:

```
PRINT 23 OR 0
```

and this:

```
PRINT -247 AND 32707
```

In assertions, the APPLE II regards not only 1, but *any* integer which is not zero, as true. However, when the *computer* figures out the value of an assertion, that value will always be either 0 or 1.

While the following box gives the precedence rules for **AND, OR,** and **NOT**, we strongly recommend that you use parentheses to make your statements clear.

ORDER OR PRECEDENCE FOR OPERATORS
USED SO FAR IN THIS TEXT:

1. ( )
2. NOT - (when used to indicate a negative number)
3. ^
4. * / MOD
5. + -
6. > < = >= <= #
7. AND
8. OR

THE IF STATEMENT

Suppose you want to print out numbers from 1 to 10, one number to a line. An obvious way to do this is

```
NEW
210 PRINT 1
220 PRINT 2
230 PRINT 3
```

and so on. But this would require 10 statements, and if you wanted to print the numbers from 1 to 200 this way, it would require 200 statements. Using what you have already learned, you can PRINT all the numbers from 1 to 32767 in just four statements by using a loop:

```
200 I=1
210 PRINT I
220 I=I+1
230 GOTO 210
```

The only thing that makes this program stop at 32767 is the upper limit on numbers that the APPLE can handle.

There is another way to control how long a loop runs. What you want is a statement that does a **GOTO** if **I** is, for example, less than 11, but doesn't do the **GOTO** if **I** is greater than 11. The answer to your wishes is the **IF** statement. If a condition is met, the computer will execute the instruction included in the **IF** statement. If the condition is not met, the computer will skip this instruction and execute the next one.

Here is a program that counts from 1 to 10 and then stops:

```
200 I=1
210 PRINT I
220 I=I+1
230 IF I=11 THEN END
240 GOTO 210
```

Here is another way of doing exactly the same thing:

```
200 I=1
210 PRINT I
220 I=I+1
230 IF I<11 THEN GOTO 210
240 END
```

Think about both programs, and check for yourself that they both perform as advertised.

In general, the **IF** statement works like this:

IF *arithmetic expression* **THEN** *any statement*

First, the arithmetic expression is evaluated. If it evaluates to zero (false) the **"THEN"** portion of the **IF** statement is ignored, and the computer goes on to the next instruction. If the arithmetic expression is *not* zero (true) the **"THEN"** portion of the **IF** statement is executed.

The most common statement to follow the word **"THEN"** is a **GOTO**. Because of this, you may leave out the word GOTO in a statement like

```
IF I<11 THEN GOTO 210
```

so that it may be written

```
IF I<11 THEN 210
```


This is not a recommended practice as it is less clear than writing the word GOTO—even if it is easier. The choice, of course, is up to you.

The **IF** statement is a very powerful one, and it will appear in almost every program you write. For the fun of it, try this program:

```
NEW
400 GR
410 ROW=1
420 COLOR=ROW
430 HLIN 0,39 AT ROW
440 ROW= ROW+1
450 IF ROW<16 THEN GOTO 420
460 END
```

MORE GRAPHICS PROGRAMS

Earlier, you put four colors at the corners of the screen. Now type in this program:

```
NEW
190 GR
200 COLOR=9
210 PLOT 0,0
220 PLOT 0,39
230 PLOT 39,39
240 PLOT 39,0
```

LIST the program to check that you typed it in correctly, and then RUN it. Quick, isn't it? To change the colors, just change line 200, and RUN the program again. Try to LIST the program. Notice that the listing slips through the narrow **window** at the bottom of the screen. This will happen unless you type

```
TEXT
```

to get out of GRaphics mode before you try to LIST.

This program makes the entire screen a solid color.

```
NEW
200 GR
```

```
210 COLOR=9
220 COLUMN=0
230 VLIN 0,39 AT COLUMN
240 COLUMN=COLUMN+1
250 IF COLUMN<40 THEN GOTO 230
260 END
```

Here's a blow-by-blow explanation of what happens when you RUN this program. Line 200 sets the APPLE into graphics mode. The color is chosen in line 210. The program is to start in the zeroth column of the screen and work its way over to the 39th column. Line 220 makes sure the program starts in column 0. At line 230, a vertical line is drawn in column 0. Now that the zeroth column is filled with the desired color, line 240 increments the column by one. The value of COLUMN is now 1. Line 250 checks to see if the new value of COLUMN is less than 40. If it is less than 40, the program goes back to line 230, to draw a new vertical line in the next column. However, when the value of COLUMN reaches 40 (there are only 40 columns on the APPLE II screen), the program does not go back to line 230, but *"drops through"* (as we say) to line 260. Line 260 stops the program.

LIST the program that fills the screen with a solid color. Remember that **RND (16)** will give a number between 0 and 15 inclusive. We could let the computer pick the color by changing line 210 to

```
210 COLOR=RND(16)
```

Each time the program is executed, a random color will be chosen. If the color is zero, the screen will appear to be cleared. (Why?) Execute this program a few times.

Another change eliminates the need to type RUN after the screen is filled with color. Rewrite line 260 as

```
260 GO TO 210
```

Observe what happens. When will this program stop? LIST the program and make sure you understand what it does.

When you are finished playing with the solid color program, clear the computer and try the following program. It uses a new and very important instruction: the **REM** statement. "**REM**" stands for "**REM**ark." This statement allows you to put commentary in a program. The computer ignores any **REM** statements; they are strictly for the benefit of humans. See how easy it is to follow this program where **REM**s are used liberally.

```
200 REM SET GRAPHICS MODE
```



```
210 GR
220 REM CHOOSE A RANDOM COLOR
230 COLOR=RND(16)
240 REM CHOOSE A RANDOM POINT (X,Y)
250 X=RND(40)
260 Y=RND(40)
270 REM PLOT THE RANDOM POINT
280 PLOT X,Y
290 REM CHOOSE ANOTHER RANDOM COLOR AND POINT
300 GOTO 230
```

Think about what this program will do. *Then* try it out on your APPLE.

There are many easy modifications to this program that will make it more interesting. For example, rewrite lines 270 and 280 as follows

```
270 REM PLOT A RANDOM HORIZONTAL LINE
280 HLIN X,X+RND(40-X) AT Y
```

Program statement 280 has especial interest. Clearly it draws a horizontal line starting at column X. X has been chosen at random; it may be any number from 0 to 39. The problem you face is this: how to choose a random value for the right end of the horizontal line. You can't simply use **RND (40)**, because it might give you a number less than X (the computer will not plot a line from a higher to a lower coordinate). And you can't use something like **X+RND (40)** because if X were 39, say, the only legal value you could have the program add to it would be zero, and the **RND** function might choose otherwise. You want the **RND** function to choose a number that is at least X, but does not exceed 39. Having chosen X, the amount of room to the right of column X is *(40−X)*. So that amount is used as the limit of the **RND** function.

Try the program, and then make up some of your own variations. Do not worry about making errors. Errors are part of the learning process. Nobody ever learned to walk without falling.

MAKING THINGS EASIER AND EASIER

By now you should have the habit of making your line numbers increase by at least 10 for each consecutive line. Doesn't adding 10's seem like something that a computer could do? It does, and it can. Clear any old programs that may be lying around by the usual **NEW** command, for you are about to learn how to make the APPLE II number your lines for you. Type in these statements and watch the computer provide you with line numbers **AUTO**matically:

```
AUTO 300
PRINT "S"
PRINT "E"
PRINT "A"
PRINT "M"
END
```

Now type

```
RUN
```

The APPLE responds with a *** SYNTAX ERR. This is because the command RUN is not actually part of your program and it shouldn't have a line number. You need to type the RUN without a line number. You can get rid of the line number by backspacing over it. Try. There is another way that is a bit quicker. When you want to issue an instruction without a line number, you hit

CTRL

X

Do this, and then type

```
RUN
```

and your program will execute.

You can also use CTRL X to insert lines with line-numbers out of **AUTO**matic sequence. For example, hit

CTRL

X

then type

```
305 PRINT "T"
```

and RUN your program. (Don't forget you will need a CTRL X before a RUN or a LIST.)

Do you feel like the sorcerer's apprentice? Now that you've got the **AUTO** line numbering working, it seems to want to go on forever giving you line numbers, like it or not. Well, to stop **AUTO**matic line numbering altogether, type

CTRL

X

and the command

```
MAN
```

This command stands for "**MAN**ual."

Thus far, the line-numbers have been incremented by 10. However, you may want to increase your line-numbers by more or less than 10. If you wanted to start your line numbers at 1000 and to increment by 30, you would type

```
AUTO 1000,30
```

Here is another program which uses the **AUTO** feature. Type the statements in as shown, since you don't have to type the line numbers.

```
NEW
AUTO 1000
GR
COLOR=9
REM READ PADDLE ZERO
X=PDL(0)
REM DIVIDE BY 7 SO MAXIMUM VALUE OF X IS 36
X=X/7
REM READ PADDLE ONE
Y=PDL(1)
REM LIMIT RANGE TO KEEP Y ON SCREEN TOO
Y=Y/7
REM PLOT THE POINT
```

```
PLOT X,Y
GOTO 1030
```

After you type RUN, operate the game controls. This program is called the "Etch-a-sketch" (TM) after a device that behaves similarly. The division by seven is necessary since the **PDL** function gives values between 0 and 255, whereas the screen can only accept column and row values from 0 to 39. By dividing by seven, you get values from *(0/7)* = 0 to *(255/7)* = 36. This does not utilize the full height or width of the screen. To get the full width of the screen, instead of

X = X/7

you could use the two lines

```
IF X>239 THEN X=239
X=X/6
```

The **IF** statement limits the value of X to 239. In the APPLE's integer arithmetic, 239/6 = 39. The **PDL** range of 0 to 255 is changed exactly to the screen's requirements of 0 to 39. This use of the **IF** to limit the range of a variable is very common.

Loops, whether executed by airplanes or computer programs, have a top and a bottom. In the program

```
NEW
100 NUMBER=0
110 PRINT NUMBER
120 NUMBER=NUMBER+1
130 IF NUMBER<=12 THEN GOTO 110
140 END
```

line 110 is the top of the loop, and 130 is the bottom. The program prints the integers from 0 to 12 inclusive. The number 12 is the **limit** of the loop. Another way to write a loop is to use the **FOR** statement. We can use this statement to rewrite the previous program.

```
200 FOR NUMBER=0 TO 12
210 PRINT NUMBER
220 NEXT NUMBER
230 END
```

Use RUN 200 to execute this program. If you just type "RUN," the program at line 100 (being the lowest line number around) will be executed.

Line 200 is the **FOR** statement. It starts by setting NUMBER to the value 0. This is exactly the same task that line 100 performed. Then line 210 is executed. The *bottom* of a loop that begins with a **FOR** statement is always a **NEXT** statement. The **NEXT** statement tells the computer to add one to the variable mentioned in it. If the variable is not over the limit, execution continues at the statement *immediately following* the **FOR**. If the variable is over the limit, the program drops through (out of the loop) to the statement *after* the **NEXT**. In this case, the program drops through to line 230 which terminates the program.

The most obvious advantage of the **FOR-NEXT** method of constructing loops is that it saves a statement. The most important advantage is that you don't have to think so hard when writing a loop if you use a **FOR-NEXT** loop. If you wanted to draw a line with each of the 15 colors on the screen, you could type

```
3000 GR
3010 FOR I=0 TO 15
3020 COLOR=I
3030 HLIN 0,39 AT I
3040 NEXT I
3050 END
```

Another advantage is that it is much easier to read a single **FOR** statement than to look through three statements to figure out what a loop is doing. To find the bottom of a **FOR-NEXT** loop, all you have to do is look for a **NEXT** which has the same variable as the **FOR**.

It might be well to mention that, although you should know how the **FOR** statement works, you don't have to use it. It doesn't add any new abilities to those you already have. It just makes some programs easier to write (for some people).

At this point, if you have been following along on your APPLE II, you should remove the portion of the programs between lines 3000 and 3050, inclusive. So type

```
DEL 3000,3050
```

To PRINT just the *even* numbers from 0 to 12, you could use the program

```
100 THING=0
110 PRINT THING
120 THING=THING+2
130 IF THING<=12 THEN GOTO 110
140 END
```

The secret is in line 120, where 2 is added to THING. We say that the loop *steps* by two. To step by two in a **FOR** loop, you would type

```
200 FOR THING=0 TO 12 STEP 2
```

the rest of the program would look like lines 210 through 230 above, except that the name "NUMBER" would have to be changed, wherever it occurs, to the name "THING". Try it. The **STEP** may be any number in the range of the APPLE. It can even **STEP** backward, for example

```
200 FOR THING=39 TO 15 STEP-3
```

Type this and try it by typing RUN 200.

You should play with the **FOR** statement for a while, if you wish to learn to use it. A number of the example programs from this point on will use the **FOR** statement.

Along with the convenience of the **FOR** statement come some limitations. For example, **FOR-NEXT** loops may be nested, but may not cross; a few examples (which generate graphics) demonstrate the idea.

```
NEW
AUTO 200
GR
FOR HUE=1 TO 15
COLOR=HUE
FOR ROW=0 TO 39
HLIN 0,39 AT ROW
NEXT ROW
COLOR=HUE-1
FOR COLUMN=0 TO 39
VLIN 0,39 AT COLUMN
NEXT COLUMN
NEXT HUE
END
```

This is an example of two-level nesting. Think about it and RUN this program before going on to the next. Remember, when writing programs using **FOR** statements, that

EACH **FOR** MUST HAVE A MATCHING **NEXT**.

A WRONG PROGRAM:

Loop Crossing

```
NEW
AUTO 500
FOR I=10 TO 20
PRINT I
FOR J=30 TO 40
PRINT J
NEXT I
NEXT J
```

This program won't work. Its loops are **crossed**, which not only gives an error message, but doesn't make any sense. Whenever you find yourself writing crossed loops, it means that your thinking has gotten tangled. If you are *sure* that you know what you are doing, and *still* want to cross loops, use loops made with **IF** statements. You can cross those all you want, for what good it will do you.

A LAST EXAMPLE OF NESTED LOOPS:

```
NEW
AUTO 100
GR
FOR ROW=0 TO 30 STEP 10
FOR COLUMN=0 TO 35 STEP 5
COLOR=RND(16)
FOR I=0 TO 9
HLIN COLUMN,COLUMN+4 AT ROW+I
NEXT I
FOR DELAY=1 TO 100
NEXT DELAY
NEXT COLUMN
NEXT ROW
GOTO 110
```

This program has three-level nesting, and draws quilts. Try removing lines 170 and 180. What happens? Add the line

```
145 COLOR=RND(16)
```

What happens? Is it what you expected?

PRINTS CHARMING

As an experiment, type in this program and see what it does when you RUN it.

```
NEW
100 PRINT "HELLO"
110 GOTO 100
```

Now change line 100 by just one symbol

```
100 PRINT "HELLO",
```

and RUN the program again. As you can see, this PRINTs the word in *columns*. Now substitute a semicolon (;) for the comma (,)

```
100 PRINT "HELLO";
```

and RUN the program again. This time the output is *packed*. This means that there are no spaces between what you told the computer to PRINT. It prints HELLO after HELLO, until the screen is quickly filled.

Change the program by adding this statement

```
90 V=99
```

and changing line 100 to read

```
100 PRINT V
```

RUN this program. Now change line 100 to

```
100 PRINT V,
```

and RUN it again. Then change line 100 to

```
100 PRINT V;
```

and observe that the semicolon can be used with numerical values. The ability to place numbers one after the other without intervening spaces is sometimes quite useful.

Commas and semicolons can be used *within* a PRINT statement. Clear the old program with NEW, and type

```
100 STRIKES=2
110 BALLS=3
```



```
120 PRINT STRIKES,BALLS
130 END
```

You can make clearer output by including messages in the PRINT statement. For example, change line 120 into

```
120 PRINT "THE STRIKES AND BALLS ARE ";S
TRIKES,BALLS
```

Notice that you probably want to have a space after the word "ARE," lest the number of strikes gets printed too close to it. If you don't think that the large space between the number of strikes and balls looks nice, you could use the statement

```
120 PRINT "THE STRIKES AND BALLS ARE ";S
TRIKES;" ";BALLS
```

In this version, a blank is put between the number of strikes and balls. Perhaps the prettiest way of doing this (are you trying all of these on your APPLE?) is

```
120 PRINT "STRIKES ";STRIKES;" BALLS ";BALLS
```

This gives you a scoreboard-like display.

Let's say that you wanted to PRINT the word "HERE" starting in the 10th column (the screen is 40 columns across, by the way), you could use this statement

```
120 PRINT "         HERE"
```

(You have to take our word for it that there are nine blanks before the word "HERE"). Or you could use the **TAB** feature. Just as on typewriters, you can set a tab on the APPLE. The statements

```
110 TAB 10
120 PRINT "HERE"
```

have the same effect as putting 9 blanks in the quotes as we did above. Try it, you'll like it.

By combining the **TAB** with the **FOR** loop you can program some neat visual effects. For example:

```
200 FOR I=1 TO 24
210 TAB I
220 PRINT "X"
230 NEXT I
240 END
```

You can also **TAB** up and down with the APPLE. There are 24 (not 40) horizontal printing lines. That, by the way, is why the upper limit in the loop in the program above is 24. To print on a particular line, you can vertical tab (**VTAB**) to that line. Here is a small program that demonstrates the use of the vertical tab:

```
600 FOR X=1 TO 24
610 FOR Y=1 TO X
620 TAB X
630 VTAB Y
640 PRINT "APPLE"
650 NEXT Y
660 NEXT X
670 GOTO 600
```

Before you RUN this program, try (it ain't easy!) to figure out what it will do. It's both surprising and pretty.

VTAB but not **TAB** works for immediate execution. You can only use **TAB** in programs. While **TAB** and **VTAB** act a bit like the co-ordinates in **PLOT**, there are some differences. The 40 columns for the **TAB** instruction are numbered from 1 to 40, as they would be on a typewriter, while the first co-ordinate of a **PLOT** instruction can run from 0 to 39, which is more convenient for programming graphics. Since characters are taller than the "bricks" we build graphics with, there is only room for 24 lines of printing on the screen. Therefore **VTAB**'s limits are 1 and 24. A zero or a number that is too large for **TAB** or **VTAB** will give the message

```
*** RANGE ERR
```

A number larger than 255 or a negative number causes the message

```
*** >255 ERR
```

to appear.

The largest value for **VTAB** is 24, but the largest value for **TAB** is 255, so you can **TAB** past the length of a screen line. To see this in action, try

```
400 FOR I=1 TO 255
410 TAB I
420 PRINT I
430 NEXT I
440 END
```


TALKING TO A PROGRAM ON THE RUN

Here is a program that makes a dot of color move across the screen, bouncing off the right and left sides.

```
NEW
400 REM   CHOOSE A COLOR FOR THE "BA
    LL"
420 BALL=9
440 REM   SET GRAPHICS MODE
460 GR
480 REM  CHOOSE A STARTING POSITION
    FOR THE BALL
500 XOLD=20
520 REM   MOVE THE BALL BACK AND FOR
    TH BY ADDING AN INCREMENT (CALLE
    D XSPEED) TO THE X POSITION
540 REM  TO MOVE THE BALL TO THE LEF
    T, MAKE THE INCREMENT NEGATIVE
560 XSPEED=1
580 REM  CALCULATE THE NEW X POSITIO
    N BY ADDING THE INCREMENT XSPEED
     TO THE OLD X POSITION
600 XNEW=XOLD+XSPEED
620 REM  CHECK THAT THE BALL WOULD B
    E ON THE SCREEN
640 IF (XNEW>-1) AND (XNEW<40) THEN
    GOTO 740
660 REM  THE BALL WOULD BE OFF ONE S
    IDE OF THE SCREEN, SO CHANGE THE
     DIRECTION OF X MOTION
680 XSPEED=-1*XSPEED
700 GOTO 600
720 REM  PLOT THE BALL IN ITS NEW P
    OSITION
740 COLOR=BALL
```

```
760 PLOT XNEW,20
780 REM ERASE THE OLD BALL POSITION
800 COLOR=0
820 PLOT XOLD,20
840 REM  SAVE THE CURRENT BALL POSIT
    ION FOR NEXT MOVE
860 XOLD=XNEW
880 REM  MOVE AGAIN
900 GOTO 600
```

The reason that the variable **XSPEED** was called "XSPEED" will be evident if you change its value. Try **XSPEED = 2** for example. If you set **XSPEED** too high, the ball will appear to jump wildly across the screen, with no trace between positions.

This kind of program is the basis for many typical TV games. It is worthwhile to spend some time playing with the program, changing this and that, just to see what can be done with it.

SAVING PROGRAMS

If your changes should prove inexplicably fatal to the program's operation, you can always retype the original program and start over. However, to save yourself a lot of typing, why not **SAVE** the working program now, on a tape cassette? Then anytime you want that program, you can simply **LOAD** it back into the computer, from your tape. To save this program, type

CTRL

C

which stops the program, if it is running. Insert a blank cassette into your recorder and rewind it to the beginning, where your recorded program will be easy to find. On the recorder, hold down the *Play* button while pressing down the *Record* button. Both should stay down. Back at the APPLE, type

```
SAVE
```

When you hit RETURN, the blinking cursor will disappear. After 10 or 15 seconds, the computer will give a "beep" to let you know the recording has begun. Another "beep" will sound when the recording is completed, and the cursor will reappear. Push *Stop* on the recorder, and you are ready to go back to programming. Your program in the computer has not been affected in any way by SAVEing it.

Though you are now in a position to understand the program above, you might have friends who aren't. Suppose you wanted a friend to be able to choose the color of the ball. You could explain how to change line 420, but you'd also have to explain the possible error messages, and what to do if . . . well, it *would* take a bit of explaining. It would be better to let your friend *interact* with the program. To do this, you can use an **INPUT** statement. Change line 420 to read

```
420 INPUT BALL
```

When the program executes this statement it will put a question mark (?) on the screen, followed by the blinking cursor, and then *wait* until someone types a number and hits RETURN. The number typed will become the value of **BALL** and the program will resume execution. It might be a good idea to have the computer tell your friends what they are expected to do. You could put in PRINT statements such as

```
280 REM   SET TEXT MODE
300 TEXT
320 PRINT "TO SELECT A COLOR FOR THE
     BOUNCING BALL,"
340 PRINT "TYPE A NUMBER FROM 1 TO 15"
360 PRINT "AFTER THE QUESTION MARK."
380 PRINT "THEN PRESS THE KEY LABELL
    ED RETURN."
```

You may also incorporate a message into the **INPUT** statement:

```
420 INPUT "WHAT COLOR WOULD YOU LIKE THE
    BALL TO BE (1-15)",BALL
```

Notice that in an **INPUT** statement the message must be in quotes, that there must be a comma before the variable name, and that the question mark appears right at the end of your message. It usually makes sense to make the message ask a question.

Your friends can use the right- and left-pointing arrows to correct mistakes in typing. But if they make a mistake and then press RETURN, they will get an error message. If the character entered is not a number,

```
*** SYNTAX ERR
RETYPE LINE
?
```

will appear on the screen. If too great a number is entered,

```
*** >32767 ERR
RETYPE LINE
?
```

will appear. Tell the person using the program (the "user") to simply ignore these error messages and retype the number. However, if the user types a negative number or a number between 255 and 32767, the error message

```
*** >255 ERR
```

will appear, and the program will stop. For the most part, the user will not know how to restart the computer—*and shouldn't have to*. Therefore you should make the program check that all numbers typed by the user are correct. These lines will do it:

```
424 REM  CHECK THAT THE BALL COLOR TYPED
    IS BETWEEN 1 AND 15, INCLUSIVE.
428 IF (BALL>0) AND (BALL<16) THEN GOTO
    460
432 PRINT "THAT WASN'T BETWEEN 1 AND 15"
436 GOTO 420
```

Are you beginning to see why we advised you to leave so much room between line numbers?

It is good programming practice to make a program as foolproof as possible. You have advanced to the point where you are writing error messages for others to read. It may be all right for a programmer like you to read jargon such as "SYNTAX ERROR" but it is most definitely not all right to force an innocent user to deal with such nonsense.

Each time you use an **INPUT** statement, your program must check that what the user types is within certain limits, so that the program won't "blow up" or fail in any way. Dealing with the untutored user (and you must assume users are not programmers) is an art in itself. **Use of clear English sentences and careful checking of what the user types are always required.**

By the way, you can **INPUT** several values with one input statement. The statement

```
3000 INPUT X,Y,Z
```

would display a question mark as usual, and then wait for three numbers to be typed in. The first number would be stored in the variable named **X**, the second number in the variable named **Y**, and the third in the variable named **Z**. The three numbers must be separated by RETURN's or commas, and the last number must be followed by a RETURN .

SAVE your best version of the bouncing ball program, just in case. Then, if you have not done so already, try to add vertical motion to it. Use the new variables **YNEW**, **YOLD** and **YSPEED**. A solution is given on the next page, but try to work this out, yourself, *before* you look.

When you have this program running the way you want it to, **SAVE** it on your tape cassette. We will use it again, later on.

OFF THE WALLS

Here is one way to make the ball bounce off all four walls. The statements in black are ones that have been added to—or changed from—the program which bounced the ball between two walls.

```
280 REM  SET TEXT MODE
300 TEXT
320 PRINT "TO SELECT A COLOR FOR THE
     BOUNCING BALL,";
340 PRINT "TYPE A NUMBER FROM 1 TO 1
    5"
360 PRINT "AFTER THE QUESTION MARK."

380 PRINT "THEN PRESS THE KEY LABELL
    ED RETURN."
400 REM   CHOOSE A COLOR FOR THE "BA
    LL"
420 INPUT "WHAT COLOR WOULD YOU LIKE
    ",BALL
424 REM   CHECK THAT THE BALL COLOR
    TYPED IS BETWEEN 1 AND 15, INCLU
    SIVE."
428 IF (BALL>0) AND (BALL<16) THEN
    460
432 PRINT "THAT WASN'T BETWEEN 1 AND
     15"
436 GOTO 420
440 REM   SET GRAPHICS MODE
460 GR
480 REM  CHOOSE A STARTING POSITION
    FOR THE BALL
500 XOLD=20
510 YOLD=38
520 REM   MOVE THE BALL BACK AND FOR
    TH BY ADDING AN INCREMENT (CALLE
    D XSPEED) TO THE X POSITION
```



```
540 REM  TO MOVE THE BALL TO THE LEF
    T, MAKE THE INCREMENT NEGATIVE
560 XSPEED=1
565 REM   MOVE THE BALL UP AND DOWN
    BY ADDING AN INCREMENT (CALLED Y
    SPEED) TO THE Y POSITION
570 REM   TO MOVE THE BALL UP, MAKE
    THE INCREMENT NEGATIVE
575 YSPEED=1
580 REM  CALCULATE THE NEW X POSITIO
    N BY ADDING THE INCREMENT XSPEED
     TO THE OLD X POSITION
600 XNEW=XOLD+XSPEED
620 REM  CHECK THAT THE BALL WOULD B
    E ON THE SCREEN
640 IF (XNEW>-1) AND (XNEW<40) THEN
    GOTO 706
660 REM  THE BALL WOULD BE OFF ONE S
    IDE OF THE SCREEN, SO CHANGE THE
     DIRECTION OF X MOTION
680 XSPEED=-1*XSPEED
700 GOTO 600
704 REM  CALCULATE THE NEW Y POSITIO
    N BY ADDING THE INCREMENT YSPEED
     TO THE OLD Y POSITION
706 YNEW=YOLD+YSPEED
708 REM   CHECK THAT THE BALL WOULD
    STILL BE ON THE SCREEN
710 IF (YNEW>-1) AND (YNEW<40) THEN
    GOTO 740
712 REM THE BALL WOULD BE OFF THE TO
    P OR BOTTOM OF THE SCREEN, SO CH
    ANGE THE DIRECTION OF Y MOTION
714 YSPEED=-1*YSPEED
719 GOTO 706
```

```
720 REM  PLOT THE BALL IN IT'S NEW P
    OSITION
740 COLOR=BALL
760 PLOT XNEW,YNEW
780 REM ERASE THE OLD BALL POSITION
800 COLOR=0
820 PLOT XOLD,YOLD
840 REM  SAVE THE CURRENT BALL POSIT
    ION FOR NEXT MOVE
860 XOLD=XNEW
870 YOLD=YNEW
880 REM  MOVE AGAIN
900 GOTO 600
```

As you will see when you RUN this program, the result is a bit repetitive. You can alter the pattern of bouncing by changing the starting values of **XOLD** and **YOLD** (lines 500 and 510), but here is a change you might like better:

```
600 XNEW=XOLD+XSPEED* PDL (0)/70
706 YNEW=YOLD+YSPEED* PDL (1)/70
```

To see what this does, play with the paddles.

One more suggestion. Why not have another **INPUT**, giving a value to a variable called **BACKGROUND**? Fill the screen with the color **BACKGROUND** once, at the beginning of your program (right after GR). Then, to erase the old ball position, use

```
800 COLOR=BACKROUND
```

or even

```
800 COLOR=BACKROUND+3
```

SAVE your favorite version of this program on tape.

MAKING SOUNDS

Clicks, ticks, tocks, and various buzzes are easily generated. You can make sounds on your APPLE if you tap it, scratch your fingers across it or drop it, but the sounds covered here are produced by *programming* it. So go to a quiet place and try working through this section.

To construct any sound-producing program on the APPLE II, you will need this magic formula.

```
150 SOUND=PEEK(-16336)
```

There is no easy explanation for this formula. The number, −16336, is related to the memory address of the APPLE's loudspeaker, and was built into the electronics of the computer. You are just going to have to look this number up

when you need it.

PEEK returns the numerical code stored at a certain location in the computer. At most locations **PEEK** only returns a numerical value, but at some locations, such as −16336, it can cause something to happen. In this case, it causes the speaker to make a click. Since you are programming in BASIC, the **PEEK** function must be assigned to a variable. In this instance we have named the variable **SOUND**. Every time the program executes this statement, the APPLE will produce a miniscule "click." Add the statement

```
160 END
```

so that the APPLE won't beep loudly at the end of the program. RUN the program. Listen to your computer closely.

Now substitute this line

```
160 GOTO 150
```

and RUN the program. No problem hearing this!

To make your program beep for a limited period of time, add statements such as

```
140 FOR I=1 TO 300
160 NEXT I
170 END
```

Try it.

A tone is generated by a rapid sequence of clicks. Any program that uses **PEEK(-16336)** repeatedly will generate some sort of noise. Since −16336 is such a bother to type, we will insert another statement that will allow us to substitute a symbol which is easier to type. Enter the statement

```
100 S=-16336
```

To produce a nice, resonant click change line 150 to

```
150 SOUND=PEEK(S)-PEEK(S)-PEEK(S)-PEEK(S)-
    PEEK(S)-PEEK(S)
```

Different numbers of PEEKs in the statement will produce different quality clicks. Try RUNning some variations. For more buzzy tones, put one of your variations into a loop. In general, the faster the loop, the higher the pitch.

Now, to use these sounds, LOAD your bouncing-ball program (**OFF THE WALLS**, in the last chapter) back into the computer, from your tape cassette. Try adding a "bounce" sound each time the ball rebounds from a wall.

One possible solution is given on the next page, but try to work it out for yourself, first. (Hint: a bounce occurs whenever either **XSPEED** or **YSPEED** changes value.)

NOISE FOR THE BOUNCING BALL

Here is one way to make the bouncing audible. Add these lines to the **OFF THE WALLS** program from the last chapter:

```
240 REM  SET S TO ADDRESS OF SPEAKER
260 S=-16336

683 REM  MAKE A BOUNCE NOISE
685 FOR I=1 TO 5
690 BOUNCE=PEEK(S)-PEEK(S)+PEEK(S)-PEEK(S)
695 NEXT I

715 REM MAKE A BOUNCE NOISE
716 FOR I=1 TO 5
717 BOUNCE=PEEK(S)-PEEK(S)+PEEK(S)-PEEK(S)
718 NEXT I
```

Now try your own sounds. Why not make a different sound off each wall?

FOR HIGHER NOTES, MULTIPLE STATEMENTS ON ONE LINE

To get still higher tones, another feature of APPLE BASIC can be introduced. It is possible to put more than one statement on the same line. Try this one-line *program*:

```
50 S=PEEK(-16336):GOTO 50
```

The colon (:) can be used to separate statements in any program where you wish to have more than one statement on a line. However, only the first statement on the line has a statement number, so you can only branch to the first statement with a GOTO.

Now add:

```
40 FOR DELAY=1 TO 2500: NEXT DELAY
```

The advantages of multiple statements with a common line-number are these:

1. The statements are executed faster. (This is an advantage only if you need more speed.)
2. More of your program can fit on the screen.

3. It can save some typing when you are not using AUTOmatic line-numbering.
4. You can group statements together that collectively perform one function, such as the delay in line 40 above.
5. It requires less memory. (This is an advantage only if you are running out of space, and the computer gives you a `*** MEM FULL ERR` while entering a program.)

But there are also some disadvantages:

1. The program is harder to read.
2. It is harder to modify or correct the program.
3. You can't branch to any but the first statement.
4. It is very discouraging to type in a long multiple statement only to have it return a `*** SYNTAX ERR`, making it necessary to retype the whole statement.

MULTIPLE STATEMENTS ON A LINE AND THE IF STATEMENT

The multiple statement

```
10 IF 4<2 THEN PRINT "YES":PRINT "NO"
```

will print the word "NO" when executed. The word "YES" is not printed since the assertion

4<2

is false. The program then goes on to the next statement (*not* to the next line number) and prints "NO" since that is what the next statement in the program tells the computer to do.

CHAPTER 4
STRINGS, ARRAYS AND SUBROUTINES

STRINGING ALONG

Would you like to see your name backwards? So far we have played with graphics and numbers. But computers can also manipulate letters and symbols. Instead of handling them one at a time, as with the numerical values of variables, your computer handles a whole **string** of characters at a time. This will seem fairly natural, since we humans also usually deal with characters in bunches. Strings, just like variables, have names. The names follow the same rules as variable names except that they end with a dollar sign (**$**). Here are some examples of string names:

MYNAME$
A$
SENTENCE$

Since there are going to be many characters in a string, you must tell the APPLE, prior to using the string's name, the maximum number of characters you will ever have in the string. Suppose you know that you will have 30 or fewer characters in a string called **NAME$** (pronounced "NAME-dollar"). Then you would warn the computer with the statement (including a line-number if part of a program, of course)

```
DIM NAME$(30)
```

This is called "setting the **DIM**ension of the string."
You are now permitted to type

```
NAME$="HARRY S. TRUMAN"
```

Notice that the characters put in a string must be enclosed in quotes. The statement

```
PRINT NAME$
```

will print the contents of **NAME$**: in this case, the name of the 33rd President of the United States. Thus, when you have a string of characters that you need often, you can store the string in a variable with a short name.

There are several more instructions that manipulate strings. Suppose you want to know what the 12th through 14th characters of **NAME$** are. You could type

```
PRINT NAME$(12,14)
```

and the computer would print the requested characters. You must not ask for more characters than there are—you will get an error message. A simple experiment will show you what the error message is. Do one and see.

Consider these immediate commands:

```
DIM MAX$(255)
MAX$="ANYGARBAGEATALLOFSOMELENGTHSOTHAT
YOUCANNOTSEEHOWMANYLETTERSTHEREARE"
```


A question comes to mind: How many letters are stored at the present time in **MAX$** ? Don't count. There is a *function* that counts the letters in a string. Type

```
PRINT LEN(MAX$)
```

and you will find out what you want to know. Incidentally, there cannot be more than 255 characters in a string (and as usual, you can only enter about 100 characters in a single statement), and *all spaces count as characters*.

When you use the name of a string, such as **MAX$**, you mean the whole string. You can refer to any portion of the string by giving the numerical positions of the first and last characters in that segment. The segment of **MAX$** from character number A to character number B is specified as **MAX$ (A,B)**. If you are interested in a segment which ends where the string ends, you may omit the second positional value. Thus every character in a string, from the Nth character to the end, is specified as **MAX$ (N)**. Consider this program:

```
200 DIM ALPHABET$(100),X$(30)
210 ALPHABET$="ABCDEFGHIJKLMNOPQRSTU
    VWXYZ"
220 INPUT "TYPE A NUMBER BETWEEN 1 A
    ND 26, AND I WILL TELL YOU WHICH
     LETTER HAS THAT POSITION IN THE
     ALPHABET: ",P
230 IF P> LEN(ALPHABET$) OR P<1
     THEN GOTO 220
240 PRINT ALPHABET$(P,P);" IS LETTER
     NUMBER ";P;" IN THE ALPHABET."
250 PRINT
300 INPUT "TYPE A LETTER, AND I WILL
     TELL YOU WHERE IT IS IN THE ALP
    HABET: ",X$
310 FOR I=1 TO LEN(ALPHABET$)
320 IF ALPHABET$(I,I)=X$ THEN GOTO 500
330 NEXT I
340 PRINT "THAT IS NOT A LETTER OF T
    HE ALPHABET."
350 GOTO 300
500 PRINT X$;" IS LETTER NUMBER "
    ;I;" IN THE ALPHABET."
```

```
510 PRINT
520 GOTO 220
```

It is customary, and a good idea, to **DIM** strings a bit longer than you expect them to be, especially if you aren't sure of their exact length. It is also a good practice to use the **LEN**gth function, as shown here in lines 230 and 310, instead of a particular number. That way, if you change the string, the program will continue to work. For example, if you change line 210 to

```
210 ALPHABET$="0123456789"
```

the program will still run. But if you had used 26 instead of LEN(ALPHABET$) in line 310, a

```
*** STRING ERR
```

message would have resulted.

This program illustrates several other common programming practices. Notice how this program finds the position of a character in a string. This method of using a loop to *scan* through a string, one position at a time, is very common. The program also shows that you can **DIM**ension more than one string at a time in a **DIM** statement, simply by separating the items with commas. Notice the function of the blanks in the quotes in line 500. What would happen to the output without these blanks?

You can substitute one string for another with a replacement statement such as

```
X$=ALPHABET$
```

This statement copies the contents of **ALPHABET$** into **X$**. However, you must make sure that the receiving variable was **DIM**ensioned large enough to contain the replacement variable. In the example, the **DIM** for **X$** must be at least as large as **LEN (ALPHABET$)** . If the **DIM** of **X$** is too small, you may get the message

```
*** STR OVFL ERR
```

(which just means **STR**ing **OV**er**FL**ow **ERR**or).

YOU CANNOT USE THE PARTIAL STRING NOTATION ON THE LEFT SIDE OF A REPLACEMENT STATEMENT
For example, the statement

```
X$(3,5)="XYZ"
```

is illegal. However, the statement

```
X$=ALPHABET$(12,24)
```

are both permitted. There is no logical reason for this; it's just the way things are.

Here is a program that generates random words. How frequently a letter is

chosen depends on how many times it appears in **HEAP$**.

```
 90 COUNTER=1: REM  INITIALIZE CHARA
    CTER COUNTER
100 DIM HEAP$(255): REM  LETTER STRI
    NG
110 HEAP$="AAAAABCDDEEEEEEEEEFGGHIII
    JKLLLMMMNNNOOOOOPPRRRSSSSSTTTTTU
    UUUVWYY"
120 REM  PICK A RANDOM WORD LENGTH,
    FROM 2 TO 6 LETTERS
130 LENGTH= RND (5)+2
140 REM  WILL NEXT WORD + COUNTER EX
    CEED 40 CHARACTERS?
150 IF COUNTER+LENGTH+1<40 THEN
    GOTO 200
160 REM  LINE LENGTH EXCEEDED
170 COUNTER=1: REM  RESET CHARACTER
    COUNTER
180 PRINT : PRINT : REM  SKIP A LINE

190 REM  PICK 'LENGTH' NUMBER OF RAN
    DOM LETTERS FROM HEAP$
200 FOR W=1 TO LENGTH
210 L= RND ( LEN(HEAP$))+1
220 REM   PRINT LETTER
230 PRINT HEAP$(L,L);
240 COUNTER=COUNTER+1: REM  INCREMEN
    T CHARACTER COUNTER
250 NEXT W
260 REM  SEPARATE "WORDS" WITH A BLA
    NK SPACE
270 PRINT " ";
280 COUNTER=COUNTER+1: REM  INCREMEN
    T CHARACTER COUNTER
290 REM  START A NEW WORD
300 GOTO 130
```

Oh, yes—still want to see your name spelled backwards? Here's a program that will do just that:

```
NEW
100 CALL -936: REM   CLEAR SCREEN
110 DIM NAME$(100)
120 INPUT "TYPE YOUR NAME. I WILL SH
    OW IT TO YOU, SPELLED BACKWARDS.
     ",NAME$
130 REM  STEP BACKWARDS THROUGH THE
    NAME
140 FOR I= LEN(NAME$) TO 1 STEP
    -1
150 REM  PRINT ONLY THE NEXT LETTER
160 PRINT NAME$(I,I);
170 NEXT I
180 PRINT : REM  "CANCEL" THE SEMICO
    LON.
190 PRINT : REM  SKIP A LINE.
200 REM  DO IT ALL AGAIN
210 GOTO 120
```

CONCATENATION GOT YOUR TONGUE?

It is possible to add a second string to the end of an existing string—assuming the DIM statement for the existing string allocated sufficient room to contain both strings, joined end to end. You remember that the statement

```
100 DIM A$(75)
```

permits A$ to be a string of characters *up to* 75 characters long. The actual length of A$ is LEN(A$). For example, if you type

```
100 DIM A$(75)
200 A$="XYZ"
210 PRINT LEN(A$)
500 END
RUN
```

the number printed should be 3. To add a character onto the end of A$—as it stands now—you can type

```
220 A$(4)="A"
```


Add this statement to your program. Then type

```
230 PRINT A$,LEN(A$)
```

and RUN the program. Does A$ now contain XYZA? Next, retype line 220 as

```
220 A$(4)="BCDE"
```

and RUN the program again. Surprised at the result? In concatenation, the notation

A$(4)

represents that part of A$ *beginning* at the fourth element. Thus, in this last version of the program, A$(4) became the letter A, A$(5) became the letter B, and so on. You have just *concatenated* the string

ABCDE

onto the string

XYZ

Erase the program you just used and type this portion of a new program:

```
NEW
100 DIM FIRST$(100),SECOND$(100)
110 INPUT "GIVE ME ABOUT HALF OF A S
    ENTENCE: ",FIRST$
120 INPUT "AND NOW THE SECOND HALF O
    F THE SENTENCE: ",SECOND$
```

Now, suppose you wish to concatenate FIRST$ and SECOND$, storing the combined string in FIRST$. From the last program, you know that you simply start SECOND$ one element after the last character in FIRST$. But you don't know how long FIRST$ will be, except by use of the LEN function. So you can concatenate the two strings by using this statement:

```
130 FIRST$( LEN(FIRST$)+1)=SECOND$
```

This line reveals the trick. You know that you want to place the second string *one element beyond* the end of the first string. Since LEN(FIRST$) tells where the end of FIRST$ is, then LEN(FIRST$)+1 is *one element* after it ends. So that is where you want SECOND$ to begin.

To watch this program work, type

```
140 PRINT FIRST$
150 PRINT : PRINT : GOTO 110
RUN
```

And that's how you can do concatenation.

ARRAYS

In this section on arrays we use examples from mathematics, but they are from recreational mathematics and require nothing beyond elementary arithmetic. Nonetheless, if you feel put off by such things, just skim through this chapter. The APPLE II can do interesting numerical work, as well as string and graphical processing, and we thought it would be good to have a few examples of this type.

Arrays are neat. The programming power they give you more than compensates for the bit of thinking and experimenting it will take you to become familiar with them. They are like strings, except that an array holds *numbers* instead of letters. To create an array, you use a **DIM** statement, just as with strings.

DIM A (400)
Unlike strings, there is no upper limit for the amount of room you set aside for an array (except as dictated by the size of your computer's memory). Simply keep in mind that the longer your program, the less room you have for arrays, and vice versa.

The **DIM** statement above has given us 400 new variables. They behave exactly like the variables you have come to know and love. They are:

A(1)
A(2)
A(3)
and so on, down to
A(400)

Although you may find them awkward to type, they can be used just as any other variable is used. The statement

A(34) = 45 + A(192)
is perfectly correct. The number in parentheses is called a **subscript**, and the notation A(192) is read *"A-sub-one-ninety-two."*

Here is an interesting application which illustrates using arrays—it finds *prime numbers*. The prime numbers are whole numbers that can't be divided evenly (except by themselves, and one). For instance, 12 is not prime since it can be divided by 6 or by 4 or by 3 or by 2. But 17 is prime since the only numbers that divide it evenly are 17 and 1. Prime numbers have had a strange influence over people's minds since the ancient Greeks, if not before. They seem very mysterious, since they don't appear to occur in a pattern. Prime numbers have many useful properties in mathematics, and people have been trying for 2000 years to figure out a formula that would give them these numbers. So far, nobody has even found a formula that gives *only* primes (nevermind *every* prime). But while there is no formula (yet) that produces primes, we *can* write a program that will generate every prime—from 2 to the largest number the APPLE can handle, 32767. (*Note:* By mathematical convention, 1 is not considered a prime number.)

The idea is this: the program will look at every number (called **ISIT**, since we are asking *is it* a prime or not) and try to see if the number can be evenly divided by some other number (called the **TEST**). If **ISIT** can be divided, the program will go on to a higher **ISIT** and try that. If **ISIT** can't be divided, the program will print the value of **ISIT**, since it is a prime.

APPLE BASIC has an excellent way of testing whether **ISIT** is divisible by **TEST**: the **MOD** operator. If **ISIT MOD TEST** is zero, then **TEST** divides **ISIT** evenly. Our first try at the program might look like this:

```
NEW
 90 PRINT 2,
100 FOR ISIT=3 TO 32767
110 FOR TEST=2 TO ISIT-1
120 IF ISIT MOD TEST=0 THEN GOTO
    160
130 NEXT TEST
140 REM  IF WE GET HERE, WE HAVE A P
    RIME NUMBER
150 PRINT ISIT,
160 NEXT ISIT
```

Some questions to think about—if you can answer them, you understand how the program works.

Why does the loop at line 100 start with 3 and not with 2?

Why does the loop at line 110 have an upper limit of **ISIT**-1?

This program is dreadfully slow. It will take days before it is done. Maybe weeks. We can easily double the speed by changing line 100 to read

```
100 FOR ISIT=3 TO 32767 STEP 2
```

which will skip over the even numbers, which (except for two) are never prime because they can all be divided by two. The next improvement comes in the second loop. We are trying to divide **ISIT** by every number less than itself. This is unnecessary. We only have to try dividing **ISIT** by *primes*. For if **ISIT** is divisible by some non-prime (call it Q) then **ISIT** is certainly divisible by a prime that divides into Q. So if we save all the primes we calculate, then we can just divide by these, and the program will speed up a lot. Now here is where we need an array—we will use one called **PRIME** to save all the primes that the program generates.

```
NEW
500 REM  PROGRAM TO FIND PRIMES.
510 REM  MAKE SOME ROOM FOR PRIMES.
520 DIM PRIME(4000)
530 REM  USE "TOP" TO HOLD THE SUBSC
    RIPT OF LAST PRIME FOUND SO FAR.
540 TOP=1
550 REM  SET THE FIRST PRIME.
560 PRIME(TOP)=3
600 REM  MAIN LOOP LOOKS FOR POSSIBL
    E PRIMES.
610 FOR ISIT=3 TO 32767 STEP 2
620 REM  SCAN THROUGH THE PRIMES SAV
    ED SO FAR AND DIVIDE BY EACH OF
    THEM.
630 FOR SUBSCRIPT=1 TO TOP
640 IF ISIT MOD PRIME(SUBSCRIPT)
    =0 THEN GOTO 800
650 NEXT SUBSCRIPT
690 REM  IF THE PROGRAM GETS HERE, I
    SIT HAS BEEN DIVIDED BY ALL THE
    PRIMES AND WASN'T DIVIDED EVENLY
     BY ANY OF THEM.
700 REM  ISIT IS PRIME.
710 PRINT ISIT,
720 REM  SAVE THIS PRIME IN THE ARRA
    Y.
730 TOP=TOP+1
740 PRIME(TOP)=ISIT
790 REM  LOOK FOR NEXT PRIME.
800 NEXT ISIT
900 END
```

This program prints all the primes up to 32767 except 2 and 3. That problem can be remedied by adding the line

```
490 PRINT 2,3,
```


A program such as this one would be very hard to write without the help of arrays. We have included a lot of REMarks to make it easier for you to read.

Note: This program could run much faster if you divided **ISIT**, not by *every* prime, but only by those less than the square root of **ISIT**. If you are intrigued by such things, think of a way to do this in the program.

ERROR MESSAGES ABOUT ARRAYS

If a subscript is negative, or larger than the amount specified in the **DIM** statement for that array, the message

```
*** RANGE ERR
```

will appear. Expert programmers will always notice at what line the program failed. For example, the computer might say

```
*** RANGE ERR
STOPPED AT 740
```

At this point you would type

```
LIST 740
```

and the computer would print

```
740 PRIME(TOP)= ISIT
```

and you would know that the variable **TOP** had gone out of range. To see just what happened, you would type

```
PRINT TOP
```

and see exactly what value caused the problem. If you DIM **PRIME** to a small value, such as 5, you can see this error message in operation.

INITIAL VALUE OF ARRAYS

When **RUN** is typed, all variables without subscripts are given the value zero. Elements of arrays, however, are *not* set to zero by **RUN**, **CLR**, or RESET. If you want an array to contain zeros, you must include a few program lines that set each array element to zero. Incidentally, notations such as **A(0)** and **PRIME(0)** are allowed. These are not array elements but are alternate (and generally useless) names for the simple variables **A** and **PRIME**.

DEBUGGING

By now you have typed in some programs that didn't run the way you expected. Whenever this happened you had to figure out what was wrong and fix it. This fixing is called "debugging" since a programming error is often called a "bug." The APPLE II has some features that will help you stomp out any bugs that might be in your programs. To demonstrate how to find bugs, we need a program that has some.

First, a bit of background. If you multiply a number by itself, for example 5 * 5, you get the *square* of the number, in this case 25. Conversely, we say that 5 is the *square root* of 25. Most integers, for example 20, don't have an integer for a square root. 5 is too big to be the square root of 20, and 4 (whose square is 16) is too small. Clearly, the square root is between 4 and 5.

It is easy to write a program to find the square of a number on the APPLE II (try it!), but how would you find the *approximate* square root of a number? First, notice that the largest square the APPLE II can handle is 32761. The square root of 32761 is 181, so any square root the APPLE can find must lie between 0 and 181. Here's how the square root program will work:

We know the square root of a number (which we will call **NUMBER**) must be between 0 and 181. Therefore we set up two counters **MIN** (for "minimum") and **MAX** (for "maximum"):

```
100 MIN=0
110 MAX=181
```

Now, here comes a neat trick. The program will guess a number somewhere between **MIN** and **MAX**. Then it will see if this number which will be called **GUESS**, is too large or too small. One way of getting between **MIN** and **MAX** is to choose a number halfway between them. This is done in the next line:

```
120 GUESS=(MIN+MAX)/2
```

Now we need to know whether this **GUESS** is larger than the correct square root, or smaller. The correct square root, when squared, would equal **NUMBER**, so let's square our **GUESS**

```
130 SQUARE=GUESS*GUESS
```

so that we can compare *its* square to **NUMBER**.

These statements contain the heart of the program. If **SQUARE** is too large, then we know that the true square root is less than **GUESS**. So we lower the value of **MAX** to **GUESS**. If **SQUARE** is too small, then we know the true square root is more than **GUESS**, so we raise the value of **MIN** to **GUESS** . As long as **MAX** and **MIN** are not equal, we must keep "guessing" new values for

GUESS. The safest guesses are between the new **MIN** and **MAX**, so we add this line

```
180 GOTO 120
```

which tells the computer to try again.

This way we squeeze **MIN** and **MAX** closer and closer together until the approximate square root is caught between them.

```
140 IF SQUARE>NUMBER THEN MAX=GUESS
150 IF SQUARE<NUMBER THEN MIN=GUESS
```

When **MAX** and **MIN** are equal, we have found the square root, so we program this line

```
160 IF MAX=MIN THEN GOTO 200
```

and at line 200 we can say

```
200 PRINT "THE SQUARE ROOT OF ";NUMBER
210 PRINT "IS APPROXIMATELY ";GUESS
```

Here is the program so far:

```
100 MIN=0
110 MAX=181
120 GUESS=(MIN+MAX)/2
130 SQUARE=GUESS*GUESS
140 IF SQUARE>NUMBER THEN MAX=GUESS
150 IF SQUARE<NUMBER THEN MIN=GUESS
160 IF MAX=MIN THEN GOTO 200
180 GOTO 120
200 PRINT "THE SQUARE ROOT OF "
    ;NUMBER
210 PRINT "IS APPROXIMATELY ";GUESS
```

To test the program, add the statement

```
 90 INPUT " OF WHAT NUMBER SHALL I
    FIND THE SQUARE ROOT",NUMBER
```

Run the program and when it asks

```
OF WHAT NUMBER SHALL I FIND THE SQUARE
ROOT ?
```

answer with

```
34 RETURN
```

so that we can follow the program together. Oops. Nothing happens. The program is doing something, but it is not finding the answer. The program is not quite right: it has some bugs. Stop it with

CTRL

C

and find out what was happening. Type

```
PRINT NUMBER,GUESS,MIN,MAX,SQUARE
```

to see what these variables contained when you stopped the program. This will give you a valuable clue to what is going on. **NUMBER**, of course, was 34. You might wonder why we bother printing this value, since the program isn't supposed to change it. Good question. **NUMBER** isn't *supposed* to be changed, but by printing it you *know* that it hasn't been changed. There can be a big difference between what a program is supposed to do and what it actually does. Just remember: don't trust anything you haven't tested. **GUESS** was 5, which is the correct answer, **MIN** was 5, which is fine, but **MAX** was 6. Hmmm . . .

Before you get carried away with fixing up the program, let's see what happens *while it is running*. Type

```
10 DSP MIN
20 DSP MAX
```

This means to **DiSP**lay the values of **MIN** and **DiSP**lay the values of **MAX**. You could also use

```
DSP MIN
```

without a line number, but you could not say RUN, as this clears the **DiSP**lay feature. You would have to start the program with a

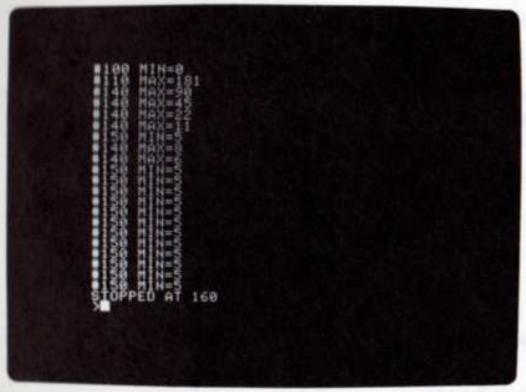
```
GOTO 90
```

The program immediately asks for a value for **SQUARE**, and you type the 34 again—but get ready to hit

CTRL

C

immediately. With good timing your screen will look like this photograph:



This display shows that at line 100, **MIN** was set to 0 (correct); and then at line 110, **MAX** was set to 181 (as you expected). Then you can see **MAX** coming down to 11, while **MIN** (since it is not shown) stayed at zero. This demonstrates clearly how **DSP** works: every time a **DiSP**layed variable is changed, the screen shows the line number and the variable name and its value. Handy, no?

After **MAX** got to 11, **MIN** moved up to 5, then **MAX** went to 8 and then to 6. Nothing ever changed after that, **MIN** just kept on being set to 5 in line 150. Think about the program. Will **MIN** and **MAX** ever be the same number? **SQUARE**, as you can tell by a PRINT, is at 25, since **GUESS** is at 5. Another DiSPlay will convince you that **GUESS** is at 5 forever. Try it. So for the number 34, **SQUARE** will always be less than **NUMBER**, **MIN** will be set to **GUESS** forever, and **MAX** will never equal **MIN**. Try changing the program with this instruction

```
160 IF MAX=MIN+1 THEN GOTO 200
```

RUN the program for several values of **NUMBER**. It always gives an answer with a maximum error of 1. Sometimes it doesn't get the *closest* integer to the square root, but it is never more than 1 away from a correct answer. But! The program gives no answer at all when you give it a **NUMBER** which is a perfect square, such as 25. Try to use **DSP** and your head to find a fix. Our fix is next. No peeking.

Our fix, after a bit of head scratching, is to change line 140 to

```
140 IF SQUARE>=NUMBER THEN MAX=GUESS
```

Changing **MAX** when **SQUARE** is exactly equal to **NUMBER**, as well as when **SQUARE** is greater than **NUMBER**, makes the program work for exact squares. But then **GUESS** may or may not be the square root of **NUMBER**. However, **MAX** always is, so we change line 210 to read

```
210 PRINT "IS, ROUNDED UP TO THE NEAREST
    INTEGER IF NECESSARY, ";MAX
```

Now the program works fine, and the answer is exact for perfect squares, and is the correct answer rounded up to the nearest integer if **SQUARE** is not a perfect square. You can fiddle with the program to make it find the *nearest* integer, but we won't bother doing it here.

A LAST WORD ON PRIMES

Now that we have a square root program and a prime program, it is hard to resist making a really fast prime-finding program by combining the two. When we were testing primes, we divided by *every* prime less than the number we were testing. But, it is not necessary to divide by every prime—only by every prime less than or equal to the square root of the number being tested. The reason is clear with a few minutes of thought, which we leave to you.

So we combined the two programs, and for still greater speed put as many statements as we could on one line, inside the main loop. The listing shows our fastest program so far. The steps above line 150 are only done once, so it doesn't matter how fast they are. They have been written out separately for clarity—which you should always do, unless there is a pressing need for speed or compactness. We set the **DIM** at 3600 because there are 3512 primes between 0 and 32766. We ran the program and had it print **TOP**. Notice that we have the loop stop at 32766. If you run it to 32767, you get a

```
*** >32767 ERR
```

32767 isn't a prime. 32767 = 7 * 31 * 151.

```
100 DIM PRIME(3600)
110 PRIME(1)=2
120 PRIME(2)=3
130 PRINT 2,3,
140 TOP=2
150 FOR GUESS=3 TO 32766 STEP 2
    :J=2
160 IF GUESS MOD PRIME(J)=0 THEN
    500: IF PRIME(J)>=MAX THEN
    400:J=J+1: GOTO 160
400 PRINT GUESS,:TOP=TOP+1:PRIME(
    TOP)=GUESS:MIN=0:MAX=181
410 ROOT=(MAX+MIN)/2:TRIAL=ROOT*
    ROOT: IF TRIAL>=GUESS THEN
    MAX=ROOT: IF TRIAL<GUESS THEN
    MIN=ROOT: IF MAX=MIN+1 THEN
    500: GOTO 410
500 NEXT GUESS
510 END
```

SUBROUTINES

Imagine that there is a game for which you need a piece that looks like a blue horse with orange feet and a white face. Here is a program that draws such a piece:

```
NEW
1000 REM  PROGRAM TO DRAW BLUE HORSE
     WITH WHITE FACE AND ORANGE FEET
1010 GR
1020 COLOR=7: REM  LIGHT BLUE
1030 PLOT 15,15
1040 HLIN 15,17 AT 16
1050 COLOR=9: REM  ORANGE
1060 PLOT 15,17
1070 PLOT 17,17
1080 COLOR=15: REM  WHITE
1090 PLOT 14,15
2000 END
```

There is nothing wrong with this program; it does draw a blue horse with orange feet and a white face. Now, suppose you needed to draw another horse somewhere else on the screen. You could rewrite this program with new values for X and Y. But that is a bother. There should be some way of using the same program to put a figure anywhere on the screen without having to rewrite it each time.

The key to doing this begins with the observation that you can *move* a point which is at co-ordinates (A,B) to the *right* by *adding* to A, the first co-ordinate. For example, the point (4,17) moves 10 columns to the right if you add 10 to the first co-ordinate, making the point (14,17). Likewise, a point moves *left* if you *subtract* from the first co-ordinate (or add a negative value). A simple experiment will show you that adding to and subtracting from the second co-ordinate moves points down and up, respectively.

With these facts in mind, you can rewrite your program to "center" the horse at almost any point (X,Y) on the screen. Why "almost" any point? Because, if you choose a center point at an edge of the screen, the horse will go off the screen, and this might give you a `*** RANGE ERR` message. Here is an improved program:


```
NEW
1000 REM  PUT A HORSE ANYWHERE ON THE
     SCREEN
1010 COLOR=7: REM  LIGHT BLUE
1020 PLOT X,Y-1
1030 HLIN X,X+2 AT Y
1040 COLOR=9: REM  ORANGE
1050 PLOT X,Y+1
1060 PLOT X+2,Y+1
1070 COLOR=15: REM  WHITE
1080 PLOT X-1,Y-1
```

You notice that both the **GR** and the **END** have been left out. We want to use this part of the program to put *several* horses on the screen. A **GR** here would clear the screen before each new horse was drawn. An **END** statement here would stop the program right after the first horse was displayed.

This program can't be run, just as it is. First you must set graphic mode, and choose X and Y. A good first try at using the horse program might be:

```
20 GR
30 REM  CHOOSE THE FIRST HORSE CENT
   ER
40 X=12
50 Y=35
```

If you try to RUN this, you do get a horse at the desired location, but the program ends there. We want to put *two* horses on the screen. What if you could write

```
60 Do the program at line 1000 and then come back to line 70
70 REM CHOOSE THE SECOND HORSE CENTER
80 X=33
90 Y=2
100 Do the program at line 1000 and then come back to line 110
110 END
```

Wouldn't that be nice and easy? You know that the computer can't read those strange instructions at lines 60 and 100. But it can read

```
GOSUB 1000
```

in BASIC. The **GOSUB** instruction tells the computer to **GO** to the **SUB**routine beginning at line 1000 and start executing at that statement. It also tells the computer to **come back to the line that follows the GOSUB statement** when it is finished with the subroutine. The computer knows it is finished when it encounters a **RETURN** statement. To make your horse-drawing partial-program into a **subroutine**, add the line

```
1090 RETURN
```

Now you can write that "what if you only could" program:

```
 20 GR
 30 REM  CHOOSE THE FIRST HORSE CENT
    ER
 40 X=12
 50 Y=35
 60 GOSUB 1000
 70 REM CHOOSE THE SECOND HORSE CENT
    ER
 80 X=33
 90 Y=2
100 GOSUB 1000
110 END
```

RUN the program.

In effect, you have added a new statement to BASIC: a horse-drawing statement. Now whenever you use the statement

```
GOSUB 1000
```

the computer will draw one of these special horses at whatever X,Y location you have chosen.

TRACES

The portion of the program from line 1000 to line 1090 is called a **subroutine** or *subprogram*. The portion of the program from line 20 to line 100 is called the *main program*.

To see the program's flow, or path of execution, we can invoke a special feature. Add this line to the main program:

```
10 TRACE
```

and, for a moment, delete line 20. Put the APPLE II into TEXT mode and RUN the program.

The numbers you see on the screen are the line-numbers of each statement, as it is executed. You can see how the program begins at line 10, continues through the main program until the subroutine call, then executes the subroutine, goes back to the main program, executes the subroutine again, and finally finishes the main program. **TRACE** is very handy when you are having problems with a program. If you want to **TRACE** only part of a program, you can use the **NOTRACE** statement. Add this line:

```
65 NOTRACE
```

Now the program will be **TRACE**d only up to the execution of line 65.

TRACE can also be issued in the immediate mode, like **DSP**. Simply type

```
TRACE
RUN
```

and your program will be **TRACE**d.

Note: **DSP** without a line number is cleared by a RUN command. But once you have issued the **TRACE** command, whether in immediate mode or as a statement in your program, your program will be **TRACE**d every time you RUN it, from then on. To stop **TRACE**, you must issue a **NOTRACE** , either in a line of your program, or in immediate mode:

```
NOTRACE
```

A BETTER HORSE-DRAWING SUBROUTINE

Subroutines should be written so that problems from possible errors do not arise when the program is RUN. One problem with our horse-drawing subroutine is that some values of X and Y will cause the horse to go off the edge of the screen. This can be prevented by a set of statements such as:

```
1012 IF X<1 THEN X=1
1014 IF X>37 THEN X=37
1016 IF Y<1 THEN Y=1
1018 IF Y>38 THEN Y=38
```

(Why should the maximum Y value be 38, while X must be limited to 37?)

If there is any attempt to locate a horse off the screen, the horse will be moved to the nearest edge. There are other possible strategies, such as giving an error message and stopping the program. However, our choice has the advantage that it doesn't stop the program, and you can see that *something* is happening.

Sometimes you want to be able to change the values in a subroutine for different program **GOSUB**s. For example, a second player may want to place a piece, and that should be a horse of a different color. One way to do this would be to type in the whole subroutine again, with different colors. However, let's try using *variables* rather than numbers. Instead of line 1010 saying COLOR = 7, it could say

```
1010 COLOR=BODY
```

Similarly, you could write

```
1040 COLOR=FEET
1070 COLOR=FACE
```

Then the main program could go like this:

```
 20 GR
 30 REM  CHOOSE COLOR OF FIRST PLAYE
    R'S HORSE
 40 BODY=7: REM  LIGHT BLUE
 50 FEET=9: REM  ORANGE
 60 FACE=15: REM  WHITE
 70 REM  CHOOSE CENTER OF FIRST PLAY
    ER'S HORSE
 80 X=15
 90 Y=30
100 GOSUB 1000
```

and so on (be sure to follow with an END statement, before you try to RUN it). That's a lot of statements each time you want a horse, but it is still fewer than if you had to type out the entire horse program each time. For additional programming ease, a rather subtle trick is to have a subroutine for each color horse—and have those subroutines call the horse-drawing subroutine, in turn.

```
2000 REM  ROUTINE DRAWS BLUE HORSE WI
     TH ORANGE FEET AND WHITE FACE
2010 BODY=7: REM  LIGHT BLUE
2020 FEET=9: REM  ORANGE
2030 FACE=15: REM  WHITE
2040 GOSUB 1000
2050 RETURN

2500 REM  ROUTINE DRAWS ORANGE HORSE
     WITH PINK FEET AND GREEN FACE
2510 BODY=9: REM  ORANGE
2520 FEET=11: REM  PINK
2530 FACE=12: REM  GREEN
2540 GOSUB 1000
2550 RETURN
```

Now all you need, to put a blue horse with white face and orange feet at (10,11), is

```
 30 REM  FIRST PLAYER'S HORSE
 40 X=10
 50 Y=11
 60 GOSUB 2000
```

To put an orange horse at (19,2) all you need is

```
 70 REM  SECOND PLAYER'S HORSE
 80 X=19
 90 Y=2
100 GOSUB 2500
```

Each of these subroutines, 2000 and 2500, calls subroutine 1000. Things get to be quite efficient at this stage. Once you have written a good subroutine that checks for errors, that uses variables you can set in the calling program (which may be the main program or another subroutine), then you can pyramid other subroutines upon it. This makes main programs very easy to write. Using the three subroutines, it is very easy to put up an attractive display of horses. But first, another handy routine:

```
3000 REM  CHOOSE A RANDOM PAIR OF CO-
     ORDINATES
3010 X= RND (38)+1
3020 Y= RND (39)+1
3030 RETURN
```

And, *now* for the main program

```
10 REM  SET GRAPHICS MODE
20 GR
30 REM  CHOOSE A RANDOM POINT
40 GOSUB 3000
50 REM  PUT A BLUE HORSE THERE
60 GOSUB 2000
70 REM CHOOSE ANOTHER RANDOM POINT
80 GOSUB 3000
90 REM  PUT AN ORANGE HORSE THERE
100 GOSUB 2500
110 REM  DO IT ALL AGAIN
120 GOTO 30
```

THIS is how a main program should look if you are a good programmer: mostly **REM**s and **GOSUB**s. The work should be done in relatively short subroutines, each of which is easy to write, and complete in itself. To see how this sample program does its stuff, feel free to use **TRACE**.

To make this program even easier to read, you might substitute variables—with easily recognized names—for the numbers in the **GOSUB** statements:

```
22 CHOOSEPOINT=3000
24 BLUEHOSS=2000
26 ORANGEHOSS=2500
28 DRAWHOSS=1000
```

Now see how easy it is to understand statements such as

```
40 GOSUB CHOOSEPOINT
```

or

```
60 GOSUB BLUEHOSS
```


CONCLUSION

This book has presented the core of APPLE's BASIC. If you now go through this book again, writing your own programs with the statements that have been presented here, you will solidify your knowledge considerably. There are many more abilities in the APPLE II; and once you have mastered these, there are whole new worlds for you to explore.

APPENDICES

INDEX

APPLE COMPUTER MESSAGES: THEIR CAUSES AND CURES

At times, your APPLE will print a message to you that is not the result of a **PRINT** statement in your program. These messages usually indicate that some kind of error has been made. All computer systems give error messages. They are part of the milieu. If you think of them as friendly suggestions, rather than as nagging reminders, you will find programming all the more enjoyable. Messages fall into four natural groups. The first can occur when you are typing a program. The second group occurs during execution of a program. The third group is associated with responding to an **INPUT** statement. The last group occurs when you are using the cassette tape recorder. Some messages are associated with more than one group.

GROUP I MESSAGES

These messages usually occur when you are typing in a program or instruction.

```
*** SYNTAX ERR
```

This is the most common message. The APPLE II is saying, *"I don't understand that."* What you have typed is not a well-formed BASIC statement. Rethink (if necessary) and retype the statement. Sometimes these errors are a bit obscure, such as typing the letter 0h for the number zero or vice versa. It is never difficult to correct a syntax error.

```
*** TOO LONG ERR
```

You have typed a statement that is too long—that is, one containing more than about 127 characters. Break it up into two or more shorter statements.

```
/
```

The backslash is printed, after several warning "beeps," when you have typed a line exceeding 255 characters. Although the characters remain on the screen, the computer forgets all of the statement up to the backslash and lets you start over.

```
*** >32767 ERR
```

You have typed a number greater than 32767 or less than −32767. Think small.

```
*** >255 ERR
```

Some things in BASIC must not be less than zero or greater than 255: for example, the numbers in a **PLOT** statement. This is the message you get. Think smaller, but remain positive.

GROUP II MESSAGES

Messages of this type occur most commonly during the execution of a program. These messages are usually accompanied by a further message such as

```
STOPPED AT 390
```

which tells you what line the computer was trying to execute when the problem was discovered.

```
*** BAD BRANCH ERR
```

If you attempt a **GOTO** or **GOSUB** to a certain line number, and that line number doesn't exist (perhaps it was erased accidentally), you will get this error.

```
*** BAD RETURN ERR
```

To execute a **RETURN**, you must be in a subroutine which you reached by executing a **GOSUB**. This message occurs when you try to **RETURN** from some point in the program that was not reached by executing a **GOSUB**. In other words, you have just tried to execute one more **RETURN** than the number of **GOSUB**s you've executed.

```
*** BAD NEXT ERR
```

To execute a **NEXT**, there must be some **FOR** statement that has been executed, and that hasn't finished its job. This message occurs when you attempt to execute a **NEXT**, without the previous execution of a corresponding **FOR**. A **FOR** and **NEXT** correspond when they are followed by the same variable name, such as **FOR X = 2 TO 5** and **NEXT X**.

```
*** 16 GOSUBS ERR
```

An unlikely error to happen. It means that one subroutine had a **GOSUB** to another subroutine that had a **GOSUB** that went to another subroutine that had a **GOSUB** . . . 16 times. The cure? Depends on how the subroutine got nested. A program like

```
100 GOSUB 200
200 GOTO  100
```

which **GOSUB**s over and over without any **RETURN**s will give the message. Make sure that each subroutine gets back to where it was called from by means of a **RETURN**. If the message arose because you really had 16 *nested* subroutines, you will have to make your program less ambitious or write out one of the subroutines where it is needed. (You may have as many sub-

routines as will fit in your APPLE, this restriction only applies to nested subroutines.)

*** 16 FORS ERR

There were more than 16 nested **FOR** loops. Replace one or more of the **FOR** loops with written-out loops.

*** NO END ERR

If you don't like this message, make sure that the last statement executed is an **END**. Usually, when this message appears, nothing at all is wrong with your program.

*** MEM FULL ERR

One way this can occur is if you write a program, or the program uses data, that requires more memory than you have installed in your APPLE II. This message can also occur when loading tapes, in which case it may not mean that memory is full. See GROUP IV messages below.

*** TOO LONG ERR

If someone is typing in response to an **INPUT** statement and they type more than 128 characters, they will get this message. Tell the person to be less verbose. This message can also arise if there are more than 12 nested sets of parentheses, as in a complicated arithmetic operation. In this case you can break up the expression into two or more simpler expressions.

*** DIM ERR

If a variable has been dimensioned, and you attempt to dimension it again under certain conditions, this message will appear. Just dimension strings and arrays once, and this message will not occur.

*** RANGE ERR

This can occur a number of ways: if a subscript to an array is larger than allowed by the corresponding **DIM** statement; if a subscript is less than 1; if arguments to **HLIN**, **VLIN**, **PLOT**, **TAB** or **VTAB** are too large. In any case, make sure that the subscript or argument does not go out of range.

*** >32767 ERR

An **INPUT** statement or a calculation has given a number greater than 32767 or less than −32767. Tell the user to **INPUT** smaller numbers, or make sure your program generates smaller numbers.

*** >255 ERR

Some things in BASIC must not be less than zero or greater than 255: for example, the arguments to **COLOR**, **PLOT**, and **TAB** statements. If your program or a user's response to an **INPUT** statement exceeds that range, this message will appear.

*** STR OVFL ERR

You get this message when you try to put more characters in a string than you said you would in your **DIM** statement.

*** STRING ERR

Almost any mistake involving strings can give this message. Inspect the offending statement, print out any values, and correct the cause.

STOPPED AT XXX

Where *xxx* is a line number. This message usually follows an error message, and tells you in which statement the error occured. This message is also given when **(CTRL) C** is typed to stop a program in the middle of execution.

#

There are two kinds of messages that begin with a pound sign (#). An example of the first kind is

```
#120 #130 #250
```

and so on, perhaps filling the screen. These are line numbers being shown, in order of execution, as a result of the **TRACE** feature being enabled. To turn it off, type

```
NOTRACE
```

and the numbers and pound signs will stop appearing. The other kind looks like

```
#15 I=55  #20 J=3
```

and so on. This is the result of a **DSP** statement in the program, which causes each variable named in the **DSP** statement to be displayed, with its current value and the line number, every time it is used in a statement. Eliminate the **DSP** statement(s) if this output is unwanted.

GROUP III

These messages occur when someone is typing in response to an **INPUT** statement's request.

```
*** TOO LONG ERR
```

You have typed more than 128 characters. Make your reply shorter next time you run the program. This can really mess up a non-programmer who is using your program, so warn people not to just type a lot of garbage when using your programs that require input. See this message in GROUP II.

```
RETYPE LINE
```

Notice that this message does not have three asterisks in front of it. It means that the information typed was not of the sort expected by the **INPUT** statement. The information should be retyped. The program has not been stopped. This message usually arises when someone types characters when numbers are expected, or types the wrong number of numbers. The best cure is prevention: make sure the message that preceeds the user's response is clear.

GROUP IV

These messages arise when **LOAD**ing programs from the cassette tape recorder.

```
ERR
*** MEM FULL ERR
ERR*** MEM FULL ERR
```

Any one of these three messages means the same thing: the tape has not read correctly. When loading, the `*** MEM FULL ERR` usually means that the tape has not read correctly. But if it is a long tape and you have a small (4K for example) machine, it *might* mean that the program on the tape is too long. If the tape was made on your computer, then it can't be too big. If it is a tape provided by APPLE or any other source, it should be marked with the memory requirements.

If the program on the tape is not too long, then either the volume control is too low or too high, the tone control is not at maximum, or some other problem exists in your tape recorder. To check out your tape recorder, follow the instructions in the material that came with the APPLE.

MAKING PROGRAMS RUN FASTER

You may occasionally wish to make a program run as fast as possible. To do this, some of the niceties of programming may have to be sacrificed. For example, since it takes time to skip over **REM** statements, when you are going for speed these should all be moved to the end of the program (after the **END** statement). This, like the other tricks being described here, makes the program less readable and more difficult to debug and modify. But racing cars are not made to be comfortable and carry a family of six to a picnic.

Here are some tips which can be used to speed up your programs:

1. Omit **REM**s or relocate them to the end of the program.
2. Place all subroutines *before* the main program , with the most commonly used subroutines coming first.
3. Use one-letter variable names. Shorter names run faster. Commonly used variable names should appear early in the program. Minimize the number of different variable names: reuse the same variable names wherever possible.
4. Use **FOR** loops instead of written-out loops.
5. Place as many statements as possible on one line, separating them with colons. It takes time to process line numbers.
6. Calculate common subexpressions once instead of each time they are needed. An example: You wish to test an element of an array **T(M)**, and if it is not zero, to let **B** equal the element squared divided by 5. You *could* write

   ```
   IF T(M)#0 THEN B=(T(M)*T(M))/5
   ```

   But it would be much faster in execution if you wrote

   ```
   A=T(M):IF A#0 THEN B=A*A/5
   ```

   Removing extra parentheses also adds speed—again at the expense of clarity. Use the rules of precedence.
7. If a subroutine is called from only one place in a program, write out the subroutine as part of the program. **GOSUB**s and **RETURN**s take time. If it is a short subroutine and is called from only two or three places, you may wish to write it out each time it is needed. This trades some memory space for speed.
8. Parts of a program that are executed only once do not need to be compressed for speed. Save your attention for those parts of the program that run repeatedly, in loops. For example, eliminating a **GOSUB** (as suggested in tip 7) is only worthwhile if the subroutine is called many times, from the same point in a loop.

Remember: these techniques are generally *poor* programming techniques and should *only* be used if you need to make a program run more rapidly. Many of these same tips will make a program fit into a smaller space as well as run faster. The exception is tip 7. Unless a subroutine is only one or two brief statements long or is called from only one place, using the subroutine

takes less room in memory than writing it out each time it is needed. Good use of subroutines can save a great deal of space.

There is one method of saving time and space in a program that beats all the others combined: *find a better method for solving the problem*. Since this is an encyclopaedic topic (there are hundreds of books and articles describing better techniques for solving problems with computers), we offer only one idea: when your program works correctly, and you want it to run faster, throw the program away (or hide the tape for a few days). Then reprogram the entire problem from the ground up. This method really works!

SOME OTHER FUNCTIONS AND ABILITIES

These are some miscellaneous items that are part of APPLE II BASIC, but were not mentioned in the main body of the book.

LET

The verb "LET" is allowed (as in LET T=6) for compatibility with earlier BASICs. It is not necessary.

ASC

This function returns a unique numerical value for each character. An example of its use is:

```
PRINT ASC("X")
```

which prints a numerical value for the letter "X." This numerical value is the ASCII code for the letter. If the value returned is greater than 128 then you must subtract 128 from the value to get the standard ASCII code.

SCRN

The function SCRN(X,Y) returns a number representing the color of the screen at the point (X,Y).

ABS

The function ABS(M) returns the absolute value of M. ABS (4) is 4, ABS (−67) is 67, and ABS (0) is 0.

SGN

The Function SGN(M) returns −1 if M is negative, 0 if M is zero, and 1 if M is positive.

There are two instructions, IN# and PR#, that are used to control accessories. Directions on how to use these instructions are included with the accessories.

When two or more NEXT statements occur one immediately after the other, as in

```
200 NEXT J
210 NEXT K
220 NEXT R
```

they may be combined in one statement

```
200 NEXT J,K,R
```

There is also a POP command. See the APPLE II Reference Manual for a description of this command, which allows you to leave a subroutine without using a RETURN .

POKE, PEEK AND CALL

Underneath the friendly plastic case and convenient BASIC language of your APPLE II lurks an even more powerful, but somewhat harder to use, naked microcomputer. You may never need to summon this genie (whose master, the **Monitor**, is covered in a separate manual), but you can get in touch with it from BASIC. This genie, who is called the 6502, is programmed in a language known as **Assembly Language**—which you do not need to know in order to use the APPLE II. However, when you master BASIC, and are looking for new worlds to conquer, remember that you can learn what lies beneath!

Some handy programs have been written in Assembly Language and are available to you even if you don't program in that language. To invoke these programs, you use statements that begin with the word "**CALL**." For example, you might want to clear the screen in the course of a program. Since you can't make a program type

SHIFT
ESC P

you would include a statement such as

```
310 CALL -936
```

This instruction inserts an assembler program for clearing the screen into your BASIC program. Other **CALL**s will be explained later.

Every computer has memory locations. On the APPLE they are numbered from −32767 to 32767. Normally, you do *not* even need to know that these locations exist, since BASIC uses them automatically. But, as you will see, there are times when you may want to put something into a particular memory location, or to see what is in a memory location. Putting something into a memory location is done with a **POKE** command, and examining such a location is done with a **PEEK** command. Some useful **POKE**s and **PEEK**s are covered next.

REFERENCE LIST OF POKES, PEEKS, AND CALLS

There are various **POKE**s, **PEEK**s and **CALL**s available in APPLE BASIC. The ones most commonly used are explained in full.

FULL SCREEN GRAPHICS

To understand the first set of **POKE**s, you need to understand how the TV screen's area is allocated. Up to now, graphics on the screen have been on a

40 by 40 grid, with four lines at the bottom for text. If you wish, you can devote the entire screen to graphics, with no area saved for text. The instruction to do this is

```
POKE -16302,0
```

To get back to mixed text and graphics use

```
POKE -16301,0
```

When the entire screen is graphics, you can use Y values from 0 to 47 instead of 0 to 39. Here's a program that uses full-screen graphics.

```
 90 REM  STARRY NIGHT PROGRAM
100 GR
110 REM SET TO FULL-SCREEN GRAPHICS
120 POKE -16302,0
130 REM  CLEAR BOTTOM OF SCREEN
140 COLOR=0: FOR I=40 TO 47: HLIN
    0,39 AT I: NEXT I
150 REM  PLOT STARS AND SPACES
160 COLOR= RND (15)+1
170 PLOT RND (40), RND (47)
180 COLOR=0
190 FOR I=1 TO 60
200 PLOT RND (40), RND (47)
210 NEXT I
220 GOTO 160
```

HOW TO INTERRUPT A PROGRAM BY TYPING A KEY

The starry night program runs forever until you stop it with a CTRL C or turn off the computer. But let's say that you wanted it to run until you directed it to do something else. You need to be able to *interrupt* the program without stopping it. Change line 190 as follows

```
190 FOR I=1 TO DARK
```

Add this line

```
95 DARK=60
```


Test the program, it should run as it did before. Now change line 220 and add some new lines:

```
220 REM  HAS ANY KEY BEEN PRESSED?
230 IF PEEK (-16384)<127 THEN GOTO
    160
240 POKE -16368,0
250 DARK=DARK-30: IF DARK<0 THEN
    DARK=60
260 GOTO 160
```

Try the new program. The value at location −16384 is usually less than 127. When any key is hit (except for RESET, CTRL, REPT or SHIFT) this value suddenly changes to be greater than 127. Thus by testing this location every time we go through the loop, we can tell approximately when a key has been hit, and make the program do this or that accordingly. It is good form always to have the statement **POKE −16368,0** right after the **PEEK** that reads the keyboard. This resets the keyboard so that your program can see when the next key is hit. At each press of any key (the space bar is especially handy) the "sky" will get brighter. Wait a while to see the effect and then hit the key again. Finally the value of **DARK** will be zero and the sky will be fully bright—good morning! Then you can press a key again, and have your starry sky.

GENERATING SOUNDS

As you have seen,

```
PEEK (-16336)
```

clicks the speakers of the APPLE II.

```
POKE -16336,0
```

will also click the speaker, and any program which repeatedly **PEEK**s or **POKE**s the address −16336 will produce a steady tone.

THE GAME-CONTROL BUTTON

You can tell if the buttons on the controllers are being pressed by **PEEK**ing
−16287 for the button on controller number zero, and
−16286 for the button on controller number one.

Try this program:

```
10 CALL -936
20 VTAB 10
30 PRINT "BUTTON ZERO= "; PEEK
   (-16287)>127
```

```
40 PRINT "BUTTON  ONE= "; PEEK
   (-16286)>127
50 GOTO 20
```

The result of the **PEEK**s in lines 30 and 40 will be greater than 127 if the appropriate button is being pressed.

TIRED OF WHITE ON BLACK?

The statement

```
POKE 50,63
```

will make all text printed by the computer appear in inverse (black on white), while the text you type will remain white on black.

```
POKE 50,255
```

will set things back to normal.

TEXT WINDOWING

When BASIC comes up, the text appears in all 40 columns. When the cursor reaches the end of the bottom line, the whole screen *"scrolls"* up. Using a **POKE** statement, you can make a smaller area scroll and the rest of the screen stand still. The area that scrolls is called the *"scrolling window."* All text activity will occur *inside* this "window." The **POKE**s that set the dimensions of this window are:

POKE 32, LEFTEDGE
POKE 33, WIDTH
POKE 34, TOP
POKE 35, BOTTM (we can't use the variable name BOTTOM because of the TO in bot**TO**m)

The values are normally set to:
LEFTEDGE = 0
WIDTH = 40
TOP = 0
BOTTM = 24
which results in the scrolling window being the entire screen (40 characters per line by 24 lines per screen). **POKE**ing these locations with different values will change the dimensions of the scrolling window. After changing the dimensions of the scrolling window, you must always move the cursor "into" the new window by means of either **VTAB**s and **TAB**s or the statement **CALL-936**. If you forget to move the cursor into the new window, any **PRINT** statements will print in wrong places on the screen and the screen will seem to behave very strangely (it will not damage any programs, however). Type this program in:


```
NEW
10 REM  CLEAR THE SCREEN
20 CALL -936
30 REM  SET NEW WINDOW DIMENSIONS
40 LEFTEDGE=15
50 WIDTH=12
60 TOP=10
70 BOTTM=17
80 REM  CHANGE SCROLLING WINDOW
90 POKE 32,LEFTEDGE: POKE 33,WIDTH:
    POKE 34,TOP: POKE 35,BOTTM
100 REM  MOVE THE CURSOR INSIDE THE
    WINDOW
110 CALL -936
120 REM  PRINT SOME STARS
130 PRINT "*";: GOTO 130
```

Try changing the dimensions of the scrolling window in lines 40 through 70 and RUNing the program. To return to the original scrolling window (the entire screen, 40 × 24) you can set the variables back to the original values and run the program or you can hit RESET and re-enter BASIC using

CTRL

C

RESET will always set the scrolling window to be the entire screen. Have fun.

COMPARING STRINGS

Two strings may be compared for equality as was done at line 320 in the program on page 89. Two strings may be compared for *inequality* using the # symbol. The other symbols used for comparing the magnitude of numbers (> >= < <= <>) may not be used with strings. You may not print the result of a string compare directly:
10 PRINT A$=B$
will give you a syntax error, however
10 C = A$=B$: PRINT C
is legal, and will print a 1 or a 0 depending on whether A$ is equal to B$ or not. This is a quirk in the language.

INDEX TO THE APPLE II BASIC PROGRAMMING MANUAL

-Cast of CHARACTERS-

In order of their appearance in the American Standard Code for Information interchange (ASCII). The number in parentheses is the ASCII code for the symbol.