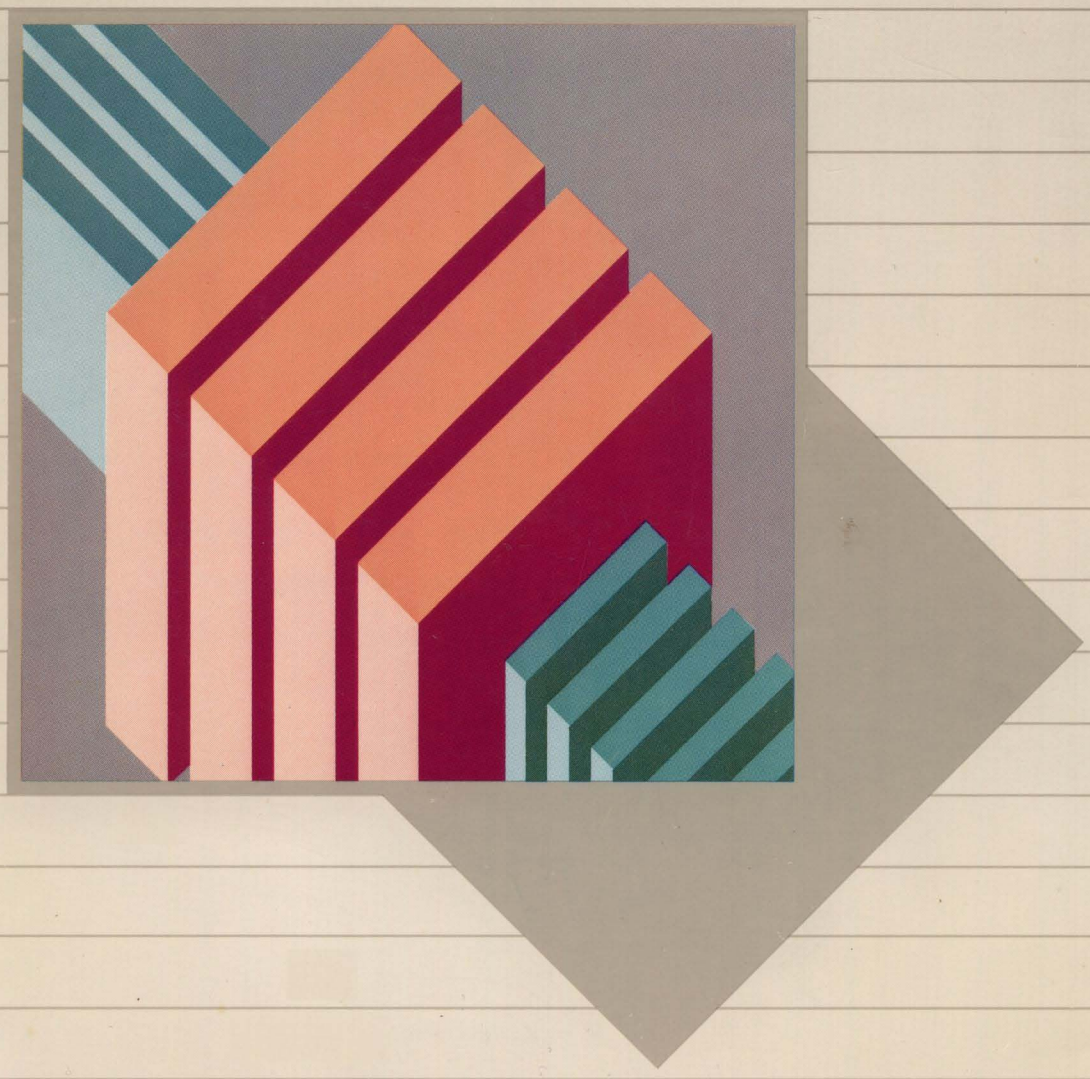
Apple *IIe*

Applesoft Tutorial

For *IIe* Only



Notice

Apple Computer, Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

Disclaimer of All Warranties and Liabilities

Apple Computer, Inc. makes no warranties, either express or implied, with respect to this manual or with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer, Inc. software is sold or licensed "as is." The entire risk as to its quality and performance is with the buyer. Should the programs prove defective following their purchase, the buyer (and not Apple Computer, Inc., its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer, Inc. has been advised of the possiblity of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

This manual is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer, Inc.

© 1982 by Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

The word Apple and the Apple logo are registered trademarks of Apple Computer, Inc.

Simultaneously published in the U.S.A and Canada.

Written by Meg Beeler of the Apple PCSD
Publications Department
Based on a manual by Jef Raskin

Reorder Apple Product #A2L2003

Apple IIe

Applesoft Tutorial

Radio and Television Interference

The equipment described in this manual generates and uses radio-frequency energy. If it is not installed and used properly, that is, in strict accordance with our instructions, it may cause interference with radio and television reception.

This equipment has been tested and complies with the limits for a Class B computing device in accordance with the specifications in Subpart J, Part 15, of FCC rules. These rules are designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that the interference will not occur in a particular installation, especially if you use a "rabbit ear" television antenna. (A "rabbit ear" antenna is the telescoping-rod type usually contained on TV receivers.)

You can determine whether your computer is causing interference by turning it off. If the interference stops, it was probably caused by the computer or its peripheral devices. To further isolate the problem:

- Disconnect the peripheral devices and their input/output cables one at a time. If the interference stops, it is caused by either the peripheral device or its I/O cable. These devices usually require shielded I/O cables. For Apple peripheral devices, you can obtain the proper shielded cable from your dealer. For non-Apple peripheral devices, contact the manufacturer or dealer for assistance.

If your computer does cause interference to radio or television reception, you can try to correct the interference by using one or more of the following measures:

- Turn the TV or radio antenna until the interference stops.
- Move the computer to one side or the other of the TV or radio.
- Move the computer farther away from the TV or radio.
- Plug the computer into an outlet that is on a different circuit than the TV or radio. (That is, make certain the computer and the radio or television set are on circuits controlled by different circuit breakers or fuses.)
- Consider installing a rooftop television antenna with coaxial cable lead-in between the antenna and TV.

If necessary, you should consult your dealer or an experienced radio/television technician for additional suggestions. You may find helpful the following booklet, prepared by the Federal Communications Commission:

"How to Identify and Resolve Radio-TV Interference Problems"

This booklet is available from the U.S. Government Printing Office, Washington, DC 20402, stock number 004-000-00345-4.

2 Elementary Programming 37

3 Making Changes 73

4 Lots of Graphics 89

E

More Programs To Play With 171

Learning To Program

Welcome! This is the manual for you if you want to learn to program or want to familiarize yourself with programming. Either way, it is designed so you can have fun while you learn.

You will learn, for example, how to tell your computer to say hello to your friends; how to get your computer to act like a calculator; and how to program your computer to draw horses all over the screen.

If you start at the beginning, try everything as it comes along, and make up your mind to take your time, it is pretty much guaranteed that you will learn how to program. The real secret is taking your time and trying everything.

You cannot learn how to program by reading this or any other book. When you learn to ride a bicycle, grow vegetables, or drive a car, you learn by doing. Making mistakes and correcting them is an important part of the process. It's the same with programming: as you follow the directions in this manual, don't worry if something doesn't work the first time. Figuring out what went wrong will help you learn that much more.

This manual is organized so each skill you learn is a stepping stone to the next. It also is organized to help you pace yourself; "pause" marks are noted at convenient stopping points.

What Is Programming?

A computer program is a set of instructions written in a code called a *computer language.* Programs make things happen. They may instruct a computer to display a message on the screen, to do complex tax preparation calculations, to create a game, or to make designs. Programs, in other words, enable you to accomplish many things with your computer.

BASIC is a commonly used personal computer language. It is easy to learn and easy to use. There are many varieties of BASIC; each type of personal computer uses a slightly different version. It is rather like regional dialects: even though people speak differently, they generally use the same vocabulary and can understand each other.

Applesoft BASIC, which is built into the Apple IIe, is a particularly powerful variety of BASIC. It has some graphics capabilities that many other varieties of BASIC don't have. This manual introduces you to Applesoft BASIC, but once you have learned Applesoft, you also will be able to understand other versions of BASIC.

Learning a computer language is like learning a second language: there are new words to learn, and there are rules about the relationships between them. There are about 100 words in the Applesoft BASIC programming language.

Getting Ready

Before you begin Chapter 1, get your machine and materials ready:

- A video monitor and a disk drive should be connected to the Apple IIe. The computer's and the monitor's power cords should be plugged into a grounded outlet. If you haven't gotten this far, go back to the *Apple IIe Owner's Manual* for installation instructions.
- Make sure you have the DOS 3.3 SYSTEM MASTER disk, the APPLESOFT SAMPLER disk, and an initialized blank disk. If you do not know how to prepare a new disk to receive information, see the *Apple IIe Owner's Manual.*
- If you use a cassette recorder instead of a disk drive, the special commands you'll need are listed in the *Applesoft BASIC Programmer's Reference Manual.*
- If you have an 80-column text card, keep it inactive while you use this manual: it is easier to learn Applesoft this way. Later you can read the *Apple IIe 80-Column Text Card Manual* to see how Applesoft works with the card active.

Things You Should Know

This tutorial is not a complete guide to Applesoft. To make things easier, information that a beginning programmer doesn't need is sometimes left out. In other words, don't be surprised that there is more to learn—that is the reason for the *Applesoft BASIC Programmer's Reference Manual.*

Word Meanings

Computer terms with which you may be unfamiliar are in italics. All italicized words are defined in the Glossary.

There are also several appendices that will help you with word meanings.

- Appendix A defines and summarizes all Applesoft statements.
- Appendix B lists reserved words (don't worry—you'll find out what they mean in due time).
- Appendix C explains error messages.

Symbols

There are three special symbols used in this manual.

The pause symbol marks points at which you can easily take a break. You don't have to stop working when you see this symbol—it's just an aid.

The gray box is used to clarify information or to remind you about some useful technique. It is set off from the main text with shading.

The warning symbol lets you know when there is some vital piece of information that you shouldn't miss. It also warns you when you must do something for your programs to work.

Where Else To Look

Before you begin this manual, you should read the *Apple IIe Owner's Manual* and run the APPLE PRESENTS...APPLE disk.

The companion to this manual is the *Applesoft BASIC Programmer's Reference Manual.* The reference manual contains detailed and complete information about Applesoft, has a handy reference card, and discusses the design of programs and good programming practices. Use the reference manual to increase your programming knowledge and skill once you have learned the basics from the *Applesoft Tutorial.*

The *DOS Manual* contains complete information about Disk Operating System commands and how DOS works. When you move on to advanced programming you may also want to read this manual.

Introducing Applesoft

Introducing Applesoft

Now it is time to get started! Find the disk labeled DOS 3.3 SYSTEM MASTER. Place it in the disk drive and close the drive door. Turn on your computer. When the red light on the disk drive goes off, you should see a square bracket *prompt* (]) and a blinking *cursor* at the left edge of your screen.

Figure 1-1. The Cursor and the Prompt

If you don't see them, check your system. Is the disk drive connected? Is the video monitor plugged in and turned on? Is the Apple IIe turned on?

> Applesoft BASIC is built into the Apple IIe, so a disk drive is not required. If you use a cassette recorder, see the *Applesoft Reference Manual.*

Once you have a cursor, press the CAPS LOCK key down into its on position. It is located at the lower-left corner of the keyboard. When the key is locked into position, you should hear a click and notice that it is lower than the other keys.

Applesoft BASIC instructions must be uppercase. The CAPS LOCK key makes it easy. Just like a typewriter's shift lock for alphabetic characters only, CAPS LOCK makes all letters uppercase, but also allows you to type numbers. If you want the upper character of a non-alphabetic key, you still must use the SHIFT key. For example, you'll use the SHIFT key to get the double quotation mark.

Warning

Whenever you are using Applesoft BASIC with the Apple IIe, make sure CAPS LOCK is on. If it is not on and you try to give an instruction in lowercase, you will be rewarded with a beep from your computer and a ?SYNTAX ERROR message. Applesoft only understands instructions in uppercase.

A First Statement

You are ready to give the Apple IIe some instructions, so type the words you see below. Remember that to get the double quotation mark you have to press the SHIFT key. Type

```
PRINT "HELLO"
```

and press the RETURN key.

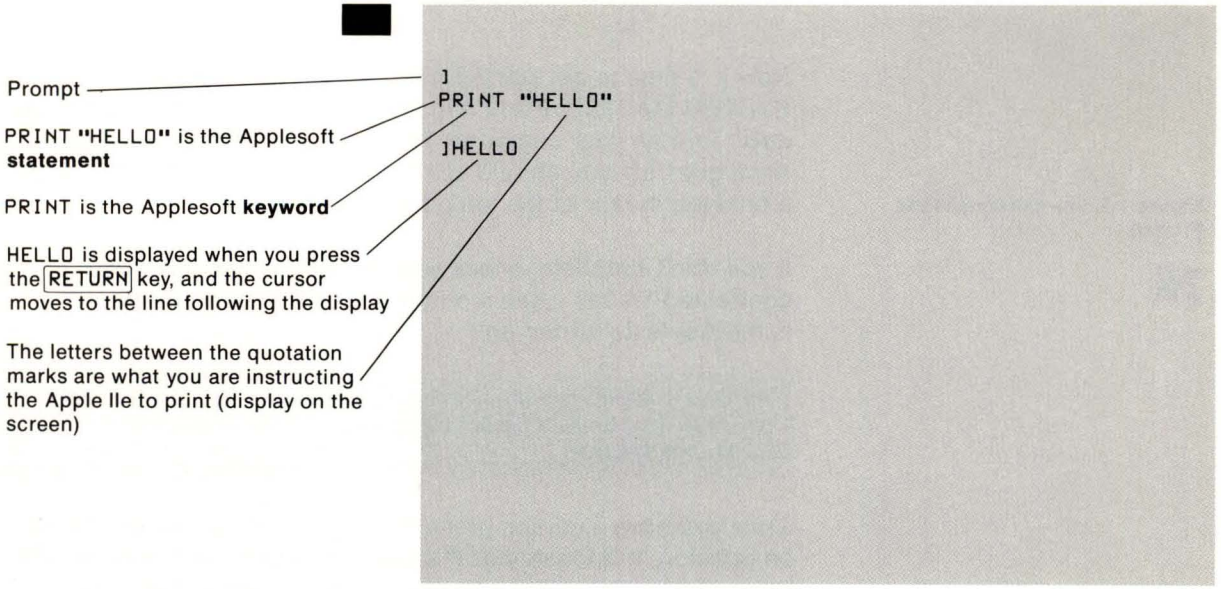


Figure 1-2. The PRINT Statement. When you give instructions like this to the Apple IIe, you should type exactly what you see in this manual, including spaces, uppercase letters, and quotation marks.

The Apple IIe should carry out your instructions by displaying the word

```
HELLO
```

on the next line. If it followed your instructions, congratulations!

If your screen doesn't look like the one in this manual, no problem. Try again. Look at your screen and carefully check what you typed. Notice that each of the items below corresponds to one of the examples in Figure 1-3.

1. Did you use uppercase letters? If not, press the CAPS LOCK key and try again.

2. Is the cursor blinking next to the second quotation mark? If so, press RETURN. (Every instruction must be followed by a press of the RETURN key to tell the Apple IIe that you have completed the instruction.)

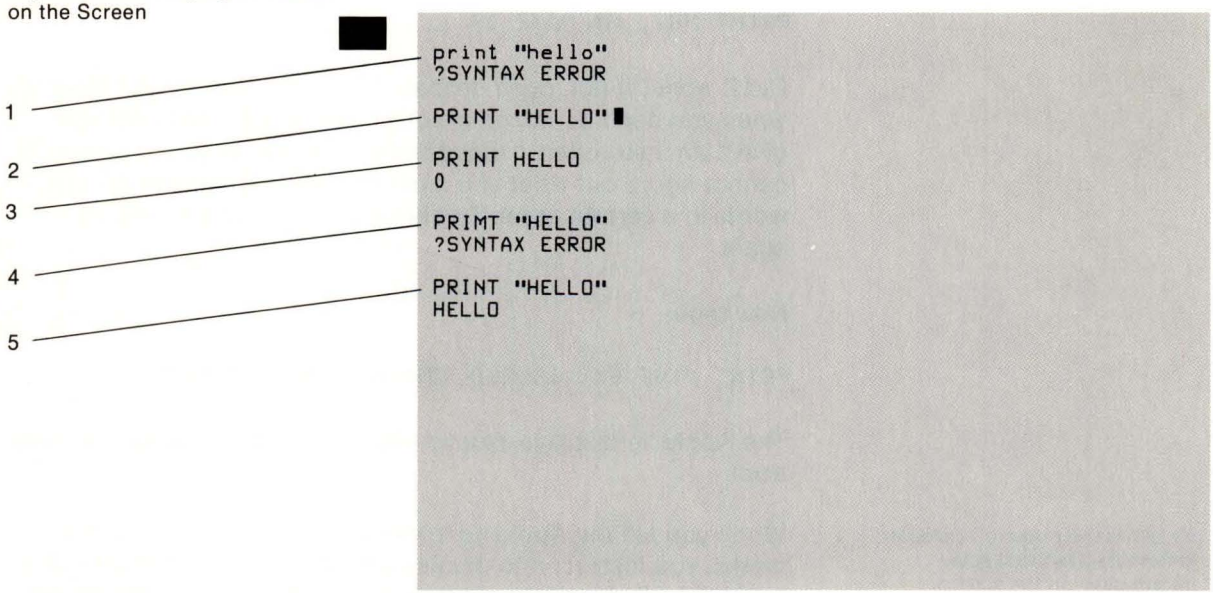
3. Are there quotation marks surrounding HELLO? If not, type the instruction again. If you used the single quotation mark, be sure to hold the SHIFT key down to produce the double quotation mark.

   If you forget the first quotation mark, the computer will print a zero right below your instruction.

4. Is PRINT spelled correctly? If not, try again.

5. Now you've got it right!

Figure 1-3. Identifying a Problem on the Screen



A **keyword** is a special word that identifies a particular Applesoft statement or command. PRINT is a keyword.

A **statement** is an instruction that is part of a computer language.

When you misspell a *keyword* (like PRINT) in a *statement* (PRINT "HELLO" is a statement), you get this error message:

```
?SYNTAX ERROR
```

If you don't notice a misspelling until you see this error message, all you need to do is type the statement again. Check to make sure it is correct before you press RETURN.

If you notice a mistake before you press RETURN, you can fix it without having to retype the whole line. Press the LEFT-ARROW key until the cursor is over the mistake. Type the correction. Then press the RIGHT-ARROW key until the cursor reaches the end of the line. Press RETURN.

Helpful Hint: A word to the wise: nobody's perfect. Unless you are familiar with a computer keyboard, it will probably take a while for your fingers to find the right keys every time. But all it takes is a little practice, so keep trying! You'll find that once you get used to the arrow keys, fixing mistakes won't take long. And soon you will realize that checking each line before you press RETURN makes life much easier.

Now type the next statement, filling in your name in place of the blank. And don't forget to press RETURN.

```
PRINT "HI, MY NAME IS _____."
```

Did it work? If not, don't despair. The computer doesn't blow up when you don't do things exactly right. It just waits until you give it an instruction it recognizes. Unlike a person, a computer cannot figure out what you mean; it only responds to certain words in a certain order. So check what you typed, and try again.

Now type

```
PRINT "THE SKY IS RED, THE GRASS IS BLUE"
```

The Apple IIe displays exactly what you told it to. Even if it's not true!

PRINT is the primary Applesoft statement used to display information on the screen.

When you tell the Apple IIe to PRINT something in quotation marks, you instruct it to display all the *characters* between the quotation marks on the display screen. You can use the PRINT statement to tell the computer to display any message you wish. Try it!

If you type much beyond 240 characters at once, (255 to be exact), the computer will beep and produce a backward slash when it reaches the *character limit*. Then you will have to start over again. Figure 1-4 illustrates the character limit.

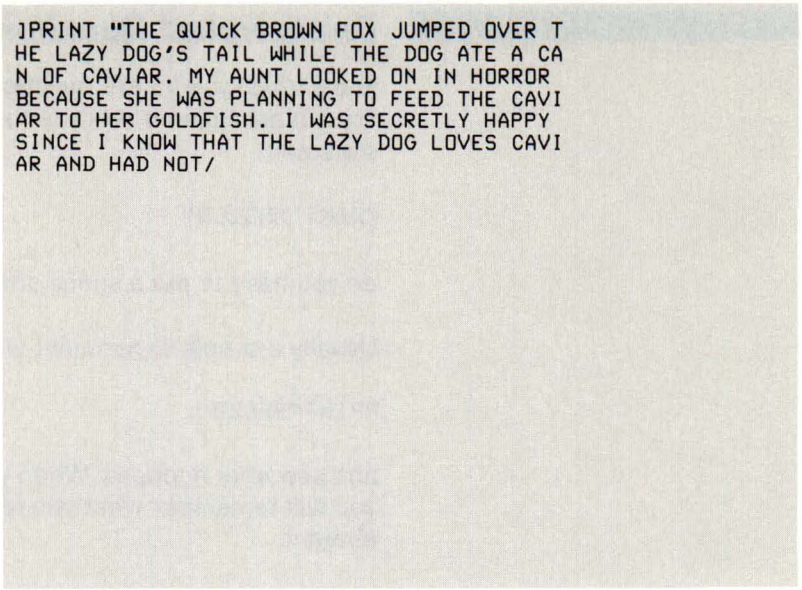


Figure 1-4. The Screen Character Limit. The length of a line on the screen is 40 characters. It takes 6 1/2 screen lines to reach the character limit of an Applesoft line.

When you want your computer to PRINT characters, you must use quotation marks. But if you type

```
PRINT 150
```

the computer prints the number 150 on the next line without any error message about the missing quotation marks. Type

```
PRINT "150"
```

and it does the same thing. This only works with numbers. If you try typing

```
PRINT HELLO
```

all you'll get is a zero.

Although both PRINT statements cause the same result, the difference between characters and numbers is very important. This difference will become clearer as you learn more about programming.

Unanswered Questions

Many times you'll have questions about Applesoft BASIC that are not answered directly in this book. For instance, in the statement

```
PRINT "HELLO"
```

do you have to put a space after PRINT?

Usually a simple experiment will answer your question. Type

```
PRINT"HELLO"
```

and see what happens. When you take the time to try it yourself, you will remember what you learn better than if you merely read about it.

When trial and error doesn't work, see this manual's appendices and the *Applesoft Reference Manual*.

Warning

The Apple IIe uses less electricity than an 11-watt light bulb. Whenever you take a short break, leave the computer turned on. You should, however, turn the monitor off, since it uses much more energy than the computer.

If you need to turn the computer off at one of the Pause marks, be sure to restart it with the DOS 3.3 SYSTEM MASTER disk.

Pause

Fancy Printing

Now that you've been introduced to the PRINT statement, how would you like to try another? Type

```
INVERSE
```

The INVERSE statement sets the video mode so that characters are displayed as black letters on a white background.

and press RETURN. Notice that the prompt looks different: it is black on a white background. To see what INVERSE does, type

```
PRINT "HELLO, HI, BONJOUR, BUENOS DIAS"
```

Isn't that nice? Try some more PRINT instructions of your own. Notice that you don't have to type INVERSE with each statement. Once you instruct the Apple IIe to use black-on-white, it will do so until you tell it to stop. To do that, type

NORMAL sets the video mode to the usual white letters on a black background.

```
NORMAL
```

and any instructions that follow will be white-on-black.

Another statement you can use, whenever you want to clear your screen and move the cursor to the upper-left corner of the screen, is

HOME moves the cursor to the upper-left corner of the text window and clears the screen.

```
HOME
```

Try it. Then type a few PRINT statements. Try HOME again. A cleared screen is at your command.

Here is another experiment. Don't forget to press RETURN after typing each statement. Type

```
INVERSE

HOME

PRINT "NOW THE SCREEN IS CLEARED AND THE WORDS
    APPEAR IN BLACK-ON-WHITE"
```

Try a few more PRINT instructions and see what happens.

Each of these statements tell the Apple IIe to perform a specific function: HOME clears the screen and returns the cursor to the upper-left corner; INVERSE produces black letters on a white background; and NORMAL returns the screen to white letters on a black background.

Doing Calculations

Without further study, the Apple IIe can be used as an ordinary desk calculator.

Try this on the Apple IIe:

```
PRINT 3 + 4
```

The answer, 7, appears on the next line.

The Apple IIe can do five different elementary *arithmetic operations:*

1. **Addition** is indicated by the usual plus sign (+).
2. **Subtraction** uses the conventional minus sign (−). To try subtraction, type

   ```
   PRINT 1086-99
   ```

3. **Multiplication** is indicated with an asterisk (*). (If an *X* were used for multiplication, it could be confused with the letter *X*). To find 7 times 8 (in case you don't remember the answer), just type

   ```
   PRINT 7 * 8
   ```

   and have your memory jogged.

4. **Division** is indicated by a slash (/). To divide 63 by 7, type

   ```
   PRINT 63 / 7
   ```

   and the correct answer will appear.

   Try dividing 3 by 2. The Apple IIe gives the answer to you in the decimal form: `1.5`.

5. **Exponentiation** is indicated by a caret (^). It is often handy to multiply a number by itself a given number of times. Instead of writing

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

   you can substitute the shorthand

   ```
   PRINT 4 ^ 5
   ```

   To type the caret (sometimes called a circumflex or an upward-pointing arrow), press the SHIFT key while holding down the 6 key.

   There is nothing special about exponentiation. It is just an abbreviation for repeated multiplication. In noncomputer notation, it is referred to as 4 to the 5th power and is written like this:

   4^5

You can combine several of these arithmetic operations within the same statement. For example, you can instruct the Apple IIe to

```
PRINT 3 + 51 + 9 + 400
```

or

```
PRINT 3 + 5 / 9 * 4 ^ 2 - 1
```

```
PRINT 3 + 51 + 9 + 400
463

PRINT 3 + 5 / 9 * 4 ^ 2 - 1
10.8888889
```

The exact rules for such usage are given later in the chapter, but you can experiment now if you wish.

Remember, Applesoft can't use numbers in quotation marks for arithmetic operations.

When you type

```
PRINT 45.340
```

your computer responds with

```
45.34
```

and doesn't display the trailing zero. The Apple IIe does not display leading or trailing zeros, that is, zeros that are at the beginning of a number and to the left of the decimal or zeros that are at the end of a number and to the right of the decimal.

Very, very small numbers (between about .0000000000000000000000000000000000000003 and −.0000000000000000000000000000000000000003) will be converted to zero by the Apple IIe. Don't take our word for it. Try it yourself. An easier way to write these numbers is `PRINT 3 * 10 ^ - 39` and `PRINT -3 * 10 ^ -39`.

Now type

```
PRINT 985788.6898
```

Surprise! The last two digits are lost, and the number left behind is the closest approximation the computer can find. This process is called rounding. Try typing

```
PRINT 788.6898
```

The Apple IIe did not round the number, but displayed it just the way you typed it. Madness you say? Ah, but there is a method to this seeming madness. Numbers are rounded only if they have more than nine digits. Any number that has fewer than ten digits will not be rounded. Applesoft does the best it can, but it can only work with nine digits.

If you type a PRINT statement with a large number like

```
1234567890
```

the Apple IIe responds with

```
1.23456789E+09
```

If you PRINT the number 10 billion (which has ten zeros), the Apple IIe responds with

```
1E+10
```

The numbers `10000000000` and `1E+10` and the numbers `1234567890` and `1.23456789E+09` have the same value. Really. The number displayed by your computer is in a form called *scientific notation.* If you need numbers like this you probably know how to read them. The *Applesoft Reference Manual* has more information if you are curious about this notation.

What About RETURN?

So far, you have been pressing the RETURN key after every line. You might like to know why this key gets so overworked. The reason is simple: without the RETURN, the computer does not know when you have completed an instruction. For example, if you typed

```
PRINT 4 + 5
```

and the computer immediately answered with a 9, you might be upset because you had planned to type

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

which nets a different answer entirely. Since the computer can't tell when you have finished typing an instruction, you must tell it. You do this by pressing the RETURN key.

Warning

Since you always have to press the RETURN key after typing an instruction, this manual will no longer remind you. Pressing RETURN after each instruction should be a habit by now, if you have been doing all the examples.

Pause

A Graphic Change of Pace

So far, you have been playing with words and numbers in Applesoft. Now it's time for a change of pace. Read on, and you will discover some of the wonderful graphics capabilities of the Apple IIe.

To prepare the screen for drawing, type

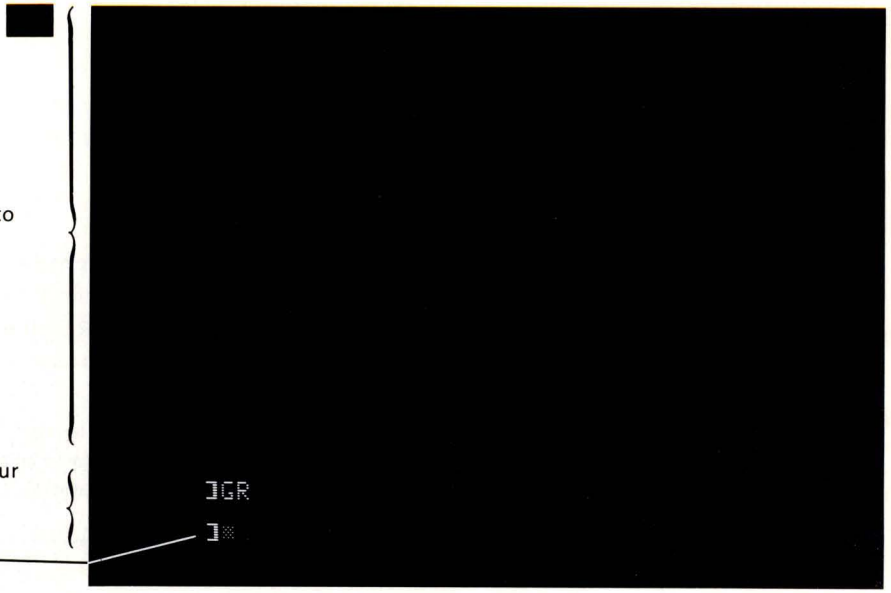
The GR statement sets the stage for low-resolution graphics.

```
GR
```

When you type GR, short for "graphics," you'll notice that a design quickly flashes past, the screen clears, and the cursor moves to the bottom of the screen. (If your screen didn't clear, you probably forgot to press RETURN. And forgot that you won't be reminded any more.)

The GR statement instructs the computer to set up an invisible grid of 40 vertical columns and 40 horizontal rows on which you can draw. This is called the *low-resolution graphics mode.* The GR statement also instructs the computer to leave enough space at the bottom of the screen for four lines of text, called the *text window.* GR also clears the screen, and sets the color to black.

Figure 1-5. The Low-Resolution Graphics Mode



The GR statement sets the stage for doing all kinds of drawing and fancy visual work. However, before you can see the results of what you do on the screen, you have to tell the computer to use a color that will show up on the black background. Try this now, by typing these two instructions:

To set the color in low-resolution graphics use the statement COLOR=, followed by an integer from 0 to 15. The PLOT statement places a "brick" at the specified location.

```
COLOR = 15
PLOT 20,20
```

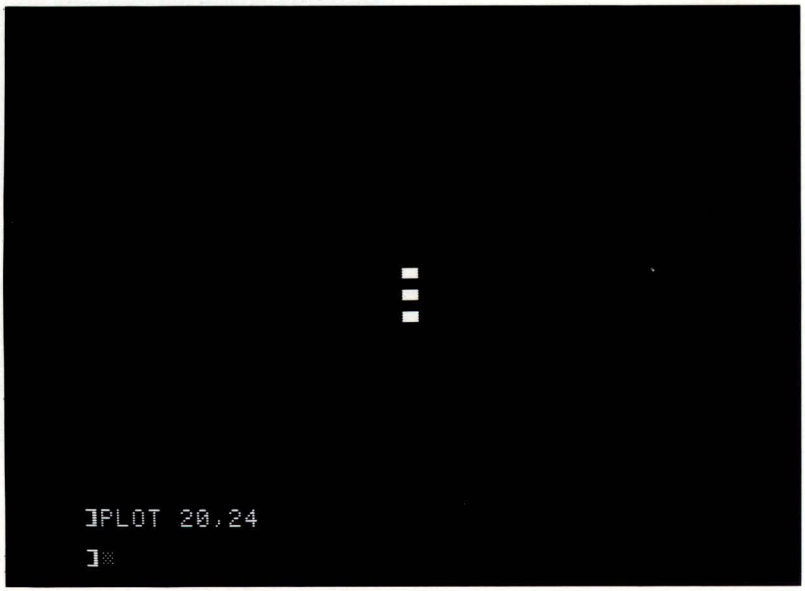
Do you see the white brick (a small rectangle) in the middle of your screen? To place two more bricks along the same vertical line, type

```
PLOT 20,22
PLOT 20,24
```

Note: If nothing appears on your screen, chances are you forgot to specify a color. You must do this, no matter what kind of display device you are using.

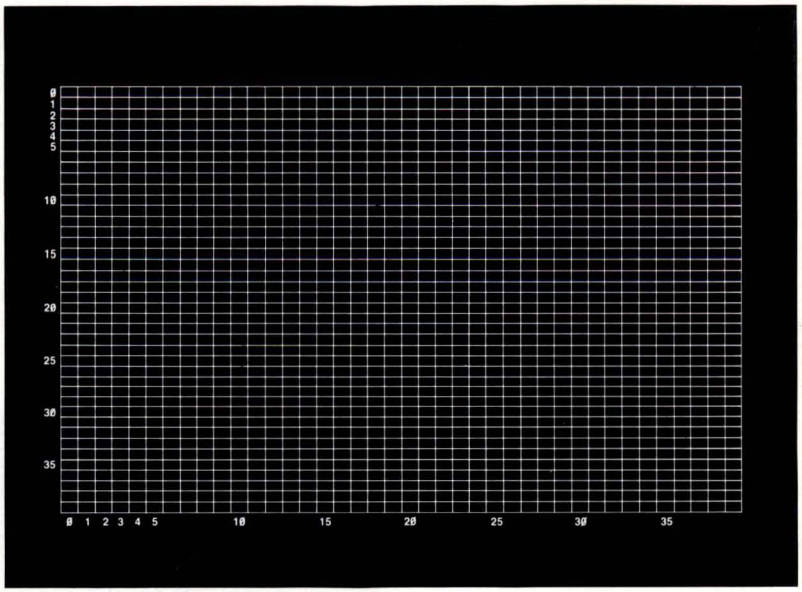
The PLOT statement tells the computer where to put each brick of color by assigning two numbers on the invisible grid. The first number in the three PLOT statements you have used so far, 20, names the 20th vertical column out of 40. The vertical column number is always the first in a PLOT statement. The second number (you have used 20, 22, and 24) names the horizontal row.

Figure 1-6. Using PLOT Statements



Now try plotting some bricks on your own. Can you put some in the leftmost vertical column? (Hint: the first number after PLOT should be 0, although 1 also will work.) What about the vertical column on the far right? (Hint: the first number after PLOT should be 39.)

Figure 1-7. Screen Grid Numbering System in Low-Resolution Graphics



Notice in Figure 1-7 that the numbering system for the vertical columns goes from 0 to 39, left to right. The numbers for the horizontal rows also go from 0 to 39, beginning at the top of the screen and moving down. Those of you who know algebra will recognize that this is a system of Cartesian coordinates. This book refers to the coordinates as columns and rows. Since the screen is already divided into 40 vertical columns and 40 horizontal rows, all you have to do is name the coordinate points that go with each `PLOT`.

The color of the bricks you are plotting is determined by the `COLOR=` statement. The Apple IIe will keep using whatever color you assign until you instruct it to change to another color. Try that now with

```
COLOR = 5
```

Notice that nothing new has happened on your screen yet. You have to tell your Apple IIe where to put each brick, so type

```
PLOT 20,21
```

And what happens? A brick of a different shade should appear in the same vertical column you have been using, but in a different row. Try

```
PLOT 20,23
PLOT 20,0
```

You should see two more bricks in the same column.

So far, if you have been trying these exercises, you have produced two shades of bricks on your screen. How does this work?

There are sixteen numbers assigned to `COLOR=` in Applesoft. Look at Figure 1-8 to see which number goes with which color.

Figure 1-8. Applesoft `COLOR=` Names and Numbers

0	black	8	brown
1	magenta	9	orange
2	dark blue	10	gray
3	purple	11	pink
4	dark green	12	green
5	gray	13	yellow
6	medium blue	14	aqua
7	light blue	15	white

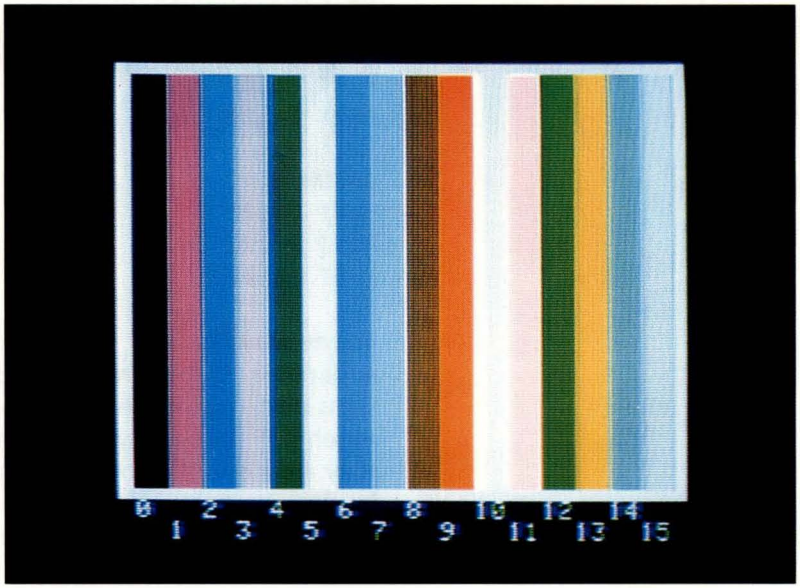
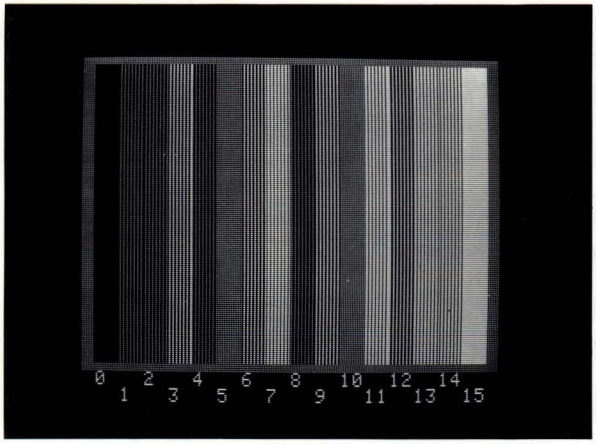
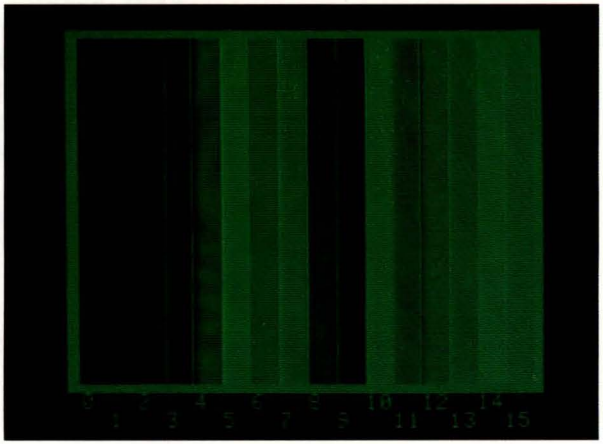


Figure 1-9. Color Groups. For good contrast on a black-and-white or green-phosphor monitor, use a number from each group.





Dark gray: 1, 2, 4, 8
Medium gray: 5, 10
Light gray: 3, 6, 9, 12
Pale gray: 7, 11, 13, 14
White: 15

If you are using a black-and-white or a green-phosphor monitor, there are five groupings of colors that will contrast well on your screen. All of the sixteen color numbers can be used, and you should feel free to experiment with them.

If You Have a Color Television Set: You can hook up a color television set to the Apple IIe with a *radio-frequency (RF) modulator.* All of the graphics examples in this manual work well in color. You don't have to stick with the color groups, however, because all 16 colors contrast well on a color screen.

Since you have bricks on your screen that have been plotted with `COLOR =15` and `COLOR = 5`, try typing these instructions to see some more contrasts:

```
COLOR = 3
PLOT 21,0
PLOT 21,20

COLOR = 1
PLOT 22,22
PLOT 22,25
PLOT 22,26
```

To see why `COLOR = 1, 2, 4,` and `8` are grouped together, experiment by plotting bricks of those colors near each other. On a black-and-white or green-phosphor monitor you can't tell much difference between them. On a color monitor, of course, they are distinct colors (magenta, dark blue, dark green, and brown). Now try

```
COLOR = 0
PLOT 22,27
```

Nothing happens, right? Take a look at Figure 1-8 to see why. The same thing will happen whenever you give the `GR` statement without indicating a color. This is because the color is initially set to zero, or black. Remember?

Now try some more `COLOR=` and `PLOT` statements. Keep practicing until you get the hang of it.

When you want to clear your screen and do some new `PLOT` statements, type

```
GR
```

When you are doing graphics, `HOME` only clears the text window. `GR` must be used to clear the graphics portion of the screen.

PLOT *Error Messages*

There are two error messages that often turn up when using the PLOT statement. You probably already know that if you type

```
PLAT
```

or

```
PLOP
```

instead of

```
PLOT
```

you get the message

```
?SYNTAX ERROR
```

A different error message occurs when you use a number larger or smaller than those permitted for coordinates in a PLOT statement. Type

```
PLOT 13,85
```

and you get the message

```
?ILLEGAL QUANTITY ERROR
```

This message means that you have tried to plot a point out of range and off the screen. The highest number you should try to use in a PLOT statement is 39 because the coordinates range from 0 to 39.

Aside: There are, in fact, some ways to plot row numbers larger than 39. They are discussed in the *Applesoft Reference Manual.*

Trying to use negative values in a PLOT statement is another way to get the

```
?ILLEGAL QUANTITY ERROR
```

message.

Pause

Drawing Lines

Prepare your screen for some new graphics by typing

```
GR
COLOR = 1
```

You must have discovered by now that it takes a lot of instructions to plot a line on the screen. It would, for example, take 40 statements to draw a horizontal line all the way across the middle of the screen:

```
PLOT 0,20
PLOT 1,20
PLOT 2,20
```

and so on, until

```
PLOT 39,20
```

However, there is an easier way to make horizontal lines. Just type

The HLIN statement is used to draw horizontal lines in low-resolution graphics.

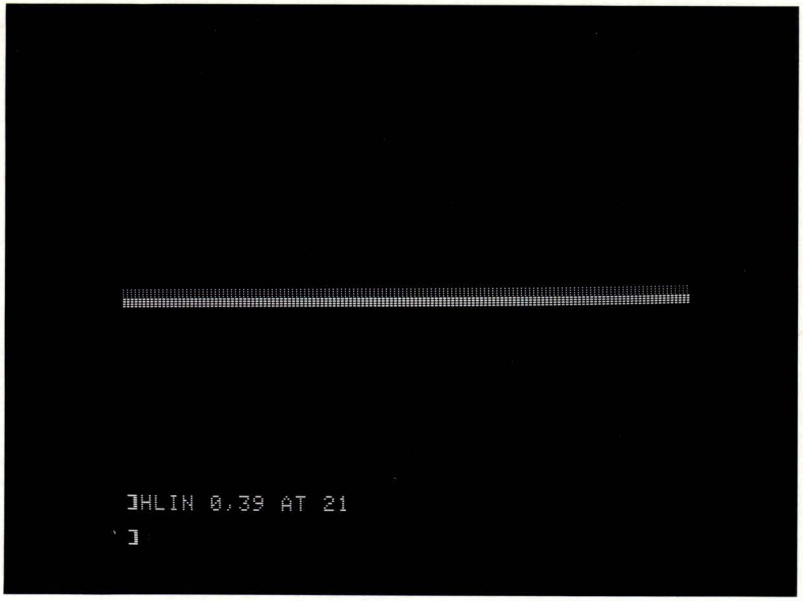
```
HLIN 0,39 AT 20
```

And there you have it: an instant horizontal line from column 0 to column 39 at row 20.

To draw a contrasting horizontal line just below the first, type

```
COLOR = 7
HLIN 0,39 AT 21
```

Figure 1-10. The HLIN Statement



Now you figure out how to draw a third contrasting line at row 22. This time, however, have it run from column 10 to column 30.

To understand the HLIN statement, look carefully at the order of the numbers after HLIN. The *syntax*, or rules for writing the statement, require the use of three numbers in a certain order. The third contrasting line, for example, goes from column 10 to column 30 at row 22 and is written as shown in Figure 1-11.

Figure 1-11. HLIN Syntax

```
HLIN 10,30 AT 22

From left column number (10), to right
column number (30) at row number (22)
```

To see how PLOT and HLIN can be used together, try the following:

```
COLOR = 8
HLIN 29,39 AT 35
HLIN 29,39 AT 37
COLOR = 10
PLOT 29,36
PLOT 39,36
```

There is a statement for vertical lines similar to that for horizontal lines. To draw another, larger, rectangular box, type

In low-resolution graphics, VLIN draws a vertical line in the color indicated by the most recent COLOR= statement.

```
COLOR = 15
VLIN 0,20 AT 0
VLIN 0,20 AT 15
COLOR = 5
HLIN 0,15 AT 0
HLIN 1,14 AT 20
```

Notice that when the first horizontal line is drawn over the ends of the vertical lines, the new color takes over, and the old color disappears. Practice making more vertical lines. Then, to clear the screen, use the GR statement.

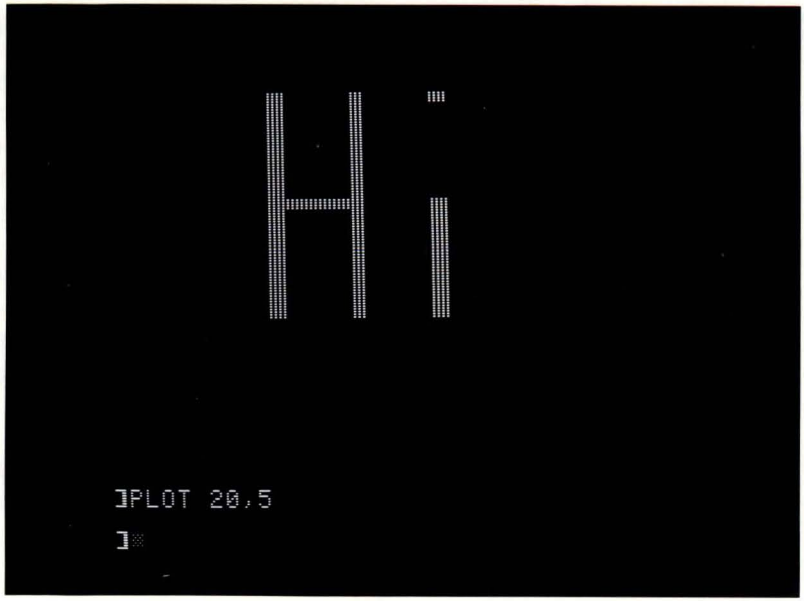
Now test your proficiency with horizontal and vertical lines by drawing a border around the screen in five statements. Put a cross on the screen. Play with PLOT, HLIN, and VLIN until you can put lines exactly where you want them.

Now, here's a test. Look at Figure 1-12. Use the number coordinates on the grid to determine how to give one HLIN and three VLIN instructions to write "Hi" on your screen. Start with

```
COLOR = 3
VLIN 5,25 AT 10
```

and figure out the rest. Then dot the *i* with a PLOT statement. If you like, add an exclamation point at column 28 (just for show).

Figure 1-12. Using Low-Resolution Grid Coordinates



The TEXT statement sets the screen to the nongraphics text mode: 40 characters per line and 24 lines. When used to leave the graphics mode, it is best used in conjunction with the HOME statement.

When you are finished with graphics, the TEXT statement returns the computer to text mode so you can use the full screen to work with programs. When you type

```
TEXT
```

you'll see a screenful of funny symbols. The technical term for this is garbage. It is just an indication that Applesoft is switching modes from graphics to text. To remove the garbage from your screen, follow the TEXT statement with HOME. As you already know, HOME clears text from the screen.

Since this manual is leaving the graphics mode for a while, make sure you return the Apple IIe to text mode before you continue.

Pause

Variables and More Calculator Abilities

In the *main memory* of your Apple IIe are a large number of special storage spaces. In each one of these spaces you can store a number, the result of a computation, words, or a group of characters. To accomplish this you need symbols to represent the various locations, or storage spaces.

It is a bit like saving a number for later use on a simple calculator. Usually this is done on the calculator by pressing a memory key. On the Apple IIe, because it has many more storage spaces than a calculator, you give a name to each space.

Say you want to save the number 77. If you wanted to name the storage space A, you would instruct the Apple IIe to

The LET statement is used to define a variable.

```
LET A = 77
```

The number, or value, 77, is not printed. It is stored in the space you have called A. If you now type

If the statement LET A = 77 has disappeared from your screen, you forgot to leave the graphics mode by typing TEXT. If you have a lot of garbage on your screen, you forgot to clear the screen with HOME.

```
PRINT A
```

the computer will print the value of the variable A, which is 77.

Try typing the two statements.

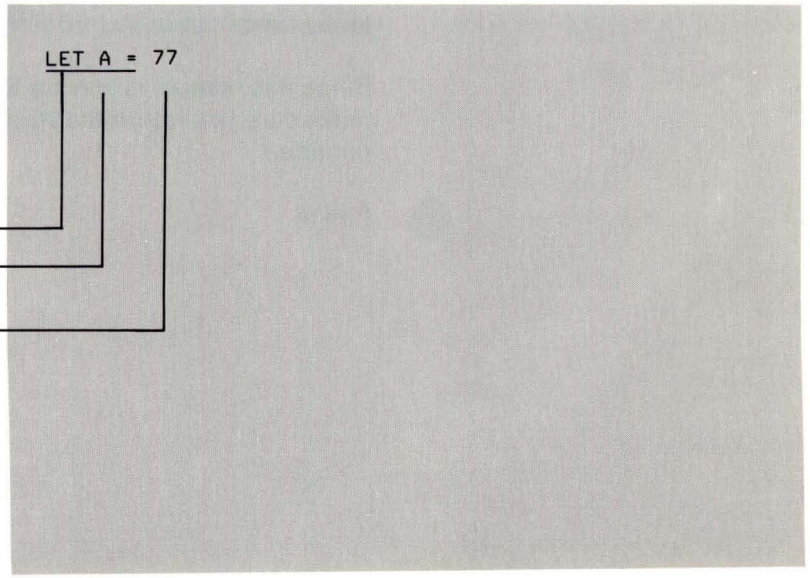


Figure 1-13. Defining a Variable with LET. A **variable** is a symbol that represents a location in memory. You can think of it as a place where one value, such as 77, is stored.

A variable can have almost any name as long as it starts with a letter. For example:

```
LET A = 77
LET RED3 = 77
LET CLOTHING = 77
```

Each of these variables represents a different location in memory. They all store the same value, 77. Now if you type

```
CLOTHING = 100
```

and print the value of CLOTHING, you get 100, right? The 77 is gone, and you have assigned a new value to CLOTHING.

> Whenever you put a new value in a variable, the old value is erased from memory.

However, the variables A and RED3 still contain the value 77. Try typing

```
PRINT RED3
```

to see. Is the same true for variable A?

You may have noticed that in the CLOTHING = 100 statement you didn't type LET. When naming a variable, LET is optional. In other words, the statements

```
LET A = 45
```

and

```
A = 45
```

give the same result. Sometimes it is easier to recognize a variable, when you are first learning Applesoft, if it is preceded by LET. But it's your choice.

You may find that it is easier on your fingers to use brief names for variables. You'll also discover that some names are not allowed because they include a word that has a special meaning in Applesoft. 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 is the ?SYNTAX ERROR message, which means you have unwittingly included a reserved word in the name. Don't worry. Just choose another name. In this case, a change to the British spelling will solve the problem and still give you a meaningful variable name: COLOURFUL.

A list of reserved words that cannot be used as variables or as part of variable names can be found in Appendix B. There is also a good deal more information on variables in the *Applesoft Reference Manual*.

Distinguishing Variables

Applesoft uses only the first two characters of a variable name to distinguish one variable from another.

To see how this works, try typing

```
BIRD = 11
PRINT BIRD
```

Did you get what you expected? Now type

```
PRINT BITE
```

What happens? Try

```
PRINT BILLOW
```

All these names begin with BI. So BIRD, BITE, and BILLOW all refer to the same variable.

There is a big difference between the way Applesoft interprets

```
PRINT CAT
```

and

```
PRINT "CAT"
```

What happens when you try them?

It is just like the difference between these two sentences: mice have four feet; "mice" has four letters.

The first sentence refers to little furry creatures with long tails. The second sentence refers to the word itself. This is how quotation marks are used in Applesoft. When you type

```
PRINT "CAT"
```

you are instructing the Apple IIe to print the word. When you type

```
PRINT CAT
```

you want the Apple IIe to print what the word stands for.

> Whenever you omit quotation marks in a PRINT statement, Applesoft will treat the word or letter like a variable and try to print the value of the word or letter.

So far, you have only used variables with a single number. You also can store the result of a computation in a variable. Type

```
A = 4 + 5
PRINT A
```

The value of A is 9, right? Now you can use the value of A in further computations. For example, try this on your computer:

```
PRINT A + 2
```

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

> Applesoft takes everything to the right of the equal sign (=), figures it out, and puts the result into the variable on the left of the sign.

Now let's say FOOD has the value of 28, and you want to increase this value by 5. If you type

```
FOOD = 28
PRINT FOOD
PRINT FOOD + 5
```

you'll get 33. But when you type

```
PRINT FOOD
```

again, Applesoft gives you the original value of the variable FOOD. The way to increase the value of the variable in memory is to type

```
FOOD = FOOD + 5
PRINT FOOD
```

The statement FOOD = FOOD + 5 may seem irrational until you recall that Applesoft doesn't require you to type in the assumed LET. What you really are saying is "let FOOD now equal what FOOD equaled up until now plus 5."

So FOOD has a new value—until you assign another value to it. Try that now. Type the statements below in order:

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

At the end of this sequence of statements, you should have the value 3. Is this correct? Is this what you expected?

In contrast to the math you learned in school, where X always is equal to X, the computer adds the element of change to variables. In the statement X = X + 1, the equal sign means "receives the value" or "is assigned the value."

The Apple IIe is able to store each new value of the variable (FOOD) as it is assigned or computed. You don't have to PRINT FOOD each time for that to happen. Just to make sure, try it now.

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

Look at the sequence below. What answer do you expect? Try it.

```
APPLES = 55
BANANAS = 11
QUOTIENT = APPLES / BANANAS
PRINT QUOTIENT
```

You see from this example that once you have stored several variables in the Apple IIe you can do computations using the variable names. Apples can't actually be divided by bananas, but Applesoft recognizes the words as variable names and uses their values in the division.

This section has explained how to store numbers and the results of computations in variables. As you may recall, it is also possible to store words and groups of characters in variables. Chapter 5 will return to this subject.

Summary of Variable Rules

1. A variable is a symbol that represents a location in memory.
2. The LET statement is used to define a variable.
3. LET is optional; LET A = 23 and A = 23 give the same result.
4. Applesoft takes everything to the right of the equal sign, figures it out, and puts the result into the variable on the left of the sign.

5. A variable name must begin with a letter. It is a good idea to give meaningful names to your variables for easier identification.
6. Applesoft uses only the first two characters of a variable name to distinguish one variable from another.
7. All Applesoft keywords and some other words are reserved and cannot be used as variable names.

Pause

A Big Timesaver

You may have noticed by now that the PRINT statement is used a lot in Applesoft.

However, you are about to learn a wonderful timesaver: a question mark can be used in place of PRINT. In fact, ? and PRINT mean the same thing in Applesoft. Try it out now by typing

```
?"WOW!"
```

Precedence

Now that you are using the Apple IIe for computations, you need to know what order, or *precedence*, Applesoft uses in carrying out your instructions. In a calculation like

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

will the answer be 6 or 8? It depends on which computation Applesoft does first. If Applesoft adds 4 to 8, then divides 12 by 2, the result is 6. If Applesoft divides 8 by 2, then adds 4, the result is 8. Look at Figure 1-14 to see the order Applesoft follows.

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

Figure 1-14. Precedence Example. Division is done first. Then the computation becomes 4 + 4, and the addition is done.

Here are some more examples:

1. When a minus sign is used to indicate a negative number, it is called a *unary* minus sign. In the example

```
PRINT -3 + 2
```

   Applesoft applies the unary minus sign to its appropriate number or variable before doing any arithmetic operations. Thus `-3 + 2` evaluates to `-1`. If it 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 *arithmetic expression* `5 -3` the minus sign is indicating subtraction, not a negative number.)

2. After identifying all negative numbers, Applesoft then does exponentiations. The statement

```
PRINT 4 + 3 ^ 2
```

   is evaluated by multiplying `3` by itself (`3 * 3 = 9`) and then adding `4` for a grand total of `13`. When there are a number of exponentiations, they are done from left to right, so that

```
PRINT 2 ^ 3 ^ 2
```

   is evaluated by multiplying `2` by itself three times (`2 * 2 *2`), which is `8`, and then multiplying that by itself (`8 * 8`). The answer is `64`.

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

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

Be assured that you don't have to memorize the order of precedence to use the computer. You can always refer to these pages when you need to know.

Figure 1-15. Applesoft's Order of Precedence for Carrying Out Arithmetic Operations

First:	-	Unary, or minus, sign used to indicate a negative number
Second:	^	Exponentiations, from left to right
Third:	* /	Multiplications and divisions, from left to right
Fourth:	+ -	Additions and subtractions, from left to right

Some arithmetic expressions to evaluate follow. With each one, first do it yourself and then try it on the Apple IIe. If your answer is different from the computer's, try to find out why.

Remember to preface each computation with ? or PRINT. Unless you already understand the way computers evaluate expressions, you should do these examples one at a time, checking your answer against the computer's as you go.

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

No answers are given in this book. The Apple IIe will give you the correct answers. If you like doing these, try some of your own. If not, keep going!

Parenthetically Speaking

Now, suppose you want to divide 12 by the result of 4 + 2. If you type

```
PRINT 12 / 4 + 2
```

you will get the answer 5. But this is not what you meant. To accomplish what you meant in the first place, use parentheses to modify the precedence. Type

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

The rule Applesoft follows is simple: it does what is in parentheses first. If there are parentheses within parentheses, Applesoft does what is in the innermost parentheses first. Here is an example:

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

In this case, doing the innermost parentheses, Applesoft first adds `1 + 2`. Now the expression is:

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

Then Applesoft follows the rules of precedence to determine that: `3 + 3 ^ 2` is `3 + 9`, or `12`, and that `12 / 12` is `1`.

In a case like `(9 + 4) * (1 + 2)`, where there is more than one set of parentheses, Applesoft does the operations within each set, starting at the left and working to the right. This expression becomes `13 * 3`, or `39`.

Here are some more expressions to evaluate. Remember that you can substitute a question mark (`?`) for `PRINT`. Incidentally, most of these rules for precedence and parentheses hold good for most computer systems anywhere in the world, not just the Apple computer.

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

Pause

Chapter Summary

Here is what you have learned in this chapter. Impressive, isn't it? Each category lists the terms in order of their appearance in the chapter. Definitions of statements are in Appendix A. Glossary terms are defined in the Glossary.

Applesoft Statements

PRINT
INVERSE
NORMAL
HOME
GR
COLOR=
PLOT
HLIN
VLIN
TEXT
LET

Arithmetic Operations

addition
subtraction
multiplication
division
exponentiation

Glossary Terms

prompt
cursor
keyword
statement
character
character limit
arithmetic operations
scientific notation
low-resolution graphics mode
text window
radio-frequency modulator
syntax
main memory
value
variable
reserved word
precedence
unary
arithmetic expression
operator

Keys

CAPS LOCK
SHIFT
RETURN
LEFT-ARROW
RIGHT-ARROW

Error Messages

?SYNTAX ERROR
?ILLEGAL QUANTITY ERROR

Chapter 2

Elementary Programming

Elementary Programming

When a computer performs according to the instructions you have given it, it *executes* the statement. Up to now, when you typed

```
PRINT 3 + 4
```

and pressed RETURN, the Apple IIe would do what you told it to do, immediately. This is called *immediate execution*. You can try out nearly every Applesoft statement in immediate execution.

But what if you want to produce some computer magic for your friends without typing while they watch? Or you want to write some instructions, take a break, and return later to add some more? To do these things, you need to be able to store the statements for execution at a later time. This is called *deferred execution*.

Deferred Execution

To make sure that the computer's memory is cleared, type

```
NEW
```

The NEW statement clears the main memory in the Apple IIe and should be used each time you begin a new program.

Now type the following line:

```
100 PRINT "MY FAVORITE FOOD IS ARTICHOKES"
```

When you press RETURN, nothing appears on the screen. If you press RETURN again, the cursor moves down the screen—but your statement still hasn't been executed.

```
]NEW
]100 PRINT "MY FAVORITE FOOD IS ARTICHOKES"
]
]
]
```

The key to this response is simple: when you typed `100` you gave the statement a *line number*, indicating that you want the computer to defer execution. The Apple IIe will hold line 100 in its *memory,* temporarily, until you instruct it not to. (If, for example, you turn off the Apple IIe, line 100 will be lost.)

Of course, you probably don't want to defer execution of that line forever. And you may be wondering how you can be sure about this temporary storage business. To find out what is being held in the Apple IIe's memory, type

The `LIST` statement displays the program lines that are in the Apple IIe's memory.

```
LIST
```

and, unless you mistyped something,

```
LIST
100 PRINT "MY FAVORITE FOOD IS ARTICHOKES"
```

appears on the screen.

The `LIST` statement instructs the computer to display whatever numbered lines are in its memory. To instruct the computer to execute line 100, type

The `RUN` statement causes the computer to execute, or carry out, whatever instructions are contained in the program lines in memory.

```
RUN
```

and the sentence

```
MY FAVORITE FOOD IS ARTICHOKES
```

should appear on your screen.

Look carefully at the difference between the line when it is listed and when it is executed. You'll notice that the line number, 100, has disappeared along with `PRINT`. As you recall, the `PRINT` statement is used to display information on the screen. So when you run line 100 only the characters enclosed in quotation marks are displayed. You can list and run line 100 as many times as you like.

Figure 2-1. Using LIST and RUN in Deferred Execution

It takes a while to understand the difference between what you see on the screen, what is in main memory, and what is stored on a disk. As you work with the Apple IIe and this manual, the difference will become clearer. More information on this subject can also be found in the *Apple IIe Owner's Manual*.

To see what happens when you give a different instruction with line 100, type

```
100 PRINT "THE SUM OF 3 + 4 IS"
110 PRINT 3 + 4
```

The old line 100 has been replaced and is no longer in memory. Type LIST to check. Now instruct the Apple IIe to

```
RUN
```

What do you get? If you now type

```
NEW
```

and then

```
LIST
```

the computer's memory is erased; both lines 100 and 110 are lost. Type

```
RUN
```

and nothing is executed—since there is nothing in memory. You'll have to give the Apple IIe some more instructions, so type

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

Now list these instructions. Notice that the computer stores statements in order of increasing line number: it has rearranged what you typed to read

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

When you run what is in memory, you see that the computer also executes the statements in order of increasing line number. But that's not good enough. There is a P missing. To add it, so the computer will print

```
A
P
P
L
E
```

you have to retype the statements with line numbers 3 and 4 as statements 4 and 5 and add a new line number 3. To make the corrections, type this:

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

To see what has happened, use the LIST statement, and then type RUN.

Congratulations! You have just written a *program*. Putting together a series of statements preceded by line numbers is, in its most simple form, what programming is about. A program is a stored sequence (that's what the line numbers are for) of instructions (like PRINT) that directs a computer to perform some function (in this case, to display APPLE on the screen).

When one of the P's was left out of APLE, it was a bother to retype those statements. There is, however, an easier way. It is good programming practice to leave some room between line numbers and before the first line. If the line numbers had been 10, 20, 30, and 40, you could have added the missing P with line 15.

One of the advantages of deferred execution is that you can add to or modify your instructions without having to type everything over and over again. To see this in action, type

```
NEW
```

to eliminate the old instructions. Now put in these:

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

When you run this program it doesn't quite print CAT vertically. But you can go back and type

```
105 PRINT "A"
```

List and run this program.

Here is a longer list of instructions to try:

```
NEW
10 HOME
20 PRINT "MY APPLE GIVES MESSAGES:"
30 PRINT
40 PRINT "HI, THERE, PROGRAMMER!"
50 PRINT
60 PRINT "MY APPLE DOES COMPUTATIONS:"
70 PRINT
80 PRINT 60 / 12
90 PRINT 4 ^ 5
```

Another advantage of deferred execution is that it is possible to store many statements at once. Of course, each statement must have a different line number.

Since you probably want to see the results of these instructions right away, type

```
RUN
```

and watch the results appear.

```
MY APPLE GIVES MESSAGES:
HI, THERE, PROGRAMMER!
MY APPLE DOES COMPUTATIONS:
5
1024
```

Does your screen look like this? If not, list your program to see what went wrong. (Notice that you can list a program after you have run it as well as before.) Here are some things to look for when you list a program:

- Does each line have a different number?
- Did you spell PRINT correctly?
- Did you remember the quotation marks after each PRINT statement? (Lines 30, 50, and 70 introduce blank lines between the statements, just as using a typewriter's carriage return would.)

If you need to correct any lines simply type them over. Then list again to make sure they look right. (In the next chapter you'll learn some faster ways to correct and edit your program lines.)

In this section you have learned several statements that help you work with whole programs. They are outlined in Figure 2-2.

Figure 2-2. The Capabilities of NEW, LIST, and RUN

NEW	Erases the current program from the computer's memory. After using NEW, you have to enter something from the keyboard or load something from a disk.
LIST	Displays the program that is in memory.
RUN	Executes the program in memory, beginning with the statement with the smallest line number. (It is also possible to start execution with whatever line number you indicate.)

Pause

Loops: the GOTO Statement

Suppose you want to print the integers 1 through 200, one number to a line. An obvious way to do this is

```
100 PRINT 1
110 PRINT 2
120 PRINT 3
```

and so on. But this would require 200 statements and a lot of careful typing! Luckily, there is an easier way. You can print the positive integers using just five statements:

```
NEW
100 N = 1
110 PRINT N
120 N = N + 1
130 GOTO 110
```

Before you run this program, look at Figure 2-3 to see how the program works.

The GOTO statement causes a program to branch to the indicated line. It is used to create a loop in a program.

Figure 2-3. The GOTO Statement

```
100 N = 1
110 PRINT N
120 N = N + 1
130 GOTO 110
```

Line 100 names the variable N and sets its value at 1. See "Variables and More Calculator Abilities," Chapter 1.

Line 110 prints the variable N.

Line 120 increases the value of variable N by 1.

Line 130 does just what it says: it causes the program to go back to line 110. Line 110 prints N, line 120 increases N by 1, line 130 says to do line 110 over again, and so on. Each time, a new value of N is printed, over and over.

Run the program. You will learn how to stop this program shortly. Meanwhile, admire the power of the program. If you typed RUN when instructed, two sentences back, the Apple IIe has executed the statement PRINT N a few hundred times already.

To stop the program press and hold the CONTROL key while pressing the C key.

The CONTROL-C command stops program execution and tells you where the execution was stopped by displaying the line number, such as:

```
BREAK IN 110
```

Try it. By the way, this is an exception to the rule about pressing RETURN after every instruction. Pressing RETURN is usually not necessary when a program is stopped with CONTROL-C.

Use CONTROL-C to stop program execution.

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

The CONT statement resumes, or continues, program execution after CONTROL-C, STOP, or END is used to halt execution.

```
CONT
```

which stands for "continue." Do this now.

As you look at the screen notice that the numbers keep rippling up and out of sight. As each new number is printed at the bottom of the screen all the others are moved up one line. This is called *scrolling.* You've been seeing it all along, but at a much slower rate.

Stop program execution again with CONTROL-C. Practice using CONT and CONTROL-C until you get used to them. They will come in handy.

If you want to start the program again, type RUN (instead of CONT). When you are ready for another example of the GOTO statement in action, type

```
NEW
100 PRINT "RAIN"
110 PRINT "IS"
120 PRINT "FALLING"
130 PRINT "DOWN"
140 PRINT "DOWN"
150 PRINT "DOWN"
160 GOTO 100
RUN
```

CONTROL-C stops this program, and CONT continues it.

Earlier you learned that the RUN statement starts program execution at the smallest line number. However, if you want to start at some other line, such as line 130, type

```
RUN 130
```

and you'll see

```
DOWN
DOWN
DOWN
```

on your screen before the program goes back to line 100 and begins at the beginning.

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

```
LIST 130
```

the Apple IIe will list only line 130 (if there is one). If you type

```
LIST 130,150
```

or

```
LIST 130-150
```

the computer will list all the program lines in its memory between and including line 130 and line 150. Try it. You cannot, however, specify a range of lines in the RUN statement.

A summary of Applesoft statements and a brief description of each statement can be found in Appendix A. When you need a quick reference (for example, if you want to know the difference between the LIST and RUN statements) Appendix A will come in handy.

Pause

Interacting with Your Program: INPUT

The programs you have learned so far are sure to impress your less experienced friends. But what if you want to write a program to ask someone a question—to fill in the blank, so to speak? There is a statement you can use to do just that: the INPUT statement.

The INPUT statement allows the programmer to interact with a program user from within a program.

Say you want a program that asks a person's age and uses whatever number the person enters to display a message. You already know how to use PRINT to display messages. So the first step is to include in the program the message you want used. You also know how to assign a variable name to a number. Since age is a variable, you'll use what you already have learned for this. What you haven't yet learned is how to use the INPUT statement.

Type this program, then list it to make sure you haven't made any typing errors. Before you run it, look carefully at Figure 2-4. The order, or syntax, of each statement is especially important.

```
NEW
10 HOME
20 INPUT "HOW MANY YEARS OLD ARE YOU?" ; AGE
30 PRINT "YOU ARE "; AGE ; " YEARS OLD!"
```

Now run the program. When you see the screen clear and the question

```
HOW MANY YEARS OLD ARE YOU?
```

on the screen, type in an answer (the number of your age). What happens? Run it again, and pretend you have a different age.

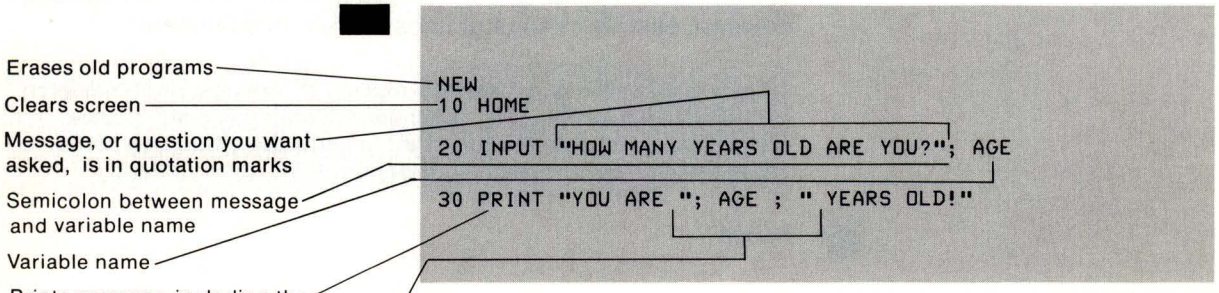


Figure 2-4. Using the INPUT Statement

When you execute the INPUT statement, in its simplest form, it prints a question mark and waits for you to type something in. What you type is stored in a variable.

When you use INPUT, you have to decide, or define, several things:

- What question, or message, do you want the person using your program to see? Since INPUT, like PRINT, displays exactly what you write it is important to ask a specific question. The message must be in quotation marks, and it is separated from the variable name by a semicolon.
- What do you want the user to type in? If you want a number, you must define a variable name.

It is also possible to have the user type in a word or group of characters. You will see an example of this in a program later in this chapter and will learn more about INPUT and variables in Chapters 4 and 5.

Here is an example of how to add a variable computation to your INPUT program. Leave lines 10 and 20 as they are. Add this new line:

```
25 AGE = AGE * 365
```

and change line 30 to read:

```
30 PRINT "YOU ARE ABOUT "; AGE; " DAYS OLD!"
```

Use the LIST statement to double-check your new lines. Then run the program. You'll notice that it doesn't give the exact number of days old a person is—unless it happens to be the birthday of the user. It estimates. That is why you added ABOUT to line 30. Each time you want to see this program in action, type RUN to begin execution.

So You Want To Save Your Programs?

So far, you have been using deferred execution by writing programs with line numbers, but each time you typed NEW, your previous program was erased from the computer's memory. This section explains how to save your programs by using the computer's Disk Operating System.

Preparations

Initializing, or **formatting**, is a process used to prepare a disk to receive information. This process is explained in the *Apple IIe Owner's Manual*.

To save a program you need two things: a program to save, and an *initialized* disk or a cassette tape on which to save the program. Of course, you also can save programs by writing them down, but that's not nearly as much fun as using the Apple IIe.

- You should have initialized a disk before you started this manual. If you did, find it, and proceed to the next section, "Saving."
- If you are not sure whether a disk is initialized, find out by typing

  CATALOG

  and press the RETURN key. Something similar to this should appear on your screen:

CATALOG is a DOS command that displays a list of all the files on a disk in the specified disk drive.

```
DISK VOLUME 254
 A 002 HELLO
```

If this message appears on your screen, your disk is initialized and you can proceed to "Saving." If no such message appears, the *Apple IIe Owner's Manual* explains how to initialize a disk.

- If you have two disk drives connected to Apple IIe, you need to know which drive to use. DOS automatically uses Drive 1 when you boot the system. DOS continues to use Drive 1 until you tell it differently. When you follow a DOS command with `,D2`—as in `CATALOG,D2`—DOS then uses Drive 2 (that is what `D2` stands for) until further notice. To use Drive 1 by default again follow the DOS command with `,D1`.

  Normally, with two disk drives, you have a boot disk (like the DOS 3.3 SYSTEM MASTER disk) in Drive 1 and another disk (like the initialized disk you are about to use) in Drive 2.

  If you want more information on using two disk drives, see the *Apple IIe Owner's Manual* and the *DOS Manual.*

- If you are using a cassette recorder instead of a disk drive, see the *Apple IIe Owner's Manual* for how to store information on a cassette tape and the *Applesoft Reference Manual* for using cassette commands.

Saving

If you have been following along, the age-asking program is still in memory. To make a permanent copy of this program on a disk, give the `SAVE` command followed by the name you want to give the program. To see this in action, make sure your initialized disk is in the disk drive and type

`SAVE` , when followed by a file name, is a DOS command that stores the program currently in memory onto a disk. `SAVE` used without a file name stores the program currently in memory on cassette tape.

```
SAVE AGE
```

Your disk drive will whir briefly, and you'll see the red light go on. When the red light goes off, type

```
CATALOG
```

and you'll see a list of all the programs on your disk. There should be two, unless you have already saved others.

Helpful Hints: If you turned your system off and restarted it recently, the age-asking program may be gone from memory. To save it, type it again, and then give the SAVE AGE command.

If you are using two disk drives and want to save the program on the disk in Drive 2, type ,D2 after the program name.

Once you have saved a program on a disk, you can retrieve it and load it into the computer's memory by typing

```
LOAD AGE
```

The LOAD command, when followed by a program name, looks for the named program on the disk in the specified or default drive. When used without a program name, LOAD reads a program from cassette tape into the computer's memory.

When you give the LOAD command, the disk drive will whir as it searches for the program. As soon as the red light goes off, list or run the program you have just loaded.

If nothing happens, type the program in again, and then save it. If still nothing happens, ask yourself:

- Did I use an initialized disk?
- Is a HELLO program displayed when I give the CATALOG command?
- Am I trying to save my program on the DOS 3.3 SYSTEM MASTER disk? (If so, you'll have to change disks because the SYSTEM MASTER disk is *write-protected,* which means you can't add anything to it.)
- If I have two disk drives connected to the Apple IIe, did I remember to type ,D2 after the SAVE command (that is, assuming the disk you want to save on is in Drive 2)?

Pause

Conditions: Determining the "Truth"

Lots of things in life are conditional: one thing has to happen before another thing can happen. For example, you have to reach a certain age before you can register to vote or apply for Social Security. The government has ways of determining whether the condition has been met, what happens when the condition has been met, and what happens when it hasn't; and so does Applesoft.

When you make a conditional statement in Applesoft, there are two possible responses: the condition is true or it is false. The false condition is represented by zero (0); the true condition is represented by one (1). If, for example, you type

```
PRINT 17 >= 18
```

the Apple IIe will respond with 0 on the next line.

```
PRINT 17 >= 18
0
```

Applesoft understood the statement to mean "is 17 greater than or equal to 18?" and, since it is not, responded with 0 to indicate the falseness of the condition. If, on the other hand, you type

```
PRINT 75 >= 18
```

the computer answers with 1 because the condition is true: 75 is indeed greater than or equal to 18. For more examples of Applesoft's response to the condition of greater than or equal to, try each of the following statements:

```
PRINT 18 >= 18
PRINT 3 >= 18
PRINT 27 >= 18
```

Conditional statements in combination with statements covered later in this chapter give Applesoft the ability to make choices. If a condition is true, Applesoft will follow one set of instructions; if a condition is false, Applesoft will follow another set of instructions.

Applesoft's system of evaluating conditions is based on the *binary* numbering system, which consists of ones and zeros. Binary is a word you will hear often in computer circles since computers store and manipulate information in binary form. In other words, information you type into the computer is translated into ones and zeros; this is called *machine language*.

Symbols Used in Conditional Statements

Six symbols are used in Applesoft to determine the relationship between values. Figure 2-5 lists them and gives some examples of each symbol.

Figure 2-5. Applesoft Symbols Used in Conditional Statements

Symbol	Meaning	Examples	
		True (1)	False (0)
=	EQUAL TO	3 = 3	3 = 1
>	GREATER THAN	78 > 55	78 > 124
<	LESS THAN	10 < 20	10 < 9
>=	GREATER THAN OR EQUAL TO	4 >= 4	4 >= 25
<=	LESS THAN OR EQUAL TO	32 <= 33	32 <= 30
<>	NOT EQUAL TO	32 <> 33	32 <> 32

To type the symbols for greater than or equal to and less than or equal to on the Apple IIe keyboard, first type a < or a > and then an =. To type the symbol not equal to, press < and then >.

Think about and then test these conditional statements. Which are true? Which are false? (Remember, you can substitute a question mark for PRINT.)

```
PRINT 5 <> 5
PRINT 8 <= 8
PRINT -8 < -7
PRINT -2 >= -5
PRINT 9 <> -9
PRINT (45 * 6) <> (-45 + 6)
```

Rules for Using Conditional Statements

- Conditional statements can include variables (like `AGE`), numbers (like `5`), and arithmetic expressions (like `45 * 6`).
- When the computer determines the value of a condition, that value will always be one or zero.
- In conditional statements, all numbers that are not zero are regarded as true.
- Applesoft evaluates conditions in statements after it does arithmetic operations like multiplication and subtraction.

Applesoft follows a particular order when evaluating any statement containing a condition or an arithmetic operation.

- First, parenthetical operations are evaluated.
- Second, unary minus signs used to indicate negative numbers are evaluated.
- Third, exponentiations are evaluated.
- Fourth, multiplications and divisions are evaluated from left to right in an expression.
- Fifth, addition and subtraction are evaluated. They are executed from left to right when they are on the same line.
- Sixth, symbols used in conditional statements are evaluated. All six have the same priority; they are evaluated from left to right within the same statement.

Pause

Conditional Loops: the `IF . . . THEN` *Statement*

When you first tried loops, using the `GOTO` statement, your programs ran on and on, indefinitely, until you pressed `CONTROL`-`C`. Another way of limiting such programs is to use conditionals in combination with the `IF . . . THEN` statement.

Say you want a program that counts to 20, then stops. What you want is a conditional statement that will limit the variable, `N`, to `20` or less. This would be written `N < = 20`.

The IF . . . THEN statement creates a conditional program loop.

The idea of the program is if N is less than or equal to 20, then counting continues; if the assertion is false (if N is greater than 20), then counting stops. Not surprisingly, the Applesoft statement for this is IF . . . THEN. Here is a program that uses a conditional in the IF . . . THEN statement:

```
NEW
200 N = 1
210 PRINT N
220 N = N + 1
230 IF N <= 20 THEN GOTO 210
```

So long as N is less than or equal to 20 the program will loop back to line 210. When the condition is no longer met, the program won't return to line 210, so execution ends. Try the program.

Loops, whether executed by airplanes or computer programs, have a top and a bottom. In this program line 210 is the top of the loop and line 230 is the bottom. The number 20 is the limit of the loop.

In general, the IF statement works like this: the condition following IF is evaluated; if the result is zero (false), all the rest of that program line is ignored, and the computer goes on to the next line; if the result is not zero (true), the statement following THEN is executed.

To transfer the program AGE from the disk back into memory, type

```
LOAD AGE
```

and then LIST. What you should see on your screen is

```
10 HOME
20 INPUT "HOW MANY YEARS OLD ARE YOU?" ; AGE
25 AGE = AGE * 365
30 PRINT "YOU ARE ABOUT "; AGE; " DAYS OLD!"
```

To see how conditionals work with IF...THEN statements, change lines 25 and 30 to

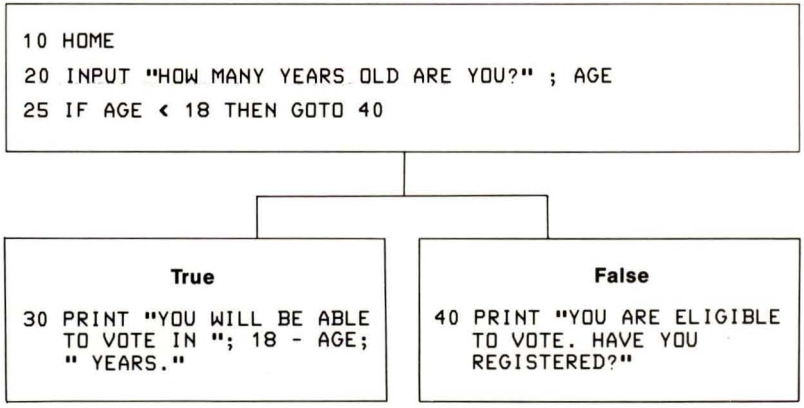
```
25 IF AGE < 18 THEN GOTO 40
30 PRINT "YOU WILL BE ABLE TO VOTE IN ";
   18 - AGE; " YEARS."
```

and add line 40:

```
40 PRINT "YOU ARE ELIGIBLE TO VOTE. HAVE YOU
   REGISTERED?"
```

Figure 2-6 shows the two paths this program can take. Run the program after you have studied the diagram.

Figure 2-6. A Conditional Loop



Unless you want to erase the old program AGE, you must save this new version with a different name. Since it is helpful to use program names that remind you what the program does, you could use VOTE-AGE for the name of the new version.

Using the APPLESOFT SAMPLER: COLORLOOP

The COLORLOOP program uses graphics to demonstrate a conditional loop. It is already stored on the APPLESOFT SAMPLER disk that came with the Apple IIe, so you can take a break from typing!

Loading this program from a disk into the computer's memory is not much different from loading a program you have saved yourself. Just make sure you have the right disk in the right drive.

Remove the disk you have been saving programs on from the disk drive. Insert the APPLESOFT SAMPLER disk and close the drive door. Type

```
RUN COLORLOOP
```

and you will hear the drive whir as it looks for the program. When COLORLOOP is loaded into the computer's memory, it will run automatically. If COLORLOOP doesn't execute:

- Type CATALOG to make sure the APPLESOFT SAMPLER disk is in the disk drive and the program COLORLOOP is listed on the disk.
- Give the RUN instruction again, making sure everything is spelled correctly and that you have specified the appropriate drive. (Remember that if the APPLESOFT SAMPLER disk is in Drive 2, you must type ,D2 after the program name.)

After execution stops, look at the program by typing TEXT, HOME, and LIST. You should see this on your screen:

```
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
```

If You Do Not Have a Disk Drive: All of the programs on the APPLESOFT SAMPLER disk are listed in this manual. Whenever you are instructed to load something from that disk, find the listing, and type the program in.

ROW is the name of the variable in this program. In line 420, the COLOR= statement makes the color equal to the value of the variable ROW: each time the value of ROW changes, the value of COLOR= changes with it.

Observe this program as it is executed. It uses graphics to demonstrate the same kind of conditional loop you added to AGE and used in the counting program. So long as the variable ROW is less than 16, the program returns to line 420 and executes another loop. When ROW is not less than 16, the condition is false and the program ends. You are beginning to see how to combine Applesoft statements in some interesting ways!

More on the IF *Statement*

The IF statement is a powerful one, and you will use it in almost every program you write. The following program begins without an IF statement, then shows what happens when you add one. It is not on the APPLESOFT SAMPLER disk because you will be changing it several times. Making the changes will give you a sense of how the statements you've been learning about can be used in combination with one another. Remember that you need to type

```
TEXT
```

if you want to see all the program lines as you type them in.

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

List the program to check your typing; then run it. Quick, isn't it? Now change line 210 to another color with

```
210 COLOR = 15
```

and run the program again. Try to list the program. Notice that you only see lines 240 and 250. The rest of the listing slips through the narrow text window at the bottom of the screen. This will continue to happen unless you type

```
TEXT
```

to get out of the graphics mode and then

```
HOME
```

to get rid of the garbage on the screen before you use the LIST statement.

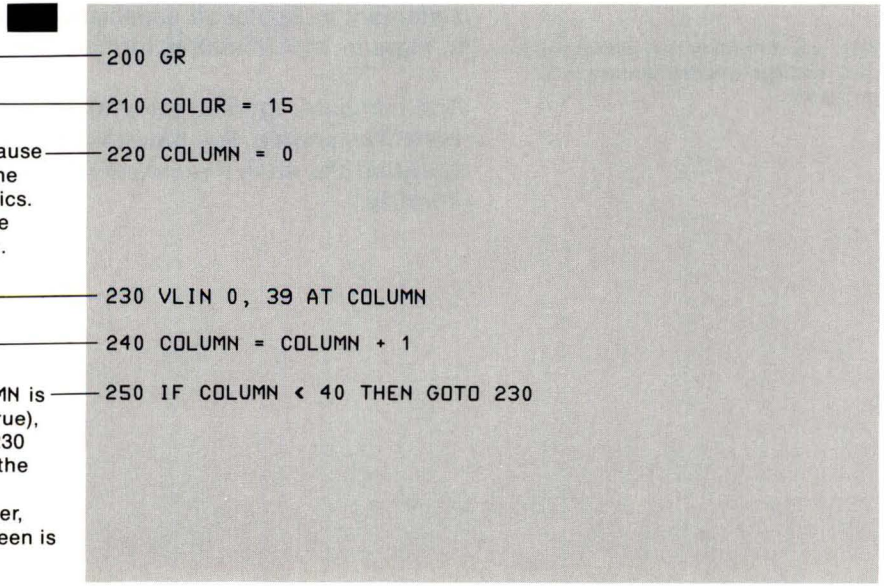
Now replace lines 220-250 with the following, and then list the program (while you are still in text mode).

```
220 COLUMN = 0
230 VLIN 0, 39 AT COLUMN
240 COLUMN = COLUMN + 1
250 IF COLUMN < 40 THEN GOTO 230
```

This program makes the screen a solid color. Run it now.

Because you haven't typed NEW, the old line 200 is still around. So long as it is in the computer's memory (and it will be until you type NEW, turn the computer off, or change the lines), you can use it in your program.

Figure 2-7. The VLIN Loop Program



To eliminate typing RUN each time you want to fill the screen with color, add the line

```
260 GOTO 210
```

Observe what happens when you run the program now. Try typing something else. What you type doesn't show up because the program is running continuously: it keeps going back to line 210 over and over. To stop the program, type CONTROL-C. To get it going again, type CONT.

Now stop it again with CONTROL-C. Type TEXT, HOME, and then LIST. Line 260 creates an infinite loop with the GOTO statement—since there is nothing in the program to stop it. If you want to save this program without the infinite loop, remove line 260 before saving.

A Note About Line Numbers: Line numbers can begin with 10, 100, 1000, or whatever number you choose. What is important is to leave space between the numbers so it is easy to add lines when necessary. While you are doing the programs in this manual, however, it is important to use the same line numbers as in the examples. If you don't, you will get confused when it comes to additions and changes like the ones you made in the last program.

Remarks

REM is a statement that allows you to put remarks, or commentary, in a program.

As you can see from the description of the last program, what happens on any given line can be complicated. There is a statement in Applesoft especially for explaining what is going to happen: REM. It stands for "remark."

The computer ignores REM statements; they are strictly for the benefit of people. See how easy it is to follow the VLIN loop program shown in Figure 2-8 when REM statements are used liberally:

Figure 2-8. Using REM Statements in a Program

Program with Remarks

```
195 REM SET GRAPHICS MODE
200 GR
205 REM SET COLOR
210 COLOR = 15
215 REM START AT COLUMN 0
220 COLUMN = 0
225 REM DRAW VERTICAL LINE
    AT COLUMN
230 VLIN 0,39 AT COLUMN
235 REM PROCEED TO NEXT
    COLUMN BY ADDING 1
240 COLUMN = COLUMN + 1
245 REM LOOP MAKES IT
    EXECUTE OVER AND OVER
    UNTIL COLUMN 39 IS
    REACHED
250 IF COLUMN < 40 THEN GOTO
    230
```

Program without Remarks

```
200 GR
210 COLOR = 15
220 COLUMN = 0
230 VLIN 0,39 AT COLUMN
240 COLUMN = COLUMN + 1
250 IF COLUMN < 40 THEN
    GOTO 230
```

If you look carefully you will see that the program still gives the same instructions as it did before. The added REM statements simply explain what the line (or lines) following them will do.

REM statements are optional. Applesoft does not execute them because they are informational. You can use them as you wish. The longer your programs, the more helpful REM statements are. They help anyone who looks at the program to understand what it does.

FOR/NEXT *Loops*

The FOR/NEXT statement sets up a program loop that is carried out the number of times specified by the TO portion of the statement.

The FOR/NEXT statement creates a program loop that works within a range of numbers. You define the range with a variable. The FOR/NEXT statement is a more efficient way of creating a loop if you are using a variable that regularly increments. Type TEXT to return to text mode, HOME to clear the screen, and NEW to clear memory. Then try this program:

```
NEW
100 FOR NUMBER = 0 TO 12
110 PRINT NUMBER
120 NEXT NUMBER
130 PRINT "PROGRAM IS FINISHED WHEN LAST NUMBER
    IN THE FOR STATEMENT IS DISPLAYED"
```

Run this program. You see that it does essentially the same thing as the number counting program you produced with IF...THEN. Figure 2-9 explains how the FOR/NEXT program works.

Figure 2-9. The FOR/NEXT Statement. The line-by-line explanations could be inserted in the program as REM statements.

Sets range of the variable NUMBER to 0-12.

Value of variable, NUMBER, is displayed.

This is the bottom of the loop. The variable is increased by 1 and then compared to the upper limit specified in line 100: 12. When the limit of the range is reached, the program stops, as line 130 indicates.

```
100 FOR NUMBER = 0 TO 12

110 PRINT NUMBER

120 NEXT NUMBER


130 PRINT "PROGRAM IS FINISHED WHEN LAST NUMBER IN THE
    FOR STATEMENT IS DISPLAYED"
```

In a FOR/NEXT statement the variable acts as a counter: it specifies the number of times the loop will run. In any FOR/NEXT loop, 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 (out of the loop) to the statement after the NEXT. In this program line 130 is not displayed until the limit 12 is reached.

The most important advantage of the FOR/NEXT method of constructing loops is that you don't have to think so hard when writing the loop. If you want to draw a series of horizontal lines on the screen using each of the 16 colors, substitute this program for COLORLOOP:

```
NEW
3000 GR
3005 REM SETS VARIABLE RANGE TO 0-15
3010 FOR ROW = 0 TO 15
3015 REM MAKES VALUE OF COLOR THE SAME AS VALUE
     OF VARIABLE
3020 COLOR = ROW
3025 REM DRAWS HLIN AT EACH ROW WITHIN RANGE
3030 HLIN 0, 39 AT ROW
3035 REM WHEN LIMIT OF RANGE IS REACHED, PROGRAM ENDS
3040 NEXT ROW
```

To leave the graphics mode, type TEXT; type HOME to clear the screen. Now look at the two programs that follow. They show two ways to print the even numbers from 0 to 12. The first uses IF...THEN:

```
NEW
100 X = 0
110 PRINT X
120 X = X + 2
130 IF X <= 12 THEN GOTO 110
```

In line 120, 2 is added to the variable X. This loop increases by 2.

Use STEP only within a FOR/NEXT statement. Its use is optional.

The second example shows how to display the even numbers from 0 to 12 using the STEP command within a FOR/NEXT loop.

```
200 FOR X = 0 TO 12 STEP 2
210 PRINT X
220 NEXT X
```

You may be wondering why you see two sets of even numbers from 0 to 12 on your screen. Type LIST to see if you can figure out why.

Hint: The two sets of numbers appear on the screen because lines 100 to 130 as well as lines 200 to 220 were executed when you typed RUN. To see only the second program, type RUN 200. To remove a previous program from memory, you can manually delete it or you can begin each program with NEW.

A STEP may involve any number in Applesoft's range, from approximately -32768 to +32767. A program can STEP by 5, 50, or 500. A program can also STEP backward, as in

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

Type this line; execute it by typing

```
RUN 200
```

Notice that the statement RUN 200 also executes lines 210 and 220 because all line numbers larger than the one indicated in a RUN statement are executed.

Now try some of your own FOR/NEXT statements. Several of the example programs from this point on will use the FOR/NEXT statement.

Nesting and Crossing Loops

Along with the convenience of the FOR statement come some limitations. For example, FOR/NEXT loops can be *nested*, meaning one loop may be contained inside another, but FOR/NEXT loops must not cross. Figures 2-10 and 2-11 demonstrate nested and crossed loops.

Figure 2-10. Nested Loops: the HUE Program

```
NEW
300 GR
310 FOR HUE = 1 TO 15
320 COLOR = HUE
330 FOR ROW = 0 TO 39
340 HLIN 0, 39 AT ROW
350 NEXT ROW
360 COLOR = HUE - 1
370 FOR COLUMN = 0 TO 39
380 VLIN 0, 39 AT COLUMN
390 NEXT COLUMN
400 NEXT HUE
```

The program in Figure 2-10 is an example of two-level nesting. It is called HUE on the APPLESOFT SAMPLER disk. Run it now. Figure out how it works before going on.

Warning

When writing programs using FOR statements, remember that *each* FOR *must have a matching* NEXT.

The program in Figure 2-11 illustrates what are called crossed loops.

Figure 2-11. Crossed Loops

```
NEW
500 FOR N = 10 TO 20
510 PRINT N
520 FOR J = 30 TO 40
530 PRINT J
540 NEXT N
550 NEXT J
```

This program doesn't work correctly. Type TEXT, HOME, and RUN. It is an example of what happens when each FOR doesn't have a matching NEXT. The program's loops are crossed, so it jumbles the numbers as it displays them. It also gives an error message. See if you can fix the program by uncrossing the loops.

The quilt drawing program in Figure 2-12 has three-level nesting. QUILT can be loaded from the APPLESOFT SAMPLER. This program avoids using COLOR as a variable name because, as you may recall, COLOR is a reserved word in Applesoft. Applesoft.

Figure 2-12. Three-Level Nesting: the QUILT Program

```
NEW
400 GR
410 HUE = 0
420 FOR COLUMN = 0 TO 35 STEP 5
430 FOR LINE = 0 TO 30 STEP 10
440 HUE = HUE + 1
450 IF HUE > 15 THEN HUE = 0
460 COLOR = HUE
470 FOR ROW = LINE TO LINE + 9
480 HLIN COLUMN,COLUMN + 4 AT ROW
490 NEXT ROW
500 NEXT LINE
510 NEXT COLUMN
```

Now try an experiment. Type TEXT. Remove line 400 by typing 400 and pressing RETURN. Run the program.

When you remove the GR statement, the program executes in text mode. When you put line 400 back, the program will execute in graphics mode again. You can do this with any program that uses graphics.

Pause

Controlling Spaces in Your Programs

The comma and the semicolon are used in Applesoft to create different display effects in program execution. There are also three Applesoft statements that control spacing in programs. The examples in this section let you experiment so you become familiar with the differences.

As a first experiment, type this program and see what it does when you run it.

```
NEW
100 PRINT "MELLOW"
110 GOTO 100
```

Stop the program with CONTROL-C. Then add a comma to line 100.

```
100 PRINT "MELLOW",
```

and run the program again. The program now prints the word in columns. Use CONTROL-C again to stop the program, and then substitute a semicolon (;) for the comma (,).

```
100 PRINT "MELLOW";
```

Run the program again. This time the display runs together, leaving no spaces between the words. It prints MELLOW after MELLOW until you stop the program with CONTROL-C.

> Using a semicolon results in output with no spaces between the words or numbers. Using a comma results in output in columns.

Change the program by adding this statement

```
90 V = 99
```

and change line 100 to read

```
100 PRINT V
```

Run this program. Now add a comma to line 100.

```
100 PRINT V,
```

Run it again. Then change the comma to a semicolon

```
100 PRINT V;
```

and observe that the comma and semicolon also can be used with numeric 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. Remove the old lines with NEW, and type

```
100 BALLS = 3
110 STRIKES = 2
120 PRINT BALLS,STRIKES
RUN
```

Your screen should look like this:

```
3               2
```

You can make clearer output by including a message in the PRINT statement. For example, change line 120 to

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

and the program will appear on your screen as

```
THE BALLS AND STRIKES ARE 3             2
```

unless you didn't include a space after ARE. Another way of writing the statement is

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

Perhaps the prettiest way to print this line (are you trying all of these on the Apple IIe?) is to add some spaces within the PRINT statement

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

This gives you a scoreboard-like display:

```
BALLS  3    STRIKES  2
```

Now let's say you wanted to display the word HERE in the 10th column (the screen is 40 columns across). You could use the statement

```
120 PRINT "         HERE"
```

making sure that you carefully added exactly nine blanks before the word HERE. Or you could use the TAB feature, which lets you set a tab in your programs that acts like a tab on typewriters. Try the statement

The display function TAB, which must be used in a PRINT statement, moves the cursor through tab fields on the screen.

```
120 PRINT TAB(10);"HERE"
```

to see TAB in action.

TAB must be used in a PRINT statement. If you try a statement like

```
105 TAB (5)
```

you'll get an error message. TAB must be followed by an *argument:* a number or variable contained in parentheses. When TAB is combined with a variable, such as N, it spaces according to the value of the variable. Here is an example:

```
NEW
200 FOR N = 1 TO 24
210 PRINT TAB(N);"X"
220 NEXT N
```

There are two related statements you can use to position the cursor in various parts of the screen.

VTAB moves the cursor to the specified vertical row on the display screen.

VTAB moves the cursor vertically up or down the 24 horizontal display lines. The top line is line 1; the bottom line is line 24. VTAB, unlike TAB, is not used within a PRINT statement.

The HTAB statement moves the cursor either left or right to the specified column (1 through 40) on the screen.

You can tab horizontally with HTAB. It works like TAB, but can cause printing to begin either to the left or to the right of the current printing position and does not have to be used with the PRINT statement. The leftmost character on a line is in position 1, while the rightmost character is in position 40.

Here is a program that shows how HTAB moves the cursor on the screen:

```
10 PRINT "RUN"
20 HTAB 5
30 PRINT "FASTER, "
40 HTAB 34
50 PRINT "FASTER,"
60 HTAB 81
70 PRINT "OR ELSE!"
```

The program SPACES on the APPLESOFT SAMPLER demonstrates the use of HTAB and VTAB together. Load it now. Or you can type it in:

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

Before you run this program, try (it isn't easy!) to figure out what it will do.

While TAB, HTAB, and VTAB act somewhat like the coordinates in PLOT, there are some differences.

1. The 40 columns for the TAB instruction are numbered from 1 to 40, as they would be on a typewriter, while PLOT coordinates are numbered from 0 to 39 for graphics display.
2. VTAB's limits are 1 through 24. Since characters are taller than the bricks used in low-resolution graphics, there is room for only 24 lines of screen display.

3. The largest value that can be used with TAB and HTAB is 255.
4. A zero or a number that is too large or too small for TAB, VTAB, or HTAB will give you an ?ILLEGAL QUANTITY ERROR.
5. TAB and HTAB will tab past the end of the screen and *wrap around* to the next line.

To see wraparound in action, type

```
NEW
300 FOR K = 1 TO 255
310 PRINT TAB(K); K
320 NEXT K
```

Normally TAB is not used in this way. A more usual method would be to use HTAB. Replace lines 310 and 320 with

```
310 HTAB K
320 PRINT K
```

and add

```
330 NEXT K
```

What happens to the program when you replace HTAB with VTAB? How could you change the program to conform to the screen limits?

Be assured that the numbering systems you are working with will become more familiar with use. The screen is always divided into 40 horizontal rows and 40 vertical columns. The horizontal numbers vary depending upon the mode (0–39 in graphics, 1–40 in text); and the vertical numbers change because characters take more space than the bricks used in graphics.

Here is a program that combines most of the statements you have used in this chapter. One part isn't familiar: the variable N$. You will find out more about this kind of variable in Chapter 5. There is also a new statement used in line 70. Type

```
NEW
10 HOME
20 INPUT "WHAT IS YOUR FIRST NAME?"; N$
30 PRINT
40 FOR A = 1 TO 5
50 PRINT TAB (A); "HI, "; N$; ", HOW ARE YOU?"
60 NEXT A
70 END
```

The END statement causes a program to cease execution and returns control to the user.

List this program to check your typing, correct any mistakes you might have made, and run it. If you like it, and you want to store it on a disk for future use, all you need to do is type

```
SAVE WELCOME
```

and then CATALOG to make sure it has been stored on the correct disk.

Chapter Summary

Statements	DOS Commands	Terms
NEW	RUN	execute
LIST	CATALOG	immediate execution
GOTO	SAVE	deferred execution
CONT	LOAD	line number
INPUT		program
IF...THEN		scroll
REM		initialize
FOR/NEXT		default
STEP		write-protected
PRINT (punctuation)		binary
TAB		machine language
VTAB		nested
HTAB		argument
END		wrap around

Keys	Error Messages
CONTROL-C	BREAK IN 110

Chapter 3

Making Changes

Making Changes

So far, you have learned several simple ways of fixing mistakes in programs. However, you are getting the hang of things now, and beginning to work with longer programs, so it is time to give you a few more tools for changing or editing your work.

Just to refresh your memory: you have learned that if you catch a typing mistake before you press RETURN, you can use the RIGHT- and LEFT-ARROW keys to fix the mistake. The LEFT-ARROW key backs the cursor up to the error, and the RIGHT-ARROW key copies over the remainder of the line after you have made the correction.

You also know that when you notice a mistake after you have pressed RETURN, you can type the line over again.

The Moving Cursor: Escape Mode

Another method uses the ESC key in combination with the four arrow keys. Called *escape mode,* this method allows you to move the cursor around the screen without affecting anything except the cursor position.

Rules for Using Escape Mode

- To get into escape mode press the ESC key. To leave escape mode press the SPACE bar. Once you are out of escape mode all you have to do to get back in is press the ESC key.
- To correct or change something using escape mode, you must position the cursor over the first digit in the line number of the line you wish to fix.
- Escape mode allows you to correct only one program line at a time.

- After you make a change in a line it is important to copy over the remainder of the line with the RIGHT-ARROW key.
- The RIGHT- and LEFT-ARROW keys work differently in escape mode than they do in normal mode. In escape mode, they move the cursor around without affecting the characters on the screen. The arrow keys used alone (without pressing the ESC key first) erase (LEFT-ARROW) and copy over (RIGHT-ARROW) the characters they go over.

A Practice Session

There are several mistakes in the lines below. If you follow the step-by-step instructions, you will get an idea of how escape mode works.

What you do . . .	What happens . . .

Type exactly what you see:
 	
`NEW`
`10 PRINT "THE MOCKINGBIRD"`
`20 PRINT TAB (5); "SINGS"`
`30 PRINT RAB (3): "IN`
`SPRING"`
	All the mistakes you type appear on the screen.

Type `LIST 10`	`10 P RUN T"THE MOCKINGBIRD"`
Press and release the ESC key once.	This puts you into escape mode.
Press the UP-ARROW key two times.	The cursor is moved to line 10.

```
]LIST 10
10 P RUN T"THE MOCKINGBIRD"

]
```

Press the LEFT-ARROW key once. The cursor must be placed over the first digit.

Notice that the cursor moves, but nothing else changes on the screen while you are in escape mode. (You can't see it, but nothing changes in the computer's memory either.)

Press the SPACE bar.

This takes you out of escape mode.

Press the RIGHT-ARROW key five times.

The RIGHT-ARROW key copies over characters when you are not in escape mode (so you don't have to type them over).

Replace the U with an I.

Copy the rest of the characters in the line with the RIGHT-ARROW key. When you reach the end of the line, press RETURN.

Type `LIST 10` **to see the corrected line.**

```
]LIST 10
10  PRINT "THE MOCKINGBIRD"
]
```

If you were to run the program now, a `?SYNTAX ERROR IN 30` would appear on the screen because there are two mistakes in line 30 that you haven't fixed yet. `RAB` should be `TAB`, and the colon should be a semicolon. To correct line 30, first type `LIST`, then get into escape mode by pressing the ESC key, and use the arrow keys to move the cursor to the beginning of line 30.

Follow the same procedure you used for line 10: leave escape mode, use the RIGHT-ARROW key to move the cursor over the offending characters, replace them, and copy the remainder of the line. List again, to make sure you have corrected all the mistakes, then run.

Using escape mode may seem a bit complicated at first, but it is easy once you get the hang of it. Promise!

Other Escape Commands

Three other commands are useful when you are doing a lot of editing. They are all entered from escape mode.

- ESC @ (press the ESC key, let go, then hold the SHIFT key to produce the @ sign) clears the entire screen, just as the `HOME` statement does.
- ESC F clears the screen from the cursor position to the bottom of the screen.
- ESC E clears from the cursor to the end of the line.

The Limits of Escape Mode

Now that you've successfully used escape mode to correct some program lines, you can take a break (of sorts). Get into escape mode and play with the four arrow keys: move the cursor to the top of the screen, down to the bottom, up to the middle, over to

the right edge, and all around. What happens when the cursor gets to the right edge of the screen? (Hint: it wraps around. You saw this when you used TAB and HTAB.) What happens when you accidentally press some other key while you are in escape mode? Try it—finding out how it works now will save you some headaches later. When you get tired of moving the cursor around the screen, press the SPACE bar to leave escape mode.

Helpful Hints: If you don't position the cursor over the first digit in the line number of the line you want to fix, some of the line will be lost. If you don't believe this, try it!

The reason you corrected line 10, listed it, and then corrected line 30 is because it is difficult to correct more than one line at a time in escape mode.

Inserting Text into an Existing Line

Sometimes you may notice that an addition to a program line would make the program a lot nicer. Suppose, for example, that you want to insert a TAB(10) statement after PRINT in the line

```
90 PRINT "THIS IS A SHORT PROGRAM"
```

Follow the step-by-step instructions below to see how text insertion works.

What you do . . .	What happens . . .

Type
 	
`NEW`
`90 PRINT "THIS IS A SHORT PROGRAM"`	

Use RUN to see how the line looks.	

```
]RUN
THIS IS A SHORT PROGRAM

]
```

Type `LIST 90`

```
]LIST 90
90  PRINT "THIS IS A SHORT PROGRA
    M"

]
```

Press the ESC key.	This puts you into escape mode.
Press the UP-ARROW key three times and the LEFT-ARROW key once.	This positions the cursor over the first digit in the line number. The cursor must be over the first digit for corrections or insertions to work properly.
Press the SPACE bar.	This leaves escape mode.
Press the RIGHT-ARROW key ten times.	This positions the cursor at the place in the line where you want to insert something.

```
]LIST 90
90  PRINT #THIS IS A SHORT PROGRA
    M"

]
```

Press ESC.	This puts you back in escape mode.
Press the UP-ARROW key once.	This positions the cursor just above the line so that you can add text to a line.

```
]LIST 90
          ▒
90  PRINT "THIS IS A SHORT PROGRA
    M"

]
```

Press the SPACE bar once.	This takes you out of escape mode.

Type `TAB(10);`

This is the text you are adding to the line.

```
]LIST 90
        TAB (10);
90  PRINT "THIS IS A SHORT PROGRA
    M"

]
```

Press ESC.

This returns you to escape mode.

Press the DOWN-ARROW **key once.**

```
]LIST 90
        TAB (10);
90  PRINT "THIS IS A SHORT PROGRA
    M"

]
```

Press the LEFT-ARROW **key eight times.**

This returns the cursor to the point at which you began the insertion. It allows you, after you leave escape mode, to copy over the remainder of the line with the RIGHT-ARROW key.

```
]LIST 90
        TAB (10);
90  PRINT "THIS IS A SHORT PROGRA
    M"

]
```

Press the SPACE **bar once.**

This takes you out of escape mode.

Press the RIGHT-ARROW **key 23 times—until the cursor is on the space following** `A` **of** PROGRAM.

```
]LIST 90
        TAB (10);
90  PRINT "THIS IS A SHORT PROGRA
    M"

]
```

Press the ESC key.

This step, and the two that follow, moves the cursor to the right without copying the spaces between PROGRA and M. See the **Helpful Hints** that follow for more information.

Press the RIGHT-ARROW key until the cursor is on the M.

```
]LIST 90
          TAB (10);
90  PRINT "THIS IS A SHORT PROGRA
    M"

]
```

Press the SPACE bar.

This takes you out of escape mode.

Copy the remainder of the statement with the RIGHT-ARROW key and press RETURN. LIST the line.

```
]LIST 90
          TAB (10);
90  PRINT "THIS IS A SHORT PROGRA
    M"

]LIST 90

90  PRINT  TAB( 10);"THIS IS A SH
    ORT PROGRAM"

]
```

RUN the line to see the difference in its execution after adding TAB (10).

```
]RUN
          THIS IS A SHORT PROGRAM

]
```

Now try adding VERY before SHORT in the same line. When you are finished, the statement should LIST as

```
]LIST
90  PRINT  TAB( 10);"THIS IS A VE
    RY SHORT PROGRAM"
]
```

Helpful Hints: Applesoft uses a screen width of 40 columns—that is, up to 40 characters can be displayed on a single line. Any line longer than 40 characters is automatically wrapped around to the next line and indented. When you are changing such a program line, you must use escape mode if you don't want the automatic indentation copied into your revised line. This is what you were doing when you went back into escape mode between the A and M of PROGRAM. When the cursor is over the first character of the second line, leave escape mode and use the RIGHT-ARROW key to finish copying.

There is also a statement you can use to narrow the screen width and to instruct Applesoft to stop adding those extra spaces: POKE 33,33. You don't need to learn about it now, but you can find out about it in the *Applesoft Reference Manual* when you are ready.

Getting Rid of Program Lines

Naturally there are times when you decide you don't like a program line and want to get rid of it. There are several ways to remove program lines.

If you have just finished typing the line and change your mind about it before you press RETURN, there is a control character you can use: CONTROL-X. Try it out by typing the line below. Before you press RETURN, hold the CONTROL key while you press the X key. If you forget and press RETURN before using CONTROL-X, you'll have to type NEW and try again.

CONTROL-X deletes a line only before the RETURN key has been pressed.

```
10 PRINT "WHY AM I DOING THIS?"
```

```
10 PRINT "WHY AM I DOING THIS?"\
```

As you see in the illustration and on your screen (you are doing these exercises, aren't you?), a slash appears at the end of the line when you use CONTROL-X. When you try to list the line, nothing appears. You really have gotten rid of that line!

Now type

```
20 PRINT "AN ELEPHANT JUST WALKED IN"
```

and press RETURN. If you change your mind about this line now, there are several alternatives. One is to replace the line by typing it over with a different statement, as in

```
20 PRINT "THIS IS VERY SILLY"
```

However, if you want to eliminate the line entirely, you can type

```
20
```

and it will disappear. If you don't believe it, type LIST 20.

Another method is to type the statement

```
DEL 20,20
```

The DEL statement removes, or deletes, the specified range of lines from the program.

which instructs the computer to delete a line or lines. Since it is obviously easier to type `20` by itself than to type `DEL 20,20`, you may be wondering why the `DEL` statement is so great. It's not apparent until you want to erase five or ten lines of a program. Instead of having to type each line number separately, you can type

```
DEL 10,90
```

This will delete every statement with line numbers between and including 10 and 90.

The `DEL` statement in Applesoft looks like, but is not the same as, the `DELETE` command used by the Disk Operating System (DOS).

`DELETE` is always used to delete entire programs from the disk and is always followed by the name of the program you want to remove from the disk.

`DEL`, as you have just learned, is followed by two line numbers separated by a comma. If you get the two commands confused and use the wrong one, don't worry. The Apple IIe will give you an error message to remind you.

The DELETE key on the Apple IIe's keyboard is not the same as the `DEL` or `DELETE` commands. See the *Apple IIe Owner's Manual* and the *Apple IIe Reference Manual* for more information.

`DELETE` is a DOS command that removes the program specified by name from a disk.

Editing Long Programs

The full value of using escape mode to make changes does not become apparent until you work with long program lines and long programs. The more you write your own programs, the more you will want to use escape mode.

There are several advantages to becoming well versed in the use of escape mode. One advantage is that it is much faster to use when editing an extremely long program line of, for example, 230 characters. Another advantage is that it makes it easier to merge two programs that have a lot of similar lines. Yet another advantage is that you can list a line that appears in similar form more than once, make the relevant changes, and copy it over for easy reproduction.

However, these uses of escape mode are a bit down the line. A more handy tool for the present involves the use of CONTROL-S.

CONTROL-S stops and resumes program listing.

When a long program is listing, the lines scroll by too quickly for most of us to read. One way to get around this problem is to list portions of a program, by typing, for example,

```
LIST 10,100
```

and then

```
LIST 110,200
```

and then

```
LIST 210,300
```

and so on until you have checked all the program lines.

Another, easier, way to control scrolling is to use CONTROL-S to stop scrolling and again to resume it after you have finished checking the lines on the screen. It takes a while to get the knack of pressing CONTROL and S fast enough, but it's worth the trouble and aggravation.

A Little History

Eight other keys, besides the four arrow keys, can be used for editing. If you are familiar with older models of the Apple, you may know what those keys are. In any case, a little history is in order.

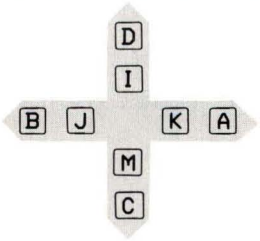
When the Apple II was first introduced, there were no UP- and DOWN-ARROW keys on the keyboard and there was no escape mode. Editing was done by alternating the ESC key with the A, B, C, and D keys: A moved the cursor to the right, B moved the cursor to the left, C moved the cursor down, and D moved the cursor up. If, for example, you wanted to move the cursor up five and over to the left four spaces, you had to type, in succession:

ESC D ESC D ESC D ESC D ESC D
ESC B ESC B ESC B ESC B

Whew! You can see how easy it would be to get confused.

Later, this method was changed: one press of ESC was enough to get you into escape mode. However, the keyboard still had no arrow keys. The four keys I, J, K, and M were used to move the cursor. If you look at the keyboard, you'll notice that those keys form a diamond. Their directions are consistent with their position: I moves the cursor up, J moves it to the left, M moves it down, and K moves it to the right.

Figure 3.1. Key Directions



Although arrow keys have been added, I, J, K, and M still work in escape mode. And you can still use A, B, C, and D to move the cursor if you precede each with ESC. If you happen to have learned one of the older methods, you can continue to use it and not worry about changing to the arrow keys. Or you may find that you like one of the older methods better. The Apple IIe was designed so that all three keys move the cursor in the same direction. Figure 3-1 summarizes the directions.

Summary of Editing Features

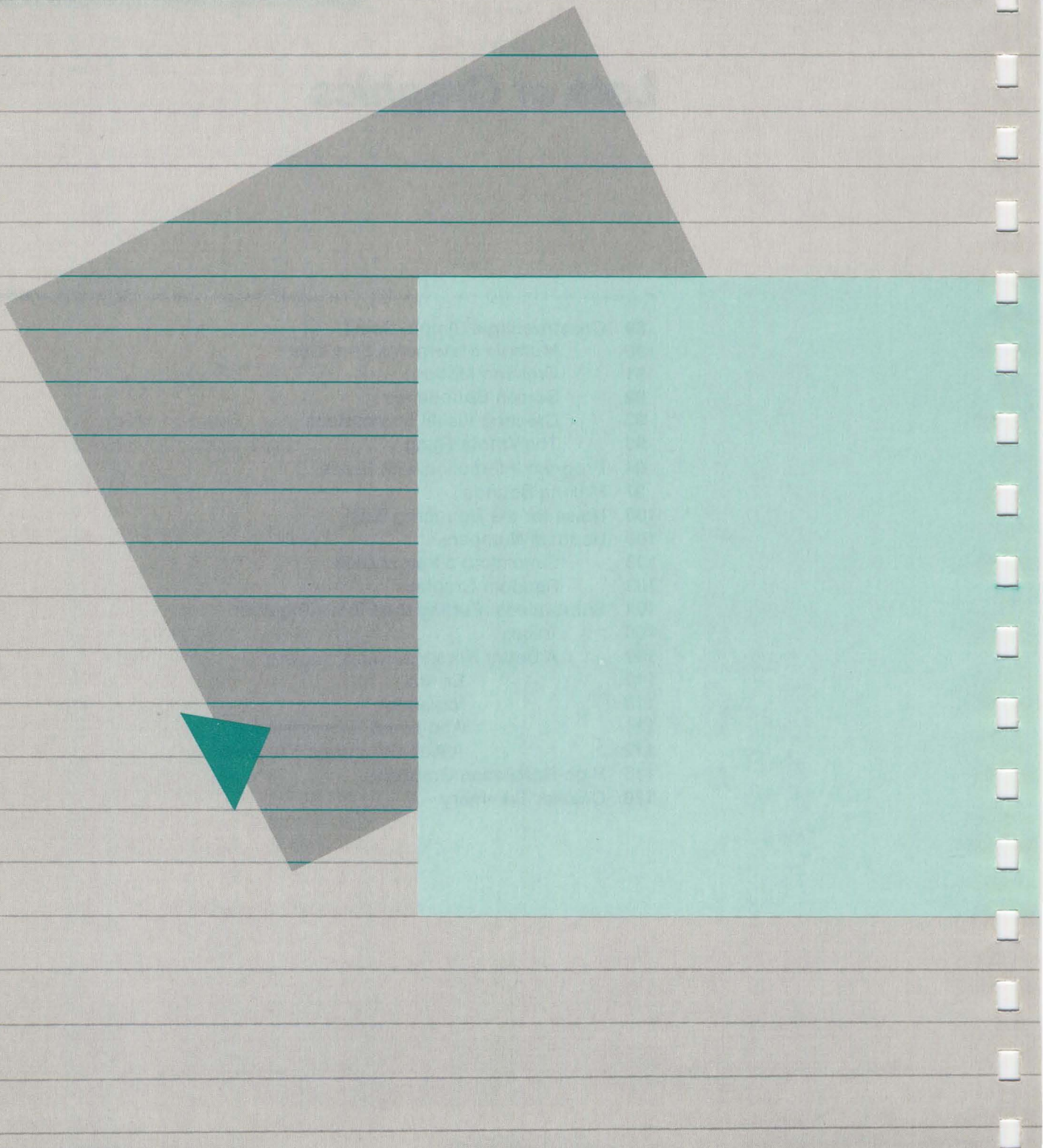
Escape Mode

To enter escape mode:	Press the ESC key
To leave escape mode:	Press the SPACE bar
To move the cursor up:	Use the UP-ARROW, I, i, D, or d
To move the cursor down:	Use the DOWN-ARROW, M, m, C, or c
To move the cursor left:	Use the LEFT-ARROW, J, j, B, or b
To move the cursor right:	Use the RIGHT-ARROW, K, k, A, or a
To clear from cursor to end of line:	Press ESC E
To clear from cursor to end of screen:	Press ESC F
To clear entire screen:	Press ESC @

Non-Escape Mode

To delete a character:	Press the LEFT-ARROW key
To copy over, or retype a character:	Press the RIGHT-ARROW key
To delete a line before RETURN has been pressed:	Press CONTROL-X
To stop a program listing:	Press CONTROL-S
To resume a program listing:	Press CONTROL-S
To delete a program line or lines:	Use the DEL statement
To delete a whole program from disk:	Use the DELETE command

Lots of Graphics



Lots of Graphics

Now that you have a few more editing tools at your fingertips, you are probably anxious for more programming challenges. This chapter takes a great leap forward, leading you through the construction and modification of longer and more complicated programs. You will see how programs are built piece by piece and how statements work as building blocks.

Don't expect to be able to do all this yourself right away! Learning how to program is a process; the more you work with the programs in this manual, the more you'll understand what to do on your own.

Constructing a Simple Game

This section introduces a number of program elements, or parts, that are typically used in video games. It demonstrates a way to use color in creating visual impressions on the screen.

See Chapter 2, "Using the APPLESOFT SAMPLER: COLORLOOP" for more information.

Before going further, load the program COLORBOUNCE from the APPLESOFT SAMPLER by typing

```
LOAD COLORBOUNCE
```

Then, to see COLORBOUNCE in action, execute it by typing RUN.

If you are not using a disk drive, turn to "The Whole Thing" in this chapter. Type the program in, check your listing, then execute it.

To stop the program, press CONTROL-C. Now read the next few sections to find out how the program works. Each section discusses a different element of the program.

Multiple Statements on a Line

Up to now most of the program lines you have worked with contained one statement in each line. With this method you might begin a program with (remember, you don't have to do anything but read this section):

```
400 REM SET COLOR GRAPHICS AREA
420 GR
```

However, the use of a colon (:) between statements allows you to combine multiple instructions on one numbered program line. Like this:

```
400 GR : REM SET COLOR GRAPHICS AREA
```

It sometimes makes program lines easier to understand if the comments are on the same line they describe, as in line 400. And sometimes it is easier if the remark is on a different line. COLORBOUNCE uses both methods.

The colon can be used to separate any two statements. Other examples:

```
390 HOME : PRINT "THAT WASN'T BETWEEN 1 AND 15!" :
    PRINT
720 COLOR = 0 : PLOT X,Y
760 X = NX : Y = NY
```

Line 390 combines three related statements: HOME clears the screen, the first PRINT displays a message to the user, and the second PRINT displays a blank line after the message. In line 720, the color is set to black (0) each time X and Y are plotted; the section "Creating Visual Impressions" explains more about this. Line 760 changes variable X to NX, and variable Y to NY.

On a program line using the colon and more than one statement, remarks must be the last on the line.

Creating Motion

The bouncing ball, which is really a brick plotted and moving on the graphics grid, is created with six variables in COLORBOUNCE:

X represents the starting position of the back-and-forth motion
Y represents the starting position of the up-and-down motion
XV represents the velocity of X
YV represents the velocity of Y
NX represents the changing position of X (NX = X + XV)
NY represents the changing position of Y (NY = Y + YV)

The program lines that introduce these variables in COLORBOUNCE look almost the same as the descriptions you just read. The use of clear REM statements can really help.

```
440 X = 0 : REM SET STARTING POSITION OF BACK-AND-
    FORTH VARIABLE
460 Y = 5 : REM SET STARTING POSITION OF UP-AND-DOWN
    VARIABLE
480 XV = 2 : REM SET X VELOCITY
500 YV = 1 : REM SET Y VELOCITY
520 REM CALCULATE NEW POSITION
540 NX = X + XV : NY = Y + YV
```

Helpful Hint: If you find it confusing to have more than one statement on a program line, it is OK to use the longer method. In this portion, for example, you would have 11 lines instead of 6.

One of the main reasons for combining statements is to save space in long programs: the more program lines, the more memory used. Another reason is that program execution is faster with multiple statements on a line. A third reason is that some statements, like IF...THEN, **control** all the statements on the same line; combining statements thus affects **how** the program works. When you begin writing longer programs of your own, you'll find helpful tips on saving space and time in the *Applesoft Reference Manual.*

Screen Boundaries

When you first began plotting points on the graphics grid you learned two crucial pieces of information: there are 40 horizontal and 40 vertical points on the grid; the numbering system of the grid goes from 0 to 39 in both directions. To keep a bouncing ball within these limits, you have to do two things:

1. Set some limits in the program so the ball doesn't appear to move off the screen.
2. Change the direction of the ball when it reaches the edge of the screen.

The COLORBOUNCE lines that follow do both these things. Look at them carefully so you understand how they work.

```
560 REM IF BALL EXCEEDS SCREEN EDGE, THEN BOUNCE
580 IF NX > 39 THEN NX = 39 : XV = -XV
600 IF NX < 0 THEN NX = 0 : XV = -XV
620 IF NY > 39 THEN NY = 39 : YV = -YV
640 IF NY < 0 THEN NY = 0 : YV = -YV
```

Figuring this out is not easy. One thing to remember is that variable values are not static: when a line says XV = -XV, the contents of the variable actually change to a negative value.

In other words, when the ball reaches the limit of the screen, 39, line 580 changes XV to a negative XV. Then when line 540 is reached in the next loop, NX becomes 38. In the loop after that, NX becomes 37, and so on, until the ball reaches the other screen edge, at which point 1 is added to the value of NX with each loop. This is how the direction is changed back and forth.

Creating Visual Impressions

By alternating between black and a color for the ball each time a new position is plotted for X and Y, you can make it look as if the ball is being erased, adding to the visual effect of the ball's motion. This is done in COLORBOUNCE with the following lines:

```
660 REM PLOT NEW POSITION
680 COLOR = 7 : PLOT NX,NY
700 REM ERASE OLD POSITION
720 COLOR = 0 : PLOT X,Y
740 REM SAVE CURRENT POSITION
760 X = NX : Y = NY
780 GOTO 540
```

The PLOT NX,NY statement in line 680 plots a brick (otherwise known as a ball) in the specified color at the new position set by NX and NY. When line 720 plots X and Y, it is really plotting the **old** coordinates NX and NY, which are saved after plotting the previous brick (line 760). The statement COLOR = 0 in line 720 sets color to black, making it look as if the brick has been erased. This is because the plotting happens so fast. The next loop, of course, resets the color and plots NX and NY in that color (line 680). The alternating between black and color in each loop adds to the appearance of motion.

If the order of these statements doesn't make sense to you, try moving them around and running the program with your changes incorporated.

The Whole Thing

Here is a listing of COLORBOUNCE. This is exactly the same as the program on the APPLESOFT SAMPLER. List that program to compare the versions. You can either list it in sections, such as

```
LIST 400,580
LIST 600,780
```

or you can list the whole thing and use CONTROL-S to stop and start the listing. Go ahead, try it!

```
400 GR : REM SET COLOR GRAPHICS AREA
420 HOME : REM CLEAR TEXT AREA
440 X = 0 : REM SET STARTING POSITION OF BACK-AND-
    FORTH VARIABLE
460 Y = 5 : REM SET STARTING POSITION OF UP-AND-DOWN
    VARIABLE
480 XV = 2 : REM SET X VELOCITY
500 YV = 1 : REM SET Y VELOCITY
520 REM CALCULATE NEW POSITION
540 NX = X + XV : NY = Y + YV
560 REM IF BALL EXCEEDS SCREEN EDGE, THEN BOUNCE
580 IF NX > 39 THEN NX = 39 : XV = -XV
600 IF NX < 0 THEN NX = 0 : XV = -XV
620 IF NY > 39 THEN NY = 39 : YV = -YV
640 IF NY < 0 THEN NY = 0 : YV = -YV
660 REM PLOT NEW POSITION
680 COLOR = 7 : PLOT NX,NY
700 REM ERASE OLD POSITION
720 COLOR = 0 : PLOT X,Y
740 REM SAVE CURRENT POSITION
760 X = NX : Y = NY
780 GOTO 540
```

Here's how COLORBOUNCE works, in summary: the program plots a brick, erases it, plots another brick one column over, erases that, and so on. When the edge of the screen is reached (in lines 580-640, when NX and NY are greater than 39 and when NX and NY are less than 0), the program reverses the plotting direction.

Program Interaction with Users

Suppose you decide to change COLORBOUNCE so that a new ball color can be entered each time the program is run. One way of doing this is to change line 680, run the program, and change line 680 again. But that would be a lot of trouble. And it would be difficult to explain if you wanted to show a friend how.

A better way is to use the INPUT statement to let the user interact with the program. As you recall from Chapter 2, an INPUT statement names a variable and asks the person using the program to enter something from the keyboard.

You have to be careful how you word the statement because COLOR is a reserved word. If, for example, you added the line

```
350 INPUT COLOR
```

the program would get stuck: COLOR can't be used as a variable name. You could, however, write line 350 as

```
350 INPUT "COLOR? "; HUE
```

and change line 680 to define COLOR= as variable HUE:

```
680 COLOR = HUE : PLOT NX,NY
```

If you haven't done so, add these lines to COLORBOUNCE. Then run it to see how the lines work. (When the program executes line 350, the word COLOR, followed by a question mark (?), will appear on the screen. The cursor will keep blinking until someone types a number and presses RETURN.)

You may know what to do when you see the question mark. However, your friend may not. So it is a good idea to have the program tell your friend (the user) what is expected. Adding some PRINT statements and changing to text mode at the beginning of the program will do the trick:

```
280 REM SET TEXT MODE
300 TEXT : HOME
310 PRINT "TO SELECT A COLOR FOR"
320 PRINT "THE BOUNCING BALL, FIRST TYPE"
330 PRINT "IN ANY NUMBER FROM 1 TO 15."
340 PRINT "THEN PRESS THE KEY LABELED RETURN.": PRINT
```

It also would help to put a more specific message in the INPUT statement:

```
350 INPUT "WHAT COLOR WOULD YOU LIKE THE BALL TO BE
    (1-15)? "; HUE
```

Writing the INPUT statement:

- The message must be in quotation marks in an INPUT statement.
- When the INPUT statement contains a message, the computer doesn't add a question mark. If you want a question mark to appear, you must include it in the INPUT message.
- Putting a space after the question mark, within the quotation marks, sets the question off from the answer.
- There must be a semicolon between the message and the variable name.

Adding the new lines 280 to 350 to COLORBOUNCE will make the program much more friendly for users. But there still are some potential pitfalls for people who are not used to computers. What if someone makes a mistake and then presses RETURN? Instant error messages and loud beeps!

If too great or too small a number is entered, the program will either let the ball move to the right side of the screen and then stop or the message

```
?ILLEGAL QUANTITY ERROR IN 680
```

will appear on the screen and then the program will stop. For the most part, your friends will not know how to restart the computer—and shouldn't have to. Therefore you should make the program check the number typed in by the user before proceeding. These lines will do it:

```
370 REM IS HUE OF BALL IN RANGE?
380 IF HUE > 0 AND HUE < 16 THEN GOTO 400
390 HOME : PRINT "THAT WASN'T BETWEEN 1 AND 15!" :
    PRINT
395 GOTO 310
```

If the character entered is not a number,

```
]?REENTER
]?
```

will appear on the screen. You'll learn how to modify the program to avoid this in Chapter 5.

It is good programming practice to make a program as foolproof as possible. You have gotten to the point where you are writing error messages for others to read. It may be all right for a programmer like you to deal with such jargon as ?SYNTAX ERROR, but an innocent user shouldn't have to.

Each time you use an INPUT statement you should make your program check what the user enters so the program doesn't fail in any way. Dealing with the untutored user (and you must assume that users are not programmers) is an art in itself. Use of specifically worded INPUT statements and careful checking of the user's response are always required.

Now that you have made some changes to COLORBOUNCE, list it again. Make sure that you have added all the new lines (280-395), that you have changed line 680, and that all the lines are typed correctly. Run the new version. Then save it as NEW COLORBOUNCE.

Each time you save a different version of a program, you must give the version a new name. Otherwise the old version will be written over and lost.

Pause

Making Sounds

Clicks, ticks, tocks, and various buzzes are easily generated by the Apple IIe. You can make sounds on your computer if you tap it, scratch your fingers across it, or drop it, but the sounds you are about to make are produced by programming it.

To construct a sound-producing program on the Apple IIe, you need this formula:

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

There is no easy explanation for this formula. The number, -16336, is related to the memory *address* of the Apple IIe's speaker and is built into the electronics of the computer.

The PEEK function returns the contents, in decimal form, of the byte at the memory address specified by the argument.

PEEK is a *function.* In Applesoft a function is something that takes one or more numbers and performs some operation on them to yield a single value. Applesoft has a number of built-in arithmetic functions (like SQR, which finds the square root of a number); some other functions can be derived. The *Applesoft Reference Manual* discusses all of Applesoft's functions.

The number that the function uses (-16336 in this case) is called its argument and is always put in parentheses after the function name. The number the function finds is said to be returned to the program. PEEK returns the numeric value of the contents of any byte in memory. You supply the address of that byte when you provide the number in parentheses; in this case, the address is -16336. The Apple IIe has memory addresses that range from 0 to 65535.

The range of memory addresses is determined by the size of memory in the machine. The Apple IIe has 64 *kilobytes* (64K) of memory, which is equal to 65535 bytes, as 1K represents 2 to the 10th power or 1,024.

At most locations PEEK only returns a numeric value, but at some locations, such as -16336, it can cause something to happen. In this case, it causes the speaker in the Apple IIe to make a click. Every time the program executes this statement, you will hear a barely audible click. Run the program and listen to your computer closely.

Now add this line:

```
160 GOTO 150
```

and run the program. No problem hearing this!

To make your program buzz for a limited period of time, add these statements:

```
140 FOR BUZZ = 1 TO 100
160 NEXT BUZZ
```

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, here is a statement that will allow you to substitute the variable S for the number. Add this to your sound-producing program:

```
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 produce different qualities of sound. Try some variations. See what happens when you use all -PEEK(S)s. How does the sound change when you use a lot of +PEEK(S)s?

For a more buzzy tone, put one of your variations into a loop. In general, the faster the loop, the higher the pitch.

Applesoft does arithmetic operations at different speeds: it takes longer to do subtraction than to do addition and longest to do division. So the fastest PEEK loop is composed of all +PEEK(S)s. A slower loop is composed of -PEEK(S)s. The slowest loop would use division: PEEK(S) / PEEK(S) / PEEK(S), for example. A lot more information on PEEKs can be found in the *Applesoft Reference Manual.*

To produce even higher tones on the Apple IIe, try these lines:

```
NEW
40 FOR PAUSE = 1 TO 2500 : NEXT PAUSE
50 S = PEEK(-16336) : GOTO 50
```

Remember, CONTROL-C stops program execution. As you might guess, line 40 causes a pause before line 50 is executed. The GOTO statement in line 50 causes a continuous sound after the pause.

To put these sound tricks to good use, load your most recent version of COLORBOUNCE into the computer. List it, and study the lines. Then try to add a sound for each time the ball bounces off a wall.

One possible solution is given in the next section, but try to work it out for yourself.

Hint: A bounce occurs whenever XV and YV change value (sign) and direction.

Noise for the Bouncing Ball

Here is one way to make the bouncing audible. Add these lines to COLORBOUNCE:

```
240 REM SET S TO ADDRESS OF SPEAKER
260 S = -16336
580 IF NX > 39 THEN NX = 39 : XV = -XV : FOR B = 1 TO 5 :
    BOUNCE = PEEK(S) + PEEK(S) + PEEK(S) : NEXT B
600 IF NX < 0 THEN NX = 0 : XV = -XV : FOR B = 1 TO 5 :
    BOUNCE = PEEK(S) + PEEK(S) + PEEK(S) : NEXT B
620 IF NY > 39 THEN NY = 39 : YV = -YV : FOR B = 1 TO 5 :
    BOUNCE = PEEK(S) + PEEK(S) + PEEK(S) : NEXT B
640 IF NY < 0 THEN NY = 0 : YV = -YV : FOR B = 1 TO 5 :
    BOUNCE = PEEK(S) + PEEK(S) + PEEK(S) : NEXT B
```

You can see the advantage of having multiple statements on a single line in a program like this. The sounds are directly connected to the values of XV and YV so it makes sense (and works more efficiently) to add to the already existing lines.

Now try your own sounds. Why not make a different sound bounce off each wall? When you find a sound combination you like, don't forget to save a new version of COLORBOUNCE with the noisemaking lines included. The remainder of this chapter will introduce some new programs and concepts; we'll return to COLORBOUNCE in Chapter 5.

Pause

Random Numbers

Here's another Applesoft function. Try it (remember you can stop it when you want with CONTROL-C):

```
NEW
100 PRINT RND(1)
110 GOTO 100
RUN
```

The arithmetic function RND returns a random real number greater than or equal to zero and less than one.

RND in line 100 stands for "random." The RND function produces random numbers. Each time you run it, you'll see a different sample of numbers. Try it!

The numbers generated by this program are random decimal fractions between zero and one. Any argument (the number in parentheses) greater than zero will return random decimal fractions between zero and one. Try, for example, changing line 100 to

```
100 PRINT RND (6)
```

Although you won't be doing it in this tutorial, you can produce different effects by using an argument of zero or an argument of less than zero. See the *Applesoft Reference Manual* for more information.

Random decimal fractions between one and zero can be a little clumsy. Often integers (numbers like three, six, and ten) are easier to use. To get random integers from zero to nine you have to change the program. Type

```
NEW
 90 REM ASSIGNS RND NUMBER TO X
100 X = RND(1)
110 REM MULTIPLIES X BY 10
120 X = X * 10
130 REM CHOPS OFF THE FRACTION
140 X = INT(X)
150 PRINT X
160 GOTO 100
```

The INT function returns the largest integer less than or equal to the given argument.

Line 140 introduces `INT`, the integer function. The statement `X = INT(X)` returns the largest integer that is less than or equal to the value of `X`. For instance, if the value of `X` is `3.6754`, then `INT(X)` is equal to `3`; if the value of `X` is `-45.12345`, then `INT(X)` is equal to `-46`. The parentheses after `INT` can contain any arithmetic expression or numeric variable.

Now run the program. Does it work the way you expected?

To change the program so that it generates numbers from one to ten (instead of from zero to nine) add this line to your program:

```
145 X = X + 1
```

This program may seem a little complicated at first. To see what happens step by step add lots of PRINT statements. The following program shows how. However, you don't have to type in the additions. The program RANDOM on the APPLESOFT SAMPLER disk is the same as what you see.

```
 90 REM ASSIGNS RND NUMBER TO X
100 X = RND(1) : PRINT "X = RND(1)",X
105 PRINT
110 REM MULTIPLIES X BY 10
120 X = X * 10 : PRINT "X = X * 10",X
125 PRINT
130 REM CHOPS OFF THE FRACTION
140 X = INT(X) : PRINT "X = INT(X)",X
145 PRINT
150 REM ADDS 1 TO THE VALUE OF X
160 X = X + 1 : PRINT "X = X + 1", X
170 PRINT : PRINT
180 FOR PAUSE = 1 TO 2000 : NEXT PAUSE
190 GOTO 100
```

Load and run this program to see what happens. To change the way the numbers are displayed, you can remove the comma (,) and the X from the end of line 160, change line 170, and add line 175, like this:

```
160 X = X + 1 : PRINT "X = X + 1"
170 PRINT X
175 PRINT
```

Amazingly enough, this whole program can be condensed to just one line:

```
100 PRINT INT (10 * RND(1) ) + 1 : GOTO 100
```

Study this line until you figure out how its parts correspond to each of the lines in RANDOM.

Simulating a Pair of Dice

Now you can use what you've learned about random numbers to make a program act like a pair of dice.

```
NEW
100 PRINT "WHITE DICE",
110 PRINT INT (6 * RND(1) ) + 1
120 PRINT "RED DICE",
130 PRINT INT (6 * RND(1) ) + 1
```

This program generates random integers from one to six for each die. Each time you run the program, you simulate a roll of the dice. What could you add to the program so you don't have to execute it over and over? Can you write a program that uses these "dice" to play a game? Try it.

Now try writing a one-line program that generates random numbers from 1 to 50. From 0 to 25. Make up your own limits. Remember to add one (1) to the random number if you don't want to generate zeros.

Helpful Hint: The number RND is multiplied by (as in 50 * RND(1)) represents the range of numbers to be generated. The number added to RND is the smallest number generated: if 1 is added, the lowest number will be 1; if 3 is added, the lowest number will be 3; if nothing is added, the lowest number will be 0.

Random Graphics

Here's a colorful way to combine the RND function with some graphics statements:

```
NEW
200 GR
210 REM CHOOSE A RANDOM COLOR
220 COLOR = INT (16 * RND(1) )
230 REM CHOOSE A RANDOM POINT
240 X = INT (40 * RND(1) )
250 Y = INT (40 * RND(1) )
260 REM PLOT THE RANDOM POINT
270 PLOT X, Y
280 REM DO IT AGAIN
290 GOTO 220
```

Try using RND in other programs. Can you write a program that draws random lines in random colors across the screen? How about a program that shades the graphics screen with random colors?

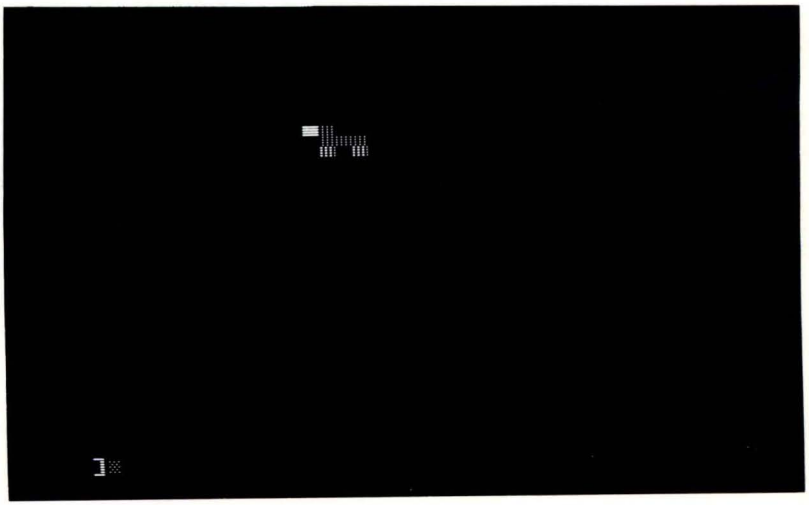
The RND function is used in many games. A somewhat longer example of its use is SCRAMBLER on the APPLESOFT SAMPLER. That program is discussed in Appendix E, "More Programs To Play With."

Subroutines: Putting the Pieces Together

This section takes you through the step-by-step process of thinking about a program and putting the pieces together. It begins with an idea: that you want to draw a horse as part of a game you are developing. Here is a program that draws a blue horse with orange feet and a white face:

```
NEW
1000 REM DRAW BLUE HORSE WITH WHITE FACE AND ORANGE
     FEET
1010 GR
1020 COLOR = 2 : REM DARK 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
```

Figure 4-1. A Blue Horse



There is nothing wrong with this program: it draws a blue horse with orange feet and a white face. But suppose you wanted 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.

You learned in COLORBOUNCE that variables can be used to change the velocity and direction of points plotted on the graphics screen. A similar method can be used to move the horse to a different place on the screen.

You can move a point that is at coordinates (A,B) to the right by adding to the value of the first coordinate, A. If A = 4 and B = 17, you could move point (A,B) 10 columns to the right by adding 10 to the first coordinate, making the point (14,17).

Likewise, a point moves left if you subtract from the first coordinate (or add a negative value). A simple experiment shows that adding to and subtracting from the second coordinate moves points down and up, respectively.

This, in essence, is the basis of all animation in programming.

With these ideas in mind, you can rewrite the horse program to place the horse at almost any point (X,Y) on the screen. Why almost any point? Because if the center of the horse is plotted at the edge of the screen, part of the horse will go off the screen. This might give you an ?ILLEGAL QUANTITY ERROR IN 1030 message. The number at the end of the error message identifies the line at which the error occurs and makes it easier for the programmer to correct the problem. Here is part of an improved program:

```
NEW
1000 REM PUT A HORSE ANYWHERE ON THE SCREEN
1010 COLOR = 2 : REM DARK BLUE BODY
1020 PLOT X, Y - 1 : REM CENTER OF HORSE
1030 HLIN X, X + 2 AT Y : REM REST OF BODY
1040 COLOR = 9 : REM ORANGE FEET
1050 PLOT X, Y + 1 : REM FRONT FOOT
1060 PLOT X + 2, Y + 1 : REM REAR FOOT
1070 COLOR = 15 : REM WHITE HEAD
1080 PLOT X - 1, Y - 1
```

Notice that the GR statement has been left out. This part of the program is supposed to put several horses on the screen. A GR statement at 1005 would clear the screen before each new horse was drawn.

This program can't be run as it is. You must set the graphics mode and choose X and Y:

```
20 GR
30 REM FIRST HORSE CENTER
40 X = 12
50 Y = 35
```

When you execute this, you will get one horse at the desired location before the program ends. But you still want to put two horses on the screen. What if you could write what you see in Figure 4-2:

Figure 4-2. A "What If" Program

```
 60 DO THE PORTION OF THE PROGRAM AT LINE 1000
    AND THEN COME BACK TO LINE 70
 70 REM SECOND HORSE CENTER
 80 X = 33
 90 Y = 2
100 DO THE PORTION OF THE PROGRAM AT LINE 1000
    AGAIN AND THEN END
```

Wouldn't that be nice and easy? The problem is that the computer can't read those strange instructions in lines 60 and 100. It can, however, read the statement

```
GOSUB 1000
```

GOSUB causes a program to branch to the line number given. The subroutine beginning at that line number should end with a RETURN statement, which causes the program to branch back to the statement immediately after the GOSUB.

A program such as the one starting at line 1000 is called a *subroutine.* GOSUB 1000 tells the computer to go to the subroutine beginning at line 1000 and execute 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 the subroutine is finished when it encounters a RETURN statement. To make a complete subroutine in your horse-drawing program, add the line

The portion of the program from line 1000 to 1090 is called a **subroutine.** A subroutine is a segment of a program that is used over and over by the main routine (lines 20-100) of the program.

```
1090 RETURN
```

Combining all these lines, you have that "what if you only could" program:

```
  20 GR
  30 REM CHOOSE CENTER OF THE FIRST HORSE
  40 X = 12
  50 Y = 35
  60 GOSUB 1000
  70 REM CHOOSE CENTER OF THE SECOND HORSE
  80 X = 33
  90 Y = 2
 100 GOSUB 1000
1000 REM PUT A HORSE ANYWHERE ON THE SCREEN
1010 COLOR = 2 : REM DARK BLUE BODY
1020 PLOT X, Y - 1 : REM CENTER OF HORSE
1030 HLIN X, X + 2 AT Y : REM REST OF BODY
1040 COLOR = 9 : REM ORANGE FEET
1050 PLOT X, Y + 1 : REM FRONT FOOT
1060 PLOT X + 2, Y + 1 : REM REAR FOOT
1070 COLOR = 15 : REM WHITE HEAD
1080 PLOT X - 1, Y - 1
1090 RETURN
```

If you run this program, you'll get the error message

```
?RETURN WITHOUT GOSUB ERROR IN 1090
```

You'll find out how to fix the program to avoid this error shortly. Keep reading! However, despite the error message, the program runs fine. You have just created a horse-drawing *routine.* Now use the statement

```
GOSUB 1000
```

to draw one of these special horses at whatever (X,Y) location you choose.

Traces

The debugging statement TRACE indicates the path the program follows as it is executed.

To follow the program's flow, or path of execution, you can invoke a special feature called TRACE. This statement will show you why the Apple IIe gave an error message when the horse-drawing program was executed. Add this line:

```
10 TRACE
```

and, for a moment, delete line 20 with

```
DEL 20,20
```

To leave the graphics mode and clear the screen, type TEXT and HOME. Then run the program.

Figure 4-3. Using TRACE. TRACE displays the line number of each statement as it is executed.

```
#10 #30 #40 #50 #60 #1000 #1010 #1020 #1
030 #1040 #1050 #1060 #1070 #1080 #1090
#70 #80 #90 #100 #1000 #1010 #1020 #1030
#1040 #1050 #1060 #1070 #1080 #1090 #10
00 #1010 #1020 #1030 #1040 #1050 #1060 #
1070 #1080 #1090
?RETURN WITHOUT GOSUB ERROR IN 1090
```

TRACE shows that the program begins at line 10, the smallest line number, and continues until it reaches line 60, which sends it to the subroutine. It executes the subroutine, returns to the main program at line 1090, then executes the subroutine again. When it reaches line 1090 again, not finding any smaller line numbers, it goes to line 1000 and executes the subroutine again. This is where the problem occurs. Do you understand the error message now?

To remedy the problem of endless repetition, add this new line to the program:

The END statement causes a program to cease execution and returns control to the user.

```
110 END
```

When the program gets to line 110, it will do just what the line says: end. Run the program once more. No error message this time. Because the RETURN statement causes the program to branch to the statement immediately following the most recently executed GOSUB, line 110 is reached and the program stops.

As you see, TRACE is extremely handy when you are having problems with a program. By adding TRACE, you can find out where the problem is.

The NOTRACE statement turns off TRACE.

If you want to TRACE only part of a program, use the NOTRACE statement. Add this line:

```
65 NOTRACE
```

and the program will be traced only up to the execution of line 65.

TRACE can also be issued when you are working without line numbers in immediate execution, TRACE and then RUN. It is not a good idea to try it out now, unless you save the current version of the horse-drawing program first.

Once you have issued the TRACE statement, whether in immediate or deferred execution, the program will be traced every time you run it. To stop TRACE you must issue a NOTRACE statement and remove the TRACE statement.

Now that you have finished tracing the horse-drawing program, issue a NOTRACE command, remove lines 10 and 65, and reinsert line 20. That way you can run this program in the future without having it traced each time.

A Better Horse-Drawing Routine

This section introduces several additions and modifications to the horse-drawing program. The changes are based on techniques that will be useful in any programming you do.

Errors

It is important to anticipate possible errors when writing programs so that problems are avoided. One problem with the 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 adding a set of statements such as:

```
1000 REM PUT A HORSE ANYWHERE WITHIN SCREEN
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
```

The format of these lines should be familiar: it is similar to that used in COLORBOUNCE. To keep the horses within the screen limits, the values of X and Y are compared with the values of the graphics grid. (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 programming strategies, such as giving an error message and stopping the program. However, limiting the range of the horse has the advantage of not stopping the program.

Note that line 1000 is slightly different from the old line 1000; the rest of these lines are additions to the subroutine.

Variables

As you have already discovered, using variables instead of specific number assignments for colors gives a program more flexibility. If, for example, you wanted to add a second player to the game and give that player a horse of a different color, you could replace line `1010 COLOR = 2` with

```
1010 COLOR = BODY
```

Similarly, you could write

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

Then the main routine would change to:

```
 20 GR
 30 REM FIRST PLAYER'S HORSE COLOR
 40 BODY = 2 : REM DARK BLUE
 50 FEET = 9 : REM ORANGE
 60 FACE = 15 : REM WHITE
 70 REM FIRST PLAYER'S HORSE CENTER
 80 X = 15
 90 Y = 30
100 GOSUB 1000
```

and so on. Now you add the lines for the second player's horse color. Be sure to include an END statement. Run the program to see the two horses displayed. Change your program lines until you have color combinations you like.

This is a good opportunity to use some of the editing skills you learned in Chapter 3, "Editing Long Programs." One of the advantages of using escape mode is that you can list a line that appears in similar form more than once in the program, make the relevant changes, and copy it over for easy reproduction. Try that now—it will save you a lot of typing.

Additional Subroutines

To refine the horse-drawing program even more, use subroutines that assign the colors for each player's horse and then have each of those subroutines go to (or call) the horse-drawing subroutine in turn. Here are two examples:

```
2000 REM DRAWS BLUE HORSE WITH ORANGE FEET AND WHITE
     FACE
2010 BODY = 2 : REM DARK BLUE
2020 FEET = 9 : REM ORANGE
2030 FACE = 15 : REM WHITE
2040 GOSUB 1000
2050 RETURN

2500 REM DRAWS ORANGE HORSE WITH PINK FEET AND GREEN
     FACE
2510 BODY = 9 : REM ORANGE
2520 FEET = 11 : REM PINK
2530 FACE = 4 : REM GREEN
2540 GOSUB 1000
2550 RETURN
```

When you add subroutines 2000 and 2500, you also must change the main routine. To put a blue horse with a white face and orange feet at (10,11), change lines 30-60 as follows:

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

To put an orange horse at (19,2), change lines 70-100 to

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

The horse-drawing subroutines at lines 2000 and 2500 call another subroutine that begins at line 1000. Things become quite efficient at this stage. The three subroutines make it very easy to put up an attractive display of horses.

But first, another handy subroutine that replaces the old one at lines 1012-1018:

```
3000 REM CHOOSES A RANDOM PAIR OF COORDINATES
3010 X = INT (RND(1) * 37) + 1
3020 Y = INT (RND(1) * 38) + 1
3030 RETURN
```

A Well-Structured Program

Here is the whole program. It combines the main routine with three subroutines. You can change the program you have been typing in to match what you see or load HORSES from the APPLESOFT SAMPLER.

```
 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 routine should look if you are writing well-structured programs: mostly REM and GOSUB statements. The work should be done in relatively short subroutines, each of which is easy to write and complete in itself.

> Appendix E introduces more examples of subroutines and how to structure them for maximum program efficiency.

If the workings of this program are still a little unclear, use TRACE to follow the program flow. Type TEXT and HOME. Then add

```
  5 TRACE
130 NOTRACE
```

and then RUN. Compare what you see on the screen with the program listing until you can follow the subroutines.

Pause

High-Resolution Graphics

So far, you have used low-resolution graphics. In this section you will be introduced to another kind of graphics called *high-resolution graphics.*

High-resolution graphics lets you draw with much more detail than you could with the low-resolution grid. The high-resolution graphics screen is 280 by 160 plotting points. The horizontal coordinates start with 0 at the left of the screen and end with 279 at the right. Likewise the vertical coordinates go from 0 at the top of the screen to 159 at the bottom.

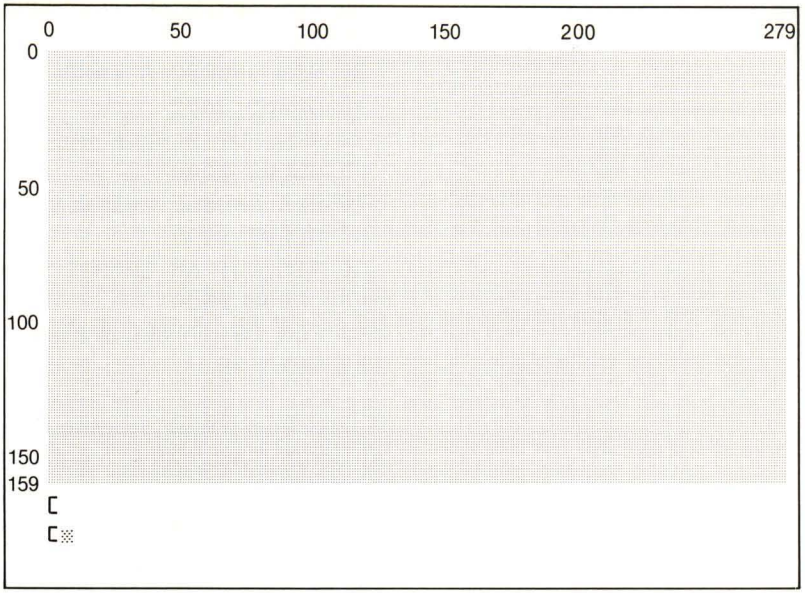


Figure 4-4. High-Resolution Graphics Screen Coordinates. The horizontal coordinates range from 0 to 279; the vertical coordinates range from 0 to 159.

High-resolution graphics commands are often the same as the corresponding low-resolution graphics commands except for the addition of an H (for "high" resolution). Your familiarity with low-resolution graphics will be helpful to you in this section.

Type

HGR

High-resolution graphics, set by HGR, uses a screen grid of 280 by 160 plotting points and leaves four lines for text at the bottom. HGR clears the screen to black.

to get into high-resolution graphics. This statement clears the screen to black and leaves four lines at the bottom for text. As in low-resolution graphics, high-resolution graphics allows you to use vertical coordinates that would be in the text area (191 is the maximum), but these points are not shown on the screen unless you do some tricky maneuvers. See the *Applesoft Reference Manual* for more information.

Helpful Hint: If the cursor is not visible, press RETURN until the cursor appears near the bottom of the screen.

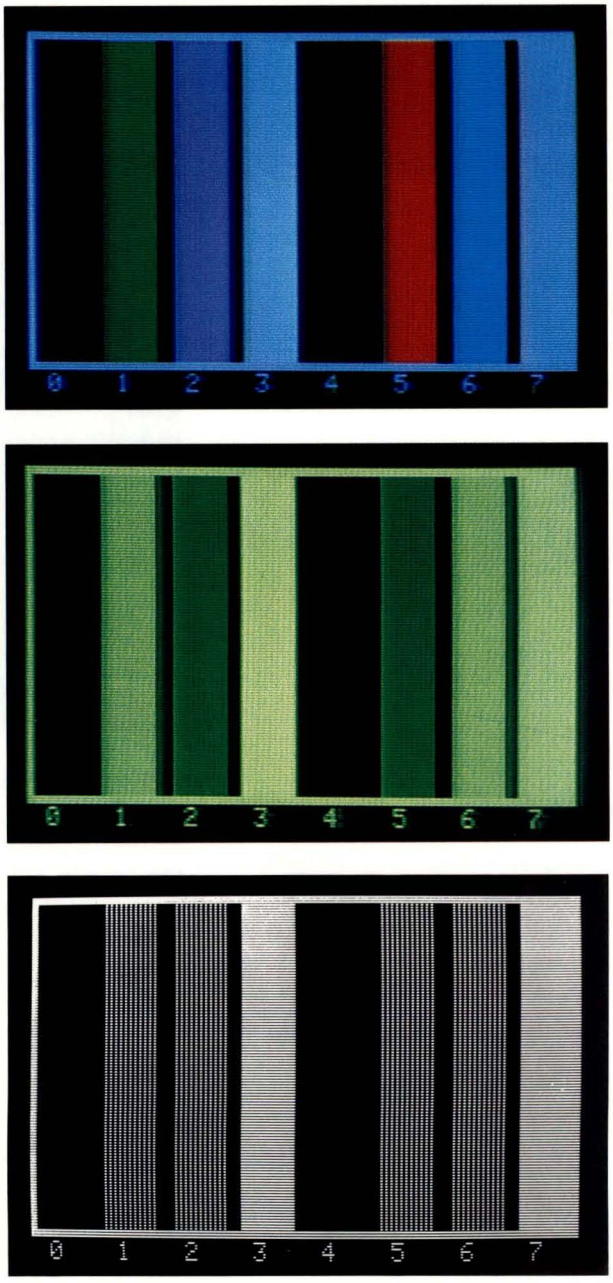
HCOLOR= sets the high-resolution graphics color to the number specified.

The color of the dots you plot in high-resolution graphics is determined by the HCOLOR= statement. As in low-resolution graphics, the Apple IIe will use whatever color you assign until you change it.

High-resolution graphics is truly wonderful, but you have to make some sacrifices to use it. There are fewer numbers assigned to HCOLOR=, and the colors vary according to their positions on the screen. Look at Figure 4-5 to see which number goes with which color.

Figure 4-5. High-Resolution Graphics Colors

0 black1	4 black2
1 green	5 orange
2 violet	6 blue
3 white1	7 white2



To try out high-resolution graphics, once you have issued the HGR statement, type

```
HCOLOR = 2
HPLOT 130,100
HPLOT 50,50
```

and so on. Notice how much smaller the plotted points appear in high resolution. This mode makes it possible to create images with much more detail than you can achieve in low resolution.

Figure 4-6. The Fine Resolution of the HGR Screen

The HPLOT statement plots dots and lines in high-resolution graphics using the most recently specified value of HCOLOR=.

Drawing lines is even easier in high-resolution graphics than in low-resolution graphics. You simply HPLOT from one point on the screen TO another point. To draw a line along the top edge of the screen, type

```
HCOLOR = 1
HPLOT 0,0 TO 279,0
```

If you then want to draw a line from the corner at point 279,0 to the bottom corner of the screen all you have to do is type

```
HPLOT TO 279,159
```

and a line appears along the right edge of the screen. When you use this last statement, the new line takes its starting point and its color from the point previously plotted (even if you have issued a new HCOLOR= command since that point was plotted). To see for yourself, type

```
HCOLOR = 4
HPLOT TO 0,159
```

The color 4 is black so you would think the line wouldn't show up. But it does because it takes its color from its starting point, which is green (number 1).

You also can combine several lines in one HPLOT statement. Clear the screen with HGR, and try this on the Apple IIe:

```
HCOLOR = 3
HPLOT 0,0 TO 279,0 TO 279,159 TO 0,159 TO 0,0
```

There should be a line around the edge of the screen, as in Figure 4-7. If there isn't a continuous line around the edge of the screen, check your typing.

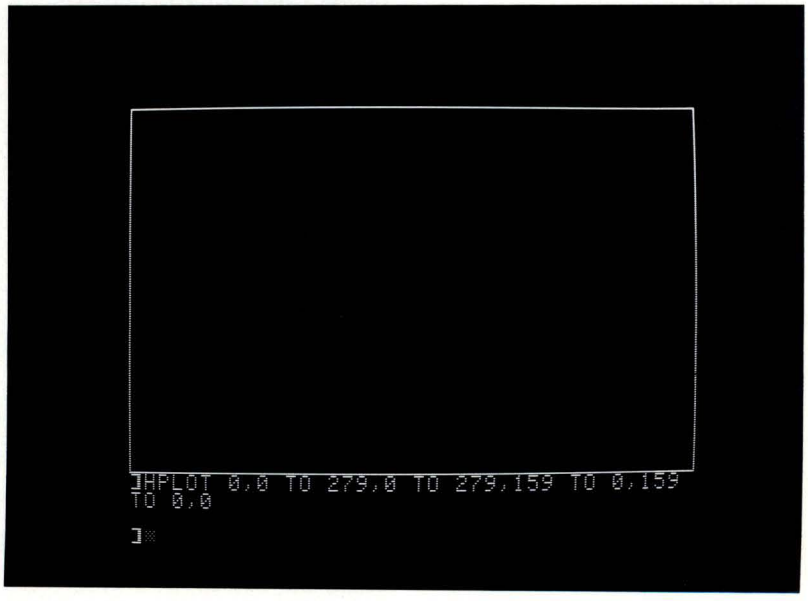


Figure 4-7. A Line Around the Edge of the High-Resolution Screen

If the top line is not visible on your screen, your television set or video monitor needs adjustment. Try adjusting it. If that doesn't work, replace `HPLOT 0,0 TO 279,0` with `HPLOT 0,2 TO 279,2` and so on.

You see how easy it is to draw straight lines in high resolution. Well, diagonal lines are just as easy. To draw a line from the top-left corner of the screen to the bottom-right corner, type

```
HPLOT 0,0 TO 279,159
```

Practice drawing high-resolution lines of varying length and color. You will discover that certain colors don't always draw vertical lines on black-and-white monitors. Colors 2 and 6 only plot vertical lines beginning with even numbers, while 1 and 5 only plot lines beginning with odd numbers. For example, if `HCOLOR = 1`, then `HPLOT 279,0 TO 279,159` will draw a vertical line on the screen, but `HPLOT 2,159 TO 0,0` will not. This is because of the way high-resolution colors are interpreted by black-and-white monitors.

To see an example of what you can design with high-resolution graphics, load the program `MOIRE` from the APPLESOFT SAMPLER. Run it; stop the program with CONTROL-C. Then type `TEXT`, `HOME`, and `LIST` to see the program lines.

Figure 4-8. `MOIRE` Program Listing. Numbers 1 and 2 correspond to those in the text.

```
NEW
 80 REM MOIRE PROGRAM
 90 HOME
100 VTAB 24 : REM MOVE CURSOR TO BOTTOM LINE
120 HGR : REM SET HIGH-RESOLUTION GRAPHICS
140 A = RND(1) * 279 : REM PICK AN X FOR
    "CENTER"
160 B = RND(1) * 159 : REM PICK A Y FOR "CENTER"
180 N = INT (RND(1) * 4) + 2 : REM PICK A
    STEP SIZE
200 HTAB 15 : PRINT "STEPPING BY "; N;
220 FOR X = 0 TO 278 STEP N : REM STEP THRU A VALUES
240 FOR S = 0 TO 1 : REM 2 LINES, FROM X AND
    X + 1
260 HCOLOR = 7 * S : REM FIRST LINE BLACK, NEXT
    WHITE
280 REM DRAW LINE THROUGH "CENTER" TO OPPOSITE SIDE
300 HPLOT X + S,0 TO A,B TO 279 - X - S,159
320 NEXT S,X
340 FOR Y = 0 TO 158 STEP N : REM STEP THRU B
    VALUES
360 FOR S = 0 TO 1 : REM 2 LINES, FROM Y AND
    Y + 1
380 HCOLOR = 7 * S : REM FIRST LINE BLACK, NEXT
    WHITE
400 REM DRAW LINE THROUGH "CENTER" TO OPPOSITE SIDE
420 HPLOT 279,Y + S TO A,B TO 0,159 - Y - S
440 NEXT S,Y
460 FOR PAUSE = 1 TO 1500 : NEXT PAUSE : REM DELAY
480 GOTO 120 : REM DRAW A NEW PATTERN
```

1 — 280
2 — 320
1 — 400
2 — 440

1. The center of the high-resolution screen is at the intersection of four points, or dots, and cannot be precisely plotted. Thus "CENTER" in lines 280 and 400 is approximate.

2. One instruction can provide the NEXT for more than one FOR statement, as you see in lines 320 and 440. Be careful that you list the NEXT variables in the right order, though, to avoid crossed loops.

Can you think of ways to change the program? For example, try making the value of HCOLOR= change randomly. Try drawing orange then blue lines, or only blue lines.

There is much more to high-resolution graphics than is presented here. When you feel confident using the high-resolution graphics statements presented in this section, refer to the *Applesoft Reference Manual* for more information on high-resolution graphics.

Chapter Summary

Statements

PEEK
RND
INT
GOSUB
TRACE
NOTRACE
END
HGR
HCOLOR=
HPLOT

Terms

address
function
byte
kilobyte
memory
subroutine
routine
high-resolution graphics

Error Messages

?REENTER
?RETURN WITHOUT GOSUB ERROR

Chapter 5

Strings and Arrays

Strings and Arrays

Computers can manipulate letters and symbols as well as graphics and numbers. In this chapter you will learn how the Apple IIe handles whole *strings* of characters. You'll also discover more about how it stores characters and numbers in what are called *arrays*.

Stringing Along

A string, or bunch of characters, is a type of variable. *String variable* names follow the same rules as numeric variable names except they end with a dollar sign (`$`). Here are some examples of possible string variable names:

```
A$
MYNAME$
SENTENCES$
```

Actually, you have already used a string variable without knowing it. In the `WELCOME` program in Chapter 2, `N$` was used as a string variable to store your first name. Look at that program now. `N$` is a good example of what string variables do: it holds whatever characters (letters) are typed in and uses those same characters in a message to the user. String variables can do other things, too. Read on.

Numeric variables (like `A`) are different from string variables (like `A$`): `A` can only contain a number (someone's age, for example) and `A$` can contain a character string (someone's name, for example). Both can be used in the same program.

String variables can be given any kind of name, so long as the name ends with a dollar sign. If you want a string variable to contain the letters HARRY S TRUMAN, you can use the variable

```
A$ = "HARRY S TRUMAN"
```

or

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

The characters you put into a string variable must be enclosed in quotation marks. The statement

```
PRINT NAME$
```

will print the contents of the variable NAME$: in this case the name of the 33rd president of the United States.

String Functions

There are several Applesoft instructions that manipulate strings. Suppose you want to know the length of a string (how many characters it contains). You can use the length function, LEN, by typing

The string function LEN returns the number of characters in a string (specified in parentheses) between 0 and 255.

```
PRINT LEN ("HARRY S TRUMAN")
```

or you can type the equivalent statement

```
PRINT LEN (NAME$)
```

and Applesoft will display the length of the string. As you see (double-check by counting the characters yourself) the length of the string NAME$ is 14. Remember that the computer counts spaces and punctuation as characters.

The number of characters in a string may range from 0 to 255. If you try to use more than 255 characters in a string you will get the ?SYNTAX ERROR or ?STRING TOO LONG ERROR message. A string with zero characters is called a *null string.* Each time you run a program, all numeric variables are automatically set to the value 0 and all sting variables are set to the null string—until you direct otherwise. Therefore, for each program you write that uses string variables, you must assign the string value within the program.

On some occasions you may want to display only a part of the character string contained in NAME$. To do this there are three handy string functions: LEFT$, RIGHT$, and MID$.

If, for instance, you want to PRINT the first five letters in NAME$, type

The string function LEFT$ returns the specified number of leftmost characters from the string.

```
PRINT LEFT$ (NAME$,5)
```

and

```
HARRY
```

will be displayed on the screen. The reason you might want to do this will become apparent soon. If you type

The string function RIGHT$ returns the specified number of rightmost characters from the string.

```
PRINT RIGHT$ (NAME$,5)
```

```
RUMAN
```

will appear.

Here's a short program that uses the functions LEN and LEFT$.

```
NEW
 90 NAME$ = "HARRY S TRUMAN"
100 FOR N = 1 TO LEN(NAME$)
110 PRINT LEFT$(NAME$,N)
120 NEXT N
```

Run this program.

Now write another program substituting RIGHT$ for LEFT$. The RIGHT$ function is just like the LEFT$ function except that it uses the rightmost characters in the string. What happens when you run it?

The string function MID$ returns the substring specified in parentheses.

If you want to use characters starting from the middle of the string instead of the beginning or end, the MID$ function is what you need. Type

```
PRINT MID$(NAME$,7)
```

and the Apple IIe will respond with

```
S TRUMAN
```

since S is the seventh character in the string. To see how the MID$ function alters the program, edit line 110 to read

```
110 PRINT MID$ (NAME$,N)
```

Do you get what you expect when you run the program?

Suppose you want to display just Y, S, and T from the string called NAME$. To do this it's necessary to add another argument to the MID$ function.

```
PRINT MID$ (NAME$,5,5)
```

The first argument specifies the fifth string character space. This is where the display will begin. The second argument indicates how many character spaces will be displayed. This is interpreted by Applesoft as "find the fifth character space in NAME$ and print five character spaces beginning at the fifth and moving to the right."

The first argument in a MID$ statement specifies the character in the string (not necessarily a letter, since punctuation and spaces are also characters in strings) where MID$ should begin. If only one argument is given, MID$ will return the characters from the one named to the end of the string. The second argument, which is optional, limits the number of characters to be returned.

Now change line 110 again, as follows, and run it.

```
110 PRINT MID$ (NAME$,N,6)
```

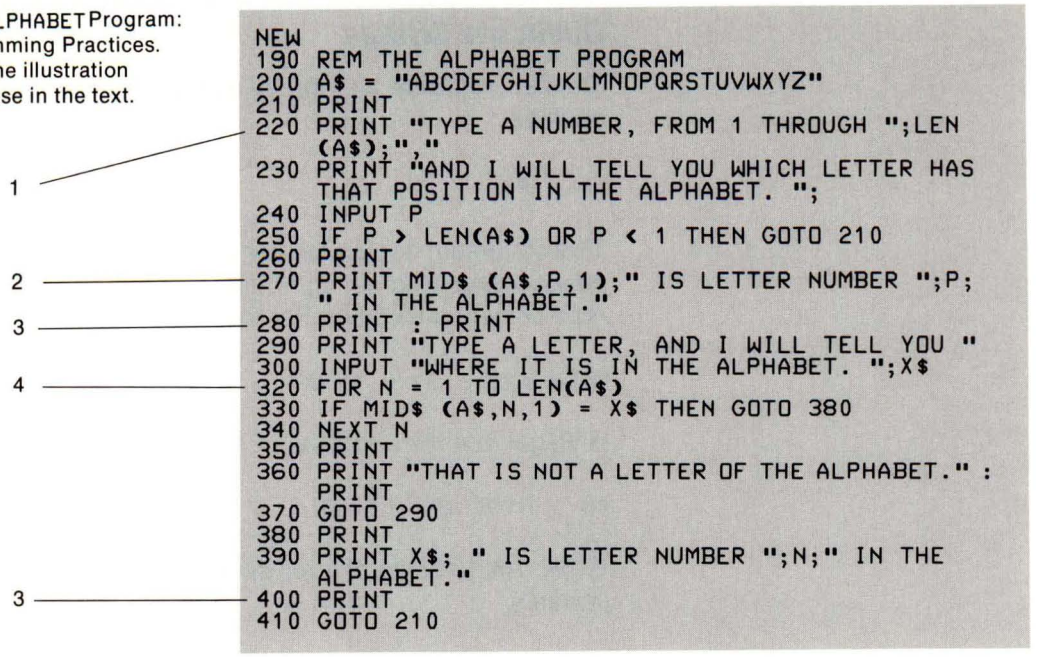
Don't go any further in this book until you've thoroughly tested the LEFT$, RIGHT$, and MID$ functions. Or else!

Common Programming Practices Using Strings

The program ALPHABET on the APPLESOFT SAMPLER disk illustrates some common programming practices using strings. Load and list it now.

The best way to understand how this program works is to study it carefully. Figure out what each variable represents and how each is used. Then look at the programming techniques pointed out in Figure 5-1.

Figure 5-1. The ALPHABET Program: Common Programming Practices. The numbers in the illustration correspond to those in the text.



```
NEW
190 REM THE ALPHABET PROGRAM
200 A$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
210 PRINT
220 PRINT "TYPE A NUMBER, FROM 1 THROUGH ";LEN
    (A$);","
230 PRINT "AND I WILL TELL YOU WHICH LETTER HAS
    THAT POSITION IN THE ALPHABET. ";
240 INPUT P
250 IF P > LEN(A$) OR P < 1 THEN GOTO 210
260 PRINT
270 PRINT MID$ (A$,P,1);" IS LETTER NUMBER ";P;
    " IN THE ALPHABET."
280 PRINT : PRINT
290 PRINT "TYPE A LETTER, AND I WILL TELL YOU "
300 INPUT "WHERE IT IS IN THE ALPHABET. ";X$
320 FOR N = 1 TO LEN(A$)
330 IF MID$ (A$,N,1) = X$ THEN GOTO 380
340 NEXT N
350 PRINT
360 PRINT "THAT IS NOT A LETTER OF THE ALPHABET." :
    PRINT
370 GOTO 290
380 PRINT
390 PRINT X$; " IS LETTER NUMBER ";N;" IN THE
    ALPHABET."
400 PRINT
410 GOTO 210
```

1. LEN(A$) determines the length of the string held in A$—when A$ changes, LEN(A$) will too. This gives the program more flexibility and allows you to specify a different alphabet without having to change the rest of the program.

2. The MID$ statement in line 270 is interpreted as "begin at the first character in A$, go to P (the number entered by the user), and display one character." This is how the program identifies the letter in the alphabet that has the position specified by P.

3. Notice the function of the blank spaces in the PRINT statements. What would happen to the display without these blank spaces? (If you aren't sure, try removing a few of them and see what happens when you run the program.)

4. The program uses a loop to find 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.

Run ALPHABET now. Then try changing line 200, and run it again. Feel free to modify this program and change it in any way you like.

Duplicate Strings

You can duplicate a string by using a replacement statement such as

```
X$ = A$
```

This statement copies the contents of `A$` into `X$`. However, you cannot use partial string notation on the left side of a replacement statement. For example, the statement

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

is illegal, but the statement

```
X$ = MID$ (A$,24,3)
```

is OK. The left side of a replacement statement must be a variable.

Backward Spelling

Would you like the Apple IIe to spell your name backward? Well, here's your big chance! This program will do just that.

```
NEW
100 REM PROGRAM TO SPELL YOUR NAME BACKWARD
110 INPUT "TYPE YOUR NAME AND I WILL SHOW IT
    TO YOU SPELLED BACKWARD.  ";N$
120 REM REVERSE ORDER OF LETTERS
130 FOR T = LEN(N$) TO 1 STEP -1
140 R$ = R$ + (MID$ (N$,T,1))
150 NEXT T
160 PRINT : PRINT "YOUR NAME SPELLED BACKWARD
    IS ";R$
170 PRINT : PRINT
180 GOTO 110
```

Run this program, trying several different names. After the program executes itself a few times you will notice that there is something wrong. Line 140 is the key to the problem. If, for instance, you input SALLY as your name, the variable N$ becomes SALLY and the variable R$ becomes YLLAS. Try it. When the program returns to line 110 and asks your name again, type in JOE. This will set N$ to JOE. But the old variable R$ is still in memory, so line 140 sets R$ to the old R$ plus N$ spelled backward. Instead of getting EOJ (JOE spelled backward) displayed, you'll get YLLASEOJ.

What is needed is some way to reset string variables to zero, so R$ can be refilled after each GOTO. Fortunately this can be done in Applesoft. The null string can be used to empty the contents of R$. Add this line to the program:

```
175 R$ = "" "
```

Now run the program again. By using the null string near the end of the program, the contents of R$ can be set to zero characters after the backward name has been displayed. Then, when the program goes back to line 110 and starts over, R$ will be set to the new name rather than filling up with and displaying all previous names.

CLEAR sets all variables, including strings and arrays, to zero. It is very tricky to use within programs.

There is an Applesoft statement, CLEAR, that resets all variables of every size and shape, but it should be used in immediate execution. Type

```
N = 254
PRINT N
```

Now type

```
CLEAR
```

and then

```
PRINT N
```

Does your computer give 0 as the value of N?

Concatenation Got Your Tongue?

It is possible to add a second string onto the end of an existing string using the plus (+) sign. This process is called *concatenation*. Try the following:

```
C$ = "GOOD MORNING"
D$ = C$ + ", " + "BILL"
PRINT D$
```

The Apple IIe will respond with

```
GOOD MORNING, BILL
```

Concatenation is especially useful if you wish to take a string apart and then put it back together with slight modifications. For instance, to create a new string containing the same characters as in `D$`, but in a different order, type

```
E$ = RIGHT$ (D$,4) + MID$ (D$,13,2) + LEFT$ (D$,12)
PRINT E$
```

and

```
BILL, GOOD MORNING
```

will appear on your screen. Now here is a program that puts concatenation to good use:

```
NEW
100 INPUT "GIVE ME ABOUT HALF OF A SENTENCE. "; HALF$
105 PRINT
110 INPUT "NOW GIVE ME THE SECOND HALF OF THE
    SENTENCE. "; OTHERHALF$
120 WHOLE$ = HALF$ + " " + OTHERHALF$
130 PRINT
140 PRINT WHOLE$
150 PRINT : PRINT : PRINT : GOTO 100
```

And that's how you do concatenation. Try removing the space in line 120. What happens to the sentence halves?

Appendix E presents a complex version of a sentence scrambling program and gives lots of help with program display.

More String Functions

Strings can be made up of almost any kind of character, including numbers. However, the characters between the quotation marks in a string cannot be interpreted arithmetically—even if they are numbers. Type

```
C$ = "123"
PRINT C$ + 7
```

The Apple IIe will give you a ?TYPE MISMATCH ERROR. However, the VAL function (short for "value") can alleviate this problem. The VAL function returns the value of the contents of a string as opposed to its actual contents. Type

The VAL function attempts to interpret a string, up to the first nonnumeric character, as a real number or an integer, and returns the value of that number.

```
PRINT C$
```

and then type

```
PRINT VAL(C$)
```

Both statements apparently produce the same result; however, appearances can be deceiving. You already know that if you type

```
PRINT C$ + 5
```

your computer will respond with ?TYPE MISMATCH ERROR. Try typing

```
PRINT VAL(C$) + 5
```

and

```
PRINT VAL(C$) + 5
]128
```

appears on the screen. Notice that the string variable name, which is the argument of the VAL function, must be in parentheses.

What if you want to put the value of C$ - 21 into an ordinary (nonstring) variable? Simple. Just type

```
Q = VAL(C$) - 21
```

Now type

```
PRINT Q
```

and see what you get. Are the contents of Q what you expect? You can even use VAL to add the numeric values of two different strings. To try this, create a new string

```
K$ = "12"
```

and then type

```
P = VAL(C$) + VAL(K$)
PRINT P
```

Try the VAL function with different strings, including strings that begin or end with letters.

The string function STR$ returns a string that represents the value of the argument.

Sometimes it is necessary to change a number into a string. The STR$ function, which works much like the VAL function in reverse, can be used to make this change. Suppose you want to change the numeric variable P to a string variable. Type

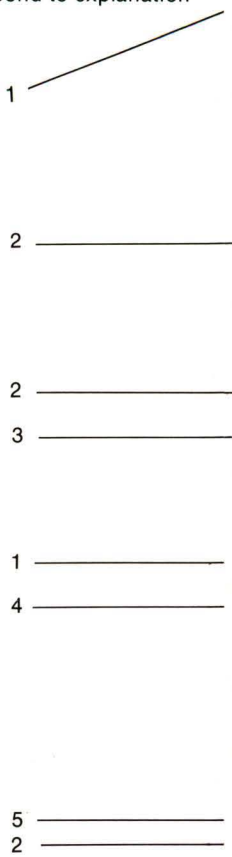
```
P$ = STR$(P)
PRINT P$
```

to see how STR$ works.

STR$ is especially useful for *formatting* text on the screen. You could use STR$, for example, if you wanted to line up all the decimal points in a list of numbers. Your program would convert each number to a string and use the LEN and MID$ functions and HTAB to line each entry up.

The program DECIMAL on the APPLESOFT SAMPLER does just that. Run it, and then look at your listing and Figure 5-2 to see how the lines work. This program has lots of remarks to help you understand the functions of particular statements.

Figure 5-2. The DECIMAL Program: Using STR$. The numbers in the illustration correspond to explanation in the text.



```
  5 REM THE DECIMAL PROGRAM
 10 GOTO 1000
100 REM * PRINT ALIGNED NUMBERS *
110 REM P$ MUST CONTAIN NUMBER
120 REM VTAB MUST BE PRE-DONE
130 REM FIND DECIMAL POINT:
135 LET DP = LEN (P$) + 1: REM SET ALIGNMENT
    BASED ON NO DP
140 FOR CHAR = 1 TO LEN (P$)
150 IF MID$ (P$, CHAR, 1) = "." THEN DP = CHAR:
    REM IF DP FOUND, USE IT
160 NEXT CHAR
165 REM LINE UP DP AND PRINT:
170 CALL -868: REM CLEAR LINE
180 HTAB 15 - DP: PRINT P$
190 RETURN
300 REM * INPUT NUMBER *
310 VTAB 21 : HTAB 1
320 PRINT "WHAT DECIMAL NUMBER"
330 PRINT "WOULD YOU LIKE TO ADD? ";
332 CALL -958: REM CLEAR FROM HERE TO BOTTOM
    OF SCREEN
335 INPUT ""; N$: REM "" KILLS QUESTION MARK
340 N = VAL (N$)
350 IF N > 999999 OR N < .01 THEN GOTO 310:
    REM POSITIVE NUMBERS ONLY
360 N$ = STR$ (N)
370 RETURN
1000 REM * MAIN ROUTINE *
1010 TEXT : HOME
1030 GOSUB 300 : REM GET NUMBER
1100 REM MOVE DOWN AND PRINT N$
1110 ROW = ROW + 1: VTAB ROW
1120 P$ = N$ : GOSUB 100
1200 REM ADD NEW NUMBER TO SUM:
1210 SUM = SUM + N
1300 REM PRINT TOTAL:
1310 P$ = STR$ (SUM)
1320 PRINT "TOTAL: ";: GOSUB 100
1400 REM REPEAT UNTIL FULL:
1410 IF ROW < 19 THEN 1030
1420 CALL -958: END
```

1. Asterisks are used in REM statements for visual identification. They are generally used to mark the beginning of subroutines and main routines, as in lines 100 and 1000.

2. CALL causes execution of machine-language subroutines that do magic things on the screen: CALL-868 clears the current line; CALL-958 clears to the bottom of the screen. It is advisable to use these CALL statements in conjunction with INPUT statements. For more information, see Appendix E and the *Applesoft Reference Manual.*

3. The null string (" ") in line 335 keeps the INPUT statement from adding an extra question mark.

4. Note that the main routine of this program begins at line 1000; at line 1030, GOSUB 300 causes it to branch back to the INPUT statement. At 370, the program returns to line 1100.

5. The screen will only show 19 rows of numbers without scrolling, so the variable ROW is conditional in line 1410.

Lines 335–360 demonstrate the first steps toward making a truly error-proof input routine. Try entering all different kinds and forms of numbers to see what will stop this program. Then figure out ways to catch those kinds of errors so they won't cause the program to stop.

Trapping More Errors

As you are finding out, there are many potential pitfalls a programmer has to be aware of and plan for in writing programs. It is no fun for the user to find himself stuck in the middle of a program with no idea of how to get out, and it is no fun for the programmer to see that happen.

Now that you know how VAL and STR$ can be used if the character entered isn't a positive number (lines 335-360 in DECIMAL), it would be helpful to see how to add these functions to a program you have already worked on. Load your most recent version of COLORBOUNCE, and type

```
LIST 350-400
```

You should see these lines on your screen:

```
350 INPUT "WHAT COLOR WOULD YOU LIKE THE BALL TO BE
    (1-15)? ";HUE
370 REM IS HUE OF BALL IN RANGE?
380 IF HUE > 0 AND HUE < 16 THEN GOTO 400
390 HOME : PRINT "THAT WASN'T BETWEEN 1 AND
    15!": PRINT
395 GOTO 310
400 GR : REM SET COLOR GRAPHICS AREA
```

Line 380 checks that the number entered is within the correct range, and lines 390 and 395 give the user a message and another chance to enter a number within range. However, the program doesn't cover the possibility that a user might enter ONE instead of 1. As you may recall from "Program Interaction with Users," Chapter 4, such an entry would be rewarded with the ?REENTER error message, but no indication of what was wrong.

Now you can fix the program to take care of just such a possibility. Follow these steps:

1. Change the variable HUE in line 350 to a string variable, HUE$, so that if a user happens to spell out a number, Applesoft will be able to do something with it (remember, a numeric variable cannot contain a string of characters).
2. Add these lines to the program:

```
352 REM IS ENTRY A NUMBER?
354 HUE = VAL(HUE$) : REM VALUE OF HUE$ BECOMES
    NUMERIC VARIABLE
356 IF HUE > 999999 OR HUE < .01 THEN GOTO 300 : REM
    INPUT MESSAGE WILL REAPPEAR
358 HUE$ = STR$(HUE) : REM HUE CHANGED BACK INTO
    STRING HUE$
```

Now run the new version and see what happens when you spell out an entry. The COLORBOUNCESOUND program on the APPLESOFT SAMPLER contains these and all the other changes you have made to COLORBOUNCE.

Introducing Arrays

An array is a type of variable that is used to represent lists of values linked together in some logical pattern. You can think of an array as a table of numbers from which you can select pieces, or elements. (See Figure 5-3.) The programming power they give you more than compensates for the time spent becoming familiar with them.

An array name can be any legal variable name, but it always ends with a set of parentheses. Within the parentheses are the elements of the array.

When the DIM statement is executed, it sets aside space for an array containing the specified number of elements.

To create an array you must first tell the computer the maximum number of elements you want the array to accommodate. To do this use the DIM statement (DIM stands for "dimension"). The elements in an array are numbered from zero, so to dimension an array called A that will have a maximum of 16 elements, type

```
DIM A(15)
```

This DIM statement creates a one-dimensional array variable with space for 16 elements. The elements behave exactly like the variables you have come to know and love. They are:

```
A(0)
A(1)
A(2)
```

and so on, up to

```
A(15)
```

The elements of an array can be used just as any other variable is used. For example, you can take two elements of array A and write

```
A(9) = 45 + A(12)
```

The elements of the array (the numbers in parentheses) are called *subscripts.* The subscript can be an arithmetic expression or it can be represented by a variable.

The following program illustrates the use of variables in the subscript and displays the contents of each array element. Type

```
 90 REM DIMENSION ARRAY CALLED DAYS
100 REM HOLDS 7 NUMBERS
110 DIM DAYS(6)
120 REM FILL THE ARRAY
130 FOR NUM = 0 TO 6
140 DAYS(NUM) = NUM + 1
150 NEXT NUM
160 REM DISPLAY THE ARRAY ELEMENTS
170 FOR I = 0 TO 6
180 PRINT "DAYS(";I;") = ";DAYS(I)
190 NEXT I
```

If an array is used in a program before it has been dimensioned, Applesoft automatically reserves space for 11 elements (subscripts 0 through 10). However, it is good programming practice to dimension all arrays.

Arrays can have one dimension, as in the arrays `A(15)` and `DAYS(6)`, or more than one dimension. The statement

```
DIM TWO (3,7)
```

sets up a two-dimensional array. The first dimension has four elements (`0, 1, 2, 3`), and the second dimension has eight elements (`0, 1, 2, 3, 4, 5, 6, 7`). The `TWO` array has space for a total of 32 separate elements (4 times 8). Figure 5-3 illustrates the concept of dimensions.

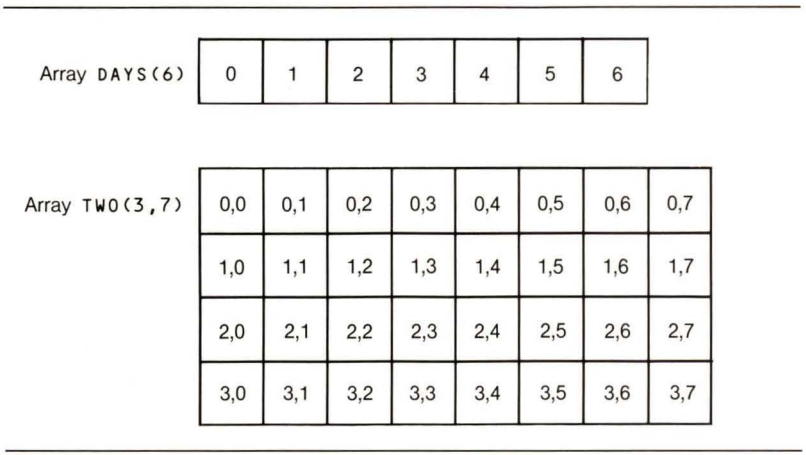


Figure 5-3. Array Dimensions. Array `DAYS(6)` has one dimension and seven elements; array `TWO(3,7)` has two dimensions and 32 elements.

Suppose you want to write a program that scrambles the numbers from one to eight. To accomplish this you need to manipulate tables of data. This is just the kind of thing arrays are good for. The following program accomplishes this.

```
NEW
200 REM DIMENSION THE ARRAY
210 DIM GLASS(8)
220 REM FILL THE ARRAY
230 FOR I = 1 TO 8
240 GLASS(I) = I
250 NEXT I
260 REM SCRAMBLE THE ARRAY AND CHOOSE EACH ELEMENT
270 FOR WINE = 1 TO 8
280 REM CHOOSE SOME OTHER ELEMENT
290 MILK = INT (RND(1) * 8) + 1
300 REM WAS MILK DIFFERENT FROM WINE?
310 REM IF NOT, TRY AGAIN
320 IF MILK = WINE THEN GOTO 280
330 REM INTERCHANGE GLASS(WINE) AND GLASS(MILK)
340 TEMP = GLASS(WINE) : GLASS(WINE) = GLASS(MILK) :
    GLASS(MILK) = TEMP
350 NEXT WINE
360 REM PRINT CONTENTS OF ARRAY
370 FOR C = 1 TO 8
380 PRINT GLASS(C)
390 NEXT C
```

Run the program. Do you understand how it works? The program requires that no element of the array be what it was originally, so it first fills an array with numbers and then scrambles the contents of the array. Notice that you don't have to start filling the array at zero.

Here's a description of what some of the more complex program lines do. Lines 230 through 250 fill the array and assign each array element a number corresponding to its array number (GLASS(1) = 1, GLASS(2) =2, etc.). Line 270 sets the variable WINE to numbers 1 through 8. Line 290 sets variable MILK to random integers from 1 to 8. Then line 320 makes sure that the value of variable WINE is not equal to the value of variable MILK at any given time. The contents of variables GLASS(WINE) and GLASS(MILK) are switched in line 340. Finally the array is printed with lines 370 through 390.

Array Error Messages

Here are a few error messages you might generate while programming with arrays.

- ?REDIM'D ARRAY

  This error message occurs when an array is dimensioned more than once in the same program. For example, the lines

  ```
  10 A(9) = 15
  20 DIM A(50)
  ```

  would produce such an error message. Often, however, this error occurs because the default dimension is used and then later a dimension statement is added to the program.

- ?BAD SUBSCRIPT ERROR

  If an attempt is made to use an array element that is outside the dimension of the array, this error message occurs. For instance, if A has been dimensioned to 25 with the statement DIM A(25), referring to the element A(52) or any other element whose subscript is less than 0 or greater than 25 will result in the ?BAD SUBSCRIPT ERROR.

- ?ILLEGAL QUANTITY ERROR

  You will get this message if you try to use a negative number as an array subscript.

There are several programs in Appendix E that use arrays. Going on to those programs now will give you a broader sense of what can be done with arrays.

Chapter Summary

Statements

LEN
LEFT$
MID$
RIGHT$
CLEAR
VAL
STR$
DIM

Terms

string
array
string variable
numeric variable
null string
concatenation
format
element
subscript
application programs

Error Messages

```
?STRING TOO LONG ERROR
?TYPE MISMATCH ERROR
?REDIM'D ARRAY
?BAD SUBSCRIPT ERROR
```

Conclusion

Now that you have been introduced to some of the programming tools of Applesoft, you have several options.

If you feel you have learned enough programming for now, you will probably want to explore application programs: those written by other people, available for purchase, that let you use the Apple IIe for all kinds of practical purposes. The *Apple IIe Owner's Manual* has a section on application programs.

If you are ready and eager for more programming challenges, here are some suggestions.

- If you go through this book again, writing your own programs with the statements that have been presented, you will solidify your knowledge considerably.
- Appendix E, "More Programs To Play With," presents four new programs. They provide examples of some of the practical things you can do with Applesoft and are designed to help you build your programming skill and understanding. Experimenting with and studying the programs in this appendix will help you learn good programming practices and will give you lots of ideas about writing your own programs.
- Use the *Applesoft Reference Manual* to learn about statements as you need them. You can also add to your programming skills by trying out the examples in that manual.

One of the pleasures of owning a computer is developing your own ideas into programs that you and others can use. This book has presented the core of Applesoft BASIC. Applesoft has many more capabilities, and once you have mastered those presented here, there are whole new worlds for you to explore!

Appendices

E

Summary of Statements and Commands

This appendix is a summary of the Applesoft BASIC statements and DOS commands used in this manual. The number and/or letter in square brackets at the end of each description refers to the chapter and/or appendix where more detailed information about the statement can be found.

You have been introduced to about half of all the Applesoft BASIC statements available on the Apple IIe. The *Applesoft BASIC Programmer's Reference Manual* presents the remainder and provides a comprehensive discussion of all Applesoft statements.

Included in this appendix are some statements that were not introduced in the body of this manual. They are used in Appendix E, "More Programs To Play With."

ASC

The `ASC` function returns the decimal ASCII code for the first character of the argument. [E]

CALL

`CALL` causes execution of a machine-language subroutine at the memory location whose decimal address is specified. For example, `CALL -868` will clear the current line from the cursor to the right margin. It is possible to obtain the same result using an ESC or CONTROL sequence. [5, E]

CATALOG

This Disk Operating System (DOS) command displays a list of all the files on a disk in the specified disk drive. For example, `CATALOG,D2` instructs the operating system to display the files on whatever disk is in the second disk drive. Drive 1 (`D1`) is used by default unless another drive is specified, as in the example.

The file type and the number of disk sectors occupied by the file are indicated to the left of the file name in the CATALOG listing. The file types are represented by letters:

- I means the file is a program written in Integer BASIC.
- A means the file is a program written in Applesoft BASIC. (All the programs you have saved on a disk while using this manual are this file type.)
- T means the file contains text and was created by a WRITE command.
- B means the file is stored in binary form, also known as machine language.

[2]

CHR$

The CHR$ function returns the ASCII character that corresponds to the value of the argument, which must be between 0 and 255. For example, CHR$(65) returns the letter A. [E]

CLEAR

This Applesoft statement sets all variables, including arrays and strings, to zero. It should be used in immediate execution. Because it throws away the values of all variables, it is tricky to use within a program. [5]

COLOR=

The color for plotting in low-resolution graphics is set with the statement COLOR= followed by an integer from 0 to 15, as in COLOR = 5. Color is set to zero by the GR statement, so GR must always be followed by the COLOR= statement for anything to appear on the display screen. Color names and their associated numbers are:

0	black	4	dark green	8	brown	12	green
1	magenta	5	gray	9	orange	13	yellow
2	dark blue	6	medium blue	10	gray	14	aqua
3	purple	7	light blue	11	pink	15	white

On black-and-white or green-phosphor screens the 16 colors appear as four shades of gray, as shown:

Dark gray: 1, 2, 4, 8
Medium gray: 5, 10
Light gray: 3, 6, 9, 12
Pale gray: 7, 11, 13, 14
White: 15

[1]

CONT

The CONT statement causes program execution to resume, or continue, after CONTROL-C, STOP, or END is used to halt execution. Execution resumes at the next instruction (like GOSUB)—not the next line number. Variables are not cleared.

If you modify, add, or delete any program line or get an error message after stopping execution, CONT won't work.

If there is no halted program, CONT has no effect. [2]

DATA

The DATA statement creates a list of elements that can be used by READ statements. The elements can be constants, strings, real numbers, integers, or a combination, as in the example

```
DATA SMITH, "TRUMAN", 3.17, -6
```

[E]

DEL

The DEL statement removes, or deletes, the specified range of lines from the program and is written DEL 23,56. Other syntax will be followed by a ?SYNTAX ERROR. To delete a single line, say line 35, use the form DEL 35,35 or type the line number and then press the RETURN key. [3]

DELETE

This DOS command removes the program specified by name from a disk. Like other DOS commands, it can be followed by a disk drive number if, for example, the disk containing the program is not in the default drive. `DELETE BOUNCE,D2` would remove the program named `BOUNCE` from the disk in Drive 2, unless that disk is write-protected or the file is locked. [3]

Note: The DELETE key on the keyboard is not the same as the `DELETE` command. See the *Apple IIe Owner's Manual* and the *Apple IIe Reference Manual* for more information.

DIM

When the `DIM` statement is executed, it sets aside space for an array containing the specified number of elements. The elements in an array, called subscripts, are numbered from zero. `DIM A(50)` will dimension, or set aside space for, the array `A` containing up to 51 elements. `DIM N$(25)` will allot 26 strings of any length to the array `N$`.

If an array element is used in a program before it is dimensioned, Applesoft automatically reserves space for 11 elements (subscripts 0 through 10). Array elements are set to zero when `RUN` or `CLEAR` is executed. [5]

END

The `END` statement stops a program and returns control to the user. No message is printed. [4]

FOR

A `FOR` statement in combination with a `NEXT` statement sets up a program loop. The loop operation is carried out the number of times specified with the `TO` portion of the statement. The use of `STEP` is optional.

In the statement `FOR W = 1 TO 20...NEXT W`, the variable `W` counts how many times to do the instructions. The instructions inside the loop will be executed 20 times, for `W` equal to `1`, `2`, `3`, up to `20`. The loop ends with `W = 21`, and the instruction after `NEXT W` is then executed.

The statement FOR Q = 2 TO -3 STEP -2...NEXT Q illustrates how to use STEP to count in regular increments other than one.

Checking takes place at the end of a loop, so in the example FOR Z = 5 TO 4 STEP 3...NEXT Z the instructions inside the loop are executed once.

A ?NEXT WITHOUT FOR ERROR appears if you try to run a program with crossed loops. [2]

GOSUB

GOSUB causes the program to branch to the line number given. The subroutine beginning at that line number should end with a RETURN statement, which causes the program to branch back to the statement immediately after the GOSUB. For example, GOSUB 500 would cause the program to branch to line 500 and continue until RETURN. [4]

GOTO

The GOTO statement causes the program to branch to the indicated line. It is used to create a loop and to run a program without resetting all variables. [2]

GR

The GR statement sets the stage for low-resolution graphics. In the low-resolution graphics mode set by GR, the screen has an invisible grid of 40 vertical columns and 40 horizontal rows numbered 0-39 and space for four lines of text at the bottom. GR clears the screen and sets COLOR= to zero, or black. [1]

HCOLOR =

The color for plotting in high-resolution graphics is determined by the HCOLOR= statement. Color numbers and their associated names are:

0	black1	4	black2
1	green	5	orange
2	violet	6	blue
3	white1	7	white2

[4]

HGR

The high-resolution graphics mode, set by HGR, creates a screen grid of 280 by 160 plotting points and leaves four lines for text at the bottom. The screen is cleared to black, and page 1 of memory is displayed. Neither HCOLOR= nor text screen memory is affected when HGR is executed. The cursor may not be visible unless it is moved into the bottom four lines of the screen (text window) by pressing RETURN. [4]

HLIN

The HLIN statement is used to draw horizontal lines in low-resolution graphics. It uses the most recently specified color. The syntax of HLIN is

```
HLIN 10,30 AT 20
```

This example draws a horizontal line from column 10 to column 30 at row 20. [1]

HOME

Using HOME when you are in text mode will clear all text and move the cursor to the upper-left corner of the screen. Using HOME when you are in one of the graphics modes only clears the four lines available for text at the bottom of the screen. (To clear the screen of graphics, use GR or HGR.) [1]

HPLOT

The HPLOT statement is used to plot dots and lines in high-resolution graphics using the most recently specified value of HCOLOR=. There are three syntax variants; each has a different result:

```
HPLOT 10,20
```

Plots a high-resolution dot at column 10, row 20 on the (invisible) screen grid.

```
HPLOT TO 70,80
```

Plots a line from the last dot plotted to a dot at column 70, row 80, using the color of the last dot plotted (not necessarily the most recent HCOLOR=). If no previous point has been plotted, no line is drawn.

```
HPLOT 30,40 TO 50,60
```

Plots a high-resolution line from the dot at column 30, row 40 to a dot at column 50, row 60.

The plotted line may be extended in the same instruction almost indefinitely. The single statement

```
HPLOT 0,0 TO 279,0 TO 279,159 TO 0,159 TO 0,0
```

plots a rectangular border around all four sides of the high-resolution screen. (If it doesn't work for you, your screen needs adjustment.)

HPLOT must be preceded by HGR to avoid erasing the Apple IIe's memory, including your program and variables. [4]

HTAB

The HTAB statement moves the cursor either left or right to the specified column (1 through 40) on the screen.

HTAB's moves are relative to the left margin of the text window, but independent of the line width. A line has 255 character positions (this is the character limit of any line). Since the screen has a limit of 40 characters per line, HTAB causes the cursor to wrap around to the next screen line for positions 41-80, the next line down for positions 81-120, and so on. [2]

IF

The IF...THEN statement creates a conditional program loop. It is very useful for limiting the range of a program variable. When the condition following the keyword IF is evaluated as false (0), all the rest of that program line is ignored, and the computer goes on to the next line. When the condition is true (1), the statement following the keyword THEN is executed.

```
IF N <= 5 THEN GOTO 100
```

In this example the variable N will be compared to the condition IF N <= 5 and evaluated. Each time the condition is true (when N is 1, 2, 3, or 4, for example), the remainder of the statement will be executed and the program will branch to line 100 (specified by GOTO). When the condition is false (when N is 6 or 10, for example) the GOTO is ignored.

String expressions are evaluated by alphabetic ranking.

A THEN without a corresponding IF or an IF without a corresponding THEN will cause a ?SYNTAX ERROR. [2]

INPUT

The INPUT statement enables you to interact with a program user from within a program. An INPUT statement must name a variable to be entered by the user and may contain a question or statement. In the example

```
INPUT A
```

a question mark is displayed on the screen when the statement is executed, and the program waits for the user to enter a number, which will be assigned to the numeric variable A. In the example

```
INPUT "TYPE AGE THEN A COMMA THEN YOUR NAME "; B, C$
```

the optional string is displayed exactly as shown (if the programmer wanted to include a question mark, she would have to include it within the quotation marks). The program waits for the user to type a number, which is assigned to the variable B, then a comma, then a name, which is assigned to the string variable C$. Multiple entries may be separated by commas or presses of the RETURN key. INPUT cannot be used in immediate execution. [2]

INT

The INT function returns the largest integer less than or equal to the given argument. In the example INT (NUM), if NUM is 2.389, then 2 will be returned; if NUM is -45.123345 then -46 will be returned. [4]

INVERSE

The INVERSE statement sets the video mode so that characters are displayed as black letters on a white background. Use NORMAL to return to white letters on a black background. [1]

LEFT$

The string function LEFT$ returns the specified number of leftmost characters from the string. If you type PRINT LEFT$("APPLESOFT",5) the five leftmost characters, APPLE, will be returned. [5]

LEN

The string function LEN returns the number of characters in a string (specified in parentheses) between 0 and 255. In the example LEN("AN APPLE A DAY"),14 will be returned. In the example LEN(A$), the number returned will be the count of characters in the string A$. [5]

LET

The LET statement is used to define a variable. The variable name to the left of the equal sign, which is used in conjunction with LET, is assigned the value of the string or expression to the right of the equal sign. LET is optional; the statements LET A = 23 and A = 23 give the same result. [1]

LIST

The LIST statement displays the program lines that are in the Apple IIe's memory.

LIST	Displays entire program.
LIST 150	Displays only line 150.
LIST -200	Lists from the start of the program through line 200.
LIST 200-	Lists from line 200 to the end of the program.
LIST 200,3000 or LIST 200-3000	Lists program lines 200 through 3000.

CONTROL-S is used to halt and to resume a listing. Listing is aborted by CONTROL-C, and the CONT command cannot be used. [2]

LOAD

The LOAD command, when followed by a file name, attempts to find the named program file on the disk in the specified or default drive. If the program is found, it is transferred into the computer's memory. LOAD erases any program in the computer before placing the new program in memory.

LOAD is also an instruction used with cassette recorders. When not followed by a file name, LOAD reads an Applesoft program from cassette tape into the computer's memory. No prompt is given: the user must rewind the tape and press the play button on the recorder before loading. A beep is sounded when information is found on the tape being loaded. When loading is successful, a second beep sounds and the Applesoft prompt (]) is displayed. Only pressing the RESET key can interrupt the command. See the *Applesoft Reference Manual* for a list of all instructions used with cassette recorders. [2]

MID$

The string function MID$ returns the substring specified in parentheses. If you type PRINT MID$("AN APPLE A DAY",4), the fourth through the last characters of the string will be returned: APPLE A DAY. In the example MID$("AN APPLE A DAY",4,9), the nine characters beginning with the fourth character in the string will be returned: APPLE A D. [5]

NEW

The NEW statement deletes the current program from memory and sets all variables to zero. [2]

NEXT

NEXT is used within a FOR/NEXT loop. See the FOR statement for explanation. [2]

NORMAL

NORMAL sets the video mode to the usual white letters on a black background. [1]

NOTRACE

The NOTRACE statement turns off TRACE. See TRACE. [4]

ONERR GOTO

ONERR GOTO is used to avoid an error message that halts execution when an error occurs. When executed, ONERR GOTO sets a flag that causes an unconditional jump to the indicated line number if any error is later encountered. [E]

PEEK

The PEEK function returns the contents, in decimal form, of the byte at the specified memory address. In the example PEEK(-16336), the address, or argument, is -16336; this particular address is related to the memory address of the Apple IIe's speaker. When the function is executed, the speaker makes a barely audible click. [4,E]

PLOT

In low-resolution graphics, the PLOT statement places a brick at the specified location. In the example PLOT 10,20, a brick will be placed at column 10, row 20. The color of the brick is determined by the most recent value of COLOR=, which is black if not specified. [1]

POKE

POKE stores the binary equivalent of the second argument (3 in the example below) in the memory location whose decimal address is given by the first argument (34 in the example). POKE statements are useful for doing things like switching the graphics/text window mix and for controlling the size and scrolling of the text window. In the example POKE 34,3, the top margin of the display is set three lines down from the top and text only scrolls up to that margin. Another example is POKE 33,33, which narrows the width of the text window. [E]

PRINT

PRINT is the primary Applesoft statement used to display information on the screen. The PRINT statement can display a number, as in PRINT 150; it can display the contents of a variable, as in PRINT N; it can display a group of characters contained in quotation marks, as in PRINT "HELLO THERE"; and it can display a blank line, as in PRINT, which causes a line feed and RETURN to be executed.

To display a list of items without any intervening spaces between them, use semicolons in the PRINT statement. To display a list of items in separate tab fields, use commas. [1]

PR#

PR# is a DOS command that sends information to a specified slot, 1 through 7. To send information to slot 1, where a printer is customarily connected, you type PR#1. PR#0 returns display to the screen, although it is not the best method to use with some peripheral cards. [E]

READ

When a program executes a READ statement it looks for a DATA statement. It uses the first element in the DATA statement as a variable in the READ statement. Successive DATA elements are assigned to successive variables in the READ statement each time READ is executed. In the example READ A, B, C$, the first two elements in the DATA statement must be numbers and the third element must be a string. The elements will be assigned, respectively, to the variables A, B, and C$. [E]

REM

REM is a statement that allows you to put remarks, or commentary, in a program. REM statements are not displayed or executed in a program; they are used by programmers to explain program lines. [2]

RETURN

The RETURN statement is used at the end of subroutines. It causes the program to branch to the statement immediately after the most recently executed GOSUB. [4]

RIGHT$

The string function RIGHT$ returns the specified number of rightmost characters from the string. If you type PRINT RIGHT$("SCRAPPLE",5), APPLE (the five rightmost characters) will be returned. [5]

RND

The arithmetic function RND returns a random real number greater than or equal to zero and less than one. Every time RND is used with any positive argument a new random number from zero to one is generated, unless it is part of a sequence of random numbers initiated by a negative argument. RND(0) returns the most recently generated random number. [4, E]

RUN

The RUN statement clears all variables and begins execution at the indicated line number.

`RUN`	Executes entire program, beginning at lowest line number.
`RUN 130`	Begins execution at line 130 (or whatever line is specified) and continues to end of program.

RUN followed by a file name is a DOS command. It loads the named file from the specified or default drive and then runs the program that has been loaded. [2]

SAVE

SAVE, when followed by a file name, is a DOS command that stores the program currently in memory. If the command SAVE AGE is given, and no file called AGE is found on the disk in the specified or default drive, a file is created on that disk, and the program currently in memory is stored under the given file name. If the disk already contains a file in the same language with the specified file name, the original file's contents are lost and the current program is saved in its place. No warning is given.

SAVE used without a file name stores the program currently in memory on cassette tape. No prompt or signal is given: the user must press the record and play buttons on the recorder before SAVE is executed. SAVE does not check that the proper recorder buttons are pushed; beeps signal the start and end of a recording. See the *Applesoft Reference Manual* for a list of all instructions for cassette recorders. [2]

STR$

The string function STR$ returns a string that represents the value of the argument. In the example `STR$(12.45),12.45` is returned. [5]

TAB

The display function TAB, which must be used in a PRINT statement, moves the cursor through tab fields on the screen. Its arguments must be between 0 and 255 and enclosed in parentheses.

If the argument is greater than the value of the current cursor position, TAB moves the cursor to the specified printing position, counting from the left edge of the current cursor line. If the argument is less than the value of the current cursor position, the cursor is not moved. [2]

TEXT

The TEXT statement sets the screen to the usual nongraphics text mode, with 40 characters per line and 24 lines. When used to leave the graphics mode, it is best used in conjunction with the HOME statement. It resets the text window to full screen. [1]

TRACE

The debugging statement TRACE causes the line number of each statement to be displayed on the screen as it is executed. TRACE is not turned off by RUN, CLEAR, NEW, DEL, or RESET. NOTRACE turns off TRACE. [4]

VAL

The VAL function attempts to interpret a string, up to the first nonnumeric character, as a real number or an integer and returns the value of that number. If no number occurs before the first nonnumeric character, zero is returned. [5]

VLIN

In low-resolution graphics, VLIN draws a vertical line in the color indicated by the most recent COLOR= statement. The line is drawn in the column indicated by the third argument. In the example VLIN 10,20 AT 30, the line is drawn from row 10 to row 20 at column 30. [1]

VTAB

VTAB moves the cursor to the specified vertical row on the display screen. The top row is row 1; the bottom row is row 24. VTAB will move the cursor up or down, but not left or right. [2]

Reserved Words in Applesoft

The following list contains all of the reserved words in Applesoft BASIC. They are reserved for use as keywords in Applesoft statements: when Applesoft sees them in a program, it tries to execute them and only understands their meaning within the language. In most cases these reserved words cannot be used as variable names. The comments at the end of the list note the exceptions. See the *Applesoft BASIC Programmer's Reference Manual* for an explanation of the statements not discussed in this manual.

&	GET	NEW	SAVE
	GOSUB	NEXT	SCALE=
ABS	GOTO	NORMAL	SCRN(
AND	GR	NOT	SGN
ASC		NOTRACE	SHLOAD
AT	HCOLOR=		SIN
ATN	HGR	ON	SPC(
	HGR2	ONERR	SPEED=
CALL	HIMEM:	OR	SQR
CHR$	HLIN		STEP
CLEAR	HOME	PDL	STOP
COLOR=	HPLOT	PEEK	STORE
CONT	HTAB	PLOT	STR$
COS		POKE	
	IF	POP	TAB(
DATA	IN#	POS	TAN
DEF	INPUT	PRINT	TEXT
DEL	INT	PR#	THEN
DIM	INVERSE		TO
DRAW		READ	TRACE
	LEFT$	RECALL	
END	LEN	REM	USR
EXP	LET	RESTORE	
	LIST	RESUME	VAL
FLASH	LOAD	RETURN	VLIN
FN	LOG	RIGHT$	VTAB
FOR	LOMEM:	RND	
FRE		ROT=	WAIT
	MID$	RUN	
			XPLOT
			XDRAW

Applesoft "tokenizes" these reserved words: each word takes up only one byte of storage space. Usually, one character takes up one byte.

- The ampersand (&) is intended for the computer's internal use only; it is not a proper Applesoft statement. This symbol, when executed as an instruction, causes an unconditional jump to location $3F5.
- XPLOT is a reserved word that does not correspond to a current Applesoft statement.

Some reserved words are recognized by Applesoft only in certain contexts:

- COLOR=, HCOLOR=, SCALE=, SPEED=, and ROT= are interpreted as reserved words only if the next nonspace character is the replacement sign (=). In the case of COLOR= and HCOLOR=, this is of little benefit because the included reserved word OR prevents their use in variable names anyway.

  When you attempt to execute a statement like 10 COLORFUL = 5, a ?SYNTAX ERROR results. When you attempt to list the same statement, it will be broken down as

```
10 COL OR FUL = 5
```

- SCRN, SPC, and TAB are recognized as reserved words only if the next nonspace character is a left parenthesis [(].
- HIMEM must have a colon (:) to be recognized as a reserved word.
- LOMEM also requires a colon (:) to be recognized.
- ATN is recognized as a reserved word only if there is no space between the T and the N. If a space occurs between the T and the N, the reserved word AT is recognized, instead of ATN.
- TO is interpreted as a reserved word unless preceded by an A with a space between the T and the O. If a space is between the T and the O, the reserved word AT is recognized instead of TO.

Sometimes parentheses can be used to get around reserved words:

```
100 FOR A = LOFT OR CAT TO 15
```

lists as

```
100 FOR A = LOF TO RC AT TO 15
```

Whereas, `100 FOR A = (LOFT) OR (CAT) TO 15` lists as

```
100 FOR A = (LOFT) OR (C AT ) TO 15
```

Error Messages

All of Applesoft BASIC's error messages are listed and described in this appendix. The *Applesoft BASIC Programmer's Reference Manual* has more information about trapping errors and debugging programs.

After an error occurs, Applesoft BASIC returns to command level as indicated by the] prompt character and a cursor. Variable values and the program text remain intact, but the program cannot be continued with CONT and all GOSUB and FOR loop counters are set to zero.

The format for error messages:

- Errors in immediate execution statements display as
  ?XX ERROR.
- Errors in deferred execution statements display as
  ?XX ERROR IN YY.

In both formats XX is the name of the specific error. In deferred execution YY is the line number of the statement where the error occurred. Errors in a deferred execution statement are not detected until that statement is executed.

All Applesoft error messages are preceded by a question mark (?). If an error message is displayed on your screen without a question mark, it is either an Applesoft message of the non-error variety (such as BREAK IN 110) or it is a DOS error message; see the *DOS Manual* for more information. An error message preceded by three asterisks (***), as in *** SYNTAX ERR, is an Integer BASIC error message.

Applesoft error messages and their explanations follow.

?BAD SUBSCRIPT ERROR

An attempt was made to reference an array element that is outside the dimensions of the array. This error can occur if the wrong number of dimensions is used in an array reference: for instance, `LET A(11) = Z` when `A` has been dimensioned using `DIM A(2)`.

?CAN'T CONTINUE ERROR

Attempt to continue a program when none existed; or after an error occurred; or after a line was changed, deleted from, or added to a program.

?DIVISION BY ZERO ERROR

Dividing by zero generates this message.

?FORMULA TOO COMPLEX ERROR

More than two statements of the form `IF...THEN` were executed.

?ILLEGAL DIRECT ERROR

You cannot use an `INPUT`, `DEF FN`, `GET`, or `DATA` statement as an immediate execution command.

?ILLEGAL QUANTITY ERROR

The parameter passed to a built-in arithmetic or string function was out of range. Illegal quantity errors can occur because of

- a negative array subscript (e.g., `LET A(-1) = 0`)
- using `LOG` with a negative or zero argument
- using `SQR` with a negative argument
- `A ^ B` with `A` negative and `B` not an integer
- use of `MID$`, `LEFT$`, `RIGHT$`, `WAIT`, `PEEK`, `POKE`, `TAB`, `SPC`, `ON...GOTO`, or any of the graphics functions with an improper argument.

?NEXT WITHOUT FOR ERROR

The variable in a NEXT statement did not correspond to the variable in a FOR statement that was still in effect, or a nameless NEXT did correspond to any FOR that was still in effect. The three most common causes for this error are forgetting to type the appropriate FOR or NEXT statement; typing the wrong variable after the NEXT statement; or accidentally branching into the body of a FOR loop.

?OUT OF DATA ERROR

A READ statement was executed, but all of the DATA statements in the program already had been read. Either the program tried to read too much data or insufficient data was included in the program.

?OUT OF MEMORY ERROR

Any of the following can cause this error: program too large; too many variables; FOR loops nested more than 10 levels deep; GOSUBs nested more than 24 levels deep; too complicated an expression; parentheses nested more than 36 levels deep; attempt to set LOMEM: too high; attempt to set LOMEM: lower than present value; attempt to set HIMEM: too low.

?OVERFLOW ERROR

The result of a calculation was too large to be represented in Applesoft's number format. If an underflow occurs, zero is given as the result, and execution continues without any error message being printed.

?REDIM'D ARRAY ERROR

After an array was dimensioned, another dimension statement for the same array was encountered. This error often occurs if an array has been given the default dimension and then is followed later in the program by a DIM statement. This error message can be useful if you want to know on what program line a certain array was dimensioned: just insert a dimension statement for that array in the first line, run the program, and Applesoft will tell you where the original dimension statement is.

?RETURN WITHOUT GOSUB ERROR

A RETURN statement was encountered without a corresponding GOSUB statement.

?STRING TOO LONG ERROR

Attempt was made by use of the concatenation operator (+) to create a string more than 255 characters long. This error tends to occur when a string variable is used more than once without being cleared.

?SYNTAX ERROR

There is a missing parenthesis in an expression, an illegal character in a line, incorrect punctuation, or some other format error. Usually this is caused by a simple typing error.

?TYPE MISMATCH ERROR

The left-hand side of an assignment statement was a numeric variable and the right-hand side was a string, or vice versa; or a function that expected a string argument was given a numeric one, or vice versa. This happens most often because the string sign ($) is left off.

?UNDEF'D STATEMENT ERROR

An attempt was made to GOTO, GOSUB, or THEN to a statement line number that does not exist. Causes include accidentally deleting a line, changing a line number without making corresponding changes in references, and simple typing errors.

?UNDEF'D FUNCTION ERROR

A reference was made to a user-defined function that had never been defined.

Help

This appendix gives you some ideas about what to do if something goes wrong and where to look for more information.

If You (or Your Program) Get Stuck

One day, when you are happily working away, you will look up to discover that nothing is happening on the display screen. When you press the RETURN key, nothing moves. When you try other keys, nothing appears. The cursor may even have disappeared. This state is affectionately referred to as a "hung" system by some people.

There are at least six ways to get the Apple IIe back to normal. The methods are listed in order of increasing severity. It is always better to try steps 1 and 2 first.

1. Press the ESC key. This key is in the upper-left corner of the keyboard. Its full name is escape, and if you're lucky, that's what it will help you do.
2. Press CONTROL-C. Many programs think of CONTROL-C as "cancel."
3. Press CONTROL-C and then RETURN.
4. Hold down the CONTROL key while pressing the RESET key (on the far right of the keyboard). CONTROL-RESET restarts the resident program.
5. Press the OPEN-APPLE key and hold it until you have pressed and let go of CONTROL-RESET. This is a power-on restart and is pretty drastic: whatever is going on is stopped and main memory is cleared. You can find out more about this procedure in the *Apple IIe Reference Manual*.

6. Turn off the power. You will rarely have to go this far, and it is easier on the computer if you use the power-on restart instead of turning the power off and on.

You can use these same methods to exit from a program. Usually a program (if it is well written) will tell you what to do, but there are times when programs don't offer such options.

Errors

In this manual, Applesoft error messages are explained in the text and in Appendix C. More information about Applesoft errors can be found in the *Applesoft Reference Manual*.

Error messages also are given by operating systems, such as DOS and Pascal. Whenever you get an unfamiliar error message, check the operating system's manual for information.

Statements and Commands

Applesoft statements are discussed in full in the *Applesoft Reference Manual*.

Commands used by the Disk Operating System are discussed in the *DOS Manual*.

The Apple IIe has certain input/output (I/O) subroutines built into the firmware. See the *Apple IIe Reference Manual* for more information.

Cassette Recorders

All of the programs in this manual can be saved on cassette tape if you are not using a disk drive. The *Applesoft Reference Manual* explains the use of cassette recorders with the Apple IIe.

More Helpful Information

The information in this section is a bit miscellaneous, but helpful nonetheless.

Printing Applesoft Programs

If you have a printer connected to the Apple IIe, or have access to a printer, you can use these general instructions to print your programs.

1. Load the program.
2. Type `PR#1` if the printer card is connected in slot 1. If the printer card is in a different slot, use the number of that slot with `PR#`.
3. Type `RUN` if you want the results of your program printed.
4. Type `LIST` if you want the listing of your program printed.

That's all there is to it!

The Apple IIe's Memory

The memory in the Apple IIe is used in a surprising number of ways:

1. To store the instructions that make up your program.
2. To store your program's variables, strings, and intermediate and final results.
3. To store information that the computer needs: about the system, about your program, and about where different things are stored in memory.
4. To create the text and low-resolution graphics that normally show on your video monitor.
5. To create the high-resolution graphics that can be shown on your video monitor.

Each of these activities, in general, occupies a different portion of the computer's memory. Information is placed in various memory pigeonholes, called memory *locations*. A block of 1024 memory locations is called *1K* of memory. Each memory location has an identifying *address*, a number that lets the Apple IIe find that location and the item of information stored there. These items of information, which you rarely see in their raw, machine-language form, are called *bytes* of information. Each byte of information occupies one memory location.

The portion of the Apple IIe's memory that is used by a particular activity can be described in terms of the memory locations used, usually specified as a range of memory addresses. If a certain range of memory locations is being used to store your program, for instance, those same memory locations must not be used to create a high-resolution graphics display or your program will be lost.

In Applesoft BASIC, memory addresses and other numbers are expressed in the usual *decimal* form. The computer uses *hexadecimal* numbers. To aid advanced programmers, memory addresses are sometimes given in both forms. Hexadecimal numbers usually are preceded by a dollar sign (`$`).

What the Prompt Character Identifies

One of the functions of the prompt character, besides cuing you for input, is to identify which language the computer is using. Here are the prompt characters you are likely to see:

`*` for the Monitor program

`>` for Apple Integer BASIC

`]` for Applesoft BASIC

By looking at the prompt, you can easily tell (if you forget) which language the computer is using.

More Programs To Play With

This appendix contains annotated listings of four programs. It provides examples of some of the practical uses for Applesoft BASIC and has these purposes:

1. To build programming skill and understanding. The programs focus on good programming practices and help you learn how to use some new techniques and statements.
2. To help a new programmer bridge the gap between the introduction to programming in this manual and the advanced programming concepts presented in the *Applesoft BASIC Programmer's Reference Manual*.
3. To offer useful routines that you can incorporate into your own programs.
4. To continue to make programming fun. The programs discussed here are to be played with, modified, experimented with, and changed.

These programs are on the APPLESOFT SAMPLER disk that you have been using as you worked through this manual. They are, in order of presentation:

- `SCRAMBLER`: a sentence scrambling game that demonstrates clear instructions to users and the `RND` function. It also gives free rein to the imaginative writer.
- `MAGIC MENU`: a surprise program that offers you useful subroutines for displaying instructions, accepting input using the underline cursor, and creating friendly menus —all in a minimum of time.

- `DISK MENU`: a program that introduces the APPLESOFT SAMPLER disk, provides a sample menu, and uses the subroutine package provided in the `MAGIC MENU` program.
- `CONVERTER`: a program skeleton used for measurement conversion that leaves plenty of space for you to add and to experiment with your own conversions while learning how to tailor programs to your needs. The program guides you through this process and helps you to build your own menus.

Before you tinker with the programs on the APPLESOFT SAMPLER, make a backup copy as instructed in the *Apple IIe Owner's Manual*. Then you can run the programs, play with them, and see what they do.

Notes To New Programmers

So far you have learned many Applesoft BASIC statements: the purpose of this appendix is to show you some of the techniques used to put those statements to work effectively. As with writing, painting, and other arts, you can gain much insight by exploring the work of others.

You will learn how to avoid the pitfalls of "write-only code": programs so cryptic even the programmer can't understand them. You will sidestep "spaghetti code": programs with meandering trails of `GOTO` statements. Instead, you will see how professionals block-structure programs so that they and others can comprehend and change what they have written.

It is always a shock to be suddenly confronted with programs that are two, five, or ten pages long after dealing with programs of at most ten lines long. However, you can rest easy: `SCRAMBLER`, the first program, is quite simple. You are already equipped with all the skills you need to understand it. The final three programs are more complex, but the reward for understanding them is far more substantial, as you will see. If you begin to feel overwhelmed, stop. Work a bit on your own and then come back. As you develop more and more questions, you will be able to find more and more answers.

SCRAMBLER

Before exploring this program, run it to see what it feels like. This discussion, and the ones that follow, assume that you have the program in your computer as you are reading.

SCRAMBLER is an easy to understand, block-structured program. Block-structured programs are made up of subroutine blocks, or components, each of which performs a specific task. The main program is in the top-level block (lines 1000-1060), which calls, in turn, the four second-level blocks. The blocks are very much like the components of a stereo system, with each part designed to do its own specific task.

SCRAMBLER has five such tasks:

1. To give instructions.

2. To accept the first halves of sentences.

3. To accept the second halves of sentences.

4. To display the sentences.

5. To end the program.

Each of these tasks has been assigned to a separate subroutine, or block, with one clearly defined entrance. Try typing

```
RUN 1180
```

to see the instruction block by itself.

Each subroutine or block is called by a higher block, using the GOSUB statement, and each block ends with a RETURN statement. (The reason you got the ?RETURN WITHOUT GOSUB ERROR just now is that the top-level block of the program did not call the subroutine—you did directly.) All this probably seems easy —after all, you've learned how to write subroutines with GOSUB and RETURN. The fact is, it is easy. It is stressed here because writing unstructured code (filled with meandering GOTO statements) at first seems even easier. It is not until your first serious program, stretching 80 or 100 lines, that you'll suddenly

discover that you can no longer control the monster you have created. By taking just a few minutes to think through the overall program you are writing, you can avoid becoming ensnarled in spaghetti code. When a block becomes so complex that you no longer can see the big picture, break that block into a series of smaller subroutine blocks.

Analysis of the Program Lines

This, and all the Appendix E programs, have basically the same first three lines. The first (well, the zeroth) line has the name of the program, when it was written, and who was responsible. Including this information in your programs lets you know at a glance what program you are looking at and when you wrote it.

Line `1` sends the program off to the main routine (which is also the top-level block) with `GOTO 1000`. The main routine begins at line `1000` so that there is space to make additions later if you need to.

Line `2` is a complete mini-program. Type

```
RUN 2
```

then

```
LIST 1350
```

to see what it does. By scrunching up the lines on one side of the screen, this program forces Applesoft to abandon sticking a bunch of extra spaces in the middle of the `PRINT` statements: you can edit `PRINT` and `REM` statements with ease. (The spaces on the right side of the screen are not copied.) Type `RUN 2` whenever you are going to edit the program; type `TEXT` when you are done. (All the Appendix E programs have this same line.)

Lines `1000` to `1060` are the top-level block of the program. By reading this level, you can understand exactly what the program does and what each subroutine block is responsible for doing.

Lines 1070 to 1160 end the program. Programs should have only one end point because you may decide at some future time to have the program do something special. In this case, the programmer, some two months after finishing the program, decided to include line 1155, which returns the user to the DISK MENU program. Because the program has only one exit, it wasn't necessary to read through the entire program looking for other END statements before making the addition.

Lines 1180 to 1390 set up the program and display the directions.

Line 1200 should be included in all your programs that use 40-column display. (MAGIC MENU, the next program, has a similar line for 80-column display.) You can never assume that the user is going to run your program with the proper text mode already set. This line also has the effect of canceling a RUN 2, so you needn't type TEXT before running the program.

Line 1210 dimensions the two string arrays used to store the first and last sentence halves. Because space for 1000 elements is allotted in each string, it would take a user between six and eight hours to fill enough strings to cause an error. It is hoped that few users are so dedicated to sentence scrambling!

English words—like TITLE$ in line 1220—are used as string variable names to make them easy to identify. Use descriptive names for variables whenever practical, keeping in mind that Applesoft only looks at the first two characters. (More on variable names in the next program.)

Lines 1400 to 1540 form the second-level subroutine block that accepts the first halves of sentences. The routine works by having the user fill the strings in the FIRST$ array, and keeps track of how many are filled with a counter called F, for "first."

Variable counters keep track; you have used them before in FOR/NEXT loops. A good rule to follow, when deciding whether to use IF...THEN or FOR/NEXT statements, is

- When the use of a variable counter is determinate (when you know, for example, that you want to count from 1 to 10 and then stop), use a FOR/NEXT loop.

- When the use of a variable counter is indeterminate (when you don't know the exact number), as in SCRAMBLER, use an IF...THEN statement.

Lines 1400 to 1470 give instructions to the user. Line 1490 sets a scrolling window, so that the user's entries will not scroll the instructions off the screen.

Lines 1500 to 1530 form a loop: as long as the user does not press OPEN-APPLE RETURN, F is incremented by 1 and the user can type in another string.

Line 1510 quickly checks to see if the user is pressing the OPEN-APPLE key. It does this by looking at the switch immediately after RETURN is pressed in the INPUT statement (line 1500).

Line 1520 checks to make sure that the user did type something before incrementing F. This prevents blank lines from being introduced into the sentence halves. Try changing the line to read

```
1520 F = F + 1
```

to see how the program would work without this error-checking element.

Lines 1600 to 1720 get the second halves of the sentences. This routine works like the first; only the prompt lines are different.

Lines 1800 to 1950 contain the second-level block subroutine that combines random sentence halves and displays the combined sentences on the screen.

Line 1820 makes sure that there are halves to combine. If not, it returns to the top-level routine. Usually, the lack of sentences means the user wants to get out.

Line 1830 uses the TEXT statement to release the text window set by line 1660 so that HOME will clear the whole screen.

Lines 1910 to 1930 select and display a random sentence. Line 1910 selects the sentence variables, FF and SS, from the total number of sentence halves, F and S. Line 1930 then displays the sentence halves with an intervening space.

Fine Tuning

SCRAMBLER has two problems that have been left for you to work out:

1. Applesoft standard INPUT statements will not accept commas or colons: try typing a sentence half with either one. There are two ways to handle problems of this type: warn the user or fix the problem. In this case you might simply tell the user not to type in commas or colons, as part of the instructions.

   In MAGIC MENU you will learn how to use a special INPUT subroutine that has already taken care of the problem; and you'll learn how to put it in programs like SCRAMBLER. It is a more difficult fix, but would be easier on the user.

2. If a user types spaces at the beginning or end of the sentence halves, the two halves are pushed apart. (Try it!) This can be fixed by getting rid of extra leading and trailing spaces, a complex task requiring the use of LEFT$ and RIGHT$ string functions. Again, the simpler way of handling the problem, until you have a stronger command of Applesoft, is to give the user a warning message.

For now, try adding warnings about these problems to the user directions after line 1360. If the added instructions are too long, change the VTAB 4 in line 1230 to a simple PRINT. When you are sure the program works, save it on a disk. Later you may wish to come back to SCRAMBLER and try out some of your advanced skills by correcting the original problems.

Program Listing

```
0  REM SCRAMBLER - SEPT, 1982 BY BG & TOG
1  GOTO 1000: REM BEGINNING OF PROGRAM
2  TEXT : PRINT  CHR$ (21): POKE 33,33: HOME : END
1000  REM THE SCRAMBLER - A SENTENCE SCRAMBLING GAME - 1982
1010  GOSUB 1180: REM SET UP PROGRAM & DISPLAY DIRECTIONS
1020  GOSUB 1400: REM INPUT 1ST HALF OF SENTENCES
1030  GOSUB 1600: REM INPUT 2ND HALF OF SENTENCES
1040  GOSUB 1800: REM DISPLAY SENTENCES
1050  REM    O.K., WE'RE DONE NOW, SO...
1060  REM
1070  REM *** END OF PROGRAM ***
1080  REM
1090  REM NOTE: PROGRAMS SHOULD HAVE ONLY 1 EXIT
1100  TEXT : HOME : VTAB 12
1110  PRINT "Would you like to play again (Y or N)? ";: GET IN$
1120  IF IN$ = "Y" OR IN$ = "y" THEN  TEXT : HOME : GOTO 1020
1130  IF IN$ <  > "N" AND IN$ <  > "n" THEN 1100
1140  TEXT : HOME : VTAB 22
1150  PRINT "Thanks for playing...."
1155  IF  PEEK (6) = 99 AND  PEEK (7) = 99 THEN  PRINT  CHR$ (4);"RUN HEL
    LO": REM SEE NOTES FOLLOWING LINE 9060 IN THE DISK MENU PROGRAM
1160  END
1170  REM
1180  REM *** DIRECTIONS ***
1190  REM
1200  PRINT  CHR$ (21): TEXT : HOME : REM   TURN OFF APPLE'S 80-COLUMN TE
    XT CARD, SELECT TEXT, AND CLEAR THE SCREEN
1210  DIM FIRST$(999),LAST$(999):F = 0:S = 0: REM GIVE THE USER 1000 1ST
    AND 2ND STRINGS
1220 TITLE$ = "*** THE SCRAMBLER ***": HTAB 21 -  LEN (TITLE$) / 2: PRINT
    TITLE$
1230  VTAB 4
1240  PRINT "The scrambler is a word game.  You will"
1250  PRINT "be asked to type the first halves of"
1260  PRINT "sentences such as 'Edwin cooked'.": PRINT
1270  PRINT "You will then be asked for the second"
1280  PRINT "halves of sentences such as 'a banana"
1290  REM
1300  PRINT "squash.'": PRINT
1310  PRINT "Type as many as you like; then command"
1320  PRINT "the computer to combine random halves"
1330  PRINT "into humorous sentences.": PRINT
1340  PRINT "The hilarity rises with the number of"
1350  PRINT "participants: what seems dull to you": PRINT "alone will be
    found hysterically funny"
1360  PRINT "by a group of 10 otherwise normal people";: PRINT "-- a stra
    nge phenomenon."
1370  VTAB 23
1380  INPUT "Press the RETURN key to begin.";IN$
1390  RETURN
1399  REM
```

```
1400  REM *** GET 1ST HALVES ***
1410  REM
1420  HOME : HTAB 9
1430  PRINT "The first half..."
1440  VTAB 20
1450  PRINT "Type the first halves of sentences."
1460  PRINT "Press RETURN after each entry."
1470  PRINT "Hold down OPEN-APPLE and press the": PRINT "RETURN key after
      your last entry.";: REM ALWAYS USE A SEMI-COLON AFTER PRINTS ON BOT
     TOM LINE
1480 F = 0:OAK = 0: REM SET COUNTER TO 0: SET OPEN-APPLE KEY VARIABLE TO
     0
1490  POKE 34,2: POKE 35,18: HOME : REM SET SCROLLING WINDOW
1500  INPUT FIRST$(F)
1510  IF  PEEK ( - 16287) > 127 THEN OAK = 1: REM CHECK QUICKLY TO SEE IF
      USER IS PRESSING THE OPEN-APPLE KEY
1520  IF  LEN (FIRST$(F)) > 0 THEN F = F + 1: REM ADVANCE COUNTER IF A SU
     CCESSFUL ENTRY
1530  IF OAK = 0 THEN 1500: REM IF OPEN-APPLE-RETURN NOT PRESSED, LOOP BA
     CK FOR THE NEXT HALF-SENTENCE
1540  RETURN
1550  REM
1600  REM *** GET 2ND HALVES ***
1610  REM
1620  TEXT : VTAB 1: HTAB 9
1630  PRINT "... the second half"
1640  VTAB 20
1650  PRINT "Type the second halves of sentences."
1660 S = 0:OAK = 0: POKE 34,2: POKE 35,18: HOME : REM SET UP SECOND HALF
     SAME WAY AS FIRST
1670  INPUT LAST$(S)
1680  IF  PEEK ( - 16287) > 127 THEN OAK = 1: REM CHECK QUICKLY TO SEE IF
      USER IS HOLDING THE OPEN-APPLE KEY
1690  IF  LEN (LAST$(S)) > 0 THEN S = S + 1:
1700  IF OAK = 0 THEN 1670
1710  RETURN
1720  REM
1800  REM ** DISPLAY SENTENCES **
1810  REM
1820  IF F = 0 OR S = 0 THEN  RETURN : REM IF THERE IS NOT AT LEAST 1 COM
     PLETE SENTENCE, CANCEL
1830  TEXT : HOME : HTAB 9
1840  PRINT "Scrambled sentences:"
1850  VTAB 20
1860  PRINT "Press RETURN for a new sentence."
1870  PRINT "Press OPEN-APPLE-RETURN to end."
1880 OAK = 0: POKE 34,2: POKE 35,18: HOME
1890  VTAB 17: INPUT IN$
1900  IF  PEEK ( - 16287) > 127 THEN  RETURN : REM IS USER PRESSING OPEN-
     APPLE? YES, THEN RETURN
1910 FF =  RND (1) * F:SS =  RND (1) * S: REM SELECT RANDOM SENTENCE
1920  VTAB 17
1930  PRINT FIRST$(FF);" ";LAST$(SS): REM AND PRINT IT
1940  PRINT : PRINT
1950  GOTO 1890
```

MAGIC MENU

It is a rule in programming that the last 10 percent of a program takes 90 percent of the time. Say you've written a program called "Commodity Market Forcaster for Oleomargarine Futures." It only took you two hours to write, but you didn't include any user instructions. After all, you know when to plug in corn oil prices, so the directions consist of a series of blinking cursors. Unfortunately, Mr. Benson, your boss, just got wind of your work and would like a copy to run on his Apple II Plus. It could take you another three or four days to get the program in shape for him to use. What to do? Use MAGIC MENU.

MAGIC MENU contains five subroutine blocks that collectively take care of the most time consuming, boring, and important tasks in making a program usable by others. ("Others," by the way, includes you about two months down the road when you run the program again and try to figure out what the third blinking cursor is asking for.)

The blocks are intelligent. For example, the Computer Identifier subroutine figures out which version of the Apple II series the program is being run on, so the other blocks in the program can take advantage of the machine's features. This lets you use both upper- and lowercase characters in your programs, even though they may be run on an Apple II or Apple II Plus. (Earlier versions of the Apple II did not have lowercase characters available from the keyboard.)

The actual code that forms each of the blocks is very complex and rather obscure, but this should not keep you from using them. The blocks can be thought of as "black boxes": give them the input they want, and they will carry out their jobs. You needn't understand how a radio works to find your favorite station; in the same way, these subroutine blocks carry out very complex tasks, just so you don't have to.

The following discussion will center exclusively on how to use the blocks, not how they work. At the end of the MAGIC MENU discussion are some hints on how to build up libraries of your own favorite black boxes, along with a discussion of variable names and how to choose them.

Notes To Advanced Programmers

MAGIC MENU and the two programs that follow feature an Applesoft version of the Apple IIe standard interface.

The REM statements scattered throughout MAGIC MENU convey a lot of information about how to deal with the Apple IIe 80-Column Text Card; how to use HTAB and VTAB; and how to read the cursor location.

How the Five Subroutines Work: A Demonstration

This section will use examples, which you should try out on your computer. They demonstrate how the five subroutines (black boxes) work and how they are tied together in MAGIC MENU. The new lines you enter will allow you to look at each subroutine as a stand-alone block of code.

If you have not yet run MAGIC MENU, do so now. Then, to clear out everything except the subroutine package itself and keep from running into Computer Identifier, which starts on line 63000, type

```
DEL 1000,62999
2000 END
```

Now, to make sure the system is normal, type

```
RUN 2
TEXT
```

This should leave you in 40-column display with a clear screen. Lines 0 to 2 of MAGIC MENU are suspiciously similar to lines 0 to 2 of every other program in Appendix E; they were discussed in detail in the section on SCRAMBLER. Line 1 remains because all the examples in this section start on line 1000: this way you can type RUN instead of RUN 1000.

The INPUT *Routine*

Lines 100 to 299 are the INPUT routine. This routine is an extension of the Applesoft statement

```
INPUT "" ; AN$
```

Any prompt, such as `Do you want the answer in dollars? (y/n)`, must have already been given. You may limit how many characters the user may enter by setting the variable FL (for "field length") to a number between 1 and 250. (Even with a `y/n` response, as above, it is wise to give people a few spaces to thrash around in.) If you know that the user probably will want the answer in dollars, you may supply that answer in AN$, in this case, AN$ = "y". Then the user must only press RETURN to say "yes." (Apple II and Apple II Plus users will not be given this default answer.)

The INPUT routine, when run on an Apple IIe, will also look to see if the ESC, OPEN-APPLE, and SOLID-APPLE keys are pressed. If ESC is pressed, or the OPEN-APPLE or the SOLID-APPLE keys are held down while another key is pressed, the program will return. ESC will equal 1 if ESC is pressed, OAKEY will equal 1 if OPEN-APPLE is pressed, and SAKEY will equal 1 if SOLID-APPLE is pressed. AN$ will hold what the user typed before pressing one of these keys; and OAKEY$ or SAKEY$ will contain the key pressed at the same time as the OPEN-APPLE or SOLID-APPLE key, respectively.

You may ignore all of these features by stating (here comes an example—type it in and try it out!):

```
1000 AIIE = 1 : HOME
1010 PRINT "Do you like margarine?";
1020 AN$ = "" : FL = 0 : GOSUB 100 : IF ESCKEY <> 0
     OR OAKEY <> 0 OR SAKEY <> 0 THEN 1000
1030 PRINT AN$
RUN
```

This example doesn't use the special capabilities of the INPUT routine, namely, field length and recognition of special keys. But even ignoring these capabilities, you will be able to use the blinking-underline cursor, and users will be able to type commas and colons.

Line 130 lists all the variables used (and potentially changed by) the routine. All good black box REM statements should do this.

The GET RETURN *Routine*

Lines 300 to 399 are the GET RETURN keypress routine in the program: they do the Apple IIe equivalent of a GET loop until the user presses RETURN. There are many times in programs when you need to say something like Press RETURN to see the Oleomargarine Futures and then wait around for someone to press RETURN. This routine does that, supplying the same cursor as the INPUT routine.

To see it work, simply type

```
DEL 1000, 1030
1000 PRINT "Please press RETURN.";
1010 GOSUB 300
RUN
```

In this example you don't supply any information, and none will be supplied back. It will change the values of I, J, K, L, and P. (Line 1010 is being deleted because it was used in the INPUT routine example.)

The Screen Formatter Routine

Lines 400 to 499 are the Screen Formatter routine. This black box performs the equivalent of

```
400 PRINT AN$ ; "" ; : RETURN
```

But it does it in a very intelligent way.

First, it never breaks a word in half between lines: it looks back through the line and finds the first space it can and breaks the sentence there, continuing it on the next line. This means that you don't have to spend large amounts of time getting instruction pages just so, only to find you left out a word that throws off every line all the way down the screen. Just throw your text into AN$ and execute the Screen Formatter.

Second, the Screen Formatter routine is active while the program is running: if the specific computer has an 80-column text card in it, it will properly format the text for 80 columns.

Third, if the program is run on an Apple II or Apple II Plus, the Screen Formatter will convert all lowercase letters, which those computers do not recognize, into uppercase letters. So your whole program can take advantage of the new Apple IIe hardware, without being confined to just the Apple IIe.

The program expects the contents you want printed to be in AN$. It will display in upper- and lowercase if the Computer Identifier routine has set AIIE to true (AIIE = 1). It will format for 80 columns if you have set COL80 to true (COL80 = 1). More of that on line 1025. Try this example:

```
1000 AIIE = 1
1010 AN$ = "Margarine used to taste horrible
     and came with the coloring in a separate
     pouch. It has improved considerably. Is
     it good enough now to compete with the $4.98
     spread?"
1020 GOSUB 400
1030 GOSUB 100
```

After running it this way, try changing line 1000 to

```
1000 AIIE = 0
```

and run it again. This is the way it would look on an Apple II or II Plus: slow, but serviceable. On these models, the routine has to scan through each character (after finding the 40th character and counting back to the first word break) to see if the character is lowercase. If it is, a complicated formula is used to translate the lowercase character into an uppercase character for correct Apple II display. (See line 445.) Note also that a listing of such lines on an Apple II or Apple II Plus will display lowercase characters as garbage; the translation only works when the program is run.

The Menu Maker Routine

Lines 500-899 contain the fourth block, called Menu Maker. This routine creates a menu in minutes, instead of days, a boast about to be proved. Type

```
1000 AIIE = 1
```

and enter the name of the program with

```
1010 TITLE$ = "Oleomargarine Futures  Forecaster"
```

The name of this menu is

```
1020 SUBTITLE$ = "Type of Margarine"
```

The menu selections (yes, you should enter them so you can see Menu Maker at work) are

```
1030 MENU$(1) = "Corn Oil"
1040 MENU$(2) = "Sunflower Seed Oil"
1050 MENU$(3) = "Cottonseed Oil"
1060 MENU$(4) = "Crude Oil"
1070 MENU$(5) = "End the Program"
```

To tell Menu Maker you are through, enter

```
1080 MENU$(6) = "END"
```

It will also accept "End" or "end". In line 1090, FROM$ is set to the null string because there are no submenus in the program example. However, in a program with more than one menu, the variable FROM$ holds the name of the menu the user has come from. When FROM$ has something in it (besides a null string), it enables the ESC key and displays a message like To return to the main menu, press the ESC key.

```
1090 FROM$ = " "
```

You may offer descriptions of the various menu items within your program. MAGIC MENU does this; for now, there are no help screens (descriptions of the menu items), so OAKEY is set to false (OAKEY = 0).

```
1100 OAKEY = 0
```

Finally, add

```
1110 GOSUB 500
1120 PRINT AN, AN$
```

So you can see that the correct menu item number was placed in both AN and AN$. Writing this menu certainly took only minutes. Now type RUN.

Magic!

Of course, all options will end the program since you are just displaying AN, not using it. From here, you could have your program branch to various subroutines dealing with the different oils. See the discussion of line 1670 in the section "Notes on the Rest of MAGIC MENU" for more information.

The Computer Identifier Routine

Now you can try out the fifth and final block, Computer Identifier. Right now, line 1000 looks like this:

```
1000 AIIE = 1
```

To make Computer Identifier figure out which machine you are using, change it to

```
1000 GOSUB 63000
```

This will make AIIE = 1 if the program is run on an Apple IIe and 0 if it is run on an Apple II or Apple II Plus. It also sets the variable RESULTS to

- 0 if an Apple II or II Plus
- 32 if an Apple IIe
- 64 if an Apple IIe with an Apple IIe 80-Column Text Card
- 128 if an Apple IIe with an Extended 80-Column Text Card

To use an Apple IIe 80-Column Text Card or an Extended Text Card, and to let the Menu Maker and Screen Formatter routines know if there is such a card, add this line:

```
1005  IF RESULTS >= 64 THEN COL80 = 1 : PRINT
      CHR$(4) ; "PR#3"
```

Try running the program again. If you have neither card, nothing will happen. If you have a text card, your screen will suddenly display 80 columns. To go back to 40 columns, just type

```
RUN 2
```

Notes on the Rest of MAGIC MENU

To get the entire program back again, type

```
LOAD MAGIC MENU
```

You have explored the black boxes themselves; now take a look at the lines of a program built upon them. There is very little difficult code in these lines; for the main part, you can look at the listings and run the program to figure out what is being done. A few lines that bear further comment are discussed in this section.

Line 1010: Applesoft puts each variable it encounters during the running of a program into a variable table, where it stores the current values of those variables. Any time a program looks at or changes the value of a variable, Applesoft scans through the table, looking at each variable in order. By declaring the most often used variables first, you can speed up your program significantly.

Line 1360: users with an 80-column text card can look at more text on one screen than users with a 40-column display. The program takes advantage of this by offering more information to users with an 80-column display.

Line 1640 handles requests for descriptions.

Line 1645 handles a request to go back to the DISK MENU program.

Line 1670 sends the user off to the requested subsection of the program. ON AN GOSUB 2000, 3000, 4000 will go to a subroutine beginning on line 2000 if AN = 1; 3000 if AN = 2; 4000 if AN = 3; and so on. Note, however, that when you use ON GOSUB, there is no logical connection between the variable number and the subroutine it sends the program to. Menu Maker will not allow an illegal number, so error-checking for correct numbers in AN is already included.

Lines 1800 to 1890 are very similar to those described above; they handle the call for descriptions, branching to the appropriate subsections.

Line 2010 does the sort of error-checking you need to think about if you want your program to be universal. (You can test your program in Apple II Plus mode, by the way, by setting AIIE and RESULTS to 0, instead of using the Computer Identifier routine.)

Program Speed

When Applesoft executes programs, it goes through every line number in sequence every time. The larger the program, the longer it takes to run; the further up a variable is in a variable table, the longer it takes to find.

Applesoft executes subroutines with the smallest line numbers fastest, so the smaller line numbers are best reserved for subroutines that must do a lot of computing in a minimum of time. The main routines usually go near the end because they are only called once.

To see the difference in execution times between small line numbers and large ones, compare the two mini-programs that follow. Type

```
10 I = I + 1 : IF I < 300 THEN 10
20 STOP
RUN 10
```

and then try

```
50000 I = I + 1 : IF I < 300 THEN 50000
50010 STOP
RUN 50000
```

A Few Words about Variable Names

Once you become familiar with how the black boxes in MAGIC MENU work, you will probably be interested in creating some black boxes of your own. These notes on variable names are designed to help you do just that.

It is usually better to use descriptive variable names. Descriptive variable names make a program far more readable: `AIIE = 1` gives you a hint; `L3 = 1` does not.

Often, however, the speed at which your Applesoft programs execute is an issue. If you stick with a few variables, declared early in your program, using them in all high-speed routines, your program will run much faster. Often three or four times faster.

Fortunately, there are some guidelines, based on conventions going back to the early 1950's, for high-speed variables. There are eight standard, nondescriptive variable names that are used to speed up execution. And, of course, they break the rule that it is always better to use descriptive variable names. The standard variable names are:

```
I, J, K, L, M, N, O, P
```

`I` is the most common variable name of all. `J` is second, and so forth. `P` is generally used for the `PEEK` function—to look at the keyboard and other hardware locations.

These variables should only be used within a subroutine, not between subroutines, nor between subroutines and the calling routine. To communicate between routines, use variable names that mean something.

If you need to call a low-level subroutine (such as the `INPUT` routine) while carrying out a high-speed activity in a higher-level routine (such as the Menu Maker routine), use the letter twice in the higher-level routine:

```
II, JJ, KK, LL, MM, NN, OO, PP
```

This will keep the lower-level routine from destroying values in the higher-level routine. Menu Maker, for example, uses all double letter variables because it calls both the INPUT and Screen Formatter routines.

Make sure you document with REM statements which variables you are affecting within such subroutines.

Finally, don't use these variables unless you really need the speed, or you are converting your favorite routines to black boxes. You have to be able to read and understand your own program: use descriptive variables anywhere they won't seriously affect your program.

By following these guidelines in the creation of the five subroutine blocks in this program, the program author has given programmers using the package the greatest lattitude in constructing their own routines.

A Few Notes on Logic

Applesoft BASIC uses a kind of logic called Boolean logic. It is based on false and true, zeros and ones.

When you say

```
X = 0 : IF A = 23 THEN X = 14
```

you are calling on Applesoft to decide if it is true that A = 23. If true, it will carry out your further instructions. The same decision could be made in a decidedly different way:

```
X = 14 * (A = 23)
```

This is a little hard to believe at first. So try it in immediate execution, first making A equal to 23 and then making A equal to something other than 23.

Type

```
A = 23
X = 14 * (A = 23)
PRINT X

A = -32.68
X = 14 * (A = 23)
PRINT X
```

When Applesoft encounters a Boolean argument (A = 23), it decides whether it is true (1) or false (0). (14 * 0) is 0 and (14 * 1) is 14. (You may explore such Boolean logic further in the section on IF . . . THEN in the *Applesoft Reference Manual.*)

MAGIC MENU uses many true or false flags to communicate between routines. A flag is a variable whose contents (usually 1 or 0) indicate whether some condition holds or whether some event has occurred. It is used to control the program's actions at some later time.

Using a flag, the Computer Identifier routine communicates to the Screen Formatter routine that the program is running on an Apple IIe by making AIIE = 1. Then, Screen Formatter can make decisions based on IF AIIE = 1 THEN or, more simply, IF AIIE THEN. The INPUT routine reports to you that the OPEN-APPLE key has been pressed by setting the flag, OAKEY, to true. Then you may go ahead and see what is in OA$ to find out what key was pressed at the time.

A good programming practice is to declare the two variable names TRUE and FALSE early in your program:

```
1000 FALSE = 0 : TRUE = 1
```

Having done so, you may then clearly state what you are doing:

```
1010 AIIE = TRUE
1020 IF AIIEE = TRUE THEN PRINT "TRUE"
1030 IF AIIE = FALSE THEN PRINT "FALSE"
```

Many new programmers will avoid flags, choosing instead some other kind of test. A couple of years ago, an Apple programmer needed to get the user's age in the variable, YEARS. His subroutine for doing this first asked the user for the age in months and years. The routine then divided the months by 12, added them to the years, and finally took the integer of YEARS + .5 to round it to the nearest year:

```
YEARS = INT (YEARS + .5) : RETURN
```

When the subroutine ended, the routine that called it needed to know if the user had answered successfully, rather than just pressing RETURN. The programmer discoved that he could tell if an age had been successfully found by simply testing the variable YEARS:

```
1000 IF YEARS THEN   [go on with the program]
```

That worked fine until a colleague typed in the age of her new baby: no years, 2 months. The program rounded this reply to: YEARS = 0. It didn't matter how many times she typed it in, it kept asking her how old "Betty" was.

You will not run out of variable names—there are more than 5000 of them. Use a flag and always give it a descriptive name.

Program Listing

```
0  REM MAGIC MENU - SEPT, 1982 BY TOG
1  GOTO 1000
2  TEXT : PRINT  CHR$ (21): HOME : POKE 33,33: END
100  REM *** INPUT ROUTINE ***
102  REM SEPT, 1982 BY B. TOGNAZZINI
105  REM USES FLASHING UNDERLINE CURSOR
110  REM YOU MAY SET THE NUMBER OF CHARACTERS THE USER MAY TYPE (THE FIEL
     D LENGTH) IN THE VARIABLE FL
115  REM  A FIELD LENGTH OF 0 (FL=0) ALLOWS THE MAXIMUM INPUT
120  REM IF YOU WANT A "DEFAULT" (SUPPLIED-BY-YOU) ANSWER, PUT IT IN AN$.
      FOR EXAMPLE: AN$= "CAT"
124  REM UPON RETURN, AN$ (ANSWER$) WILL CONTAIN THE INPUT FROM THE USER
125  REM ESCKEY WILL BE SET TO TRUE (WILL EQUAL 1) IF THE USER PRESSED IT
      TO EXIT THE INPUT
126  REM SAKEY OR OAKEY WILL BE SET TO TRUE (WILL EQUAL 1) IF THE USER PR
     ESSED ANY CHARACTER KEY WHILE HOLDING THE SOLID-APPLE OR OPEN-APPLE
     KEY, RESPECTIVELY
127  REM ABOVE 3 KEYS ARE ONLY READ IN AN APPLE IIE: PROVIDE SOME OTHER M
     ETHOD OF SIGNALLING FOR AII AND AII+ OWNERS
130  REM THE ROUTINE USES (AND MAY CHANGE THE VALUES IN) I,J,K,L,M,P,FL,E
     S (FOR ESCAPE KEY),OA (FOR OPEN-APPLE KEY),SA (FOR SOLID-APPLE KEY),
     FL,I$,OA$,SA$,AN$
135  REM * TAKE INPUT *
140 I = 5:J = 0:K = 0:L = 0:M = 0:I$ = "":ESC = 0:OA = 0:OA$ = "":SA = 0:
     SA$ = "": IF FL = 0 THEN FL = 245
142  IF  NOT AIIE THEN  INPUT "";AN$: RETURN
145  PRINT AN$;:J =  LEN (AN$)
149 M =  PEEK (37)
153 L =  PEEK (36): IF COL80 THEN  IF L =  PEEK (1147) THEN L =  PEEK (14
     03): REM FIND CURRENT HORIZ POSITION WITH 80 COLUMN CARD TURNED ON
155  PRINT " ";I$;" ";
160 N = 1: IF L +  LEN (I$) >  =  PEEK (33) - 3 AND  PEEK (37) =  PEEK (3
     5) - 1 THEN N = 0
165  POKE 36,L: POKE 1403,L: VTAB M + 1
170 I = I - 1: IF I < 0 THEN K = 1 - K:I = 5: PRINT  CHR$ (32 + 63 * K);
175  POKE 36,L: POKE 1403,L: VTAB M + 1
180 P =  PEEK ( - 16384): IF P < 128 THEN 170
185  IF  PEEK ( - 16287) > 127 THEN OAKEY = 1
190  IF  PEEK ( - 16286) > 127 THEN SAKEY = 1
195  POKE  - 16368,0:K = 0:I = 0
```

```
200  IF OAKEY THEN OA$ =  CHR$ (P - 128):AN$ = AN$ + I$: PRINT I$;" ": RETURN
     : REM * OPEN-APPLE KEY
205  IF SAKEY THEN SA$ =  CHR$ (P - 128):AN$ = AN$ + I$: PRINT I$;" ": RETURN
     : REM * SOLID-APPLE KEY
210  IF P > 159 AND P <  > 255 THEN  IF J +  LEN (I$) < FL THEN  IF N THEN
     AN$ = AN$ +  CHR$ (P - 128):J = J + 1: PRINT  CHR$ (P);: GOTO 149
215  IF P <  > 255 THEN 240: REM DELETE KEY
220  IF J THEN  PRINT " ";: POKE 36,L: VTAB M + 1: PRINT  CHR$ (136);:J =
     J - 1
225  IF J = 0 THEN AN$ = ""
230  IF J THEN AN$ =  LEFT$ (AN$,J)
235  GOTO 149
240  IF P <  > 136 THEN 265: REM * BACK ARROW KEY
245  IF J THEN  PRINT " ";: POKE 36,L: VTAB M + 1: PRINT  CHR$ (136);:I$ =
      RIGHT$ (AN$,1) + I$:J = J - 1
250  IF J = 0 THEN AN$ = ""
255  IF J THEN AN$ =  LEFT$ (AN$,J)
260  GOTO 149
265  IF P = 141 THEN AN$ = AN$ + I$: PRINT I$;" ": RETURN : REM * RETURN
     KEY
270  IF P <  > 149 THEN 294: REM * FORWARD ARROW KEY
275  IF  NOT  LEN (I$) THEN 149
280 AN$ = AN$ +  LEFT$ (I$,1):J = J + 1: PRINT  LEFT$ (I$,1);
285  IF  LEN (I$) = 1 THEN I$ = ""
290  IF  LEN (I$) THEN I$ =  RIGHT$ (I$, LEN (I$) - 1)
292  GOTO 149
294  IF P = 155 THEN ESCKEY = 1: PRINT : RETURN : REM ESCAPE KEY PRESSED
296  GOTO 149
298  REM


300  REM  ** GET RETURN ROUTINE **
305  REM USES I,J,K,L,P
310  IF AIIE THEN 325
315  GET AN$: IF  ASC (AN$) <  > 13 THEN 315
320  PRINT : RETURN
325 I = 0:J = 0:K = 0
330 I = I - 1: IF I < 0 THEN K = 1 - K:I = 5: PRINT  CHR$ (32 + 63 * K);
335  IF I <  > 5 THEN 355
340 L =  PEEK (36): IF COL80 THEN  IF L =  PEEK (1147) THEN L =  PEEK (14
     03): REM FIND CURRENT HORIZ POSITION WITH 80 COLUMN CARD TURNED ON
345  IF L = 0 THEN  POKE 36, PEEK (33): VTAB  PEEK (37)
350  IF L <  > 0 THEN  POKE 36,L - 1
355 P =  PEEK ( - 16384): IF P <  > 141 THEN 330: REM NOT A RETURN
360  PRINT " ";
365  IF  PEEK (37) = 23 THEN  VTAB 23: REM PEEK(37) CONTAINS CURRENT VTAB
      POSITION -1.  IF ON BOTTOM LINE (VERY COMMON WITH WAIT-FOR-RETURN-K
     EYS) MOVE UP ONE TO PREVENT SCREEN FROM SCROLLING
370  PRINT
375  POKE  - 16368,0: RETURN
399  REM


400  REM  *** SCREEN FORMATTER ROUTINE ***
401  REM STRING TO BE PRINTED IN AN$
402  REM IF 80 COLUMN BOARD IS TURNED ON, MAKE SURE COL80 = 1.  IF BOARD
     IS NOT TO BE USED, MAKE SURE COL80 = 0.
403  REM USES I,J,I$
404  REM ROUGH EQUIVALENT OF  PRINT AN$;" ";
405  REM  USES AIIE SET BY COMPUTER IDENTIFIER ROUTINE
406  REM USUALLY LEAVES 1 EXTRA BLANK AT END OF LINE
407  REM PERFORMS WORD-WRAP AND WILL CONVERT LOWER- TO UPPER-CASE IF USED
      INSIDE AN APPLE II OR II+
410 I =  LEN (AN$): IF  NOT I THEN  RETURN
411 P =  PEEK (36): IF COL80 THEN  IF P =  PEEK (1147) THEN P =  PEEK (14
     03): REM FIND CURRENT HORIZ POSITION WITH 80 COLUMN CARD TURNED ON
412  IF  NOT P THEN  IF I > 1 THEN  IF  ASC (AN$) = 32 THEN AN$ =  RIGHT$
     (AN$,I - 1)
413  IF P + 2 + I <  PEEK (33) AND AIIE THEN  PRINT AN$;" ";:AN$ = "": RETURN
     : REM EXPRESS CHECK-OUT
```

```
414  IF I > 1 THEN  IF  RIGHT$ (AN$,1) = " " THEN AN$ =  LEFT$ (AN$,I - 1
     )
417  IF P + I <  PEEK (33) THEN I$ = AN$:AN$ = "": GOTO 440
420 J =  PEEK (33) - P + 2:I = J
425 I = I - 1: IF I THEN  IF  MID$ (AN$,I,1) <  > " " THEN 425
430  IF I = 1 THEN I = J
431  IF I = 0 THEN  PRINT : GOTO 410
435 I$ =  LEFT$ (AN$,I - 1): IF  LEN (AN$) > I THEN AN$ =  RIGHT$ (AN$, LEN
     (AN$) - I): REM ISOLATE 1 LINE IN I$
440  IF AIIE THEN  PRINT I$;
445  IF  NOT AIIE THEN K =  LEN (I$) + 1: FOR I = 1 TO  LEN (I$):J =  ASC
     ( RIGHT$ (I$,K - I)): PRINT  CHR$ (J - 32 * (J > 96 AND J < 123));: NEXT
     I
447 P =  PEEK (36): IF COL80 THEN  IF P =  PEEK (1147) THEN P =  PEEK (14
     03)
450  IF  LEN (AN$) THEN  IF P <  > 0 THEN  PRINT
455  IF  LEN (AN$) THEN 410
460  IF P <  > 0 THEN  IF  MID$ (I$, LEN (I$),1) <  > " " THEN  PRINT " "
     ;
465  RETURN
499  REM



500  REM  *** MENU MAKER ROUTINE ***
505  REM SUPPLY PROGRAM TITLE IN TITLE$, MENU TITLE IN SUBTITLE$, CALLING
      MENU TITLE (WHERE USER CAME FROM) IN FROM$, OA=1 IF HELP IS AVAILAB
     LE


510  REM PUT MENU ITEMS IN MENU$(1) THROUGH MENU$(12).  REMEMBER TO DIM M
     ENU$(12) IF MORE THAN 10 ITEMS

515  REM PROGRAM USES INPUT ROUTINE AND SCREEN FORMATTER.  ALL VARIABLES
     AFFECTED BY THEM AS WELL AS II, JJ, KK, LL, MM, NN, OO,PP, AN, II$ A
     RE AFFECTED
520  REM USER'S CHOICE IS RETURNED IN BOTH AN$ AND AN

525  REM  IF FROM$ HAS A TITLE IN IT, ESCKEY WILL BE 1 IF USER WANTS TO G
     O BACK TO CALLING MENU.  MAIN MENUS SHOULD MAKE FROM$="" (NOTHING)

530 J = 0:K = 0:JJ = 0:KK = 0:LL = 0:MM = 0:NN = 0:OO = 0:AN = 0
535 OO = OA: REM STORE WHETHER DESCRIPTIONS ARE AVAILABLE
540 AN$ = ""
545 II = 1
550 KK = KK +  INT ( LEN (MENU$(II)) / (27 + 35 * (COL80 = 1))): REM APPR
     OXIMATE NUMBER OF EXTRA LINES MENU ITEM WILL TAKE
555  IF II < 12 THEN  IF MENU$(II + 1) <  > "End" AND MENU$(II + 1) <  >
     "end" AND MENU$(II + 1) <  > "END" THEN II = II + 1: GOTO 550
560 LL = II: REM NUMBER OF MENU ITEMS
565 NN = 0: IF LL * 2 + KK < 14 THEN NN = 1: REM DETERMINE IF MENU CAN BE
      DOUBLE-SPACED
570 MM = 3: IF COL80 THEN MM = 9: REM SELECT LEFT MARGIN OFFSET: 3 IF 40
     COLUMN, 9 IF 80 COLUMN
575 JJ = 3 + MM: IF LL > 9 THEN JJ = 4 + MM: REM IF MORE THAN 9 ITEMS, IN
     DENT NAME OF EACH BY 1 MORE
580  REM


585  REM   * DISPLAY MENU ROUTINE *
590  TEXT : HOME
595 AN$ = TITLE$: GOSUB 400: POKE 36, PEEK (33) -  LEN (SUBTITLE$) - 1:AN
     $ = SUBTITLE$
600  GOSUB 400
605  FOR II = 1 TO  PEEK (33): PRINT "_";: NEXT : PRINT
610  REM PRINT SELECTIONS
615  FOR II = 1 TO LL
620  HTAB MM: PRINT II;". ";
625  VTAB  PEEK (37): IF COL80 THEN  VTAB  PEEK (1531): REM BELOW PRINT M
     OVES DOWN 1 LINE: THIS COMMAND IS A TRICK TO MOVE UP EXACTLY 1 LINE
     FIRST
```

```
630  POKE 32,JJ: POKE 33, PEEK (33) - JJ: PRINT : REM SET "WINDOW" SO THA
     T TEXT OF MENU ITEM IS INDENTED
635 AN$ = MENU$(II): GOSUB 400: REM PRINT MENU ITEM
640  POKE 32,0: POKE 33, PEEK (33) + JJ: PRINT : REM "RESTORE" FULL WINDO
     W
645  IF NN THEN  PRINT
650  NEXT II
655  IF  PEEK (37) > 16 THEN  PRINT "TOO MANY MENU ITEMS OR TOO LONG LINE
     S.": STOP
660  TEXT : VTAB 17:AN$ = "Select option >": GOSUB 400: PRINT
665  FOR II = 1 TO  PEEK (33): PRINT " ";: NEXT
670  IF  NOT 00 OR  NOT AIIE THEN  PRINT : REM  IF ROOM, SPACE DOWN 1
675  IF  LEN (FROM$) THEN AN$ = "For " + FROM$ + ": press ESC": GOSUB 400

680  PRINT
685  IF AIIE THEN  PRINT "To erase: use the DELETE key"
690 AN$ = "To select: type a number from 1 to " +  STR$ (LL)
700  GOSUB 400: PRINT
705  IF  NOT 00 THEN 720
710  IF AIIE THEN  PRINT "For descriptions: press OPEN-APPLE-?"
715  IF  NOT AIIE THEN  PRINT "FOR DESCRIPTIONS: FOLLOW ANSWER WITH ?"
720 AN$ = "To go to selected item: press RETURN": GOSUB 400: PRINT
723  IF  NOT 00 THEN AN$ = "(There are no descriptions available)": GOSUB
     400
725 FL = 3:AN$ = "": REM SET-UP VALUES FOR INPUT ROUTINE BEFORE CALLING I
     T
730  REM



735  REM ** GET INPUT FROM USER **
740  VTAB 17: HTAB 17: CALL  - 868: HTAB 17
745  IF AIIE OR  NOT  LEN (FROM$) THEN  GOSUB 100: GOTO 795: REM THE LINE
     S THAT FOLLOW ARE TO PICK UP AN ESC PRESS ON AN APPLE II OR II+
750 I = 0:J = 0:M =  PEEK (37):L =  PEEK (36):0AKEY = 0
755 I = I - 1: IF I < 0 THEN K = 1 - K:I = 5:J = 1 - J: NORMAL : IF J THEN
      INVERSE
760  PRINT " ";: POKE 36,L: VTAB M + 1
765 P =  PEEK ( - 16384): IF P < 128 THEN 755
770  NORMAL : REM ABOVE PRINTS A BLINKING CURSOR THE HARD WAY
775  IF P = 155 THEN  POKE  - 16368,0:ESCKEY = 1: GOTO 795: REM ESC PRESS
     ED FIRST
780 L =  PEEK (36): VTAB 20: HTAB 1: CALL  - 868: VTAB M + 1: HTAB L + 1
785  INPUT "";AN$
790  REM
         END OF AII AND AII+ ROUTINE

795  VTAB 19
800  IF  NOT OAKEY THEN 825
805  IF  NOT 00 OR OA$ <  > "?" AND OA$ <  > "/" THEN 735: REM HELP NOT A
     VAILABLE (00 WAS SET TO VALUE OF OAKEY AT BEGINNING OF ROUTINE) OR W
     RONG KEY PRESSED
810  REM  JJ WILL NOW BE USED TO HOLD LENGTH OF AN$:
815 JJ =  VAL (AN$): IF JJ > 0 AND JJ <  = LL THEN II = JJ: GOTO 880: REM
     EXIT MENU FOR HELP
820  IF JJ = 0 THEN  PRINT "--> PLEASE SELECT A NUMBER FIRST <--";: CALL
      - 868: PRINT :AN$ = "": GOTO 735
825  IF SAKEY THEN 735: REM SOLID-APPLE KEY NOT USED FOR MENU
830  IF ESCKEY THEN  IF  LEN (FROM$) THEN 880: REM IF ESCAPE KEY IS PRESS
     ED, MENU IS EXITED
835 JJ =  VAL (AN$): IF JJ > 0 AND JJ <  = LL THEN II = JJ: REM ISOLATE N
     UMBER FROM AN$ AND MAKE II EQUAL IT
840  IF  LEN (AN$) = 0 THEN 735
845  IF 00 THEN  IF  NOT AIIE AND ( RIGHT$ (AN$,1) = "?" OR  RIGHT$ (AN$,
     1) = "/") THEN  IF II > 0 AND II <  = LL THEN OAKEY = 1: GOTO 880
850  IF  LEN (AN$) THEN K = 0: FOR I = 1 TO  LEN (AN$):J =  ASC ( RIGHT$
     (AN$,I)):K = K + (J <  > 32 AND (J < 48 OR J > 57)): NEXT I: IF K THEN
      PRINT "       --> PLEASE USE DIGITS <--";: CALL  - 868: PRINT :AN$ =
     "": GOTO 735
855  REM ABOVE LINE CHECKS FOR PRESENCE OF NON-NUMERIC CHARACTERS OTHER T
     HAN SPACE IN THE LINE; IF FOUND, THE LINE IS REJECTED
```

```
860  IF  NOT JJ THEN  IF  LEN (AN$) THEN K = 0: FOR I = 1 TO  LEN (AN$):J
      =  ASC ( RIGHT$ (AN$,I)):K = K + (J = 48): NEXT I: IF  NOT K THEN  PRINT
     "       --> PLEASE USE DIGITS <--";: CALL  - 868: PRINT :AN$ = "": GOTO
     735
865  REM ABOVE LINE CHECKS FOR A SINGLE SPACE AS ANSWER (OTHERWISE, A SPA
     CE IS A ZERO)
870  IF JJ > LL THEN  PRINT "       --> ";JJ;" IS TOO LARGE <--";: CALL  -
     868: PRINT :AN$ = "": GOTO 735
875  IF JJ < 1 THEN  PRINT "       --> 0 IS TOO SMALL <--";: CALL  - 868:A
     N$ = "": GOTO 735
880 AN = II: TEXT : HOME : RETURN : REM EXIT POINT FOR MENU PROGRAM
885  REM



1000  REM *** MAIN PROGRAM ***
1010 I = 0:AN$ = "":J = 0:I$ = "":K = 0: REM DECLARE MOST OFTEN USED VARI
     ABLES FIRST FOR SPEED.
1020  GOSUB 63000: REM FIND OUT  IF AIIE OR NOT
1021  DIM ME$(20)
1025  IF RESULTS >  = 64 THEN COL80 = 1: PRINT  CHR$ (4);"PR#3": REM IF A
     N 80-COLUMN CARD IS PRESENT, USE IT.  IF YOU DON'T WANT IT, CHANGE T
     HE LINE TO:
                1025 PRINT CHR$(21)

1030  TEXT : PRINT : HOME : REM TEXT CLEARS ANY OLD WINDOWS: PRINT CLEARS
      OUT ANY OLD HTAB AND VTAB INFORMATION: HOME CLEARS THE SCREEN
1040  PRINT : HOME
1050 AN$ = "*** Magic Menu ***": POKE 36,( PEEK (33) -  LEN (AN$)) / 2: GOSUB
     400: PRINT : REM CENTER TITLE
1060  PRINT :AN$ = "Magic Menu has five basic subroutines upon which you
     can build your programs:"
1070  GOSUB 400: PRINT : IF COL80 THEN  PRINT : REM BY CHANCE, THE SECOND
      LINE IS JUST LONG ENOUGH TO CAUSE AN EXTRA CARRIAGE RETURN IN 40-CO
     LUMN MODE
1080 AN$ = "1. COMPUTER IDENTIFIER: says you are now using an Apple": GOSUB
     400
1090  IF RESULT = 0 THEN AN$ = "II OR II+."
1100  IF RESULT = 32 THEN AN$ = "IIe."
1110  IF RESULT = 64 THEN AN$ = "IIe with an 80-Column Card."
1120  IF RESULT = 128 THEN AN$ = "IIe with a Memory-Expansion Card."
1130  IF COL80 THEN  GOSUB 400:AN$ = "The other four routines use the inf
     ormation from COMPUTER IDENTIFIER to let your software take full adv
     antage of whatever Apple computer it is run on."
1140  GOSUB 400: PRINT : PRINT
1150 AN$ = "2. SCREEN FORMATTER: ": GOSUB 400
1160 AN$ = "controls text display so that lines are ended between words i
     nstead of in the middle of them."
1170  IF COL80 THEN  GOSUB 400:AN$ = "It also automatically converts all
     lower-case letters to capitals when the program is run on an Apple I
     I or an Apple II+."
1180  GOSUB 400: PRINT : IF COL80 THEN  PRINT
1190 AN$ = "3. MENU MAKER: lets you create uniform, friendly menus in min
     utes, instead of days.  (A sample menu follows.)"
1200  GOSUB 400: PRINT : PRINT
1210 AN$ = "4. INPUT: is the flashing-underline cursor routine you learne
     d on Apple Presents...APPLE.": GOSUB 400
1220 AN$ = "": IF AIIE THEN AN$ = "(Pick 1 from menu.)"
1230  IF COL80 THEN AN$ = " (To reacquaint yourself with this input routi
     ne, select option 1 from the menu that follows.)
1240  GOSUB 400: PRINT : PRINT
1250 AN$ = "5. GET RETURN: waits for you to press RETURN.  It is waiting
     now..."
1260  GOSUB 400: PRINT : PRINT
1270 AN$ = "Press RETURN to continue.": GOSUB 400: GOSUB 300: REM PRINT P
     ROMPT: WAIT FOR A RETURN
1299  REM
```

```
1300  REM *** "help" SCREEN ***
1310  HOME
1315  IF COL80 THEN  VTAB 4
1320 AN$ = "A few words about the sample menu:"
1330  GOSUB 400: PRINT : PRINT : PRINT
1340 AN$ = "The following menu has six selections, only two of which actu
     ally do something.  The first selection shows you a familiar example
      of the flashing-cursor input routine; the last selection exits the
     program."
1350  GOSUB 400
1360  IF COL80 THEN AN$ = " The other four selections are there to take u
     p space on the menu and in the program, so you can see how a more el
     aborate program would be structured.": GOSUB 400
1370  PRINT : PRINT
1375  IF COL80 THEN  PRINT
1380 AN$ = "The Menu Maker enables you to give a description of each menu
      item, so the user needn't first wade through pages of " +  CHR$ (34
     ) + "directions" +  CHR$ (34) + ":"
1390  GOSUB 400
1400 AN$ = "to see the description of option 1, type"
1410  GOSUB 400
1420 AN$ = "1? RETURN": IF AIIE THEN AN$ = "1, then press OPEN-APPLE-?"
1430  GOSUB 400: PRINT : PRINT
1435  IF COL80 THEN  PRINT
1440 AN$ = "Explore the menu, then review the listings in Appendix E of t
     he Applesoft Tutorial manual for details on how to use these routine
     s in your own programs."
1450  GOSUB 400: PRINT
1460  VTAB 24:AN$ = "Press RETURN to go to the menu.": GOSUB 400: GOSUB 3
     00
1499  REM


1500  REM ** SAMPLE MAIN MENU **
1510  REM ** MAIN MENU LOOP **
1520 TITLE$ = "Magic Menu": REM TITLE OF MENU
1530 SUBTITLE$ = "Main Menu": REM TITLE OF MENU OR SUBSECTION OF PROGRAM
1540 FROM$ = "": IF  PEEK (6) = 99 AND  PEEK (7) = 99 THEN FROM$ = "the D
     isk Menu": REM SEE NOTES FOLLOWING LINE 9060 OF DISK MENU PROGRAM
1550 MENU$(1) = "A sample of the input routine"
1560 MENU$(2) = ""
1570 MENU$(3) = ""
1580 MENU$(4) = ""
1590 MENU$(5) = ""
1600 MENU$(6) = "End the program"
1605  IF  PEEK (6) = 99 AND  PEEK (7) = 99 THEN MENU$(6) = "End the progr
     am and return to the Disk Menu": REM SEE NOTES FOLLOWING LINE 9060 O
     F DISK MENU
1610 MENU$(7) = "END": REM MENU PROGRAM KEEPS DISPLAYING ITEMS UNTIL IT F
     INDS THE WORD "END"
1620 OAKEY = 1: REM HELP AVAILABLE; OPEN-APPLE KEY WILL BE READ
1630  GOSUB 500
1640  IF OAKEY THEN  GOSUB 1810: GOTO 1510: REM "HELP" ASKED FOR
1645  IF ESKEY THEN  GOTO 7000: REM RETURN TO DISK MENU
1650 AN$ = ""
1660  VTAB 8
1670  ON AN GOSUB 2000,3000,4000,5000,6000,7000: REM RETURN PRESSED, SO G
     O TO THE SELECTED SECTION
1680  VTAB 24
1685  IF AN <  > 1 THEN  POKE 36,0: IF COL80 THEN  POKE 1147,255: REM THE
     SE 2 POKES ARE EQUIVALENT TO HTAB 1, BUT WORK IN EITHER 40 OR 80 COL
     UMN MODE
1690 AN$ = "Press RETURN for the menu.": GOSUB 400
1700  GOSUB 300: REM WAIT FOR RETURN
1705  IF COL80 THEN  VTAB 22: PRINT : PRINT  CHR$ (4);"PR#3": REM OPTION
     1 TURNS OFF 80-COL CARD (IF ANY).  THIS TURNS IT BACK ON
1710  GOTO 1510
1720  REM
```

```
1800  REM *** GO TO HELP ***
1810  TEXT : HOME : VTAB 8
1820 HOLD$ = AN$
1830  ON AN GOSUB 9000,9200,9300,9400,9500,9600,9725,9825,9925
1850 AN$ = "Press RETURN to go to the menu.": GOSUB 400
1860  GOSUB 300: REM WAIT FOR RETURN
1870 AN$ = HOLD$
1880  RETURN
1890  REM BLANK LINES ARE DONE BY PRESSING THE DOWN-ARROW KEY



2000  REM *** INPUT SAMPLE ***
2010  IF  NOT AIIE THEN AN$ = "SORRY, THIS SAMPLE WORKS ONLY ON AN APPLE
     IIE COMPUTER.": VTAB 10: GOSUB 400: PRINT : RETURN
2015  PRINT  CHR$ (21): REM TURN OFF 80 COLUMN MODE IF IT IS ON.  TRY REM
     OVING THIS LINE TO SEE EFFECT ON PROGRAM
2020  HOME :HT = 1: IF  PEEK (33) > 40 THEN HT = 21: HTAB HT
2030  PRINT "Correct the answer in the box to read": GOSUB 400: PRINT
2040 AN$ = "A herd of cattle"
2050 II = ( PEEK (33) -  LEN (AN$)) / 2: POKE 36,II: REM POKE 36,X IS THE
      SAME AS HTAB X-1, EXCEPT IT WORKS WITH EITHER 40 OR 80-COLUMN MODES

2055  GOSUB 400: PRINT
2060  POKE 36,II: PRINT "  ----"
2070  HTAB HT:AN$ = "by doing the following:": GOSUB 400: PRINT : PRINT
2075  HTAB HT + 1
2080 AN$ = "1. Press left arrow to back up to the": GOSUB 400: PRINT
2085  HTAB HT + 4
2090 AN$ = "right of the " +  CHR$ (34) + "t" +  CHR$ (34) + " in the wor
     d " +  CHR$ (34) + "lot" +  CHR$ (34): GOSUB 400: PRINT
2092  PRINT
2095  HTAB HT + 1
2100 AN$ = "2. Press the DELETE key several times": GOSUB 400: PRINT
2105  HTAB HT + 4
2110 AN$ = "to erase the words " +  CHR$ (34) + "whole lot" +  CHR$ (34):
      GOSUB 400: PRINT : PRINT
2115  HTAB HT + 1
2120 AN$ = "3. Type " +  CHR$ (34) + "herd" +  CHR$ (34): GOSUB 400: PRINT
     : PRINT
2125  HTAB HT + 1
2130 AN$ = "4. Now press the RETURN key to": GOSUB 400: PRINT
2135  HTAB HT + 4
2140 AN$ = "accept the entire answer": GOSUB 400: PRINT
2200  REM BOX
2220  HTAB HT + 2
2230  PRINT "_____"
2240  FOR I = 0 TO 4: HTAB HT + 1: PRINT "|
        |": NEXT : VTAB  PEEK (37)
2250  HTAB HT + 2: PRINT "_____"
2260  VTAB 19: HTAB HT + 4
2270 AN$ = "What is a " +  CHR$ (34) + "drift" +  CHR$ (34) + "?": GOSUB
     400: PRINT : PRINT
2280  HTAB HT + 5
2290  PRINT "> ";
2300  REM GET INPUT
2310 FL = 30: REM  LENGTH OF FIELD
2320 AN$ = "A whole lot of cattle": REM  THE "default" ANSWER (SUPPLIED B
     Y THE PROGRAM - THE USER CAN CHANGE IT)
2330  GOSUB 100: REM INPUT
2335  VTAB 24
2340  IF AN$ = "A herd of cattle" THEN AN$ = "Perfect! ": GOSUB 400
2350  IF AN$ = "A HERD of cattle" THEN AN$ = "Very good!": GOSUB 400
2360  RETURN
3000 AN$ = "Sample routine number 2": GOSUB 400
3999  RETURN
4000 AN$ = "Sample routine number 3": GOSUB 400
4999  RETURN
5000 AN$ = "Sample routine number 4": GOSUB 400
5999  RETURN
```

```
6000 AN$ = "Sample routine number 5": GOSUB 400
6999  RETURN
7000  REM END
7005  PRINT : REM DISK COMMANDS (SEE NEXT LINE) WILL NOT WORK IF THERE IS
      AN "OPEN" PRINT COMMAND SUCH AS A "PRINT;" OR "PRINT,".  THEREFORE,
      ALWAYS DO A "PRINT" BEFORE ISSUING A DOS COMMAND
7010  IF  PEEK (6) = 99 AND  PEEK (7) = 99 THEN  PRINT  CHR$ (4);"RUN DIS
    K MENU": END
7020  REM ABOVE LINE RUNS DISK MENU IF DISK MENU ASKED IT TO.  SEE NOTES
    FOLLOWING LINE 9060 OF DISK MENU.
7050  TEXT : HOME : TEXT : END : REM CLEAN UP & GO AWAY
9000  REM *HELP SCREENS*
9100  REM
9105 AN$ = "Option 1: the Apple IIe Input Routine"
9107  VTAB 1
9110  HTAB ( PEEK (33) -  LEN (AN$)) / 2
9115  GOSUB 400: PRINT : PRINT
9117  IF COL80 THEN  VTAB 7: REM A DIFFERENT SPACING LOOKS MORE PLEASING
    IN 80-COLUMN MODE
9120 AN$ = "The Apple IIe is equipped with a full ASCII keyboard, includi
    ng the DELETE key, a new key for the Apple II series.": GOSUB 400
9122  PRINT : IF COL80 THEN  PRINT : REM BY COINCIDENCE, THE ABOVE SENTEN
    CE, IN 40-COLUMN MODE, IS JUST LONG ENOUGH TO "force" AN EXTRA CARRI
    AGE RETURN.  SO, TO GET THE EXTRA SPACE IN 80-COLUMN MODE, WE MUST P
    RINT ONCE MORE
9125 AN$ = "Using the DELETE key and the flashing-cursor input routine in
    cluded within this program,": GOSUB 400
9130 AN$ = "you can scan forward and backward through your answers, inser
    ting and deleting characters at will.": GOSUB 400
9135 AN$ = "  The input routine automatically gives the standard Applesof
    t BASIC blinking cursor to users of the Apple II or II+.": GOSUB 400

9145  PRINT : PRINT
9150 AN$ = "The input routine, along with the other routines in MAGIC MEN
    U, will enable you to easily create humanized programs.": GOSUB 400
9155  GOSUB 400: PRINT : PRINT
9160 AN$ = "Since you are using an Apple II or II+, you cannot take advan
    tage of this cursor directly, but using it in your programs will mak
    e it easier for others."
9165  IF AIIE THEN AN$ = "Select option 1 to reacquaint yourself with the
     underline cursor."
9170  GOSUB 400
9180  IF COL80 THEN  PRINT : PRINT
9199  RETURN
9200 AN$ = "There is no description available for this option.": GOSUB 40
    0
9250  PRINT : VTAB 24
9299  RETURN
9300 AN$ = "There is no option available for this description.": GOSUB 40
    0
9350  PRINT : VTAB 24
9399  RETURN
9400 AN$ = "Sample help screen for option 4": GOSUB 400
9450  PRINT : VTAB 24
9499  RETURN
9500 AN$ = "Sample help screen for option 5": GOSUB 400
9550  PRINT : VTAB 24
9599  RETURN
9600 AN$ = "This option lets you gracefully exit the program.": GOSUB 400

9650  PRINT : VTAB 24
9699  RETURN
10000  RETURN
63000  REM  *** COMPUTER ID ROUTINE ***
63010  REM *** AIIE OR NOT? ***
63020  REM  USES I,J,K,RE  -- SETS AIIE TO 1 IF IT IS AN AIIE
63030  REM SETS RESULT DEPENDENT ON AVAILABLE HARDWARE
63040  REM  RESULTS OF 0 MEANS NOT AIIE; 32 MEANS AIIE BUT NO 80 COLUMNS;
      64 MEANS AIIE WITH 80 COLUMNS BUT NO AUX MEM; 128 MEANS AIIE WITH A
    UX MEM
```

```
63050  DATA  8, 120, 173, 0, 224, 141, 208, 2, 173, 0, 208, 141, 209, 2,
     173, 0, 212, 141, 210, 2, 173, 0, 216, 141, 211, 2, 173, 129, 192, 1
     73, 129, 192, 173, 179, 251, 201, 6, 208, 73, 173
63060  DATA  23, 192, 48, 60, 173, 19, 192, 48, 39, 173, 22, 192, 48, 34,
      160, 42, 190, 162, 3, 185, 0, 0, 150, 0, 153, 162, 3, 136, 208, 242
     , 76, 1, 0, 8, 160, 42, 185, 162, 3, 153
63070  DATA  0, 0, 136, 208, 247, 104, 176, 8, 169, 128, 141, 207, 3, 76,
      73, 3, 169, 64, 141, 207, 3, 76, 73, 3, 169, 32, 141, 207, 3, 76, 7
     3, 3, 169, 0, 141, 207, 3, 173, 0, 224
63080  DATA  205, 208, 2, 208, 24, 173, 0, 208, 205, 209, 2, 208, 16, 173
     , 0, 212, 205, 210, 2, 208, 8, 173, 0, 216, 205, 211, 2, 240, 56, 17
     3, 136, 192, 173, 0, 224, 205, 208, 2, 240, 6
63090  DATA  173, 128, 192, 76, 161, 3, 173, 0, 208, 205, 209, 2, 240, 6,
      173, 128, 192, 76, 161, 3, 173, 0, 212, 205, 210, 2, 240, 6, 173, 1
     28, 192, 76, 161, 3, 173, 0, 216, 205, 211, 2
63100  DATA  240, 3, 173, 128, 192, 40, 96, 169, 238, 141, 5, 192, 141, 3
     , 192, 141, 0, 8, 173, 0, 12, 201, 238, 208, 14, 14, 0, 12, 173, 0,
     8, 205, 0, 12, 208, 3, 56, 176, 1, 24
63110  DATA  141, 4, 192, 141, 2, 192, 76, 29, 3, 234
63120 J = 975:K = 724
63130  FOR I = 0 TO 249
63140  READ L
63150  POKE K + I,L
63160  NEXT
63170  CALL K
63180 RESULTS =  PEEK (J)
63190  IF RESULTS <  > 0 THEN AIIE = 1
63200  RETURN
```

DISK MENU

DISK MENU was written in two hours flat. In fact, it wasn't exactly written: the author used MAGIC MENU to create DISK MENU. (Soon you will be able to do the same!)

The process was pretty straightforward. After loading in MAGIC MENU, the author took out everything no longer needed, changed the names of the MENU$s, and polished off the submenus. All done.

Explore this program enough to become familiar with it. When you write your own programs using the MAGIC MENU package, use DISK MENU as a foundation. You can still refer to the MAGIC MENU listing for details, and you'll get the added speed and reduced size of the compressed DISK MENU version.

If you compare lines 100 to 900 in DISK MENU with the same lines in MAGIC MENU, you will notice a distinct change. The subroutine package has been compressed. Most REM statements have been removed. Any variable name greater than two characters long has been shortened to a minimum.

Line 1030 makes the program skip the initial instructions if DISK MENU has set locations 6 and 7 to 99, meaning it's not the user's first time through. See the discussion of line 9060 that follows.

Lines 1047 to 1075 are the text of DISK MENU. Because the Screen Formatter subroutine from MAGIC MENU automatically straightens out unformatted text, the author didn't have to be careful about line breaks when adding these lines. Otherwise, there is very little new and interesting in DISK MENU. This is exactly the point of using MAGIC MENU: a functional, useful program using standard inputs, clear instructions, and friendly menus was created in an extremely short period of time. It took more than a month of work to write the subroutine block package on which this program is based; it takes only a few hours to grow something out of it.

Line 3060 was actually included because the author wanted to test whether Menu Maker would handle such a long line properly.

Lines 6000 to 6999 are the little routine that simulates the typing of the CATALOG command. The command is actually issued in the normal manner by line 6080. DOS commands can be executed from within an Applesoft program by printing a string that consists of a CONTROL-D followed by the DOS command; since CHR$(4) is the ASCII code for CONTROL-D, and TITLE$ has been set to "CATALOG" in line 6030, line 6080 produces a CATALOG listing on the screen.

Line 9060 is the key to why the appendix programs all come back to DISK MENU when they are done. The problem is to communicate among programs with a flag: if the flag is set (if DISK MENU has been run) come back; if not, don't come back. One cannot use a standard variable for a flag between programs, because all variables are cleared to zero when a program is run. But many memory locations are not affected by Applesoft. In this case, the author chose locations 6 and 7, placing the arbitrary number 99 in each of them. The odds of both locations having 99 in them by chance are only 1 in 65,536 —pretty good odds.

The subroutine package in DISK MENU has been reduced to only a little over half its MAGIC MENU size, and has been considerably speeded up in the process. The DOS Programmer's Tool Kit offers programs that compress your software and enable it, at the same time, to run at maximum speed. Using such "utilities programs" gives you full reign to write code you can read and understand, while allowing you to end up with a compressed version that runs at maximum speed.

There are five kinds of utilities programs that are of particular use to Applesoft programmers:

1. Those that **compress** programs by taking out REM statements, by shortening variable names to two characters, and by combining statements that can be put together.

2. Those that **compile,** or translate, code into machine language so it will execute up to 100 times faster.

3. Those that **renumber** line numbers as well as GOSUB and GOTO line references. (See the next section for an explanation of this technique.)

4. Those that allow you to **edit** program lines more easily than escape mode does.

5. Those that provide **cross referencing,** telling you each and every variable name and where it is used, as well as which line numbers are called by other line numbers with GOSUB and GOTO statements.

Renumbering and Merging Program Parts

The exercise in this section uses a program on the DOS 3.3 SYSTEM MASTER disk called RENUMBER. This program was used by the author of DISK MENU to duplicate the MAGIC MENU menus so that only the contents of the strings had to be rewritten. After programming for a while, you will learn to avoid typing as much as you can; RENUMBER helps you do just that.

This example, and the longer one that follows in the next section, is for you to try out on your computer. Insert the DOS 3.3 SYSTEM MASTER disk into your disk drive if it is not already there. Then type

```
RUN RENUMBER
```

If you are using one disk drive, remove the SYSTEM MASTER disk and insert the APPLESOFT SAMPLER disk into the drive. If you are using two drives, the APPLESOFT SAMPLER can be placed in the second drive. (These instructions will presume you are using one disk drive. If you are using two, remember to add the ,D2 to the next command.) Type

```
LOAD DISK MENU
```

and then, when DISK MENU is loaded, type

```
DEL 0,1999
DEL 3000,63999
LIST
```

You will see that there isn't much left. Now, to renumber the portion of DISK MENU beginning at line 7000, type

```
& 7000
LIST
```

The menu is suddenly on new line numbers. RENUMBER does this. Using the ampersand (&) by itself would renumber the whole program, starting at line 10 and incrementing by 10. Using & 7000 renumbers the program beginning at line 7000 and incrementing by 10. Now to store it away, type

```
&H
```

If you list the program now, you'll find there isn't anything visible: it's on hold. &H tells DOS to hold something in the computer's main memory so that you can bring in another program on top of whatever is there. This tool is limited, of course, by the size of the programs you are working with and the size of the computer's memory.

Before you merge one program into another, make sure you are not deleting old lines. Type

```
LOAD DISK MENU
LIST 7000,7999
```

You shouldn't see any listing since those line numbers have not been used. You can therefore directly proceed to the process called merging. Type

```
&M
```

to merge the new lines into the old program. By changing the program names to new program names, including the new name of this submenu in the main menu, and giving the address of this new menu to the ON...GOSUB statement in line 1230, you could index up to 12 more programs on this disk. The author used this method (RENUMBER, &H, &M) three times to create the three submenus in the program.

RENUMBER is a very useful program to have in your computer whenever you are programming. It gives you flexibility. When you discover a routine getting so long that you can no longer get the big picture, you can break out subroutines and move them to other line numbers. RENUMBER overcomes the problem of having to fit an extra 11 lines in between lines 40 and 50: just move 50 and everything above it somewhere above 500 or 1000. That will give you all the room in the world. (See the *DOS Manual* for complete instructions for using RENUMBER.)

Program Listing

```
0  REM DISK MENU - SEPT 1982 - BY TOG
1  GOTO 1000
2  TEXT : PRINT  CHR$ (21): HOME : POKE 33,33: END
100  REM

          *** INPUT ROUTINE ***


140 I = 5:J = 0:K = 0:L = 0:M = 0:I$ = "":ESC = 0:0A = 0:0A$ = "":SA = 0:
    SA$ = "": IF FL = 0 THEN FL = 245
142  IF  NOT AIIE THEN  INPUT "";AN$: RETURN
145  PRINT AN$;:J =  LEN (AN$)
149 M =  PEEK (37)
153 L =  PEEK (36): IF COL80 THEN  IF L =  PEEK (1147) THEN L =  PEEK (14
    03)
155  PRINT " ";I$;" ";
160 N = 1: IF L +  LEN (I$) >  =  PEEK (33) - 3 AND  PEEK (37) =  PEEK (3
    5) - 1 THEN N = 0
165  POKE 36,L: POKE 1403,L: VTAB M + 1
170 I = I - 1: IF I < 0 THEN K = 1 - K:I = 5: PRINT  CHR$ (32 + 63 * K);
175  POKE 36,L: POKE 1403,L: VTAB M + 1
180 P =  PEEK ( - 16384): IF P < 128 THEN 170
185  IF  PEEK ( - 16287) > 127 THEN OAKEY = 1
190  IF  PEEK ( - 16286) > 127 THEN SAKEY = 1
195  POKE  - 16368,0:K = 0:I = 0
200  IF OAKEY THEN OA$ =  CHR$ (P - 128):AN$ = AN$ + I$: PRINT I$;" ": RETURN

205  IF SAKEY THEN SA$ =  CHR$ (P - 128):AN$ = AN$ + I$: PRINT I$;" ": RETURN

210  IF P > 159 AND P <  > 255 THEN  IF J +  LEN (I$) < FL THEN  IF N THEN
    AN$ = AN$ +  CHR$ (P - 128):J = J + 1: PRINT  CHR$ (P);: GOTO 149
215  IF P <  > 255 THEN 240
220  IF J THEN  PRINT " ";: POKE 36,L: VTAB M + 1: PRINT  CHR$ (136);:J =
    J - 1
225  IF J = 0 THEN AN$ = ""
230  IF J THEN AN$ =  LEFT$ (AN$,J)
235  GOTO 149
240  IF P <  > 136 THEN 265
245  IF J THEN  PRINT " ";: POKE 36,L: VTAB M + 1: PRINT  CHR$ (136);:I$ =
     RIGHT$ (AN$,1) + I$:J = J - 1
250  IF J = 0 THEN AN$ = ""
255  IF J THEN AN$ =  LEFT$ (AN$,J)
260  GOTO 149
265  IF P = 141 THEN AN$ = AN$ + I$: PRINT I$;" ": RETURN
270  IF P <  > 149 THEN 294
275  IF  NOT  LEN (I$) THEN 149
280 AN$ = AN$ +  LEFT$ (I$,1):J = J + 1: PRINT  LEFT$ (I$,1);
285  IF  LEN (I$) = 1 THEN I$ = ""
290  IF  LEN (I$) THEN I$ =  RIGHT$ (I$, LEN (I$) - 1)
292  GOTO 149
294  IF P = 155 THEN ESCKEY = 1: PRINT : RETURN
296  GOTO 149
300  REM

          ** GET RETURN **


310  IF AIIE THEN 325
315  GET AN$: IF  ASC (AN$) <  > 13 THEN 315
320  PRINT : RETURN
325 I = 0:J = 0:K = 0
330 I = I - 1: IF I < 0 THEN K = 1 - K:I = 5: PRINT  CHR$ (32 + 63 * K);
335  IF I <  > 5 THEN 355
340 L =  PEEK (36): IF COL80 THEN  IF L =  PEEK (1147) THEN L =  PEEK (14
    03)
345  IF L = 0 THEN  POKE 36, PEEK (33): VTAB  PEEK (37)
```

```
350  IF L <  > 0 THEN  POKE 36,L - 1
355 P =  PEEK ( - 16384): IF P <  > 141 THEN 330
360  PRINT " ";
365  IF  PEEK (37) = 23 THEN  VTAB 23
370  PRINT
375  POKE  - 16368,0: RETURN
400  REM

        *** SCREEN FORMATTER ***


410 I =  LEN (AN$): IF  NOT I THEN  RETURN
411 P =  PEEK (36): IF COL80 THEN  IF P =  PEEK (1147) THEN P =  PEEK (14
     03)
412  IF  NOT P THEN  IF I > 1 THEN  IF  ASC (AN$) = 32 THEN AN$ =  RIGHT$
     (AN$,I - 1)
413  IF P + 2 + I <  PEEK (33) AND AIIE THEN  PRINT AN$;" ";:AN$ = "": RETURN

414  IF I > 1 THEN  IF  RIGHT$ (AN$,1) = " " THEN AN$ =  LEFT$ (AN$,I - 1
     )
417  IF P + I <  PEEK (33) THEN I$ = AN$:AN$ = "": GOTO 440
420 J =  PEEK (33) - P + 2:I = J
425 I = I - 1: IF I THEN  IF  MID$ (AN$,I,1) <  > " " THEN 425
430  IF I = 1 THEN I = J
431  IF I = 0 THEN  PRINT : GOTO 410
435 I$ =  LEFT$ (AN$,I - 1): IF  LEN (AN$) > I THEN AN$ =  RIGHT$ (AN$, LEN
     (AN$) - I)
440  IF AIIE THEN  PRINT I$;
445  IF  NOT AIIE THEN K =  LEN (I$) + 1: FOR I = 1 TO  LEN (I$):J =  ASC
     ( RIGHT$ (I$,K - I)): PRINT  CHR$ (J - 32 * (J > 96 AND J < 123));: NEXT
     I
447 P =  PEEK (36): IF COL80 THEN  IF P =  PEEK (1147) THEN P =  PEEK (14
     03)
450  IF  LEN (AN$) THEN  IF P <  > 0 THEN  PRINT
455  IF  LEN (AN$) THEN 410
460  IF P <  > 0 THEN  IF  MID$ (I$, LEN (I$),1) <  > " " THEN  PRINT " "
     ;
465  RETURN
500  REM

        *** MENU MAKER ***


530 J = 0:K = 0:JJ = 0:KK = 0:LL = 0:MM = 0:NN = 0:00 = 0:AN = 0
535 00 = 0A
540 AN$ = ""
545 II = 1
550 KK = KK +  INT ( LEN (MENU$(II)) / (27 + 35 * (COL80 = 1)))
555  IF II < 12 THEN  IF MENU$(II + 1) <  > "End" AND MENU$(II + 1) <  >
     "end" AND MENU$(II + 1) <  > "END" THEN II = II + 1: GOTO 550
560 LL = II
565 NN = 0: IF LL * 2 + KK < 14 THEN NN = 1
570 MM = 3: IF COL80 THEN MM = 9
575 JJ = 3 + MM: IF LL > 9 THEN JJ = 4 + MM
590  TEXT : HOME
595 AN$ = TITLE$: GOSUB 400: POKE 36, PEEK (33) -  LEN (SUBTITLE$) - 1:AN
     $ = SUBTITLE$
600  GOSUB 400
605  FOR II = 1 TO  PEEK (33): PRINT "_";: NEXT : PRINT
615  FOR II = 1 TO LL
620  HTAB MM: PRINT II;". ";
625  VTAB  PEEK (37): IF COL80 THEN  VTAB  PEEK (1531)
630  POKE 32,JJ: POKE 33, PEEK (33) - JJ: PRINT
635 AN$ = MENU$(II): GOSUB 400
640  POKE 32,0: POKE 33, PEEK (33) + JJ: PRINT
645  IF NN THEN  PRINT
650  NEXT II
655  IF  PEEK (37) > 16 THEN  PRINT "TOO MANY MENU ITEMS OR TOO LONG LINE
     S.": STOP
```

```
660  TEXT : VTAB 17:AN$ = "Select option >": GOSUB 400: PRINT
665  FOR II = 1 TO  PEEK (33): PRINT " ";: NEXT
670  IF  NOT OO OR  NOT AIIE THEN  PRINT
675  IF  LEN (FROM$) THEN AN$ = "For " + FROM$ + ": press ESC": GOSUB 400

680  PRINT
685  IF AIIE THEN  PRINT "To erase: use the DELETE key"
690 AN$ = "To select: type a number from 1 to " +  STR$ (LL)
700  GOSUB 400: PRINT
705  IF  NOT OO THEN 720
710  IF AIIE THEN  PRINT "For descriptions: press OPEN-APPLE-?"
715  IF  NOT AIIE THEN  PRINT "FOR DESCRIPTIONS: FOLLOW ANSWER WITH ?"
720 AN$ = "To go to selected item: press RETURN": GOSUB 400: PRINT
723  IF  NOT OO THEN AN$ = "(There are no descriptions available)": GOSUB
     400
725 FL = 3:AN$ = ""
735  REM
740  VTAB 17: HTAB 17: CALL  - 868: HTAB 17
745  IF AIIE OR  NOT  LEN (FROM$) THEN  GOSUB 100: GOTO 795
750 I = O:J = O:M =  PEEK (37):L =  PEEK (36):OAKEY = O
755 I = I - 1: IF I < O THEN K = 1 - K:I = 5:J = 1 - J: NORMAL : IF J THEN
     INVERSE
760  PRINT " ";: POKE 36,L: VTAB M + 1
765 P =  PEEK ( - 16384): IF P < 128 THEN 755
770  NORMAL
775  IF P = 155 THEN  POKE  - 16368,O:ESCKEY = 1: GOTO 795
780 L =  PEEK (36): VTAB 20: HTAB 1: CALL  - 868: VTAB M + 1: HTAB L + 1
785  INPUT "";AN$
795  VTAB 19
800  IF  NOT OAKEY THEN 825
805  IF  NOT OO OR OA$ <  > "?" AND OA$ <  > "/" THEN 735
815 JJ =  VAL (AN$): IF JJ > O AND JJ <  = LL THEN II = JJ: GOTO 880
820  IF JJ = O THEN  PRINT "--> PLEASE SELECT A NUMBER FIRST <--";: CALL
      - 868: PRINT :AN$ = "": GOTO 735
825  IF SAKEY THEN 735
830  IF ESCKEY THEN  IF  LEN (FROM$) THEN 880
835 JJ =  VAL (AN$): IF JJ > O AND JJ <  = LL THEN II = JJ
840  IF  LEN (AN$) = O THEN 735
845  IF OO THEN  IF  NOT AIIE AND ( RIGHT$ (AN$,1) = "?" OR  RIGHT$ (AN$,
     1) = "/") THEN  IF II > O AND II <  = LL THEN OAKEY = 1: GOTO 880
850  IF  LEN (AN$) THEN K = O: FOR I = 1 TO  LEN (AN$):J =  ASC ( RIGHT$
     (AN$,I)):K = K + (J <  > 32 AND (J < 48 OR J > 57)): NEXT I: IF K THEN
      PRINT "      --> PLEASE USE DIGITS <--";: CALL  - 868: PRINT :AN$ =
     "": GOTO 735
860  IF  NOT JJ THEN  IF  LEN (AN$) THEN K = O: FOR I = 1 TO  LEN (AN$):J
      =  ASC ( RIGHT$ (AN$,I)):K = K + (J = 48): NEXT I: IF  NOT K THEN  PRINT
     "      --> PLEASE USE DIGITS <--";: CALL  - 868: PRINT :AN$ = "": GOTO
     735
870  IF JJ > LL THEN  PRINT "      --> ";JJ;" IS TOO LARGE <--";: CALL  -
     868: PRINT :AN$ = "": GOTO 735
875  IF JJ < 1 THEN  PRINT "      --> O IS TOO SMALL <--";: CALL  - 868:A
     N$ = "": GOTO 735
880 AN = II: TEXT : HOME : RETURN

1000  REM *** MAIN PROGRAM ***
1010 I = O:AN$ = "":J = O:I$ = "":K = O: REM DECLARE MOST OFTEN USED VARI
     ABLES FIRST FOR SPEED.
1020  GOSUB 63000: REM FIND OUT  IF AIIE OR NOT
1021  DIM ME$(20)
1025  IF RESULTS >  = 64 THEN COL80 = 1: PRINT  CHR$ (4);"PR#3": REM IF A
     N 80-COLUMN CARD IS PRESENT, USE IT.  IF YOU DON'T WANT IT, CHANGE T
     HE LINE TO:
               1025 PRINT CHR$(21)

1030  IF  PEEK (6) = 99 AND  PEEK (7) = 99 THEN 1100: REM  SEE NOTES FOLL
     OWING 9060
1035  TEXT : HOME : VTAB 2: PRINT
1040 AN$ = "*** THE APPLESOFT SAMPLER DISK ***": POKE 36,( PEEK (33) -  LEN
     (AN$)) / 2: GOSUB 400: PRINT
1045  VTAB 6 + 3 * (COL80 = 1): REM START ON LINE 6 UNLESS COL80=1, IN WH
     ICH CASE, START ON LINE 9
1047 AN$ = "Featuring": GOSUB 400
```

```
1050 AN$ = "the collection of programs to be studied in conjuction with t
     he Applesoft Tutorial, with a special appearance by ": GOSUB 400
1060 AN$ = "Postage Rates, the example program from the Applesoft Referen
     ce Manual.": GOSUB 400: PRINT : PRINT
1065 AN$ = "The following " +  CHR$ (34) + "main menu" +  CHR$ (34) + " l
     ets you select one of several " +  CHR$ (34) + "sub-menus" +  CHR$ (
     34) + " with the names of the programs on this disk.": GOSUB 400
1070 AN$ = "You can choose to pick a program from a sub-menu, or you can
     elect to end the": GOSUB 400
1075 AN$ = "program. ": GOSUB 400
1076  GOSUB 400
1080  PRINT : VTAB 24
1089 AN$ = "Press RETURN to go to the Main Menu.": GOSUB 400: GOSUB 300
1090  REM  ** MAIN MENU **
1100  REM ** MAIN MENU LOOP **
1110 TITLE$ = "The Applesoft Sampler Disk"
1120 SUBTITLE$ = "Main Menu": REM TITLE OF MENU OR SUBSECTION OF PROGRAM
1130 MENU$(1) = "Example Programs from The Applesoft Tutorial"
1140 MENU$(2) = "Tutorial Appendix E: More Programs To Play With"
1150 MENU$(3) = "Postage Rates -- From The Applesoft Reference Manual"
1180 MENU$(4) = "End the program (This option will type CATALOG for you a
     nd will leave you in Applesoft.)"
1190 MENU$(5) = "END": REM MENU PROGRAM KEEPS DISPLAYING ITEMS UNTIL IT F
     INDS THE WORD "end"
1195 OAKEY = 0: REM HELP NOT AVAILABLE; OPEN-APPLE KEY WILL NOT BE READ
1197 FROM$ = ""
1200  GOSUB 500
1215 AN$ = ""
1225  VTAB 8
1230  ON AN GOSUB 2000,3000,4000,6000: REM RETURN PRESSED, SO GO TO THE S
     ELECTED SECTION
1240  GOTO 1100
1250  REM


1999  REM BLANK LINES ARE DONE BY PRESSING THE DOWN-ARROW KEY




2000  REM *** EXAMPLES ***
2010  REM


2020 TITLE$ = "Applesoft Tutorial"
2030 SUBTITLE$ = "Examples"
2040 MENU$(1) = "COLORLOOP"
2045 MENU$(2) = "HUE"
2046 MENU$(3) = "QUILT"
2047 MENU$(4) = "SPACES"
2048 MENU$(5) = "COLORBOUNCE"
2049 MENU$(6) = "RANDOM"
2050 MENU$(7) = "HORSES"
2051 MENU$(8) = "MOIRE"
2100 MENU$(9) = "ALPHABET"
2110 MENU$(10) = "DECIMAL"
2120 MENU$(11) = "COLORBOUNCESOUND"
2140 MENU$(12) = "Return to Main Menu"
2160 FROM$ = "Main Menu":AN$ = "":OAKEY = 0: GOSUB 500
2170  IF ESCKEY THEN  RETURN
2180  IF AN = 12 THEN  RETURN
2190 TITLE$ = MENU$(AN): REM STORE NAME OF SELECTION FOR USE BY EACH ROUT
     INE
2195  PRINT  CHR$ (21)
2200  GOTO 9000
2210  GOTO 2000
2220  REM


3000  REM TUTOR EXAMPLES ***
3010  REM
```

```
3020 TITLE$ = "Applesoft Tutorial"
3030 SUBTITLE$ = "Samples"
3040 MENU$(1) = "SCRAMBLER"
3050 MENU$(2) = "MAGIC MENU"
3060 MENU$(3) = "DISK MENU (Which is this very program -- selecting it wi
     ll only result in a long wait followed by the program starting over.
       It is included here because it is included in Appendix E.)"
3070  IF  NOT COL80 THEN MENU$(2) = MENU$(2) +  CHR$ (10):MENU$(3) = MENU
     $(3) +  CHR$ (10): REM CHR$(10) IS DOWN-ARROW: ADDING DOWN-ARROWS WI
     LL ADD SPACES ABOVE AND BELOW MENU OPTION 3
3080  REM IN 80 COLUMN MODE, THERE IS ENOUGH ROOM THAT THE MENU MAKER ROU
     TINE WILL ADD SPACES AUTOMATICALLY, SO WE DO IT ONLY IF THE COMPUTER
      IS NOT IN 80-COLUMN MODE
3090 MENU$(4) = "CONVERTER"
3100 MENU$(5) = "Return to Main Menu"
3110 MENU$(6) = "end"
3120 FROM$ = "Main Menu":AN$ = "":OAKEY = 0: GOSUB 500
3130  IF ESCKEY THEN  RETURN
3140  IF AN = 5 THEN  RETURN
3150  IF AN = 1 AND  NOT AIIE THEN  HOME : VTAB 10: PRINT "SORRY, SCRAMBL
     ER CAN ONLY BE": PRINT "PLAYED ON AN APPLE IIE.": VTAB 24: PRINT "PR
     ESS RETURN TO GO BACK TO THE MENU.";: GOSUB 300: GOTO 3000: REM  "pa
     tch"
3160 TITLE$ = MENU$(AN): REM STORE NAME OF SELECTION FOR USE BY EACH ROUT
     INE
3165  IF AN = 2 THEN TITLE$ = "MAGIC MENU": REM A DOWN-ARROW MAY HAVE BEE
     N APPENDED IN LINE 3070
3170  IF AN = 3 THEN TITLE$ = "DISK MENU": REM A "PATCH" TO THE PROGRAM:
     MENU$(3) HAS AN EXTRA DESCRIPTION
3180  GOTO 9000
3190  REM


4000  REM *** REFERENCE MAN ***
4010  REM

4020 TITLE$ = "Applesoft Tutorial"
4030 SUBTITLE$ = "Example"
4040 MENU$(1) = "POSTAGE RATES"
4060 MENU$(2) = "Return to Main Menu"
4070 MENU$(3) = "end"
4080 FROM$ = "Main Menu":AN$ = "":OAKEY = 0: GOSUB 500
4090  IF ESCKEY THEN  RETURN
4100  IF AN = 2 THEN  RETURN
4110 TITLE$ = MENU$(AN): REM STORE NAME OF SELECTION FOR USE BY EACH ROUT
     INE
4120  GOTO 9000
4140  REM


5999  RETURN
6000  REM *** CATALOG ***
6010  HOME
6020  VTAB 10
6025  PRINT "]";
6030 TITLE$ = "CATALOG"
6040  FOR I = 1 TO  LEN (TITLE$)
6050  PRINT  MID$ (TITLE$,I,1);
6055 K =  PEEK ( - 16336) +  PEEK ( - 16336)
6060  FOR J = 1 TO 400 *  RND (1): NEXT
6070  NEXT I
6072  POKE 6,0: POKE 7,0: REM SEE NOTES FOLLOWING 9060
6075  PRINT
6080  PRINT  CHR$ (4);TITLE$
6999  END
```

```
 9000  REM *** GO TO DISK ***
 9060  POKE 6,99: POKE 7,99: REM LET THE PROGRAMS FROM APPENDIX E KNOW THA
     T THEY CAN RETURN TO THIS PROGRAM.  USUALLY, YOU USE A VARIABLE TO "
     TURN ON" OR "TURN OFF" AN OPTION
 9061  REM BUT VARIABLES ARE ALL CLEARED BETWEEN PROGRAMS.  LOCATIONS 6 &
     7 ARE NOT AFFECTED BY APPLESOFT, SO THE AUTHOR CHOSE THEM AS A WAY T
     O COMMUNICATE
 9075  PRINT
 9080  PRINT  CHR$ (4);"RUN ";TITLE$
 9090  END
63000  REM *** COMPUTER ID ***
63010  REM *** AIIE OR NOT? ***
63020  REM  USES I,J,K,RE  -- SETS AIIE TO 1 IF IT IS AN AIIE
63030  REM SETS RESULT DEPENDENT ON AVAILABLE HARDWARE
63040  REM RESULTS OF 0 MEANS NOT A //E; 32 MEANS A//E BUT NO 80 COLUMNS;
     64 MEANS A//E WITH 80 COLUMNS BUT NO AUX MEM; 128 MEANS A//E WITH A
     UX MEM
63050  DATA  8, 120, 173, 0, 224, 141, 208, 2, 173, 0, 208, 141, 209, 2,
     173, 0, 212, 141, 210, 2, 173, 0, 216, 141, 211, 2, 173, 129, 192, 1
     73, 129, 192, 173, 179, 251, 201, 6, 208, 73, 173
63060  DATA  23, 192, 48, 60, 173, 19, 192, 48, 39, 173, 22, 192, 48, 34,
     160, 42, 190, 162, 3, 185, 0, 0, 150, 0, 153, 162, 3, 136, 208, 242
     , 76, 1, 0, 8, 160, 42, 185, 162, 3, 153
63070  DATA  0, 0, 136, 208, 247, 104, 176, 8, 169, 128, 141, 207, 3, 76,
     73, 3, 169, 64, 141, 207, 3, 76, 73, 3, 169, 32, 141, 207, 3, 76, 7
     3, 3, 169, 0, 141, 207, 3, 173, 0, 224
63080  DATA  205, 208, 2, 208, 24, 173, 0, 208, 205, 209, 2, 208, 16, 173
     , 0, 212, 205, 210, 2, 208, 8, 173, 0, 216, 205, 211, 2, 240, 56, 17
     3, 136, 192, 173, 0, 224, 205, 208, 2, 240, 6
63090  DATA  173, 128, 192, 76, 161, 3, 173, 0, 208, 205, 209, 2, 240, 6,
     173, 128, 192, 76, 161, 3, 173, 0, 212, 205, 210, 2, 240, 6, 173, 1
     28, 192, 76, 161, 3, 173, 0, 216, 205, 211, 2
63100  DATA  240, 3, 173, 128, 192, 40, 96, 169, 238, 141, 5, 192, 141, 3
     , 192, 141, 0, 8, 173, 0, 12, 201, 238, 208, 14, 14, 0, 12, 173, 0,
     8, 205, 0, 12, 208, 3, 56, 176, 1, 24
63110  DATA  141, 4, 192, 141, 2, 192, 76, 29, 3, 234
63120 J = 975:K = 724
63130  FOR I = 0 TO 249
63140  READ L
63150  POKE K + I,L
63160  NEXT
63170  CALL K
63180 RESULTS =  PEEK (J)
63190  IF RESULTS <  > 0 THEN AIIE = 1
63200  RETURN
```

CONVERTER

CONVERTER stands on its own. It is based on the now-familiar subroutine package. It was derived from MAGIC MENU, and lies in wait for you to add the sorts of conversions you might find useful. It can provide the most valuable sort of programming experience: working intimately with a program written by experts.

When you add to CONVERTER, make good use of the RENUMBER program. Many conversions are done in very much the same way: only the text changes. Duplicate the lines and change the words.

Line 11145 is special: it is an example of the kind of care that can and should go into programming. Creating humanized programs that neither make fools of nor appear foolish to users is a strong, exciting challenge.

Program Listing

```
0  REM CONVERTER - SEPT, 1982 - BG & TOG
1  GOTO 2000
2  TEXT : PRINT  CHR$ (21): HOME : POKE 33,33: END
100  REM

          *** INPUT ROUTINE ***


140 I = 5:J = 0:K = 0:L = 0:M = 0:I$ = "":ESC = 0:0A = 0:0A$ = "":SA = 0:
    SA$ = "": IF FL = 0 THEN FL = 245
142  IF  NOT AIIE THEN  INPUT "";AN$: RETURN
145  PRINT AN$;:J =  LEN (AN$)
149 M =  PEEK (37)
153 L =  PEEK (36): IF COL80 THEN  IF L =  PEEK (1147) THEN L =  PEEK (14
    03)
155  PRINT " ";I$;" ";
160 N = 1: IF L +  LEN (I$) >  =  PEEK (33) - 3 AND  PEEK (37) =  PEEK (3
    5) - 1 THEN N = 0
165  POKE 36,L: POKE 1403,L: VTAB M + 1
170 I = I - 1: IF I < 0 THEN K = 1 - K:I = 5: PRINT  CHR$ (32 + 63 * K);
175  POKE 36,L: POKE 1403,L: VTAB M + 1
180 P =  PEEK ( - 16384): IF P < 128 THEN 170
185  IF  PEEK ( - 16287) > 127 THEN OAKEY = 1
190  IF  PEEK ( - 16286) > 127 THEN SAKEY = 1
195  POKE  - 16368,0:K = 0:I = 0
200  IF OAKEY THEN OA$ =  CHR$ (P - 128):AN$ = AN$ + I$: PRINT I$;" ": RETURN

205  IF SAKEY THEN SA$ =  CHR$ (P - 128):AN$ = AN$ + I$: PRINT I$;" ": RETURN

210  IF P > 159 AND P <  > 255 THEN  IF J +  LEN (I$) < FL THEN  IF N THEN
    AN$ = AN$ +  CHR$ (P - 128):J = J + 1: PRINT  CHR$ (P);: GOTO 149
215  IF P <  > 255 THEN 240
220  IF J THEN  PRINT " ";: POKE 36,L: VTAB M + 1: PRINT  CHR$ (136);:J =
    J - 1
225  IF J = 0 THEN AN$ = ""
230  IF J THEN AN$ =  LEFT$ (AN$,J)
235  GOTO 149
```

```
240  IF P <  > 136 THEN 265
245  IF J THEN  PRINT " ";: POKE 36,L: VTAB M + 1: PRINT  CHR$ (136);:I$ =
      RIGHT$ (AN$,1) + I$:J = J - 1
250  IF J = 0 THEN AN$ = ""
255  IF J THEN AN$ =  LEFT$ (AN$,J)
260  GOTO 149
265  IF P = 141 THEN AN$ = AN$ + I$: PRINT I$;" ": RETURN
270  IF P <  > 149 THEN 294
275  IF  NOT  LEN (I$) THEN 149
280 AN$ = AN$ +  LEFT$ (I$,1):J = J + 1: PRINT  LEFT$ (I$,1);
285  IF  LEN (I$) = 1 THEN I$ = ""
290  IF  LEN (I$) THEN I$ =  RIGHT$ (I$, LEN (I$) - 1)
292  GOTO 149
294  IF P = 155 THEN ESCKEY = 1: PRINT : RETURN
296  GOTO 149
300  REM

         ** GET RETURN **


310  IF AIIE THEN 325
315  GET AN$: IF  ASC (AN$) <  > 13 THEN 315
320  PRINT : RETURN
325 I = 0:J = 0:K = 0
330 I = I - 1: IF I < 0 THEN K = 1 - K:I = 5: PRINT  CHR$ (32 + 63 * K);
335  IF I <  > 5 THEN 355
340 L =  PEEK (36): IF COL80 THEN  IF L =  PEEK (1147) THEN L =  PEEK (14
     03)
345  IF L = 0 THEN  POKE 36, PEEK (33): VTAB  PEEK (37)
350  IF L <  > 0 THEN  POKE 36,L - 1
355 P =  PEEK ( - 16384): IF P <  > 141 THEN 330
360  PRINT " ";
365  IF  PEEK (37) = 23 THEN  VTAB 23
370  PRINT
375  POKE  - 16368,0: RETURN
400  REM

        *** SCREEN FORMATTER ***


410 I =  LEN (AN$): IF  NOT I THEN  RETURN
411 P =  PEEK (36): IF COL80 THEN  IF P =  PEEK (1147) THEN P =  PEEK (14
     03)
412  IF  NOT P THEN  IF I > 1 THEN  IF  ASC (AN$) = 32 THEN AN$ =  RIGHT$
     (AN$,I - 1)
413  IF P + 2 + I <  PEEK (33) AND AIIE THEN  PRINT AN$;" ";:AN$ = "": RETURN

414  IF I > 1 THEN  IF  RIGHT$ (AN$,1) = " " THEN AN$ =  LEFT$ (AN$,I - 1
     )
417  IF P + I <  PEEK (33) THEN I$ = AN$:AN$ = "": GOTO 440
420 J =  PEEK (33) - P + 2:I = J
425 I = I - 1: IF I THEN  IF  MID$ (AN$,I,1) <  > " " THEN 425
430  IF I = 1 THEN I = J
431  IF I = 0 THEN  PRINT : GOTO 410
435 I$ =  LEFT$ (AN$,I - 1): IF  LEN (AN$) > I THEN AN$ =  RIGHT$ (AN$, LEN
     (AN$) - I)
440  IF AIIE THEN  PRINT I$;
445  IF  NOT AIIE THEN K =  LEN (I$) + 1: FOR I = 1 TO  LEN (I$):J =  ASC
     ( RIGHT$ (I$,K - I)): PRINT  CHR$ (J - 32 * (J > 96 AND J < 123));: NEXT
     I
447 P =  PEEK (36): IF COL80 THEN  IF P =  PEEK (1147) THEN P =  PEEK (14
     03)
450  IF  LEN (AN$) THEN  IF P <  > 0 THEN  PRINT
455  IF  LEN (AN$) THEN 410
460  IF P <  > 0 THEN  IF  MID$ (I$, LEN (I$),1) <  > " " THEN  PRINT " "
     ;
465  RETURN
500  REM

        *** MENU MAKER ***
```

```
530 J = 0:K = 0:JJ = 0:KK = 0:LL = 0:MM = 0:NN = 0:00 = 0:AN = 0
535 00 = 0A
540 AN$ = ""
545 II = 1
550 KK = KK +  INT ( LEN (MENU$(II)) / (27 + 35 * (COL80 = 1)))
555  IF II < 12 THEN  IF MENU$(II + 1) <  > "End" AND MENU$(II + 1) <  >
     "end" AND MENU$(II + 1) <  > "END" THEN II = II + 1: GOTO 550
560 LL = II
565 NN = 0: IF LL * 2 + KK < 14 THEN NN = 1
570 MM = 3: IF COL80 THEN MM = 9
575 JJ = 3 + MM: IF LL > 9 THEN JJ = 4 + MM
590  TEXT : HOME
595 AN$ = TITLE$: GOSUB 400: POKE 36, PEEK (33) -  LEN (SUBTITLE$) - 1:AN
    $ = SUBTITLE$
600  GOSUB 400
605  FOR II = 1 TO  PEEK (33): PRINT "_";: NEXT : PRINT
615  FOR II = 1 TO LL
620  HTAB MM: PRINT II;". ";
625  VTAB  PEEK (37): IF COL80 THEN  VTAB  PEEK (1531)
630  POKE 32,JJ: POKE 33, PEEK (33) - JJ: PRINT
635 AN$ = MENU$(II): GOSUB 400
640  POKE 32,0: POKE 33, PEEK (33) + JJ: PRINT
645  IF NN THEN  PRINT
650  NEXT II
655  IF  PEEK (37) > 16 THEN  PRINT "TOO MANY MENU ITEMS OR TOO LONG LINE
    S.": STOP
660  TEXT : VTAB 17:AN$ = "Select option >": GOSUB 400: PRINT
665  FOR II = 1 TO  PEEK (33): PRINT "_";: NEXT
670  IF  NOT 00 OR  NOT AIIE THEN  PRINT
675  IF  LEN (FROM$) THEN AN$ = "For " + FROM$ + ": press ESC": GOSUB 400

680  PRINT
685  IF AIIE THEN  PRINT "To erase: use the DELETE key"
690 AN$ = "To select: type a number from 1 to " +  STR$ (LL)
700  GOSUB 400: PRINT
705  IF  NOT 00 THEN 720
710  IF AIIE THEN  PRINT "For descriptions: press OPEN-APPLE-?"
715  IF  NOT AIIE THEN  PRINT "FOR DESCRIPTIONS: FOLLOW ANSWER WITH ?"
720 AN$ = "To go to selected item: press RETURN": GOSUB 400: PRINT
723  IF  NOT 00 THEN AN$ = "(There are no descriptions available)": GOSUB
    400
725 FL = 3:AN$ = ""
735  REM
740  VTAB 17: HTAB 17: CALL  - 868: HTAB 17
745  IF AIIE OR  NOT  LEN (FROM$) THEN  GOSUB 100: GOTO 795
750 I = 0:J = 0:M =  PEEK (37):L =  PEEK (36):OAKEY = 0
755 I = I - 1: IF I < 0 THEN K = 1 - K:I = 5:J = 1 - J: NORMAL : IF J THEN
     INVERSE
760  PRINT " ";: POKE 36,L: VTAB M + 1
765 P =  PEEK ( - 16384): IF P < 128 THEN 755
770  NORMAL
775  IF P = 155 THEN  POKE  - 16368,0:ESCKEY = 1: GOTO 795
780 L =  PEEK (36): VTAB 20: HTAB 1: CALL  - 868: VTAB M + 1: HTAB L + 1
785  INPUT "";AN$
795  VTAB 19
800  IF  NOT OAKEY THEN 825
805  IF  NOT 00 OR 0A$ <  > "?" AND 0A$ <  > "/" THEN 735
815 JJ =  VAL (AN$): IF JJ > 0 AND JJ <  = LL THEN II = JJ: GOTO 880
820  IF JJ = 0 THEN  PRINT "--> PLEASE SELECT A NUMBER FIRST <--";: CALL
     - 868: PRINT :AN$ = "": GOTO 735
825  IF SAKEY THEN 735
830  IF ESCKEY THEN  IF  LEN (FROM$) THEN 880
835 JJ =  VAL (AN$): IF JJ > 0 AND JJ <  = LL THEN II = JJ
840  IF  LEN (AN$) = 0 THEN 735
845  IF 00 THEN  IF  NOT AIIE AND ( RIGHT$ (AN$,1) = "?" OR  RIGHT$ (AN$,
    1) = "/") THEN  IF II > 0 AND II <  = LL THEN OAKEY = 1: GOTO 880
850  IF  LEN (AN$) THEN K = 0: FOR I = 1 TO  LEN (AN$):J =  ASC ( RIGHT$
    (AN$,I)):K = K + (J <  > 32 AND (J < 48 OR J > 57)): NEXT I: IF K THEN
     PRINT "      --> PLEASE USE DIGITS <--";: CALL  - 868: PRINT :AN$ =
    "": GOTO 735
```

```
860  IF  NOT JJ THEN  IF  LEN (AN$) THEN K = 0: FOR I = 1 TO  LEN (AN$):J
      =  ASC ( RIGHT$ (AN$,I)):K = K + (J = 48): NEXT I: IF  NOT K THEN  PRINT
     "      --> PLEASE USE DIGITS <--";: CALL  - 868: PRINT :AN$ = "": GOTO
     735
870  IF JJ > LL THEN  PRINT "      --> ";JJ;" IS TOO LARGE <--";: CALL  -
     868: PRINT :AN$ = "": GOTO 735
875  IF JJ < 1 THEN  PRINT "      --> 0 IS TOO SMALL <--";: CALL  - 868:A
     N$ = "": GOTO 735
880 AN = II: TEXT : HOME : RETURN
1000  REM *** SUBROUTINES ***
1010  REM SOME SPECIAL ROUTINES FOR THIS PROGRAM
1100  REM Y OR N?
1110  REM PUT NAME OF CALLING ROUTINE IN AN$
1120  REM   UPON RETURN, AN WILL BE 1 IF YES, 0 IF NO
1130  REM USES AN, AN$, I, II$
1135  PRINT
1140 II$ = "Do you want to do another " + AN$ + " conversion? (Y OR N) "
1145  HTAB 1: VTAB 22: CALL  - 958:AN$ = II$: GOSUB 400:FL = 2: GOSUB 100

1150 I = 0: IF  LEN (AN$) THEN I =  ASC (AN$): IF I <  >  ASC ("Y") AND I
      <  >  ASC ("y") AND I <  >  ASC ("N") AND I <  >  ASC ("n") THEN 11
     45
1155  IF  NOT I THEN 1145
1160 AN = 0: IF I =  ASC ("y") OR I =  ASC ("Y") THEN AN = 1
1170  RETURN
2000  REM *** BEGIN PROGRAM ***
2010  REM THE CONVERTER - A PROGRAM SKELETON USED FOR MEASUREMENT CONVERS
     IONS - 1982
2020  GOSUB 63000: REM SET AIIE TO TRUE (1) OR FALSE (0)
2025  IF RESULTS >  = 64 THEN COL80 = 1: PRINT  CHR$ (4);"PR#3": REM IF A
     N 80-COLUMN CARD IS PRESENT, USE IT.  IF YOU DON'T WANT IT, CHANGE T
     HE LINE TO:
               1025 PRINT CHR$(21)

2030  TEXT : PRINT : HOME
2040 I = 0:AN$ = "":J = 0:I$ = "":K = 0
2050  DIM MENU$(12)
2070 AN$ = "*** Converter ***": POKE 36,( PEEK (33) -  LEN (AN$)) / 2: GOSUB
     400: PRINT : REM CENTER TITLE
2080  PRINT : PRINT : IF COL80 THEN  VTAB 6
2090 AN$ = "Converter is a program to convert"
2095  GOSUB 400
2100 AN$ = "measurements from one form to another,": GOSUB 400
2110 AN$ = "such as kilometers to miles.": GOSUB 400: PRINT : PRINT
2120 AN$ = "The program is structured to provide hundreds of conversions,
      but it is now in skeleton form": GOSUB 400
2130 AN$ = "-- that is, only a few sample conversions are provided.  The
     intent": GOSUB 400
2140 AN$ = "is for you to learn, through studying the program listings an
     d the given samples, how to alter the program": GOSUB 400
2150 AN$ = "to provide you the conversions you will find useful.": GOSUB
     400: PRINT : PRINT
2160 AN$ = "Through this learning process, you will gain insight into alt
     ering other programs to suit your needs.": GOSUB 400
2220  PRINT
2230  VTAB 23
2240 AN$ = "Press RETURN to continue.": GOSUB 400: GOSUB 300
2250  HOME
2260  REM


2270  REM *** MAIN MENU ***
2280 TITLE$ = "Converter":SUBTITLE$ = "Main Menu"
2285 FROM$ = "": IF  PEEK (6) = 99 AND  PEEK (7) = 99 THEN FROM$ = "the D
     isk Menu": REM SEE NOTES FOLLOWING LINE 9060 IN THE DISK MENU PROGRAM
2290 MENU$(1) = "LINEAR MEASURES"
2300 MENU$(2) = "TEMPERATURE MEASURES"
2310 MENU$(3) = "SPEED MEASURES"
2320 MENU$(4) = ""
2330 MENU$(5) = ""
2340 MENU$(6) = ""
```

```
2350 MENU$(7) = ""
2360 MENU$(8) = ""
2370 MENU$(9) = ""
2380 MENU$(10) = ""
2390 MENU$(11) = ""
2400 MENU$(12) = "End the program"
2405  IF  LEN (FROM$) THEN MENU$(12) = "End program and go to Disk Menu"
2410  REM

2430 AN$ = ""
2440 OAKEY = 0: REM NO HELP AVAILABLE
2450  GOSUB 500: REM DO MENU
2455  IF ESCKEY THEN  GOTO 22000: REM SEE NOTES FOLLOWING LINE 9060 IN TH
     E DISK MENU PROGRAM
2460  ON AN GOSUB 11000,12000,13020,14000,15000,16000,17000,18000,19000,2
     0000,21000,22000
2470  GOTO 2270
10999  REM

11000  REM *** LINEAR MEASURE
11001  REM

11002 TITLE$ = "Conversions"
11005 SUBTITLE$ = "Change linear measure"
11010 MENU$(1) = "Kilometers to Miles"
11020 MENU$(2) = "Miles to Kilometers"
11030 MENU$(3) = "Return to Main Menu"
11035 MENU$(4) = "end"
11040 FROM$ = "Main Menu":AN$ = "":OAKEY = 0: GOSUB 500
11050  IF ESCKEY THEN  RETURN
11060  IF AN = 3 THEN  RETURN
11065 TITLE$ = MENU$(AN): REM STORE NAME OF SELECTION FOR USE BY EACH ROU
     TINE
11070  ON AN GOSUB 11100,11200
11080  GOTO 11000
11099  REM

11100  REM ** KILOMETERS --> MILES
11110  TEXT : HOME : PRINT :AN$ = "Convert Kilometers to Miles": GOSUB 40
     0: PRINT : REM FUNCTION TITLE
11120  VTAB 10:AN$ = "Enter number of kilometers ": GOSUB 400:AN$ = "":FL
      = 10: GOSUB 100
11130 AN =  VAL (AN$): IF AN = 0 AND AN$ <  > "0" THEN  PRINT  CHR$ (7): GOTO
     11110: REM ONLY NUMBERS ALLOWED
11140 MILE = AN * .62137
11145 I$ = " mile": IF  INT (MILE * 10000) / 10000 <  > 1 THEN I$ = " mil
     es"
11146 J$ = " kilometer": IF AN <  > 1 THEN J$ = " kilometers
11150  PRINT :AN$ =  STR$ (AN) + J$ + " = " +  STR$ (MILE) + I$: GOSUB 40
     0
11160 AN$ = TITLE$: GOSUB 1100
11170  IF AN THEN 11100
11190  RETURN
11199  REM


11200  REM ** MILES --> KILOMETERS
11210  TEXT : HOME : PRINT :AN$ = "Convert Miles to Kilometers": GOSUB 40
     0: PRINT : REM FUNCTION TITLE
11220  VTAB 10:AN$ = "Enter number of miles ": GOSUB 400:AN$ = "":FL = 10
     : GOSUB 100
11230 AN =  VAL (AN$): IF AN = 0 AND AN$ <  > "0" THEN  PRINT  CHR$ (7): GOTO
     11210: REM ONLY NUMBERS ALLOWED
11240 KILO = AN / .62137
11245 I$ = " mile": IF AN <  > 1 THEN I$ = " miles"
11246 J$ = " kilometer": IF  INT (KILO * 10000) / 10000 <  > 1 THEN J$ =
     " kilometers"
11250  PRINT :AN$ =  STR$ (AN) + I$ + " = " +  STR$ (KILO) + J$: GOSUB 40
     0
11260 AN$ = TITLE$: GOSUB 1100
```

```
11270  IF AN THEN 11200
11280  RETURN
11290  REM


11300  RETURN : REM PROGRAMMER TO CREATE CONVERSION ROUTINE HERE
12000  REM *** TEMPERATURE ***
12010  REM

12020 TITLE$ = "Conversions"
12030 SUBTITLE$ = "Change temperature measure"
12040 MENU$(1) = "Fahrenheit to Celsius"
12050 MENU$(2) = "Celsius to Fahrenheit"
12060 MENU$(3) = "Return to Main Menu"
12070 MENU$(4) = "end"
12080 FROM$ = "Main Menu":AN$ = "":OAKEY = 0: GOSUB 500
12090  IF ESCKEY THEN  RETURN
12100  IF AN = 3 THEN  RETURN
12110 TITLE$ = MENU$(AN): REM STORE NAME OF SELECTION FOR USE BY EACH ROU
     TINE
12120  ON AN GOSUB 12200,12400
12130  GOTO 12000
12140  REM


12200  REM ** FAHRENHEIT --> CELSIUS
12210  TEXT : HOME : PRINT :AN$ = "Convert Fahrenheit to Celsius": GOSUB
     400: PRINT : REM FUNCTION TITLE
12220  VTAB 10:AN$ = "Enter degrees Fahrenheit ": GOSUB 400:AN$ = "":FL =
     10: GOSUB 100
12230 AN =  VAL (AN$): IF AN = 0 AND AN$ <  > "0" THEN  PRINT  CHR$ (7): GOTO
     12210: REM ONLY NUMBERS ALLOWED
12235  IF AN <  - 459.6 THEN  PRINT :AN$ = "One cannot have a temperature
      below Absolute 0 (-459.6 degrees Fahrenheit)": GOSUB 400: GOTO 1248
     0
12240 CEL = (AN - 32) / 1.8
12250 I$ = " degree": IF  INT (CEL * 10000) / 10000 <  > 1 THEN I$ = " de
     grees"
12260 J$ = " degree": IF AN <  > 1 THEN J$ = " degrees"
12270  PRINT :AN$ =  STR$ (AN) + J$ + " Fahrenheit = " +  STR$ (CEL) + I$
      + " Celsius": GOSUB 400
12280 AN$ = TITLE$: GOSUB 1100
12290  IF AN THEN 12200
12300  RETURN
12310  REM


12400  REM ** CELSIUS --> FAHRENHEIT
12410  TEXT : HOME : PRINT :AN$ = "Convert Celsius to Fahrenheit": GOSUB
     400: PRINT : REM FUNCTION TITLE
12420  VTAB 10:AN$ = "Enter degrees Celsius": GOSUB 400:AN$ = "":FL = 10:
      GOSUB 100
12430 AN =  VAL (AN$): IF AN = 0 AND AN$ <  > "0" THEN  PRINT  CHR$ (7): GOTO
     12410: REM ONLY NUMBERS ALLOWED
12435  IF AN <  - 273.1 THEN  PRINT :AN$ = "One cannot have a temperature
      below Absolute 0 (-273.1 degrees Celsius)": GOSUB 400: GOTO 12480
12440 FAHR = AN * 1.8 + 32
12450 I$ = " degree": IF AN <  > 1 THEN I$ = " degrees"
12460 J$ = " degree": IF  INT (FAHR * 10000) / 10000 <  > 1 THEN J$ = " d
     egrees"
12470  PRINT :AN$ =  STR$ (AN) + I$ + " Celsius = " +  STR$ (FAHR) + J$ +
     " Fahrenheit": GOSUB 400
12480 AN$ = TITLE$: GOSUB 1100
12490  IF AN THEN 12400
12500  RETURN
12510  REM


12600  RETURN : REM PROGRAMMER TO CREATE KELVIN CONVERSION ROUTINE HERE
13000  REM *** SPEED ***
13010  REM
```

```
13020 TITLE$ = "Conversions"
13030 SUBTITLE$ = "Change speed measure"
13040 MENU$(1) = "Kilometers/hour to miles/hour"
13050 MENU$(2) = "Miles/hour to kilometers/hour
13060 MENU$(3) = "Return to Main Menu"
13070 MENU$(4) = "end"
13080 FROM$ = "Main Menu":AN$ = "":OAKEY = 0: GOSUB 500
13090  IF ESCKEY THEN  RETURN
13100  IF AN = 3 THEN  RETURN
13110 TITLE$ = MENU$(AN): REM STORE NAME OF SELECTION FOR USE BY EACH ROU
     TINE
13120  ON AN GOSUB 13200,13400
13130  GOTO 13000
13140  REM


13200  REM ** KILO/HR --> MILES/HR
13210  TEXT : HOME : PRINT :AN$ = "Convert Kilometers per hour to Miles p
     er hour": GOSUB 400: PRINT : REM FUNCTION TITLE
13220  VTAB 10:AN$ = "Enter kilometers per hour ": GOSUB 400:AN$ = "":FL =
     10: GOSUB 100
13230 AN =  VAL (AN$): IF AN = 0 AND AN$ <  > "0" THEN  PRINT  CHR$ (7): GOTO
     13210: REM ONLY NUMBERS ALLOWED
13240 MILE = AN * .62137
13250 I$ = " mile": IF  INT (MILE * 10000) / 10000 <  > 1 THEN I$ = " mil
     es"
13260 J$ = " kilometer": IF AN <  > 1 THEN J$ = " kilometers
13270  PRINT :AN$ =  STR$ (AN) + J$ + " per hour = " +  STR$ (MILE) + I$ +
     " per hour": GOSUB 400
13280 AN$ = TITLE$: GOSUB 1100
13290  IF AN THEN 13200
13300  RETURN
13310  REM


13400  REM  ** MILES /HR --> KILO/HR
13410  TEXT : HOME : PRINT :AN$ = "Convert Miles per hour to kilometers p
     er hour": GOSUB 400: PRINT : REM FUNCTION TITLE
13420  VTAB 10:AN$ = "Enter number of mph ": GOSUB 400:AN$ = "":FL = 10: GOSUB
     100
13430 AN =  VAL (AN$): IF AN = 0 AND AN$ <  > "0" THEN  PRINT  CHR$ (7): GOTO
     13410: REM ONLY NUMBERS ALLOWED
13440 KILO = AN / .62137
13450 I$ = " mile": IF AN <  > 1 THEN I$ = " miles"
13460 J$ = " kilometer": IF  INT (KILO * 10000) / 10000 <  > 1 THEN J$ =
     " kilometers"
13470  PRINT :AN$ =  STR$ (AN) + I$ + " per hour = " +  STR$ (KILO) + J$ +
     " per hour": GOSUB 400
13480 AN$ = TITLE$: GOSUB 1100
13490  IF AN THEN 13400
13500  RETURN
13600  REM


13601  RETURN : REM PROGRAMMER TO CREATE CONVERSION ROUTINE HERE
14000  REM *** SELECTION 4 ***
14999  RETURN
15000  REM *** SELECTION 5 ***
15999  RETURN
16000  REM *** SELECTION 6 ***
16999  RETURN
17000  REM *** SELECTION 7 ***
17999  RETURN
18000  REM *** SELECTION 8 ***
18999  RETURN
19000  REM *** SELECTION 9 ***
19999  RETURN
20000  REM *** SELECTION 10 ***
20999  RETURN
21000  REM *** SELECTION 11 ***
```

```
21999  RETURN
22000  REM *** SELECTION 12 ***
22010  REM *** END THE PROGRAM ***

22020  IF  PEEK (6) = 99 AND  PEEK (7) = 99 THEN  PRINT : PRINT  CHR$ (4)
     ;"RUN HELLO": END
22050  TEXT : HOME : TEXT : END
63000  REM *** COMPUTER ID ***
63010  REM *** AIIE OR NOT? ***
63020  REM  USES I,J,K,RE  -- SETS AIIE TO 1 IF IT IS AN AIIE
63030  REM SETS RESULT DEPENDENT ON AVAILABLE HARDWARE
63040  REM RESULTS OF 0 MEANS NOT A //E; 32 MEANS A//E BUT NO 80 COLUMNS;
      64 MEANS A//E WITH 80 COLUMNS BUT NO AUX MEM; 128 MEANS A//E WITH A
     UX MEM
63050  DATA  8, 120, 173, 0, 224, 141, 208, 2, 173, 0, 208, 141, 209, 2,
     173, 0, 212, 141, 210, 2, 173, 0, 216, 141, 211, 2, 173, 129, 192, 1
     73, 129, 192, 173, 179, 251, 201, 6, 208, 73, 173
63060  DATA  23, 192, 48, 60, 173, 19, 192, 48, 39, 173, 22, 192, 48, 34,
      160, 42, 190, 162, 3, 185, 0, 0, 150, 0, 153, 162, 3, 136, 208, 242
     , 76, 1, 0, 8, 160, 42, 185, 162, 3, 153
63070  DATA  0, 0, 136, 208, 247, 104, 176, 8, 169, 128, 141, 207, 3, 76,
      73, 3, 169, 64, 141, 207, 3, 76, 73, 3, 169, 32, 141, 207, 3, 76, 7
     3, 3, 169, 0, 141, 207, 3, 173, 0, 224
63080  DATA  205, 208, 2, 208, 24, 173, 0, 208, 205, 209, 2, 208, 16, 173
     , 0, 212, 205, 210, 2, 208, 8, 173, 0, 216, 205, 211, 2, 240, 56, 17
     3, 136, 192, 173, 0, 224, 205, 208, 2, 240, 6
63090  DATA  173, 128, 192, 76, 161, 3, 173, 0, 208, 205, 209, 2, 240, 6,
      173, 128, 192, 76, 161, 3, 173, 0, 212, 205, 210, 2, 240, 6, 173, 1
     28, 192, 76, 161, 3, 173, 0, 216, 205, 211, 2
63100  DATA  240, 3, 173, 128, 192, 40, 96, 169, 238, 141, 5, 192, 141, 3
     , 192, 141, 0, 8, 173, 0, 12, 201, 238, 208, 14, 14, 0, 12, 173, 0,
     8, 205, 0, 12, 208, 3, 56, 176, 1, 24
63110  DATA  141, 4, 192, 141, 2, 192, 76, 29, 3, 234
63120 J = 975:K = 724
63130  FOR I = 0 TO 249
63140  READ L
63150  POKE K + I,L
63160  NEXT
63170  CALL K
63180 RESULTS =  PEEK (J)
63190  IF RESULTS <  > 0 THEN AIIE = 1
63200  RETURN
```

Some Final Thoughts

We hope you have enjoyed this excursion into the depths of large programs. If you feel a bit overwhelmed, take heart. You have been given a tremendous amount of new information in the space of a few hours. Examine these programs again in a few days or a week. Look at programs by other authors. Each time you do, you will discover some new trick or technique that you can incorporate into your own programs.

Glossary

address: A number used to identify something, such as a location in the computer's memory.

algorithm: A step-by-step procedure for solving a problem or accomplishing a task.

Apple IIe: A personal computer in the Apple II family, manufactured and sold by Apple Computer.

Apple IIe 80-Column Text Card: A peripheral card made and sold by Apple Computer that plugs into the Apple IIe's auxiliary slot and converts the computer's display of text from 40- to 80-column width.

Apple IIe Extended 80-Column Text Card: A peripheral card made and sold by Apple Computer that plugs into the Apple IIe's auxiliary slot and converts the computer's display of text from 40- to 80-column width while extending its memory capacity by 64K bytes.

Applesoft: An extended version of the BASIC programming language used with the Apple IIe computer and capable of processing numbers in floating-point form. An interpreter for creating and executing programs in Applesoft is built into the Apple IIe system in firmware. Compare **Integer BASIC**.

application program: A program that puts the resources and capabilities of the computer to use for some specific purpose or task, such as word processing, data-base management, graphics, or telecommunications. Compare **system program**.

argument: The value on which a function operates. An argument can be a number or a variable and is contained in parentheses that follow the function.

arithmetic expression: A combination of numbers and arithmetic operators (such as 3 + 5) that indicates some operation to be carried out.

arithmetic operations: The five actions Applesoft can perform with numbers are addition, subtraction, multiplication, division, and exponentiation.

arithmetic operator: An operator, such as +, that combines numeric values to produce a numeric result; compare **relational operator**, **logical operator**.

array: A collection of variables referred to by the same name and distinguished by means of numeric subscripts. Each variable in the array can be addressed independently by using that variable's unique subscript.

ASCII: American Standard Code for Information Interchange; a code in which the numbers from 0 to 127 stand for text characters, including the digits 0 through 9, the letters of the alphabet, punctuation marks, special characters, and control characters. The code is used for representing text inside a computer and for transmitting text between computers or between a computer and a peripheral device.

assembly language: A low-level programming language in which individual machine-language instructions are written in a symbolic form more easily understood by a human programmer than machine language itself.

BASIC: Beginner's All-purpose Symbolic Instruction Code; a high-level programming language designed to be easy to learn and use. Two versions of BASIC are available from Apple Computer for use with the Apple IIe: Applesoft (built into the Apple IIe in firmware) and Integer BASIC (provided on the DOS 3.3 SYSTEM MASTER disk).

binary: The representation of numbers in terms of powers of two, using the two digits 0 and 1. Commonly used in computers, since the values 0 and 1 can easily be represented in physical form in a variety of ways, such as the presence or absence of current, positive or negative voltage, or a white or black dot on the display screen.

binary operator: An operator that combines two operands to produce a result; for example + is a binary arithmetic operator, < is a binary relational operator, and OR is a binary logical operator. Compare **unary operator.**

bit: A binary digit (0 or 1); the smallest possible unit of information that a computer can hold, consisting of a simple two-way choice, such as yes or no, on or off, positive or negative, something or nothing.

boot: To start up a computer by loading a program into memory from an external storage medium such as a disk. Often accomplished by first loading a small program whose purpose is to read the larger program into memory. The program is said to "pull itself in by its own bootstraps"; hence the term *bootstrapping* or *booting*.

boot disk: See **startup disk**.

bootstrap: See **boot**.

branch: To send program execution to a line or statement other than the next in sequence.

buffer: An area of the computer's memory reserved for a specific purpose, such as to hold graphic information to be displayed on the screen or text characters being read from some peripheral device. Often used as an intermediary "holding area" for transferring information between devices operating at different speeds, such as the computer's processor and a printer or disk drive. Information can be stored into the buffer by one device and then read out by the other at a different speed.

bug: An error in a program that causes it not to work as intended.

byte: A unit of information consisting of a fixed number of bits; on the Apple IIe, one byte consists of eight bits and can hold any value from 0 to 255. Each character in the ASCII code can be represented by one byte, with an extra bit left over.

call: To request the execution of a subroutine or function.

catalog: A list of all files stored on a disk; sometimes called a *directory.*

central processing unit: See **processor**.

character: A letter, digit, punctuation mark, or other written symbol used in printing or displaying information in a form readable by humans.

character limit: The maximum number of characters allowed in a single Applesoft statement: 255.

chip: The small piece of semiconducting material (usually silicon) on which an integrated circuit is fabricated. The word chip properly refers only to the piece of silicon itself, but is often used for an integrated circuit and its package; see **integrated circuit**.

code: (1) A number or symbol used to represent some piece of information in a compact or easily processed form. (2) The statements or instructions making up a program.

cold start: The process of starting up the Apple IIe when the power is first turned on (or as if the power had just been turned on) by loading the operating system into main memory, then loading and running a program. Compare **warm start**.

column: A vertical arrangement of graphics points or character spaces on the screen.

command: A communication from the user to a computer system (usually typed from the keyboard) directing it to perform some immediate action.

computer: An electronic device for performing predefined (programmed) computations at high speed and with great accuracy. A machine that is used to store, transfer, and transform information.

computer language: See programming language.

computer system: A computer and its associated hardware, firmware, and software.

concatenate: Literally, "to chain together"; to combine two or more strings into a single, longer string containing all the characters in the original strings.

conditional branch: A branch that depends on the truth of a condition or the value of an expression; compare **unconditional branch**.

control: The order in which the statements of a program are executed.

control character: A character that controls or modifies the way information is printed or displayed. Control characters have ASCII codes between 0 and 31 and are typed from the Apple IIe keyboard by holding down the CONTROL key while typing some other character. For example, the character CONTROL-M (ASCII code 13) means "return to the beginning of the line" and is equivalent to the RETURN key.

CPU: Central processing unit; see **processor**.

crash: To cease operating unexpectedly, possibly damaging or destroying information in the process.

CRT: See **cathode-ray tube**.

cursor: A marker or symbol displayed on the screen that marks where the user's next action will take effect or where the next character typed from the keyboard will appear. Cursors are usually represented by a white box, an underline, or a flashing checkerboard box.

data: Information; especially information used or operated on by a program.

debug: To locate and correct an error or the cause of a problem or malfunction in a computer system. Typically used to refer to software-related problems; compare **troubleshoot**.

decimal: The common form of number representation used in everyday life, in which numbers are expressed in terms of powers of ten, using the ten digits 0 to 9.

default: (1) A value, action, or setting that is automatically used by a computer system when no other explicit information has been given. For example, if a command to run a program from a disk does not identify which disk drive to use, the Disk Operating System will automatically use the same drive that was used in the last operation.

deferred execution: The saving of an Applesoft program line for execution at a later time as part of a complete program; occurs when the line is typed with a line number. Compare **immediate execution**.

delimiter: A character that is used for punctuation to mark the beginning or end of a sequence of characters, and which therefore is not considered part of the sequence itself. For example, Applesoft uses the double quotation mark (") as a delimiter for string constants: the string "DOG" consists of the three characters D, O, and G, and does not include the quotation marks. In written English, the space character is used as a delimiter between words.

digit: (1) One of the characters 0 to 9, used to express numbers in decimal form. (2) One of the characters used to express numbers in some other form, such as 0 and 1 in binary or 0 to 9 and A to F in hexadecimal.

dimension: the maximum size of one of the subscripts of an array.

directory: A list of all files stored on a disk; sometimes called a *catalog.*

disk: An information storage medium consisting of a flat, circular magnetic surface on which information can be recorded in the form of small magnetized spots, similarly to the way sounds are recorded on tape.

disk drive: A peripheral device that writes and reads information on the surface of a magnetic disk.

diskette: A term sometimes used for the small (5-1/4-inch) flexible disks used with the Apple Disk II drive.

Disk II drive: A model of disk drive made and sold by Apple Computer for use with the Apple IIe computer; uses 5-1/4-inch flexible ("floppy") disks.

Disk Operating System: An optional software system for the Apple IIe that enables the computer to control and communicate with one or more Disk II drives.

display: (1) Information exhibited visually, especially on the screen of a display device. (2) To exhibit information visually. (3) A display device.

display screen: The glass or plastic panel on the front of a display device on which images are displayed.

DOS: See **Disk Operating System**.

edit: To change or modify; for example, to insert, remove, replace, or move text in a document.

element: A member of a set or collection; specifically, one of the individual variables making up an array. See also **subscript.**

error message: A message displayed or printed to notify the user of an error or problem in the execution of a program.

escape mode: A state of the Apple IIe computer, entered by pressing the ESC key, in which certain keys on the keyboard take on special meanings for positioning the cursor and controlling the display of text on the screen.

escape sequence: A sequence of keystrokes beginning with the ESC key, used for positioning the cursor and controlling the display of text on the screen.

execute: To perform or carry out a specified action or sequence of actions, such as those described by a program.

expression: A formula in a program describing a calculation to be performed.

file: A collection of information stored as a named unit on a peripheral storage medium such as a disk.

file name: The name under which a file is stored.

firmware: Those components of a computer system consisting of programs stored permanently in read-only memory. Such programs (for example, the Applesoft interpreter and the Apple IIe Monitor program) are built into the computer at the factory; they can be executed at any time but cannot be modified or erased from main memory. Compare **hardware**, **software**.

fixed-point: A method of representing numbers inside the computer in which the decimal point (more correctly, the binary point) is considered to occur at a fixed position within the number. Typically, the point is considered to lie at the right end of the number, so that the number is interpreted as an integer. Fixed-point numbers of a given length cover a narrower range than floating-point numbers of the same length, but with greater precision. Compare **floating-point**.

flexible disk: A disk made of flexible plastic; often called a "floppy" disk.

floating-point: A method of representing numbers inside the computer in which the decimal point (more correctly, the binary point) is permitted to "float" to different positions within the number. Some of the bits within the number itself are used to keep track of the point's position. Floating-point numbers of a given length cover a wider range than fixed-point numbers of the same length, but with less precision. Compare **fixed-point**.

floppy disk: See **flexible disk**.

format: (1) The form in which information is organized or presented. (2) To specify or control the format of information. (3) To prepare a blank disk to receive information by dividing its surface into tracks and sectors; also *initialize*.

function: A preprogrammed calculation that can be carried out on request from any point in a program.

graphics: (1) Information presented in the form of pictures or images. (2) The display of pictures or images on a computer's display screen. Compare **text**.

hang: For a program or system to "spin its wheels" indefinitely, performing no useful work.

hardware: Those components of a computer system consisting of physical (electronic or mechanical) devices. Compare **software**, **firmware**.

hexadecimal: The representation of numbers in terms of powers of sixteen, using the sixteen digits `0` to `9` and `A` to `F`. Hexadecimal numbers are easier for humans to read and understand than binary numbers, but can be converted easily and directly to binary form: each hexadecimal digit corresponds to a sequence of four binary digits, or bits.

high-level language: A programming language that is relatively easy for humans to understand. FORTRAN, BASIC, and Pascal are all examples of high-level languages. A single statement in a high-level language typically corresponds to several instructions of machine language.

high-resolution graphics: The display of graphics on the Apple IIe's display screen as a six-color array of points, 280 columns wide and 192 rows high. When the text window is in use, the visible high-resolution graphics display is 280 by 160 plotting points.

immediate execution: The execution of an Applesoft program line as soon as it is typed; occurs when the line is typed without a line number. This is a particular advantage of Applesoft, which is not available in many other programming languages. It means that you can try out nearly every statement immediately to see how it works. Compare **deferred execution**.

infinite loop: A section of a program that will repeat the same sequence of actions indefinitely.

information: Facts, concepts, or instructions represented in an organized form.

initialize: (1) To set to an initial state or value in preparation for some computation. (2) To prepare a blank disk to receive information by dividing its surface into tracks and sectors; also *format*.

input: (1) Information transferred into a computer from some external source, such as the keyboard, a disk drive, or a modem. (2) The act or process of transferring such information.

integer: A whole number, with no fractional part; represented inside the computer in fixed-point form. Compare **real number**.

Integer BASIC: A version of the BASIC programming language used with the Apple II family of computers; older than Applesoft and capable of processing numbers in integer (fixed-point) form only. An interpreter for creating and executing programs in Integer BASIC is included on the DOS 3.3 SYSTEM MASTER disk, and is automatically loaded into the computer's memory when the computer is started up with that disk. Compare **Applesoft**.

interface: The devices, rules, or conventions by which one component of a system communicates with another.

inverse video: The display of text on the computer's display screen in the form of black dots on a white (or other single phosphor color) background, instead of the usual white dots on a black background.

I/O: Input/output; the transfer of information into and out of a computer. See **input**, **output**.

K: Two to the tenth power, or 1024 (from the Greek root kilo, meaning one thousand); for example, 64K equals 64 times 1024, or 65,536.

keyboard: The set of keys built into the Apple IIe computer, similar to a typewriter keyboard, for typing information to the computer.

keystroke: The act of pressing a single key or a combination of keys (such as CONTROL-C) on the Apple IIe keyboard.

keyword: A special word or sequence of characters that identifies a particular type of statement or command, such as RUN or PRINT.

kilobyte: A unit of information consisting of 1K (1024) bytes, or 8K (8192) bits; see **K**.

language: See **programming language**.

line: See **program line**.

line number: A number identifying a program line in an Applesoft program. Line numbers are necessary for deferred execution.

load: To transfer information from a peripheral storage medium (such as a disk) into main memory for use; for example, to transfer a program into memory for execution.

location: See **memory location**.

logical operator: An operator, such as AND, that combines logical values to produce a logical result; compare **arithmetic operator**, **relational operator**.

loop: A section of a program that is executed repeatedly until the limit is met.

low-level language: A programming language that is relatively close to the form that the computer's processor can execute directly. Assembly language is an example.

low-resolution graphics: The display of graphics on the Apple IIe's display screen as a 16-color array of blocks, 40 columns wide and 48 rows high. When the text window is in use, the visible low-resolution graphics display is 40 by 40 plotting points.

machine language: The form in which instructions to a computer are stored in memory for direct execution by the computer's processor. Each model of computer processor (such as the 6502 microprocessor used in the Apple IIe) has its own form of machine language.

main memory: The memory component of a computer system that is built into the computer itself and whose contents are directly accessible to the processor.

memory: A hardware component of a computer system that can store information for later retrieval; see **main memory**, **random-access memory, read-only memory, read-write memory.**

memory location: A unit of main memory that is identified by an address and can hold a single item of information of a fixed size; in the Apple IIe, a memory location holds one byte, or eight bits, of information.

menu: A list of choices presented by a program, usually on the display screen, from which the user can select.

mode: A state of a computer or system that determines its behavior.

monitor: See **video monitor**.

Monitor program: A system program built into the Apple IIe firmware, used for directly inspecting or changing the contents of main memory and for operating the computer at the machine-language level.

nested loop: A loop contained within the body of another loop and executed repeatedly during each pass through the containing loop.

normal: Video display format made up of white (or single color) dots on a black background. (See **inverse.**)

null string: A string containing no characters.

operating system: A software system that organizes the computer's resources and capabilities and makes them available to the user or to application programs running on the computer.

operator: A symbol or sequence of characters, such as + or `AND`, specifying an operation to be performed on one or more values to produce a result; see **arithmetic operator**, **relational operator**, **logical operator**, **unary operator**, **binary operator**.

output: (1) Information transferred from a computer to some external destination, such as the display screen, a disk drive, a printer, or a modem. (2) The act or process of transferring such information.

page: (1) A screenful of information on a video display, consisting on the Apple IIe of 24 lines of 40 or 80 characters each. (2) An area of main memory containing text or graphic information being displayed on the screen.

peripheral: At or outside the boundaries of the computer itself, either physically (as a *peripheral device*) or in a logical sense (as a *peripheral card*).

precedence: The order in which operators are applied in evaluating an expression.

processor: The hardware component of a computer that performs the actual computation by directly executing instructions represented in machine language and stored in main memory.

program: (1) A set of instructions, conforming to the rules and conventions of a particular programming language, describing actions for a computer to perform in order to accomplish some task. In Applesoft, a sequence of program lines, each with a different line number. (2) To write a program.

program line: The basic unit of an Applesoft program, consisting of one or more statements separated by colons (:).

programmer: The human author of a program; one who writes programs.

programming: The activity of writing programs.

programming language: A set of rules or conventions for writing programs.

prompt: To remind or signal the user that some action is expected, typically by displaying a distinctive symbol, a reminder message, or a menu of choices on the display screen.

prompt character: A text character displayed on the screen to prompt the user for some action. Often also identifies the program or component of the system that is doing the prompting; for example, the prompt character] is used by the Applesoft BASIC interpreter, > by Integer BASIC, and * by the system Monitor program. Also called *prompting character.*

prompt message: A message displayed on the screen to prompt the user for some action. Also called *prompting message.*

radio-frequency modulator: A device for converting the video signals produced by a computer to a form that can be accepted by a television set.

RAM: See **random-access memory**.

random-access memory: Memory in which the contents of individual locations can be referred to in an arbitrary or random order. This term is often used incorrectly to refer to read-write memory, but strictly speaking both read-only and read-write memory can be accessed in random order. Random-access means that each unit of storage has a unique address and a method by which each unit can be immediately read from or written to. Compare **read-only memory, read-write memory.**

read: To transfer information into the computer's memory from a source external to the computer (such as a disk drive or modem) or into the computer's processor from a source external to the processor (such as the keyboard or main memory).

read-only memory: Memory whose contents can be read but not written; used for storing firmware. Information is written into read-only memory once, during manufacture; it then remains there permanently, even when the computer's power is turned off, and can never be erased or changed. Compare **read-write memory, random-access memory.**

read-write memory: Memory whose contents can be both read and written; often misleadingly called random-access memory, or RAM. The information contained in read-write memory is erased when the computer's power is turned off, and is permanently lost unless it has been saved on a more permanent storage medium, such as a disk. Compare **read-only memory, random-access memory.**

real number: A number that may include a fractional part; represented inside the computer in floating-point form. Compare **integer.**

relational operator: An operator, such as >, that compares numeric values to produce a logical result; compare **arithmetic operator, logical operator.**

reserved word: A word or sequence of characters reserved by a programming language for some special use, and therefore unavailable as a variable name in a program.

RF modulator: See **radio-frequency modulator.**

ROM: See **read-only memory.**

routine: A part of a program that accomplishes some task subordinate to the overall task of the program.

row: A horizontal arrangement of character spaces or graphics points on the screen.

run: (1) To execute a program. (2) To load a program into main memory from a peripheral storage medium, such as a disk, and execute it.

save: To transfer information from main memory to a peripheral storage medium for later use.

scientific notation: A method of expressing numbers in terms of powers of ten, useful for expressing numbers that may vary over a wide range, from very small to very large. For example, the number of atoms in a gram of hydrogen is approximately `6.02E23`, meaning 6.02 times ten to the 23rd power. (The letter `E` stands for "exponent.") The number is easier to understand in this form than in the form `602000000000000000000000`. Applesoft uses this method to display real precision numbers with more than nine digits.

screen: See **display screen**.

scroll: To change the contents of all or part of the display screen by shifting information out at one end (most often the top) to make room for new information appearing at the other end (most often the bottom), producing an effect like that of moving a scroll of paper past a fixed viewing window. See **viewport**, **window**.

simple variable: A variable that is not an element of an array.

software: Those components of a computer system consisting of programs that determine or control the behavior of the computer. Compare **hardware**, **firmware**.

space character: A text character whose printed representation is a blank space, typed from the keyboard by pressing the SPACE bar.

startup disk: A disk containing software recorded in the proper form to be loaded into the Apple IIe's memory to set the system into operation. Sometimes called a *boot disk*; see **boot**.

statement: A unit of a program in a high-level language specifying an action for the computer to perform, typically corresponding to several instructions of machine language.

step value: The amount by which a variable changes on each pass through a loop.

string: An item of information consisting of a sequence of test characters. Both `" flipping frogs endanger widow's mite "` and `"(A + B) / 973 = 14"` are strings.

subroutine: A part of a program that can be executed on request from any point in the program, and which returns control to the point of the request on completion.

subscript: An index number used to identify a particular element of an array.

syntax: The rules governing the structure of statements or instructions in a programming language.

television set: A display device capable of receiving broadcast video signals (such as commercial television) by means of an antenna. Can be used in combination with a radio-frequency modulator as a display device for the Apple IIe computer. Compare **video monitor**.

text: (1) Information presented in the form of characters readable by humans. (2) The display of characters on the Apple IIe's display screen. Compare **graphics**.

text window: An area on the Apple IIe's display screen within which text is displayed and scrolled.

unary operator: An operator that applies to a single operand; for example, the minus sign (-) in a negative number such as -6 is a unary arithmetic operator. Compare **binary operator**.

unconditional branch: A branch that does not depend on the truth of any condition; compare **conditional branch**.

user: The person operating or controlling a computer system.

value: An item of information that can be stored in a variable, such as a number or a string.

variable: (1) A location in the computer's memory where a value can be stored. (2) The symbol used in a program to represent such a location; compare **constant**.

video: (1) A medium for transmitting information in the form of images to be displayed on the screen of a cathode-ray tube. (2) Information organized or transmitted in video form.

video monitor: A display device capable of receiving video signals by direct connection only, and which cannot receive broadcast signals such as commercial television. Can be connected directly to the Apple IIe computer as a display device. Compare **television set**.

viewport: All or part of the display screen, used by an application program to display a portion of the information (such as a document, picture, or worksheet) that the program is working on. Compare **window**.

warm start: The process of restarting the Apple IIe after the power is already on, without reloading the operating system into main memory and often without losing the program or information already in main memory. Compare **cold start**.

window: The portion of a collection of information (such as a document, picture, or worksheet) that is visible in a viewport on the display screen; compare **viewport**.

wraparound: The automatic continuation of text from the end of one line to the beginning of the next, as on the display screen or a printer.

write: To transfer information from the computer to a destination external to the computer (such as a disk drive, printer, or modem) or from the computer's processor to a destination external to the processor (such as main memory).

write-enable notch: The square cutout in one edge of a disk's jacket that permits information to be written on the disk. If there is no write-enable notch, or if it is covered with a write-protect tab, information can be read from the disk but not written onto it.

write-protect: To protect the information on a disk by covering the write-enable notch with a write-protect tab, preventing any new information from being written onto the disk.

write-protect tab: A small adhesive sticker, usually silver, used to write-protect a disk by covering the write-enable notch.

Index

Note: Boldface page numbers indicate figures.

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

030-0358-A