nibble Collector's Edition

Hands On Applesoft

A Beginner's Guide to Applesoft BASIC

Hands On Applesoft

by Leslie R. Schmeltz and the Editors of Nibble Magazine

Nibble Publications
MicroSPARC, Inc.
Concord, MA

Editorial Direction: David A. Krathwohl
Copy Editors: Mary Lou Jordan and Mary Locke
Technical Editors: Jan Eugenides, Rich Williams and Owen Linzmayer
Cover Design: Paul J. Gagnon
Cover Art: Richard Cowdrey
Printed and bound by CSA Press, Hudson, MA

ISBN 0-912341-26-2

Printed in the United States of America

1 2 3 4 5 6 7 8 9

87 88 89 90 91 92

Introduction

Applesoft BASIC is built into every Apple II computer ever sold — over four million machines. It's undeniably the best way to introduce yourself to computer programming. Working with it is fast and immediate; you just switch on your Apple, and it's there! Whether your goal is to tinker, to gain more control over your Apple, or to become a professional programmer, Applesoft's a great place to start.

Applesoft consists of a small number of commands that are easily learned. Many books for Applesoft beginners are content to merely describe the commands, in the hopes that you'll figure out for yourself how to string them together. Not so this one. You'll also learn the art of creating Applesoft programs, from start to finish, with techniques used by the best programmers in any language.

You'll find that we go into great detail about bugs. Here's why: despite your best efforts, you'll undoubtedly run into program errors that make your programs behave unexpectedly. You'll wail, you'll gnash your teeth, but with the help of our chapters on debugging, you'll ultimately feel the immense satisfaction of exterminating all of the bugs from your programs and being left with polished gems.

Disk operations are an integral part of most Applesoft programs, but few books on the subject teach them effectively. In *Hands On Applesoft,* you'll find easily-understood, detailed instructions for mastering disk storage techniques.

For your convenience, we've assembled the sample programs from *Hands On Applesoft* on disk. We've also included two valuable tools for Applesoft programmers. Applesoft Turbo Editor is like a word processor for writing programs, enabling you to insert, delete and change lines and characters with ease. And Applesoft Tutor adds "training wheels" to your computer, actually modifying the Applesoft language to simplify debugging. Tutor expands Applesoft's standard (and somewhat cryptic) error messages to help you pinpoint program problems. Both programs are described in the Appendix, and you can order them using the card bound into the front of this book.

Your programming adventure is about to begin. So go immediately to your Apple, switch it on, turn to page one, and start programming! You may never stop.

Table of Contents

Listings

Chapter 1: Getting Started

Are you the "new person on the block" when it comes to knowing about your Apple? Do you get frustrated trying to understand articles describing advanced programming techniques? Does the computer store salesperson slough off your elementary programming questions with a terse, "It's in the manual. Look it up!"? Are you afraid to ask a "silly" question at a user group meeting? Well, friend, take heart. We've all been there. Very few Apple owners have had any experience writing programs prior to purchasing their computers. Even experienced users who have never ventured beyond the bounds of commercially available programs pale at the prospect of having to write an original program.

Writing programs for the Apple is not difficult. Once you learn to talk to your computer in a language it understands, programming becomes a simple matter of telling the machine what you would like it to do. *Hands On Applesoft* is written especially for the beginning Applesoft BASIC programmer. You will learn the three elementary aspects of Applesoft programming:

1. The basic concepts necessary to outline a program.
2. The vocabulary necessary to communicate with your Apple.
3. The process used to translate a program idea into a working program.

Each chapter focuses on a specific aspect of Applesoft and discusses relevant commands. You are encouraged to learn by doing, using examples and sample programs. Very few of the more exotic command combinations are mentioned. Our goal is to give you a solid foundation on which to build as your interest in programming continues.

We assume that you have access to an Apple II series computer (II Plus, IIe, IIc or IIGS). Memory size will be relatively unimportant for most of the sample programs. Disk storage as well as some of the peripherals you may want to manage will be discussed. The instructions provided with your Apple outline the process of loading commercial programs, accessing Applesoft, and gaining at least a cursory knowledge of how your system works. Rather than duplicate information you already have, let's build from there.

All About PRINT

Before we examine an Applesoft BASIC program, let's take a look at some of the words (or statements) that BASIC uses to communicate with your computer. These will provide the building blocks we'll use to create our first program. One statement that's used by virtually every BASIC program is PRINT.

PRINT is the command Applesoft uses to display information. (Computer types would say PRINT is the primary "output" command.) Numerous qualifiers may be used with the PRINT command to create screen displays that are both interesting and informative. Qualifiers are numbers, letters or commands that tell the computer what to PRINT or how to PRINT. A program line that contains PRINT and qualifiers is called a PRINT statement.

PRINT statements specify both the data and the format in which the data will be displayed. Printing starts at the location of the cursor (that little flashing box) and proceeds from left to right across the screen. Normally, the Apple has 40 print positions on each horizontal line and can display 24 lines vertically on the screen. After the entire screen has been filled, the topmost line "scrolls" off the screen, and a new line appears at the bottom. The Apple IIe (with 80-column card), IIc, and IIGS also have an 80-column display mode, which can be activated by typing:

```
PR#3
```

and pressing Return. For the sake of simplicity, the examples that follow use only 40 columns. However, most of the examples will work identically in the 80-column mode.

Does PRINT do anything by itself? Try it and see. Type PRINT and press the Return key.

Try it again. Each time you type PRINT and press Return, the cursor moves down a line on the screen. It looks as if you skipped a line (double-spaced) when typing on a typewriter. As a matter of fact, the word Return is a throwback to typewriters, where pressing the Carriage Return key would return you to the left margin. On computers, the process of advancing one vertical line is known as a linefeed. To make a simple action sound complicated, we could say that PRINT initiates a carriage return and a linefeed.

By the way, if you would like to save a little typing, Applesoft recognizes the question mark (provided it is not enclosed by quotation marks) as an abbreviation for the word PRINT. However, we always write it out in this book. In each instance where PRINT is called for, you may simply type a question mark.

So far, we have noted each time you needed to press Return after typing a statement or command. Since that action should be fairly automatic by now, just assume that you need to press Return at the conclusion of each line.

By typing PRINT followed by Return, you have instructed the computer to execute the command immediately. In computerese, these types of statements are referred to as immediate execution commands. If a statement is preceded by a line number, it will then be interpreted as a deferred execution command: The command will not be executed until the program is run.

In order to visualize the difference, try typing:

```
10 PRINT
```

and press Return. Did anything happen? Now type RUN (remember to press Return.) RUN told the computer to execute the deferred execution command PRINT. You have, in effect, created a one-line program. An Applesoft BASIC program is made up of a sequence of numbered lines. Each line may contain one or more statements that are executed as the computer "reads" the program. As the program is executed, BASIC uses various statements to obtain information, to manipulate it, and to display the result. As you'll learn later, some of the statements compare information and use the results of their comparison to change the sequence of execution. Try expanding the program by adding several more lines:

```
20 PRINT
30 PRINT
40 PRINT
```

Now type RUN again. As you see, your program now produced four blank lines instead of one. To take another look at the program you've just created, type the command LIST and press Return. This will display the entire program, from the lowest line number to the highest. As you can see, typing a program line (with a line number) causes the line to be stored in your Apple's memory. The LIST command displays the program currently in memory.

You can combine statements on a single line to create a program that has the same function as our four-line program. First type the command NEW to erase the program currently in memory. Then type:

```
10 PRINT: PRINT: PRINT: PRINT
RUN
```

The cursor should move down four lines just as it did before. In either immediate or deferred execution modes, you can use multiple statements in one line. There are some restrictions, though: Statements must be separated by colons, and the total length of the line may not exceed 239 characters.

You will need to use the two housekeeping commands, NEW and LIST, throughout the book as you type in the sample programs. If you want to see the program currently in memory, type LIST. To make sure you have a clean slate, you should always type NEW before entering a sample program.

PRINTING CHARACTERS

The main purpose of the PRINT statement is to let you display characters on the screen. Depending on the nature of the data to be displayed, you must add certain qualifiers to the basic PRINT command. For instance, type:

```
PRINT 53721
```

Your Apple will display on the screen:

```
53721
```

As you can see, numeric data may be displayed by typing PRINT followed by the numbers. However, the PRINT command will also perform computations and display their results. If you were to type:

```
PRINT 5 + 3 + 7 + 2 + 1
```

the sum of the numbers, rather than the numbers themselves, would be shown on the screen. Try it. Now try a few more. Type:

```
PRINT (5 + 3 + 7)/(2 + 1)
PRINT 5 + (7 * 3) + 2 + 1
PRINT 5 + (7 - 3) * (2 + 1)
```

In each case, the result displayed should be different.

The numbers, parentheses and operators (+, -, / and * for addition, subtraction, division and multiplication, respectively) in combination are called expressions, and the proper term for what we have been doing is "evaluating expressions."

Printing letters is slightly more complicated. A group of characters containing a mixture of letters and/or numbers, known as a string literal in computer circles, may be displayed by enclosing it in quotation marks. Try adding a few explanatory words to our earlier example by typing:

```
NEW
10 PRINT "THE NUMBERS ARE "
20 PRINT 53721
RUN
```

Your display should look like this:

```
THE NUMBERS ARE
53721
```

You have just been introduced to a string. A string is a group of characters enclosed by quotation marks. Now let's get a little fancier. Type:

```
NEW
10 PRINT "THE SUM OF 5 + 3 + 7 + 2 + 1 IS "
20 PRINT 5 + 3 + 7 + 2 + 1
RUN
```

Line 10 printed a string and **line 20** evaluated an expression.

PRINT WITH COMMAS

Adding commas to a PRINT statement is the equivalent of using a tab stop on a typewriter. On the 40-column Apple screen, the first tab field consists of 16 character positions beginning at the extreme left (position 1). The second tab field includes positions 17 to 32; the third field includes positions 33 to 40 (extreme right). Type:

```
PRINT 5,3,7
```

You should see one number at the beginning position of each of the three tab fields, like this:

```
5               3               7
```

Type:

```
PRINT 5,3,7,2,1
```

Note that the additional characters are displayed in the first two tab positions on the next line.

You can add commas to PRINT statements contained within program lines in the same way. Type:

```
10 PRINT 5,3,7,2,1
RUN
```

The screen should be exactly the same as it was when you used the previous immediate execution statement. An exception to this rule (there's always at least one, you know) occurs when the comma is enclosed in quotation marks. Any characters after the PRINT command that are enclosed in quotation marks are displayed exactly as entered. Type:

```
PRINT "4,3,7,2,1"
```

In this case, the commas are interpreted as characters, rather than formatting instructions, and are displayed on the screen.

PRINT WITH SEMICOLONS

The semicolon (;) is used in PRINT statements to suppress the carriage return. Remember the program example we used earlier?

```
10 PRINT "THE NUMBERS ARE "
20 PRINT 53721
```

Add a semicolon at the end of **line 10** and see what happens. Type:

```
10 PRINT "THE NUMBERS ARE ";
20 PRINT 53721
RUN
```

As you can see, the semicolon is a useful tool for formatting displays. You can also use semicolons to separate items within a single PRINT statement.

PRINT SPC(*n*)

The SPC(*n*) statement provides another formatting option. When used with PRINT, this command inserts the number of spaces specified by *n* between the last item printed and the next. The value of *n* must be an integer in the range 0 to 255.

SPC(*n*) must be used in a PRINT statement. As an example, let's add some spaces between each of a group of numbers. Type:

```
PRINT 5;SPC(5);3;SPC(5);7;SPC(5);2
```

The spacing introduced by SPC(*n*) is relative, i.e., the cursor is moved the specified number of spaces from the previously printed item, not from the beginning of the line.

PRINT TAB(*n*)

The TAB command lets you space printed characters relative to the left margin of the screen. Do you remember that using commas provides three tab fields and SPC moves the cursor the specified number of spaces from the previously printed character? Using TAB(*n*) allows you to define your own tab fields. Try it. Type:

```
PRINT TAB(5);5;TAB(10);3;TAB(15);7;TAB(20);2;TAB(25);1
```

The value contained in the parentheses following TAB may be an integer from 0 to 255. Values of 1 through 40 will place the cursor on the first line, 41 will return it to the leftmost position of line 2, and so on. Using TAB lets you move the cursor to the right only. If the value of *n* is less than the current cursor position, no movement will take place. (There is, however, a way of moving the cursor to the left that we will discuss a little later.)

This concludes our discussion of the PRINT statement for the moment. We will go into more depth later, but for now let's take a break and talk about some of PRINT's comrades.

INVERSE, FLASH AND NORMAL

Up to now, you have been viewing light characters on a dark background. Want a little variety? How about black letters on a light background? Simple. Just type INVERSE prior to your next PRINT statement. Type:

```
INVERSE: PRINT 53721
```

Notice that INVERSE is not a part of the PRINT statement itself, but merely changes the display mode. Separating INVERSE from PRINT with a colon allows both statements to be used on the same line. To return to the normal light on dark display, type NORMAL.

Now hold on to your hat, because we're going to try FLASHing! (No, you don't need a raincoat for this.) Type:

```
FLASH: PRINT 53721
```

As with INVERSE, the FLASH mode may be disabled by typing NORMAL. Let's write a little program that uses all three display modes:

```
10 PRINT "THE NUMBERS ARE 5 3 7 2 1"
20 INVERSE: PRINT "THEIR SUM IS:"
30 FLASH: PRINT 5+3+7+2+1:NORMAL
RUN
```

As you can see, using the INVERSE and FLASH statements allows you to emphasize specific portions of the screen.

MOVING THE CURSOR

We have discussed some ways of moving the cursor from within a PRINT statement. Commas, SPC, quotation marks, and TAB move the cursor from left to right and offer only incidental vertical mobility. Two additional commands, VTAB and HTAB, enable you to move the cursor up or down and left or right, respectively.

The Apple screen contains 24 lines vertically; the top line is 1 and the bottom is 24. VTAB, followed by an integer from 1 to 24, moves the cursor vertically to any specified line. Try it by typing:

```
VTAB 10
VTAB 24
VTAB 1
```

Of course, PRINT statements can be added to VTAB lines by using a colon to separate the statements. Type:

```
VTAB 10: PRINT "THIS IS LINE 10"
VTAB 1: PRINT "BACK TO LINE 1"
```

HTAB lets you move the cursor horizontally on the 40-column screen. Like VTAB, HTAB must be followed by an integer. The permissible values for the HTAB statement are from 0 to 255. Cursor position 1 is at the left margin of the current line; 40, the right margin; 41, the left margin of the next line, 80; the right margin, etc. Type:

```
HTAB 25: PRINT"GO TO THE RIGHT";: HTAB 1: PRINT "THEN TO THE
LEFT"
```

Or try typing:

```
HTAB 30: PRINT "HOW ABOUT A SCROLL": HTAB 156: PRINT "DOWN
THE SCREEN?"
```

PRINTING SPEED

As you have probably noticed, characters on the Apple screen are displayed very rapidly. At times, slower speed can be useful to increase readability or for dramatic effect. The rate of character printing can be changed with the SPEED command. The format of this command is SPEED= followed by an integer from 0 through 255. This sets the rate at which characters print to the screen or peripheral device. Normally, the speed setting is automatically set at (defaults to) 255 and you must change it to get a slower rate. For a very slow print rate, type:

```
SPEED= 1
```

then type in some PRINT statements to see the effect. Change the speed to several different settings and print a few more lines. Experimenting with the SPEED command gives you a good idea of the speed of the various values.

CLEARING THE SCREEN

One final command that you will find useful is the HOME command. HOME clears all the text within the screen and returns the cursor to the top left printing position. No additional parameters need be used. Just type HOME and press Return, or include it in a program line as follows:

```
10 HOME : PRINT "THE SCREEN WAS CLEARED"
```

SUMMARY

We have concentrated primarily on formatting and printing on the screen. Although we will elaborate on many of these points later, you now have enough information to construct some interesting programs.

Before the following brief review of printing, the short program PRINTING.DEMO demonstrates the operations discussed with an irreverent look at the terms you have encountered so far.

Before going on, though, take a few minutes to look over the chapter and re-read the sections dealing with any of the commands or words that may not be clear to you. Try some of the operations with your own computer and see exactly how they work.

PRINTING DEMONSTRATION PROGRAM

The program shown in **Listing 1** demonstrates some of the items we have been discussing. Most of the lines should be familiar to you, so I won't bore you with a line-by-line explanation.

Listing 1

```
10  REM  PRINTING.DEMO
20  TEXT : HOME : NORMAL
30  PRINT "PRINTING DEMONSTRATION"
40  VTAB 8: PRINT "THIS PROGRAM DEMONSTRATES THE PRINTING":
    PRINT "TECHNIQUES EXPLAINED IN CHAPTER 1"
50  VTAB 22: INPUT "PRESS RETURN TO CONTINUE";A$
60  HOME : PRINT "TYPE MODES"
70  VTAB 8: PRINT "THIS IS NORMAL TYPE"
80  INVERSE : PRINT "THIS IS INVERSE TYPE"
90  NORMAL : FLASH : PRINT "THIS IS FLASHING TYPE"
100  NORMAL : VTAB 22: INPUT "PRESS RETURN TO CONTINUE";A$
110  HOME : PRINT "SPEED = 100": SPEED= 100
120  VTAB 7: PRINT "USING THE SPEED= COMMAND SLOWS": PRINT
    "THINGS DOWN A LOT."
130  SPEED= 255: VTAB 22: INPUT "PRESS RETURN TO
    CONTINUE";A$
140  HOME : PRINT "AUTOMATIC TABS AND SPC()"
150  VTAB 7: PRINT 5,7,9
160  VTAB 9: PRINT 5; SPC( 6);7; SPC( 3);9
170  VTAB 22: INPUT "PRESS RETURN TO CONTINUE";A$
180  HOME : PRINT "HTAB AND VTAB"
190  FOR X = 5 TO 15: FOR Y = 1 TO 40 STEP 5
200  VTAB X: HTAB Y: PRINT "HI";
210  NEXT : NEXT
220  PRINT : VTAB 22: PRINT "THAT'S ALL"
```

Chapter 1 Summary: Getting Started

The PRINT Statement

1. Without qualifiers, PRINT produces a blank line on your screen.
2. PRINT followed by numeric data displays the numbers as entered.
3. PRINT can evaluate expressions and display the result.
4. PRINT will display a group of characters enclosed within quotation marks.
5. PRINT may be abbreviated by using the question mark (?) when typing in statements.

Program Lines

1. Lines may contain multiple statements separated by colons as long as the total length does not exceed 239 characters.
2. Lines not preceded by numbers are executed immediately after Return is pressed.
3. Lines preceded by numbers are assumed to be part of a program. Execution of numbered lines is deferred until the program is RUN.
4. Line numbers cannot be less than 0 or greater than 63999.

Formatting with PRINT

1. Commas inserted into PRINT statements display data in three tab fields (positions 1-16, 17-32, 33-40).
2. The semicolon suppresses spacing and linefeed functions. It is generally used to print items without intervening spaces.
3. SPC*(n)* inserts *n* number of spaces between the last item printed and the next. The value of *n* must be from 0 to 255.
4. TAB*(n)* provides the means to space printed characters relative to the left margin of the screen. The value of *n* must be from 0 to 255.

Display Modes

1. INVERSE sets the display to dark characters on a light background.
2. NORMAL sets the display to light characters on a dark background.
3. FLASH makes the display alternate between NORMAL and INVERSE modes.

Moving the Cursor

1. VTAB*(n)* moves the cursor in either direction vertically to any of the 24 lines on the screen. The value of *n* must be from 1 to 24.
2. HTAB*(n)* moves the cursor horizontally to any of the screen printing positions. The value of *n* must be from 0 to 255.

Printing Speed

1. The Apple's default speed is 255.
2. The speed may be changed by the user through the SPEED=*n* command. The value of *n* must be in the range 0 to 255, with 0 the slowest and 255 the fastest.

Clearing The Screen

1. The HOME command clears the screen and returns the cursor to the upper-left print position.

Chapter 2: All About Variables

With the help of PRINTING.DEMO, you should now be able to PRINT like a virtuoso. PRINT is a most versatile command, but it's time to expand our repertoire. The next few measures of our programming symphony will be played by the variable section.

VARIABLES

You have learned that PRINT is the primary output command in Applesoft. Specifying each piece of data to be displayed can be a bit tedious, but there is a better way.

Using variables opens up a whole new way of handling data from within a program. First let's take a look at the nature of the beast, and then we'll get down to the nitty-gritty of how variables can make your programming life considerably easier.

Variables are symbols that represent items of data. For instance, the variable name PI might be used to represent the number 3.1415. Variables get their name from the fact that the data they represent may be changed without changing the symbolic name. For example, the variable named HI might be used to represent the number three at one point in a program's execution, but might represent the number 300 after some calculations are performed.

Variables are named according to a few simple rules. The first character must be alphabetic, and can be followed by any combination of letters and numbers that is less than 238 characters in length. Names must not contain any of the reserved words that are used by Applesoft itself. For example:

```
A1
BT
CHECKBOOK
H2O
P
Z765
```

would all be legal variable names. On the other hand, the following

`ATE`	Contains the reserved word AT
`9B`	Starts with a number
`GR2`	Contains the reserved word GR
`HOMEWORK`	Contains the reserved word HOME
`T&B`	Contains the reserved word &
`X2COST`	Contains the reserved word COS
`X.3`	Contains a character which is neither a letter nor a digit

would not. Check the rules above and the reserved word list in **Table 1** to see why not.

TABLE 1: Applesoft Reserved Words

ABS	IF	RETURN
AND	IN#	RIGHT$
ASC	INPUT	RND
AT	INT	ROT=
ATN	INVERSE	RUN
CALL	LEFT$	SAVE
CHR$	LEN	SCALE=
CLEAR	LET	SCRN
COLOR=	LIST	SGN
CONT	LOAD	SHLOAD
COS	LOG	SIN
DATA	LOMEM:	SPC
DEF	MID$	SPEED=
DEL	NEW	SQR
DIM	NEXT	STEP
DRAW	NORMAL	STOP
END	NOT	STORE
EXP	NOTRACE	STR$
FLASH	ON	TAB
FN	ONERR	TAN
FOR	OR	TEXT
FRE	PDL	THEN
GET	PEEK	TO
GOSUB	PLOT	TRACE
GOTO	POKE	USR
GR	POP	VAL
HCOLOR=	POS	VLIN
HGR	PR#	VTAB
HGR2	PRINT	WAIT
HIMEM:	READ	XDRAW
HLIN	RECALL	XPLOT
HOME	REM	
HPLOT	RESTORE	
HTAB	RESUME	

Although you can give variables very long names, there is a catch. Applesoft uses only the first two characters to distinguish one variable name from another. So:

```
CHECKBOOK
CHECKUP
CHECKIN
```

would all be interpreted by Applesoft as the same variable, CH. If you routinely use only one- or two-character variable names in your programs, many potential problems will be avoided. This "ounce of prevention" will keep your variables separate and distinct.

TYPES OF VARIABLES

Applesoft allows three distinct types of variables -- real, integer and string. Symbols are appended to the variable name to designate which type of information it may contain.

Real variables, which may contain whole numbers or numbers with a decimal fraction, are identified by the lack of a symbol following the name. XP, D2, Z and Y1 would all designate real variables. The range of allowable values for real variables is -9.99999999E+37 to +9.99999999E+37. (The E+37 is scientific notation that means to multiply by 10 to the 37th power.)

Integer variables, indicated by a percent sign (%) following the name, may contain whole numbers in the range of -32767 to +32767. Integer variable names look like this: XP%, D2%, Z% and Y1%.

String variables have a dollar sign ($) following the name and may contain up to 255 alphabetic and/or numeric characters. The same four variable names could designate strings by adding a $: XP$, D2$, Z$ and Y1$.

In each case, we used the same variable names and simply changed the designating symbol to identify the type of variable. Applesoft recognizes each of these variables as distinctly different entities, so you could indeed use the same names for different types of variables within a program. As a programming practice, however, this leaves open many possibilities for confusion and errors. Remember the "ounce of prevention" we discussed just a few paragraphs ago?

Time for a quiz! Look carefully at the values assigned to the variable X and decide which, if any, suffix should be added and what type of variable is being illustrated:

1. X = 3.1416
2. X = -29458
3. X = "H2O"
4. X = 14.7E+10
5. X = "3.1416"
6. X = 1.5
7. X = 432
8. X = "THIS ONE IS EASY"
9. X = 45983
10. X = -675

Let's see how well you fared. I slipped in a couple of tricky ones, just to see if you were awake. Items 1, 4, 6 and 9 are real variables and require no suffix. Although items 4 and 9 are whole numbers (14.7E+10 equals 147,000,000,000), they are not within the allowable range for integer variables.

Items 2, 4 and 7 are whole numbers within the valid range for integer variables, so they may be either integer or real. If your program only required whole numbers, using integer variables (with the % suffix) would speed its operation. However, to avoid problems with out-of-range values, many programmers use real variables for all numeric values. The remaining items (3, 5 and 8) are string variables, requiring the use of the $ suffix. String variables are easy to spot, since the value assigned must be enclosed by quotation marks.

If you correctly identified all of them, you're a bona fide variable spotter! If you missed any, spend a few minutes reviewing this section. As you get further into assigning values and manipulating variables, you need a firm understanding of each type's permissible values and correct designation.

ASSIGNING VALUES TO VARIABLES

There are two primary ways to assign values to variables: from the keyboard or within the program itself. Data contained in files on disk may also be used to assign values to variables. For the moment, let's confine our discussion to values assigned from the keyboard and from within the program.

By the way, expressions containing variables may be evaluated and their results displayed through the use of the PRINT statement. Most of the PRINT rules discussed in chapter 1 apply to the display of variables as well. (Although we won't go over PRINT again here, the examples used illustrate how variables may be displayed.)

INPUT

INPUT is one of the most frequently used statements in Applesoft. Available in deferred execution mode only, this command is a common method of assigning values to variables during program execution. INPUT prints a question mark on the screen and waits for the user to type in the proper information. For example:

```
10 INPUT X%
```

allows the user to type in a value for the integer variable X%. If X were a real variable, the format would be:

```
10 INPUT X
```

As you may have guessed, soliciting input for the string variable X$ would be done by:

```
10 INPUT X$
```

Let's try a short input program to see this command in operation:

```
10 INPUT X%
20 PRINT X%
RUN
```

At the question mark prompt, type an integer between -32767 and +32767 and the program will print the number. Try substituting real and then string variables for X% in the INPUT statement, and then rerun the program with each type of variable.

One important note: You do not need to enclose your response to a string variable input statement in quotation marks. While strings are enclosed in quotation marks when specified in a program, using quotation marks when responding to an INPUT statement is unnecessary.

Multiple values for variables may be solicited from the user in a single input statement by separating the variables with commas:

```
10 INPUT X%,X,X$
```

Responses to this type of input statement must be of the proper type and must be separated either by commas or by pressing Return after each response. Too few responses will cause a double question mark (??) to appear on the screen, indicating that more data is needed. Too many responses, on the other hand, will cause an EXTRA IGNORED error message to appear. Try it. Type:

```
10 INPUT X%,X,X$
20 PRINT X%
30 PRINT X
40 PRINT X$
```

From now on, the RUN command following each program will be assumed. After you have successfully run this program once or twice, try typing too few or too many responses. Did the error message or double question mark appear? Now try putting wrong values into the variables -- decimals in the integer X%, letters in X, etc. See what happens.

After you have experimented with the variable values, you may find it difficult to remember which type of data is being requested by the INPUT statement. If you included that INPUT statement in a program to be run at some other time, hitting the right combination of data to type in would be virtually impossible. Fear not -- there is a simple solution!

INPUT PROMPTING

On a practical level, some sort of message should precede a request for data so the user knows what type of information is expected. This message is known as a prompt and may be included in the INPUT statement. The prompt must be enclosed in quotation marks and terminated by a semicolon. For instance, the statement:

```
10 INPUT "HOW MANY PLAYERS?";X%
```

when RUN, would display on the screen:

```
HOW MANY PLAYERS?
```

and the computer would pause for the appropriate integer response. Of course, you may request multiple responses with one INPUT prompt. Try the following program. (If your ego is fragile, feel free to change the INPUT prompts.)

Listing 2

```
10 INPUT "TYPE IN YOUR NAME, AGE & WEIGHT. PRESS RETURN AFTER
    EACH. ";X$,X,Y
20 PRINT "HELLO ";X$
30 PRINT
40 PRINT "YOU DON'T LOOK A DAY OVER ";X+10
50 PRINT
60 PRINT "YOU WOULDN'T KID ME ABOUT ";Y;" POUNDS, WOULD YOU?"
```

See how much easier it is to use multiple inputs with a prompt? Concise prompts in each INPUT statement make using your programs much easier.

COMMAS

Let's cover one more problem before moving on: Applesoft does not like imbedded commas in response to a string INPUT prompt. It is only possible to use commas in response to an INPUT statement if the data containing the comma is preceded by a quotation mark. Frankly, this seems to be more bother than it's worth. It seems a lot easier to use multiple responses and specify a distinct variable for each name. For example:

Listing 3

```
10 PRINT "TYPE YOUR LAST NAME"
20 INPUT A$
30 PRINT "FIRST NAME"
40 INPUT B$
50 PRINT "SPOUSE'S NAME"
60 INPUT C$
```

Of course, you already know how to combine these statements into one input if desired. (Just specify multiple variables in the input statement and separate them by commas.)

Let's assume your responses to A$, B$ and C$ are SMITH, JOHN, MARY. If you want to address an envelope, a PRINT statement like:

```
10 PRINT B$; " AND ";C$;" ";A$
```

would print JOHN AND MARY SMITH. Or

```
10 PRINT "MR & MRS ";B$;" ";A$
```

would print MR & MRS JOHN SMITH. Suppose you want an alphabetical listing by last name, husband's name and spouse.

```
10 PRINT A$;",";B$;" & ";C$
```

would do the job nicely. The point here is that separate variables are much more versatile than a single variable that contains all three data items. Make your variables as simple as possible so that you can use them in several forms throughout the same program.

GET

A convenient method of fetching a single character from the keyboard is provided by the GET command. Unlike INPUT, GET does not display the character typed and does not require that Return be pressed following the character. The format for this command is simply:

```
10 GET A$
```

The string variable A$ will be used to store the character of the next key pressed. GET may also be used with numeric variables (e.g., GET A), but Applesoft will respond with a SYNTAX ERROR if you type a letter instead of a number. It is much easier, and safer, to use a string variable and convert the string to a number using the VAL function (which we will discuss shortly).

GET includes no provision for input prompting, so your prompt must be provided by a PRINT statement preceding the GET. To display the value typed in response to the GET command, another PRINT statement must be used. The following short program illustrates the necessary sequence.

Listing 4

```
10 PRINT "TYPE ANY LETTER ";
20 GET X$
30 PRINT
40 PRINT "YOUR LETTER WAS ";X$
```

Note that your letter is displayed by **line 30** and does not appear when you type it. Although it may not seem so right now, GET is an extremely versatile command. You will learn that GET can be used for program pauses, menu selection responses and many other applications.

LET

LET could be called an unnecessary statement, since all of its capabilities can be exercised without actually typing the word. LET is an assignment statement that sets a variable equal to a specific value. In actual use, LET takes the value of the variable and/or expression to the right of the equal sign (=) and assigns it to the variable on the left. For instance:

```
LET X = 4.75
```

assigns a value of 4.75 to the variable X.

```
LET X = X + 1
```

adds one to the value of X and stores the result back in X, in this case making it 5.75.

This same procedure works for all types of variables, as long as the value or expression is compatible with the nature of the variable.

```
10 LET X$ = "4"
10 LET X = 4.12359
10 LET X$ = "THE VALUE OF X IS 4.12359"
```

All three of the above are legitimate assignments and would be executed successfully within a program. Each variable in a program may be altered any number of times through the use of LET. In fact, reusing variables is a good programming technique, because it saves both memory space and execution time.

LET is an unnecessary statement because you don't need to type the actual word to achieve the same result. In other words, the statements:

```
10 LET X = X + 4
```

and

```
10 X = X + 4
```

are identical, as far as your Apple is concerned. Whenever you use a statement that has a variable on the left followed by an equal sign, you are telling the computer to assign the value to the right of the equal sign to the variable on the left. Try it. Type:

Listing 5

```
10 INPUT "TYPE A NUMBER ";X
20 PRINT "WATCH YOUR NUMBER DOUBLE"
30 LET X = X * 2: PRINT X
40 PRINT "AND DOUBLE AGAIN"
50 X = X * 2: PRINT X
```

Notice that your number doubles in **line 30** and doubles again in **line 50**. **Lines 30 and 50** function exactly the same, even though only one of them actually contains the word LET. This program also illustrates the concept of reusing a variable. Three different values are assigned to the same variable (X) as the program runs.

If you do not assign a value to a variable, its initial value is set automatically by Applesoft. For real and integer variables, the initial value is zero. For string variables, the initial value is an empty string equivalent to the assignment:

```
X$ = ""
```

The empty string is often referred to as the null string.

READ & DATA

Sounds a bit like a comedy team, doesn't it? Laurel & Hardy, Abbott & Costello, and READ & DATA. Believe it or not, there are some similarities between these two statements and a comedy team. One really doesn't function well without the other, yet together they are capable of doing some very interesting things.

Using the DATA statement, you can create a list of items that can then be used by READ. For example, here is a DATA statement with five numeric items separated by commas:

```
10 DATA 3.14, 2, 86.2, 39.1, 17
```

By observing a few simple rules, you can construct DATA statements with ease.

1. DATA statements can appear anywhere in a program and are cumulative. That is, each DATA statement adds to the list of items built up by previous (lower line numbered) statements. For instance, if your program called for five data items, you could use a data line like the one above, or create an identical list using five separate data statements:

```
10 DATA 3.14
20 DATA 2
30 DATA 86.2
40 DATA 39.1
50 DATA 17
```

2. Data items that are to be READ into numeric variables (integer or real) must, in fact, be numeric values.
3. Any kind of data item may be read into a string variable.

4. If you attempt to READ a string item into a numeric variable, Applesoft will generate a SYNTAX ERROR for the program line that contains the erroneous assignment statement.

As noted earlier, READ and DATA are a team. Once a list of data items has been created, READ is used to assign the items to variables within the program. The first variable you READ is assigned the value of the first item in the DATA list, the second variable takes the value of the second item, etc. When Applesoft encounters the first READ statement, it pulls the first data item from the DATA list and assigns it to the variable in the first READ statement. When the next READ statement is encountered, the next DATA item is removed from the list and assigned to that READ variable. Applesoft keeps careful track of how many data items are used. If your program attempts to READ more data than is contained in the data list, the computer responds with an OUT OF DATA ERROR IN whatever line the offending READ statement is located.

Like riding a bicycle, using READ and DATA is much more difficult to explain than to do. Let's construct a short program to illustrate the use of this versatile team of commands:

Listing 6

```
 10 READ A$,B$,C$
 20 PRINT A$;B$;C$
 30 READ D
 40 PRINT D
 50 READ E%
 60 PRINT E%
 70 DATA "USING ", "READ & ", "DATA"
 80 DATA 789.12, 355, "THE END"
 90 READ F$
100 PRINT F$
```

As you can see, the program uses six data items and assigns them in sequence to the variables specified. Although we have put the DATA statements after some READ statements and before others, in each case, the READ statement located the data list with ease. Try a few programs of your own using READ and DATA to gain familiarity with these two important statements.

RESTORE

At any time during the execution of a program using READ and DATA statements, you may initialize the data list using a RESTORE command. There are no parameters, conditions or options. Just type:

```
10 RESTORE
```

and Applesoft will return to the beginning of the DATA list. The next READ statement that's encountered will pull the very first item from the DATA statement list. This command allows you to reuse the data items in your program's DATA statement.

These are the primary methods of assigning values to variables. As you can see, variables can make your programming efforts much more productive. Don't be afraid to experiment with the statements discussed in this section. Become thoroughly familiar with them, because we will return to this topic again later.

VARIABLE TYPE CONVERSIONS

As we learned earlier, there are three types of variables used in Applesoft: integer, real and string. Sometimes your program will need to convert one type of variable to another. For instance, you may want to convert a real number to an integer, find the numeric value of a string or change a real number to a string for formatting purposes. Several statements are available to make the conversion process relatively easy.

INT(*n*) is used to convert a real number an integer. It does this by simply removing the decimal part of the number. For example, type:

Listing 7

```
10 X = 3.75
20 X% = INT (X)
30 PRINT X%
```

Was the integer 3 displayed? The INT function does not round the variable, but merely prints the nearest integer less than *n*. If you want to round *n* to the nearest integer, simply add .5 before using the INT function:

Listing 8

```
10 X = 3.75: Y = 3.5: Z= 3.1
20 X% = INT (X + .5)
30 Y% = INT (Y + .5)
40 Z% = INT (Z + .5)
50 PRINT X;" IS ROUNDED TO ";X%
60 PRINT Y;" IS ROUNDED TO ";Y%
70 PRINT Z;" IS ROUNDED TO ",Z%
```

Each of the real numbers is now rounded to the nearest whole number.

The reverse conversion, that of making a real variable out of an integer, can be accomplished with LET:

```
10 LET X = X%
```

or more simply:

```
10 X = X%
```

should do nicely.

VAL(X$) is used to convert a string to a real or integer variable. The format is:

```
10 X = VAL(X$)
```

or

```
10 X% = VAL(X$)
```

At least the first character in the string X$ must be a numeric character or the entire string will be converted to zero. VAL(X$) evaluates a string expression to the first non-numeric character, then ignores the rest of the characters. For instance:

```
X$ = "3.1416"
X = VAL(X$)
PRINT X
```

displays the value of X as 3.1416. If we change X$ slightly, different results are obtained.

```
X$ = "PI = 3.1416"
X = VAL(X$)
PRINT X
```

returns zero as the value of X. Another variation returns a different result:

```
X$ = "3POINT1416"
X = VAL(X$)
PRINT X
```

Here X = 3. By carefully examining the string expression you want to convert, these variations can be avoided.

If you have mistyped and entered a string response to a numeric input statement in some of your experimenting, you are familiar with the REENTER error message. The VAL function can be used to avoid this problem. Simply use a string variable for all input responses and convert them to values with VAL immediately afterwards. For example,

Listing 9

```
10 INPUT "ENTER A NUMBER: ";X$
20 N = VAL(X$)
30 PRINT N
```

STR$ (X) is used for the reverse conversion, i.e., making a numeric variable into a string variable. Remember that the destination variable must be a string. To use STR$, type:

```
10 X$ = STR$ (X)
```

This conversion statement is very useful, as you will see later, for formatting number displays and aligning decimal points.

Take a few minutes to experiment with the type conversion statements. Try correct and incorrect conversions to get a feel for the process and error messages involved.

STRING MANIPULATIONS

No, these aren't the little games that you play with a piece of twine on the fingers! String manipulations are useful utilities that allow you to extract specified series of characters (called "substrings") from within a string. The easiest way to understand these statements is to see them in actual use. For any string expression X$, LEFT$ (X$,*n*) returns the leftmost *n* characters of string expression X$. For example, if X$ = "PROGRAMMING", then :

```
PRINT LEFT$ (X$,3)
```

returns the characters PRO.

```
PRINT LEFT$ (X$,7)
```

displays PROGRAM. Get the idea? If *n* is larger than the number of characters in the expression, only the number of characters actually in the string are returned, and the extra positions are ignored. You can, of course, assign the resulting substring to another string variable within a program:

Listing 10

```
10 X$ = "PROGRAMMING"
20 Y$ = LEFT$ (X$,4)
30 PRINT Y$
```

The Apple should display PROG. LEFT$ will also evaluate a string literal contained within the parentheses as follows:

```
20 Y$ = LEFT$ ("PROGRAMMING",4)
30 PRINT Y$
```

Were the results the same as those obtained in the short program above? They should be identical.

The RIGHT$ (X$,*n*) statement, as you would expect, is the complement of the LEFT$ statement. In this case, the rightmost *n* characters of the expression X$ are returned. Using the same string for X$ as before:

```
PRINT RIGHT$ (X$,4)
```

returns MING. if you were to specify the string in a program, the format would be as follows:

```
10 Y$ = RIGHT$ ("PROGRAMMING",3)
20 PRINT Y$
```

The substring ING should be displayed on the screen.

MID$ (X$,*n*) is a little trickier, but it can do some very specific string manipulations. It extracts *n* number of characters from the middle of a string. If used in the format:

```
MID$ (X$,n)
```

a substring consisting of all the characters from the *n* th character to the end of X$ is returned. For instance,

```
PRINT MID$ ("PROGRAMMING",4)
```

displays GRAMMING. If used with the string variable X$:

```
PRINT MID$ (X$,4)
```

also displays GRAMMING.

Here's the tricky part. You may specify another argument to limit the number of characters returned:

MID$ (X$,*n*,*m*)

In this case, MID$ returns *m* characters of X$ beginning at *n* and proceeding to the right. Let's try an example:

```
PRINT MID$ ("PROGRAMMING",4,3)
```

What do you think will be displayed? Try it and see. Did you get GRA?

Although these three statements may seem a bit frivolous at first, they have some very valuable applications, as you will see in some program examples. Take a few string expressions of your own and try extracting substrings in immediate execution mode, short programs and various combinations until you are comfortable with these statements.

LEN(X$) is a handy little statement used to determine the number of characters in a string. If we use the same string (X$ = "PROGRAMMING") as in the previous examples, then

```
PRINT LEN(X$)
```

returns the number 11. We could also specify the string expression:

```
PRINT LEN("PROGRAMMING")
```

and obtain the same result, 11. You may assign the result to a numeric variable within a program:

Listing 11

```
10 X$ = "PROGRAMMING"
20 X = LEN(X$)
30 PRINT X
```

As you can see, X = 11. The LEN statement is particularly valuable in formatting a screen display or aligning columns of figures.

Concatenation of Strings

Concatenation simply means to join or put together. Two or more string expressions may be joined by placing a plus sign (+) between them. Here's a short program that illustrates this operation:

Listing 12

```
10 X$ = "PRO"
20 Y$ = "GRAMM"
30 Z$ = "ING"
40 PRINT X$ + Y$ + Z$
```

The word PROGRAMMING should appear on the display. Easy, right?

ARRAYS

Each of the three types of variables (integer, real and string) may be set up as lists, known as arrays. These lists share the same variable name and type, but use a numeric pointer, called a subscript, to indicate which element of the list is to be used. The subscript, which may be either a positive integer or a variable that represents a positive integer, is enclosed in parentheses following the variable name. For example,

```
X$(1) = "PROGRAMMING"
```

The following short program illustrates one way in which array variables may be used:

Listing 13

```
10 X$(1) = "PROGRAMMING"
20 X$(2) = "THE APPLE"
30 X$(3) = "IS FUN"
40 INPUT "WHICH STRING? (1-3) ";N
50 PRINT X$(N)
```

In **lines 10-30,** the program assigns each string to its own array element. The number that the user inputs is assigned to the variable N in **line 40,** and then the variable is used as the array element subscript in **line 50.**

Up to now, the variables that have been discussed are referred to as simple variables. Applesoft automatically assigns memory space for simple variables as your program is run. Applesoft also automatically assigns space for up to 11 elements in an array variable (subscripts zero through ten), but requires the program to request a specific amount of space if a larger array is used. This is accomplished using the DIM statement, which is short for dimension. You must specify the dimension of the array before the array variable is used the first time, as follows:

```
DIM X$(100)
```

This statement would make room for 101 string elements. (Though the zeroth element is counted, it is often ignored to make programming easier.)

It is also possible to have lists with more than one dimension, for example, X$(n,m). These are called multi-dimensional arrays, and their use is beyond the scope of this book.

VARIABLES Program

The program in **Listing 14** demonstrates some of the new commands we have discussed. You will also notice many of the PRINT statements covered in chapter 1. Type the program into your computer and run it to see some variables in action.

Listing 14

```
10  REM  VARIABLES
20  TEXT : HOME : NORMAL
30  VTAB 7: PRINT "THIS PROGRAM DEMONSTRATES THE
     TECHNIQUES": PRINT "DESCRIBED IN CHAPTER TWO."
40  GOSUB 260
50  INVERSE : PRINT "THE INPUT STATEMENT": NORMAL
60  VTAB 7: INPUT "ENTER YOUR FIRST NAME: ";A$
70  VTAB 8: INPUT "ENTER YOUR LAST NAME : ";B$
80  VTAB 11: PRINT "YOUR NAME IS: ";A$;" ";B$
90  GOSUB 260
100  INVERSE : PRINT "LEFT$, RIGHT$ AND MID$": NORMAL
110 A$ = A$ + B$
120  VTAB 7: PRINT "THE FIRST 3 LETTERS OF YOUR NAME: ";
     LEFT$ (A$,3)
130  VTAB 10: PRINT "THE LAST 3 LETTERS OF YOUR NAME: ";
     RIGHT$ (A$,3)
140  VTAB 13: PRINT "THE MIDDLE 3 LETTERS OF YOUR NAME: ";
     MID$ (A$, LEN (A$) / 2,3)
150  GOSUB 260
160  INVERSE : PRINT "READ & DATA": NORMAL
170  DIM N$(10): VTAB 5: PRINT "THE FOLLOWING WORDS HAVE
     BEEN READ INTO": PRINT "THE ARRAY N$() FROM DATA
     STATEMENTS": PRINT "CONTAINED IN THIS PROGRAM:": PRINT
180  FOR X = 1 TO 10: READ N$(X): PRINT N$(X): NEXT
190  GOSUB 260
200  INVERSE : PRINT "ARRAYS": NORMAL
210  VTAB 5: PRINT "HERE IS THE CONTENTS OF THE ARRAY N$()":
     PRINT "READ FROM DATA STATMENTS IN THE LAST": PRINT
     "SECTION:": PRINT
220  FOR X = 1 TO 10: PRINT N$(X): NEXT
230  GOSUB 260
240  PRINT "THAT'S ALL"
250  END
260  VTAB 23: PRINT "PRESS RETURN TO CONTINUE";: GET R$:
     PRINT R$: HOME : RETURN
270  DATA  ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN
```

Chapter 2 Summary: All About Variables

1. Variable names must not contain any reserved words. The first character must be alphabetic and may be followed by up to 237 letters or numbers.
2. Only the first two characters of a variable name are significant in differentiating one from another.
3. There are three types of variables used in Applesoft:
    a. Real variables contain numbers within the range of -9.99999999E+37 to +9.99999999E+37
    b. Integer variables contain whole numbers in the range of -32767 to +32767. The % suffix is used to designate integer variables.

c. String variables may contain up to 255 alphabetic and/or numeric characters and are identified by the suffix $.

4. Values may be assigned to variables from the keyboard or within the program itself.

INPUT

1. This statement prints a question mark on the screen and allows the user to type in a value that will be stored in the specified variable.
2. Multiple variables, separated by commas, may be specified in an INPUT statement.

INPUT Prompting

1. A prompt is a message that precedes a request for data.
2. Prompts should inform the user exactly what type of response is expected.
3. Prompts in an INPUT statement must be enclosed within quotation marks and terminated by a semicolon.

GET

1. This statement fetches a single character from the keyboard. The character typed in response to GET is not displayed on the screen.
2. Return need not be pressed following a response to a GET statement.
3. Responses to GET should always be assigned to a string variable and converted later, if necessary.
4. GET includes no provision for prompting. PRINT may be used to display an appropriate prompt.

LET

1. LET is an assignment statement that allows the program to set a variable equal to another variable, constant, or expression.
2. LET assigns the value of the variable or expression to the right of the equal sign to the variable on the left.
3. Each variable in a program may be altered any number of times using LET.
4. LET is implied whenever you use a statement that has a variable on the left followed by an equal sign. The word itself is optional.

READ and DATA

1. DATA creates a list of pieces of data (items) that are used by READ.
2. DATA statements may appear anywhere in a program and are cumulative.
3. DATA items must follow the same rules as INPUT responses in terms of the variables into which they will be READ.
4. READ is used to assign DATA items to variables within a program.
5. Each READ variable takes the value of a data item in sequence from the list.
6. After each READ, the computer keeps track of the last item used. The next READ begins at the DATA element following the last item and proceeds sequentially.
7. RESTORE resets the sequence of items read so that the first item in a program's DATA statements will be the first one READ.
8. Attempting to READ more data than is contained in the data list invokes the OUT OF DATA error message.

Variable Type Conversions

1. INT (X) is used to convert a real number to the largest integer that does not exceed the value of X.
2. LET may be used to convert an integer variable to real.
3. VAL(X$) interprets a string as a real or integer value. String expressions are evaluated only to the first non-numeric character. If the first character is non-numeric, the entire string evaluates as zero.
4. STR$ (X) or STR$(X%) converts a real or integer variable to a string.

String Manipulations

1. LEFT$ (X$,*n*) returns the leftmost *n* characters of string X$.
2. RIGHT$ (X$,*n*) returns the rightmost *n* characters of string X$.
3. MID$ (X$,*n*) returns a substring consisting of all the characters from the *n*th to the end of string X$.
4. MID$ (X$,*n,m*) returns *m* characters of X$, beginning at the *n*th and proceeding to the right.
5. LEN (X$) is used to determine the number of characters in string X$.
6. Two or more strings may be joined together (concatenated) by placing the plus sign (+) between them.

Arrays

1. Variables may be set up in lists, known as arrays, by using the DIM command.
2. If you forget to dimension an array, Applesoft automatically assigns a maximum subscript of 10 for each dimension.
3. Each element of an array is identified by a subscript that designates its location. Values contained within the subscript parentheses may be numbers, numeric variables or numeric expressions.

Chapter 3: Flow of Control and Branching

In the first two chapters, we concentrated on building a foundation of some of the basic concepts necessary to write programs in Applesoft. Now that you have mastered input, output and manipulation of variables, the fun really begins. Although "Flow of Control and Branching" hardly sounds like a jocular title, the concepts presented in this chapter enable you to add some spice to your programming life.

So far, we have treated Applesoft like an interstate highway. We learned the rules of the road, adjusted our speed a little and generally followed the straight and narrow. Now we're ready to set our own course, look around a bit and make some decisions.

First, we'll add a few new capabilities for evaluating expressions. Next, line numbering will be discussed in some detail. We'll save the best for last and finish up by learning about branching. Ready? Let's get to work!

EVALUATING EXPRESSIONS

Previously, we briefly touched on the concept of operators and expressions. Operators are the signs used to designate the operations on numbers or variables specified in a program line. The most common of these are the arithmetic operators:

+	for addition
-	for subtraction
/	for division
*	for multiplication
^	for exponentiation (raising to a power).

The combination of numbers (or variables) and operators is called an expression. Expressions may be simple:

```
5 + 3
```

or somewhat more complex:

```
5 + (7 * 3)/2 + 1
```

or downright complicated:

```
((4 * 3 + 6)/2) * (6 ^ 5)/((2 - 1) * 12)
```

Of course, variables may be substituted for each of the constants shown in the expressions above. The process of solving the problem posed by an expression is referred to as the evaluating the expression.

To further complicate matters, the sequence of expression evaluation depends upon certain priorities that are assigned to each of the operators. Like most versions of BASIC, Applesoft executes operators in the following order:

1. First priority -- Exponentiation
2. Second priority -- Multiplication and division
3. Third priority -- Addition and subtraction

Operators having the same priority within an expression are executed from left to right in the line.

The normal priority of execution may be altered through the use of parentheses (). Items and operators enclosed in parentheses within an expression are evaluated prior to any other operations.

Now let's play computer to illustrate how the order of execution varies when the same expression is used with different parenthetical groupings. In each case, the expression will be followed by the intermediate step(s) and final result:

4 * 3 + 6 * 4 / 2 = 12 + 12 = 24

4 * (3 +6) * 4 / 2 = 4 * (9) * 4 / 2 = 144 / 2 = 72

4 * (3 + 6) * (4 / 2) = 4 * (9) * (2) = 72

(4 * 3 + 6 * 4) / 2 = (12 + 24) / 2 = 36 / 2 = 18

((4 * 3 + 6) * 4) / 2 = ((18) * 4) / 2 = 72 / 2 = 36

4 * (3 + 6 * 4) / 2 = 4 * (27) / 2 = 108/ 2 = 54

4 * 3 + (6 * 4 / 2) = 12 + (12) = 24

(4 * (3 + 6 * 4)) / 2 = (4 * (27)) / 2 = (108) / 2 = 54

As you can see from these examples, proper grouping of items within an expression is crucial to getting the results you want.

In two of the examples, multiple parentheses were used. This procedure, known as nesting, serves to further define the order of execution. When using these techniques in your own programs, remember that the number of left and right parentheses within an expression must be equal. Although you may never need to construct such a complex expression, it is comforting to know that Applesoft permits nesting up to 35 levels of parentheses. Whether or not you could keep track of 35 pairs of the little devils is another question entirely!

Later, we will discuss a few additional operators that have to do with branching. Take a few minutes now to try some expressions on your Apple and see what happens. No need to get fancy -- just type PRINT followed by the expression you want to evaluate and press Return at the end. After you're done, we'll explore the topic of line numbering.

LINE NUMBERS

As you have seen in the examples and sample programs, lines intended for deferred execution (i.e., lines that are part of a program) begin with a number. These numbers, known as line numbers, must be integers in the range 0-63999. Numbering simple programs is merely a matter of assigning some numeric value to each line and typing it into the computer. You could, for instance, number a three-line program like this:

```
0 PRINT "THIS IS A PROGRAM TO"
3281 PRINT "DEMONSTRATE HOW LINE NUMBERS"
63999 PRINT "ARE ASSIGNED."
```

or like this:

```
0 PRINT "THIS IS A PROGRAM TO"
1 PRINT "DEMONSTRATE HOW LINE NUMBERS"
2 PRINT "ARE ASSIGNED."
```

The screen should look like this in both cases when either program is run:

```
THIS IS A PROGRAM TO
DEMONSTRATE HOW LINE NUMBERS
ARE ASSIGNED.
```

Now, let's try a slightly different approach:

```
2 PRINT "ARE ASSIGNED."
0 PRINT "THIS IS A PROGRAM TO"
1 PRINT "DEMONSTRATE HOW LINE NUMBERS"
```

Type LIST and press Return. Notice anything different? The lines have rearranged themselves in numerical order. Now type RUN. You should get the same result as the first two examples. Let's try one more. Type NEW and then:

```
2 PRINT "THIS IS A PROGRAM TO"
1 PRINT "DEMONSTRATE HOW LINE NUMBERS"
0 PRINT "ARE ASSIGNED."
```

Your screen should display:

```
ARE ASSIGNED.
DEMONSTRATE HOW LINE NUMBERS
THIS IS A PROGRAM TO
```

Type LIST and you'll see why. We have just seen that Applesoft stores and executes program lines in ascending numerical sequence, regardless of the order in which they were initially typed.

Adding and Deleting Line Numbers

Let's say that you wanted to add the statement "ARE STORED AND EXECUTED ONCE THEY" between **lines 1 and 2** of the program above. Can it be done? Remember that line numbers must be integers and there is no integer between 1 and 2. If we had the foresight to number our program thus:

```
 0 PRINT "THIS IS A PROGRAM TO"
10 PRINT "DEMONSTRATE HOW LINE NUMBERS"
20 PRINT "ARE ASSIGNED."
```

then adding the line would simply be a matter of picking an integer between 10 and 20 and inserting the line:

```
 0 PRINT "THIS IS A PROGRAM TO"
10 PRINT "DEMONSTRATE HOW LINE NUMBERS"
15 PRINT "ARE STORED AND EXECUTED ONCE THEY"
20 PRINT "ARE ASSIGNED."
```

In fact, **line 15** could be inserted without retyping the original program. Remember, Applesoft will automatically insert the new line in proper numeric sequence.

Skeptical? Try typing NEW and **lines 0, 10 and 20** above. Now type LIST. So far, so good. Now type **line 15** as shown and type LIST again. As you can see, **line 15** has been added to the program.

What if you decided to eliminate **line 15** from our program? The simplest way is to type **15** and press Return. Then LIST the program again. Presto! No more **line 15**. You have just deleted the line. Applesoft does have a special command to delete lines: DEL. DEL may be used in two ways:

1. DEL followed by a single line number deletes only the line specified.
2. DEL followed by two line numbers, separated by a comma, deletes all lines within the range specified. For example, DEL 10,30 deletes all lines from 10 through 30.

LISTING PROGRAM LINES

We have used the LIST command several times during the preceding examples. Let's discuss it a little further. LIST followed by a Return will display the entire program in memory, from the lowest numbered line to the highest, on the screen. If the program is too long to be listed on the screen in its entirety, simply press Control-S to pause the listing and keep the lines from scrolling off the top of the screen. (Type Control-S by holding down the Control key and pressing the S key.) To resume the display of your program, press any character key. To terminate a long listing, press Control-C. To examine a portion of a program, type LIST followed by two numbers (separated by a comma) representing the lowest and highest lines for display. For example,

```
LIST 0,20
```

displays all the lines from 0-20 (inclusive). To examine a single program line, type LIST followed by the desired line number.

LIST adds extra spaces and formatting to make program lines on the screen more readable. Although this can be a problem in some circumstances, the extra readability afforded by formatting is worthwhile.

USING LINE NUMBERS EFFECTIVELY

Granted, it is difficult to get too worked up about line numbering. Normally this topic is glossed over in introductory BASIC texts. As you get a little more experience in programming, however, several line numbering techniques are usually learned the hard way:

1. Incrementing numbers by 10s, 20s or even 100s allows you to insert additional program lines that are written later to enhance (or correct) existing operations.
2. Distinctively numbering programs that could be used later as subroutines in other programs prevents the need for complete renumbering with each use. For instance, printing format routines could be numbered 5000-6000, number rounding routines 7000-7500, etc.
3. Some of the most important line numbers are those that precede REM (remark) statements. REM statements allow the programmer to insert an explanatory remark into the actual program. REM statements do not affect the way a program executes, but help to identify the function of a series of lines. Liberal use of REMs saves a lot of grief, particularly when you want to examine a program a long time after it was written. For example, the following line does not perform a function, but simply serves as a marker for a program section.

5000 REM START OF PRINTING FORMAT ROUTINE

Occasionally, we'll add a few suggestions for using line numbers. The importance of some of the preceding suggestions becomes immediately apparent as we delve into our next topic -- branching.

THE LONG AND SHORT OF BRANCHING

Applesoft normally executes programs in strict ascending line number order. **Line 1** runs before **line 2, line 2** before **line 3,** etc. To put it another way, program flow of control passes to the next highest numbered line after each line is executed. In the absence of commands to the contrary, flow of control always follows line number sequence.

Branching allows you to alter the normal flow of control in two significant ways:

1. Unconditional branching immediately shifts program execution to a specified line.
2. Conditional branching shifts program execution to a specified line only if certain conditions are met.

Let us first discuss the unconditional branching commands and then go on to the conditional variety.

GOTO *line number*

GOTO is the simplest branching command and unconditionally causes Applesoft to begin executing the statements at the line number specified. Since there are no parameters specified other than line number, GOTO simply serves as a traffic director within the program. For instance:

Listing 15

```
10 PRINT "IF YOU'RE NOT CAREFUL"
20 PRINT "WITH BRANCHING, YOU MAY END UP"
30 PRINT "WITH A PROGRAM THAT GOES ON"
40 PRINT "AND ON AND ON...AND ON"
50 GOTO 40
```

Try typing this program and RUNning it. The GOTO command in **line 50** sets up an endless loop between **lines 40 and 50**. When you get tired of seeing "AND ON" on the screen, press Control-C to stop the program. Of course, there are more meaningful uses for GOTO, as we will see a little later.

GOSUB *line number*

This command tells Applesoft to shift program execution to the subroutine beginning at the specified line number. Although we have alluded to subroutines earlier, let's take a minute to explain exactly what constitutes a subroutine.

The easiest way to visualize a subroutine is to picture a program within another program. A subroutine consists of one or more lines designed to handle a specific (and repeatedly used) function within a program. Rather than having to repeat the instructions each time, we can simply have the program execute the subroutine whenever the function is needed. The end of a subroutine is signalled by the RETURN command, which will be explained shortly.

We compared Applesoft to an interstate highway earlier in the chapter. If you visualize execution of a program as a drive down the interstate, subroutines are similar to rest stops. When you pull off the road and do whatever it is you need to do, your trip resumes where you left off. At the conclusion of a subroutine, control returns to the next numbered line in the main program.

The RETURN statement signals the conclusion of a subroutine and must be present if GOSUB is to work as intended. If no RETURN statement is present, GOSUB acts like the GOTO command and program control never returns to the intended line. Here's a short program that illustrates the use of GOSUB and RETURN.

Listing 16

```
 10 PRINT "THIS PROGRAM WILL COMPUTE THE SUM"
 20 PRINT "OF FIVE NUMBERS YOU SUPPLY"
 30 PRINT
 40 GOSUB 170
 50 T = T + A
 60 GOSUB 170
 70 T = T + A
 80 GOSUB 170
 90 T = T + A
100 GOSUB 170
110 T = T + A
120 GOSUB 170
130 T = T + A
140 PRINT
150 PRINT "THE SUM OF YOUR NUMBERS IS ";T
160 END
170 PRINT
180 INPUT "ENTER A NUMBER: ";A$
190 A = VAL(A$)
200 RETURN
```

Type the program and RUN it. The input subroutine (**lines 170-200**) handles the repetitive chore of requesting numbers.

There is another repetitive chore in the program -- keeping track of the running total T. Each time a new value for A is typed, it is added to T. Try changing the program so that the T = T + A chore is handled in another subroutine. You may also add T = T + A to the existing subroutine. Experiment with these simple subroutines a little until you are comfortable with how they operate. You will find that subroutines are one of the most effective ways of eliminating redundant program lines.

CONDITIONAL BRANCHING

As noted earlier, you may set up branching statements that execute only if certain conditions are satisfied. These statements are valuable because they enable you to build varying degrees of logic into your programs. Using conditional branching, you can develop programs that handle any number of contingencies.

You can combine the GOTO and GOSUB branching commands with conditional statements to create some interesting program possibilities. However, let's finish differentiating between the two types before we start combining them.

IF--THEN

IF--THEN is the primary conditional branching statement. In operation, the condition specified following IF must be true before the operation following THEN is executed. For instance:

```
100 IF X = 0 THEN STOP
```

would halt program execution only if X were indeed equal to zero. Any other value of X would result in the program ignoring the THEN STOP portion of the statement and continuing program execution at the next numbered line. For comparison purposes, the equal sign (=) in this case is referred to as a "relational operator" since it is charged with determining the relationship between X and 0. If X = 0, the relationship is true; if not, it is false.

Returning to our example, if **line 100** contained multiple statements:

```
100 IF X = 0 THEN PRINT "THE VALUE OF X IS 0": INPUT "ZERO IS
     NOT AN ACCEPTABLE VALUE. TRY AGAIN.";X: GOTO 100
```

none of the commands following THEN would execute unless the IF condition were met. In practical use, this feature allows the construction of short routines that take up no more than one line and are needed only if the specified condition is present.

The relationships specified in IF--THEN statements need not be limited to equalities. Numeric variables may be greater than (>), greater than or equal (>=), less than (<), less than or equal (<=) or not equal (<>) to another variable, value or expression. For example:

```
IF (X / Y) < (Y ^ 2) THEN STOP
IF X <= 100 THEN STOP
IF (X + 6)  ^ 2 >= X * Y THEN STOP
```

are all valid statements. String expressions or variables may also be used in IF--THEN commands:

```
IF X$ = "APPLE" THEN GOTO 100
IF Y$ <> "PROGRAM" THEN PRINT "WRONG!"
```

As you can see, IF--THEN is a most versatile conditional branching statement. We will be using this command repeatedly in upcoming examples. Take a little time to experiment and become familiar with it.

FOR--NEXT

FOR--NEXT provides a means of repeating an operation a specified number of times. Repetition of the same series of commands is known as iteration and the structure for providing iteration is popularly called a "loop." A simple loop like:

```
100 FOR I = 0 TO 2500 : NEXT I
```

causes the computer to count to 2500. Of course, nothing is displayed while the computer is counting, so these "empty" loops are sometimes used for creating delays.

More complex loops can easily be constructed through the use of FOR--NEXT statements. FOR must be followed by a numeric variable and range to set up a loop. For example, to construct a loop that repeats three times, the format would be:

```
100 FOR X = 1 TO 3
200 NEXT X
```

The variable X is assigned successive values of 1, 2, and 3 as NEXT X is encountered. By default, X increments by +1 on each repetition.

If a different increment is desired, a STEP statement must be added as follows:

```
100 FOR X = 3 TO 1 STEP -1 : NEXT X
```

In this instance, X would be assigned successive values of 3, 2, and 1. If this all seems rather confusing, don't despair. Like many other programming activities, FOR--NEXT loops are much easier to use than to explain.

Let's construct a short program using a loop to print the values of X as they are incremented:

Listing 17

```
10 FOR X = 1 TO 3
20 PRINT "THE VALUE OF X IS: ";X
30 NEXT X
```

When this program is RUN, the screen should display:

```
THE VALUE OF X IS: 1
THE VALUE OF X IS: 2
THE VALUE OF X IS: 3
```

NEXT may be used with or without the variable name specified in the FOR statement. Operation of the example above would be exactly the same if **line 30** read:

```
30 NEXT
```

Loops, like expressions in parentheses, may be nested. Applesoft allows nesting of no more than 10 FOR--NEXT loops. When using nested loops, use of the variable name in the NEXT statements helps to prevent confusion as to which variable is being incremented. Here's a simple nesting example that uses two loops and prints the value of each variable as it is incremented:

Listing 18

```
10 FOR X = 1 TO 3
20 FOR Y = 3 TO 1 STEP -1
30 PRINT "THE VALUE OF X IS: ";X
40 PRINT "THE VALUE OF Y IS: ";Y
50 NEXT Y
60 NEXT X
```

Type and RUN this program. Your screen should display:

```
THE VALUE OF X IS 1
THE VALUE OF Y IS 3
THE VALUE OF X IS 1
THE VALUE OF Y IS 2
THE VALUE OF X IS 1
THE VALUE OF Y IS 1
THE VALUE OF X IS 2
THE VALUE OF Y IS 3
THE VALUE OF X IS 2
THE VALUE OF Y IS 2
THE VALUE OF X IS 2
THE VALUE OF Y IS 1
THE VALUE OF X IS 3
THE VALUE OF Y IS 3
THE VALUE OF X IS 3
THE VALUE OF Y IS 2
THE VALUE OF X IS 3
THE VALUE OF Y IS 1
```

Notice that the loops do not (and should not) cross each other. The Y loop executes fully before incrementing the X loop. If you're still a little confused, try constructing some short programs using the FOR--NEXT commands to see exactly how they work.

ON--GOTO and ON--GOSUB

These two commands are used primarily to determine a program's course of action according to the value of a variable specified. Rather than launch into a lengthy explanation of these commands, let's illustrate one of them with a sample program. See if you can figure out from the example how these commands are supposed to work.

Listing 19

```
10 INPUT "TYPE AN INTEGER FROM 1 TO 5 ";X$: X = VAL(X$)
20 ON X GOTO 40, 50, 60, 70, 80
30 PRINT "WRONG VALUE, TRY AGAIN.":GOTO 10
40 PRINT "YOU TYPED THE NUMBER 1":GOTO 90
50 PRINT "YOU TYPED THE NUMBER 2":GOTO 90
60 PRINT "YOU TYPED THE NUMBER 3":GOTO 90
70 PRINT "YOU TYPED THE NUMBER 4":GOTO 90
80 PRINT "YOU TYPED THE NUMBER 5"
90 END
```

Type the program and RUN it.

Need some help understanding it? Notice that there are five line numbers following the ON X GOTO command. The value of X determines to which of these lines the program execution branches. If you enter a value of X outside the range 1-5, program execution continues at the next numbered line following the ON--GOTO statement.

The most obvious use for these commands is in program menus, which we will discuss in a future chapter. One other conditional branching command, ONERR GOTO, will be deferred for now (see chapter 8).

CALLING MEMORY LOCATIONS

In operation, **CALL** is remarkably similar to GOSUB. The primary difference is that CALL executes a machine language subroutine beginning at the memory location specified. Some predefined machine language subroutines are included in the Apple's ROM and may be called at will. CALL -936, for example, clears all of the characters inside the text window, and returns the cursor to the top leftmost printing position. (You can do the same thing with the Applesoft HOME command.)

It is possible to write your own machine language subroutines and integrate them into Applesoft programs through the use of CALL statements. Actually, machine language routines have distinct advantages in terms of faster execution and greater memory efficiency. Machine language programming is difficult, but certainly not impossible, to learn. There are many excellent manuals available if you are seriously interested in pursuing machine language programming.

THE FLAG

It is often necessary to repeatedly determine the state of certain conditions in a program. Rather than including multiple IF-THEN statements, it is usually easier to refer to a flag variable, the value of which can be changed as easily as flipping a light switch. A flag is simply a variable that is set and reset from within a program for a specific purpose. That simple little variable, however, helps you to determine program flow and perform a myriad of other chores.

By its nature, a flag must have a predefined range of acceptable values, and determine a course of action based on the present value. For instance, your flag variable SW could have three possible values in the print routines of a specific program. You may define the values something like this:

`SW=0` (Print to the screen alone)
`SW=1` (Print to the printer alone)
`SW=2` (Print to both screen and printer)

In the program itself, various features can be added or subtracted depending on the setting of SW. If SW=0, the full range of video tricks like inverse video, graphics, flashing, etc., can be utilized. If SW=1, all the features of your printer can be utilized without regard to the limitations imposed by the Apple display. If SW=2, the number of available features is limited to the capabilities common to your printer and the Apple display.

There is no need to write a completely separate PRINT routine for each condition of SW. Using a few IF--THEN statements will allow you to direct output to the appropriate slot, turn on the printer, set length of line output, and so on.

Flags are versatile little variables that can serve you well in your programming efforts. Expressive string variables can be used to help other programmers determine the nature of your flags and the meaning of their different settings. Learn to use them and you will be rewarded with many hours of effort saved.

CASH REGISTER PROGRAM

The program shown in **Listing 20** demonstrates several of the branching and other commands we have discussed. This program simulates a pretty basic cash register. You may want to type it into your computer, try it, and then see if you can add a few more features.

Listing 20

```
 10  DIM X(6):TP = 5: REM  5% SALES TAX
 20  REM  SALE TYPE SELECTION MODULE
 30  HOME : PRINT : PRINT  TAB( 14);"TYPE OF SALE:": PRINT :
     PRINT
 40  PRINT "1) CASH": PRINT : PRINT "2) CHARGE": PRINT :
     PRINT "3) REFUND": PRINT : PRINT "4) TOTALS": PRINT :
     PRINT "5) QUIT"
 50  PRINT : PRINT "CHOOSE ONE: ";: GET A$:TY =  VAL (A$)
 60  IF TY < 1 OR TY > 5 THEN  PRINT  CHR$ (7): GOTO 30
 70  ON TY GOTO 90,90,310,460,550
 80  REM   CASH/CREDIT SALE ENTRY MODULE
 90  N = 1:S = 1:TS = 0
 100  HOME : PRINT "PRESS RETURN AFTER EACH ITEM."
 110  PRINT "ENTER RETURN ALONE FOR LAST ITEM.": PRINT
 120  PRINT "ITEM #";N;"  ";: INPUT A$:X(S) =  VAL (A$)
 130  IF X(S) = 0 THEN  GOTO 160
 140  N = N + 1:S = S + 1: IF S < 6 GOTO 120
 150  REM  TOTAL THIS SALE
 160  FOR S = 1 TO N:TS = TS + X(S): NEXT S
 170  TS =  INT ((TS + .005) * 100) / 100
 180  PRINT : PRINT  TAB( 8);"TOTAL SALE:  ";TS
 190  TX =  INT (((TS * TP / 100) + .005) * 100) / 100: PRINT
     : PRINT  TAB( 9);"SALES TAX:  ";TX:TE = TS + TX: PRINT :
     PRINT " TOTAL AMOUNT DUE:  ";TE
 200  IF TY = 2 THEN  GOTO 260
 210  PRINT : INPUT "  AMOUNT TENDERED:  ";A$:TD =  VAL (A$)
 220  PRINT : PRINT  TAB( 12);"CHANGE:  "; INT ((TD - TE +
     .005) * 100) / 100
```

```
230  NT = NT + N - 1:GT = GT + TS:TT = TT + TX
240  PRINT : GOSUB 560
250  GOTO 30
260  REM   CHARGE SALES MODULE
270  NT = NT + N - 1:CT = CT + TS:TT = TT + TX
280  PRINT : GOSUB 560
290  GOTO 30
300  REM  REFUND/CREDIT ROUTINE
310  HOME : PRINT "PRESS RETURN AFTER EACH ITEM."
320  PRINT "ENTER 0 FOR LAST ITEM.": PRINT
330  N = 1:S = 1:TS = 0
340  PRINT "ITEM #";N;"    ";: INPUT A$:X(S) =  VAL (A$)
350  IF X(S) <  > 0 THEN N = N + 1:S = S + 1: IF S < 6 GOTO
    340
360  FOR S = 1 TO N:TS = TS + X(S): NEXT S
370  TS =  INT ((TS + .005) * 100) / 100
380  PRINT : PRINT "TOTAL OF ITEMS: ";TS
390  TX =  INT (((TS * TP / 100) + .005) * 100) / 100: PRINT
    TAB( 6);"SALES TAX:  ";TX: PRINT "  TOTAL CREDIT:  ";TS
    + TX
400  PRINT : PRINT : PRINT "1) CASH REFUND": PRINT : PRINT
    "2) CREDIT TO ACCT": PRINT : INPUT "SELECT ONE: ";A$:RF
    =  VAL (A$)
410  IF RF = 1 THEN NT = NT - N + 1:GT = GT - TS:TT = TT -
    TX: GOTO 430
420  CT = CT - TS:TT = TT - TX
430  PRINT : GOSUB 560
440  GOTO 30
450  REM  PRINT TOTALS
460  HOME : VTAB (3): PRINT  TAB( 14);: INVERSE : PRINT
    "SALES SUMMARY": NORMAL
470  PRINT : PRINT  TAB( 19);"ITEMS:";NT
480  PRINT :GT$ =  STR$ (GT): PRINT  TAB( 14);"CASH
    SALES:";GT$
490  PRINT :CT$ =  STR$ (CT): PRINT  TAB( 12);"CREDIT
    SALES:";CT$
500  PRINT :X = GT + CT:X$ =  STR$ (X): PRINT  TAB(
    16);"SUBTOTAL:";X$
510  PRINT :TT$ =  STR$ (TT): PRINT  TAB( 15);"SALES
    TAX:";TT$
520  PRINT :X = GT + CT + TT:X$ =  STR$ (X): PRINT  TAB(
    10);"TOTAL RECEIPTS:";X$
530  PRINT : PRINT : PRINT  TAB( 10);"PRESS RETURN FOR
    MENU";: INPUT Z$
540  GOTO 30
550  HOME : END
560  VTAB 23: INPUT "PRESS RETURN TO CONTINUE ";CH$: RETURN
```

Chapter 3 Summary: Flow of Control and Branching

Evaluation of Expressions

1. A combination of numbers (or variables) and operators is known as an expression.
2. The process of solving the problem posed by each expression is referred to as evaluation.
3. Arithmetic operators (also called algebraic operators) are + (addition), - (subtraction), / (division), * (multiplication) and ^ (exponentiation).
4. Operators are assigned priority for execution within an expression as follows:
   First priority -- exponentiation
   Second priority -- multiplication and division
   Third priority -- addition and subtraction
5. Operators having the same priority are executed from left to right within an expression.
6. Items and operators enclosed by parentheses in an expression are evaluated prior to any other operation.
7. Parentheses may be nested to further define the execution order of operators. Applesoft permits nesting of 35 levels of parentheses.

Line Numbers

1. All statements intended as part of a program must be preceded by an integer in the range 0-63999, called a line number.
2. Applesoft stores and executes program lines in ascending numerical sequence, regardless of the order in which the lines were typed.
3. Numbered lines may be added to a program at any time.
4. Program lines are displayed through the use of the LIST command:
   LIST displays the entire program.
   LIST *n* displays line *n*.
   LIST *n,m* displays all lines from *n* through *m*.
5. Numbered lines may be deleted from a program in the following ways:
   n Return deletes a single line.
   DEL *n* also deletes the single line specified.
   DEL *n,m* deletes all lines within the range of numbers stated.
   NEW deletes the entire program in memory.
6. Use line numbers with wide enough intervals to allow additional lines to be inserted with minimal effort.
7. Numbered REM (remark) statements help to identify the function of a series of lines within a program.

Subroutines

1. Subroutines can be visualized as short programs within a program and consist of one or more lines designed to handle specific (usually repeated) functions in the program.
2. Program flow of control is routed to the subroutine by the GOSUB *n* command, where *n* is the first line of the subroutine.
3. The end of a subroutine is signalled by the RETURN command. Program execution resumes at the next numbered line following the GOSUB command.

Branching

1. Allows the alteration of the normal flow of control within a program.

2. Unconditional branching commands (GOTO, GOSUB, RETURN and CALL) immediately shift program execution to a specific line.
3. Conditional branching commands (IF--THEN, FOR--NEXT, ON--GOTO and ON--GOSUB) shift program execution order only if certain conditions are satisfied.
4. Conditional and unconditional branching commands may be combined to initiate an action within a program.
5. Conditional branching commands make effective use of the relational operators.
   Equals (=)
   Not equal (<>)
   Greater than (>)
   Greater than or equal (>=)
   Less than (<)
   Less than or equal (<=)
5. CALL *memory location* is similar to GOSUB in that it directs program execution to the memory location specified.

Flags

1. A flag is simply a variable that can be set and reset from within a program for a specific purpose.
2. Flags must have a predefined range of acceptable values and determine a course of action based on the present value.

Chapter 4: The Ins and Outs of Storage

So far, we have concentrated on programming basics with little regard for what to do with completed programs. Sure, you can type a program into your computer every time you need it and then supply data from the keyboard. But that can become extremely tedious. After you have typed the same program three or four times, program storage becomes very attractive! Let's take a break from learning programming commands and concentrate on the ins and outs of storage.

One form of storage is that found in the Apple's RAM (random access memory). The computer types who sold you your Apple no doubt threw around some funny sounding numbers like 64K and 128K, impressing upon you the fact that more of those Ks are better.

Who's K? Well, K (or Kbyte) is one of those innumerable abbreviations used in computing; it stands for one kilobyte, which is actually 1024 bytes. A byte is one complete computer word consisting of 8 bits. Suffice it to say that most characters can be represented in one byte of memory.

The computer uses up some bytes for its own operations, Applesoft uses some more, and other peripherals (like disk drives) reserve a few for themselves. What's left over is used for program and data storage.

Apple also utilizes another type of storage, ROM (read only memory). ROM is permanently programmed with routines needed to operate the computer and is not accessible for use in program and data storage. It is the information contained in ROM that gives your Apple its "personality" and operating capabilities.

WHY EXTERNAL STORAGE?

With all that memory you should be able to store a lot of programs and data in the computer itself, right? Sounds like a fine idea, but your Apple has a one-track mind. Remember typing NEW before entering the demonstration programs in the first few chapters? Each time you did that, the Apple "forgot" the last program that was in memory. If you forgot to type NEW, the program being typed in was just added onto the one in memory.

To further compound the problem, every time the power is shut off (or goes out, even for a split second) your Apple has instant amnesia. When power is restored, the contents of ROM will still be there to provide the operational capacity of the system. Any program or data contained in RAM, however, is gone.

Your Apple's RAM is great for its intended purpose -- reading and writing during the execution of a program -- but it is obviously not practical for long-term storage. Disk storage is intended to solve these problems.

DISK STORAGE

The two most common forms of disk storage used by Apple owners are the 5 1/4 inch disk (sometimes called a floppy disk) and the 3 1/2 inch disk (sometimes called a micro-floppy). The disk looks very much like a thin phonograph record coated with magnetic material and enclosed in a square envelope or plastic case.

A disk drive is a mechanism specifically designed to handle these disks. Each disk drive contains a motor to spin the disk at a constant speed, a read/write head for recording and playback, and miscellaneous electronic components.

The procedures used by your computer to handle disk read and write are called its disk operating system (DOS). The disks that came with your computer are probably formatted for either DOS 3.3 or ProDOS. Our discussion is limited to operations that are identical to both operating systems.

The commands and capabilities associated with disk storage comprise a relatively lengthy Apple manual, so we will do little more than summarize them in this chapter. For more specific information, study the manual supplied with your version of DOS.

BOOTING DOS

To translate the title of this section into plain English, "booting DOS" refers to the process used to add disk commands and capabilities to your Apple. (Somehow the term always conjures up other, less attractive images in my mind.) Put your DOS 3.3 System Master disk or the ProDOS User's Disk into the drive as instructed in your DOS manual, and type:

```
PR#s
```

where *s* is the slot in which your disk controller card is located, usually slot 6. Though the IIc doesn't have slots, the built-in disk drive is treated as if it were attached to slot 6. The "in use" light on your drive should go on. In addition, you will hear some rather strange noises coming from the drive itself. After a few seconds, a message appears on the screen to verify that your disk operating system has indeed booted properly.

What then? Well, now you need to tell your Apple to do something. Your computer behaves in much the same fashion as it did before, with some exceptions, as we shall see later.

INITIALIZING DISKS

The first thing your should do is initialize a blank disk or two. When you buy blank disks, there is absolutely nothing on them. If you plan to use them to store information, the disks must be initialized or formatted, so that your computer can use them for data storage. Let's do one right now:

For ProDOS users with the ProDOS User's Disk:

1. Boot your ProDOS User's Disk
2. After the disk boots, select option F for PRODOS FILER
3. Press V for VOLUME COMMANDS. Then press F for FORMAT A VOLUME. Enter your disk drive slot and the drive number of the blank disk. If you have two disk drives, put your blank disk in the second disk drive. Enter WORK.DISK for the volume name. The disk drive will then spin for about 45 seconds.
4. After the format operation is complete, press the Escape key twice to return to the main menu.
5. Press F for FILE COMMANDS, then C to COPY FILES.
6. FILER will now ask for the disks to copy to and from. Press the question mark key (?) and Return. Then type: /WORK.DISK/? Press Return twice.
7. Answer Y to copy both PRODOS and BASIC.SYSTEM. Then press the Escape key three times to return to the main menu.

8. Place the newly created disk in drive 1 and reboot your machine by holding down the Control and Open-Apple keys while pressing the Reset key.
9. When the disk boots, it should present you with a BASIC prompt ( ]). Type NEW and press Return, then enter the following program:

```
10 REM STARTUP PROGRAM
20 HOME: VTAB 5: PRINT "WORK DISK NUMBER 1"
30 PRINT "FORMATTED (date)"
40 END
```

10. Save the program on disk with the command: SAVE STARTUP

For ProDOS users with the System Utilities Disk:

1. Boot your System Utilities disk. Apple IIGS owners should boot the Apple IIGS System Disk and select /SYSUTIL.SYSTEM from the /SYS.UTILS folder. If asked, answer either Y or N for 80-column display.
2. Choose the option to FORMAT A DISK. Then enter the slot and drive of your disk drive and choose PRODOS as the operating system. Enter /WORK.DISK for the volume name. Place the blank disk in the slot and drive that you selected and press Return. Wait for the disk to format.
3. After the format operation is complete, press the Escape key to return to the main menu.
4. From the main menu, choose the option to copy files. Copy the files BASIC.SYSTEM and PRODOS from your system disk to your work disk. Apple IIGS users should copy BASIC.SYSTEM and P8 from the system disk (The P8 file can be found either in the main directory on your system disk or in the /SYSTEM subdirectory). Then use the RENAME option to change the name of P8 to PRODOS. Once the disk is formatted and the two files are copied to your work disk, your work disk should boot.

For DOS 3.3 Users:

1. Boot your DOS 3.3 System Master disk.
2. Clear memory by entering NEW and pressing Return.
3. Enter the following sample greeting program:

```
10 REM HELLO PROGRAM
20 HOME : VTAB 5: PRINT "DATA DISK NUMBER 1"
30 PRINT "INITIALIZED (date)"
40 END
```

4. Remove the DOS 3.3 System Master disk from the disk drive and insert the blank disk you wish to initialize.
5. Type INIT HELLO.

Both disk operating systems automatically attempt to execute a greeting program upon booting. Under ProDOS, when you boot the disk, a program named STARTUP will automatically be run, if it is present. Under DOS 3.3, the program named when you initialize the disk (in most cases, HELLO) will automatically be run.

Now let's see if your disk works. Put it in the drive and type *PR#3,* just as you did when booting DOS. Soon, you should see the two lines from your greeting program printed at the top of the screen. Just for the fun of it, type LIST. Did you see the program?

A little later we will discuss some ways of getting your Apple to run another program during the boot process.

SAVING AND LOADING PROGRAM FILES

Boot one of your data disks, type NEW and then type in a program. (How about the PRINTING.DEMO program from chapter 1?) When you're finished, RUN the program to make sure you got it right.

Now type SAVE DEMO. The drive should run for a second or two. Type CATALOG to display the contents of the disk on the screen. If you are using ProDOS, simply type CAT. Under DOS 3.3, two programs should be listed -- HELLO and DEMO -- both preceded by an "A" indicating that they are Applesoft programs. ProDOS users will see ProDOS, BASIC.SYSTEM and DEMO. DEMO is followed by BAS to indicate that it is a BASIC program.

Now let's see if the SAVE worked. Type NEW and LIST to be sure the program is no longer in memory. Now type RUN DEMO. DOS will retrieve the program from the disk and automatically RUN it.

Try typing another program or two and saving them onto your data disk. Each time you do so, the disk catalog should list the new programs.

Choosing a File Name

Here are a few simple rules you should keep in mind when choosing a file name:

1. Under DOS 3.3, file names may be up to 30 characters long. Under ProDOS, they are limited to 15 characters.
2. Under DOS 3.3, file names must start with a letter and may contain any character except a comma. Under ProDOS, file names must begin with a letter and may only contain letters, digits and periods.
3. If you SAVE a program using a name that already exists on that disk, the old file is replaced by the program in memory when you issue the SAVE command.

Changing File Names

Once you have named your file, the name is not cast in concrete. File names may be changed readily using the RENAME command. To use this command, type RENAME followed by the current file name, then a comma (,), then your new file name.

Let's suppose, for instance, that you got tired of typing RUN DEMO every time you wanted to use that program. Let's choose a shorter name that's easier to type: D.D. To change the program's name, just type RENAME DEMO, D.D. Type CATALOG to see what has happened. The old DEMO program is now known as D.D. Just to prove that it's official, type LOAD DEMO. What happened? The Apple told you the file was no longer on the disk by printing a FILE NOT FOUND message. It's a good habit to give programs meaningful names so that you can tell what is in a file when you use the disk later.

DELETING FILES

We have two program files on our disk. Let's add one more the easy way. First type LOAD D.D. Once the disk has stopped running, type SAVE DEMO.COPY. If you catalog the disk again, you will see that you have saved another copy of D.D. under the name DEMO.COPY.

Now let's unceremoniously remove DEMO.COPY from the disk. Type DELETE DEMO.COPY. That's all there is to it -- just type DELETE and a program name.

LOCKING AND UNLOCKING FILES

Once a program or data file is locked, it may not be deleted from the disk or renamed without first unlocking it. Using these commands is quite simple -- type LOCK *program.name* or UNLOCK *program.name*. Locked files are indicated in the disk catalog by an asterisk to the left of the file name. If you try to delete or rename a locked file, the Apple will chastise you with a FILE LOCKED error message.

USING DISK COMMANDS

So far, all the DOS commands we have used were carried out in the immediate execution mode. That means they may simply be typed (followed by Return) and will be executed at once.

DOS commands may also be issued from within program lines, or as we defined it earlier, in deferred execution mode. To use disk commands within a program, the PRINT statement is combined with a special character to let your Apple know that that the string to be printed is really a disk command. The special character is a Control-D, and to understand its use, we need to take a quick look at how your Apple stores characters.

Like everything else in a computer, characters are really stored in the computer as numbers (often referred to as a character's ASCII code). For example, the number 65 represents the letter A. Applesoft has a special function, CHR$, which allows you to use the numeric form of a letter, rather than the letter itself, in a program. For example, the statements:

```
10 PRINT "A"
20 PRINT CHR$(65)
```

would both display the letter A on your screen. The numeric code for Control-D is 4, so to include it in our PRINT statement, we type:

```
10 PRINT CHR$(4);"CATALOG"
```

Instead of printing the word CATALOG, this one program line will cause the disk catalog to be displayed. (ProDOS users may want to substitute CAT for CATALOG for a 40-column display.)

Since disk commands are often used throughout a program, it has become standard practice to assign Control-D to the string variable named D$. Using this approach, the one-line program would become:

```
10 D$ = CHR$(4)
20 PRINT D$;"CATALOG"
```

All of the disk commands we have discussed thus far may be used in either immediate or deferred execution modes. A few of the commands we discuss -- OPEN, APPEND, READ, WRITE and POSITION -- may be used only from within a program line. The reasons for this limitation will become clear as we discuss each of these statements.

Here are a few examples of "legal" disk command statements:

```
10 PRINT CHR$(4);"CATALOG"
20 D$ = CHR$(4)
30 PRINT D$;"RENAME DEMO,D.D."
40 PRINT D$;"CATALOG"
50 PRINT D$;"LOCK D.D"
```

```
60 PRINT D$;"CATALOG"
70 PRINT D$;"RENAME D,DEMO"
80 PRINT D$;"UNLOCK DEMO"
```

DATA FILES

So far, our discussion of disk storage has centered on program files. You can, of course, save data in disk files that are called text files.

There are two types of text files commonly used in disk storage: Sequential and random access. Under DOS 3.3, text files of either type are identified by the letter T preceding the file name. Under ProDOS, TXT follows the file name.

Data items in sequential text files are stored one after the other, with a Return character (ASCII code 13) acting as a separator. Using cr as an abbreviation for CHR$(13), this is how items in a sequential text file would look:

```
FREDcrJOHNSONcr12 MAIN STcrANYTOWNcrNY
```

Several commands are used from within programs to create and manage sequential text files on disk:

1. OPEN
2. CLOSE
3. READ
4. WRITE
5. APPEND
6. POSITION

In addition, several commands we discussed earlier for program storage work pretty much the same way for text files:

1. DELETE
2. RENAME
3. LOCK
4. UNLOCK
5. VERIFY

Creating a Sequential File

There is a normal set of procedures that must be followed to create a sequential text file and save data in it. The commands, in order, are OPEN, WRITE, PRINT and CLOSE. Let's examine the purpose of each of these in greater detail.

OPEN causes DOS to set aside some workspace in memory for the contents of the file and tells the system to READ or WRITE from the beginning of the file.

The WRITE command directs the output of any subsequent PRINT commands to the disk file instead of to the screen. The file name must be the same as the one used in the OPEN statement. WRITE is cancelled by any other DOS command contained in a PRINT statement.

PRINT is used to actually place the data into the disk file. When used in this fashion, PRINT need not be followed by CHR$(4).

CLOSE releases the workspace in memory and secures the file from further use for the moment. Any file that has been opened and has received data must be closed.

Let us try a short program that will create and save a text file on disk:

Listing 21

```
10 D$ = CHR$(4): REM Control-D
20 PRINT D$;"OPEN SEQUENTIAL.FILE"
30 PRINT D$;"WRITE SEQUENTIAL.FILE"
40 PRINT "THIS IS THE FIRST ITEM"
50 PRINT "FOLLOWED BY THE SECOND"
60 PRINT "THIRD"
70 PRINT "AND FOURTH"
80 PRINT D$;"CLOSE SEQUENTIAL.FILE"
90 END
```

After running this program, your disk directory should list SEQUENTIAL.FILE when you issue a CATALOG command.

Appending Your File

Before we retrieve the data written earlier, let's add a few more items using the APPEND command. Here's a short program that will show you how to do it:

Listing 22

```
10 D$ = CHR$(4): REM Control-D
20 PRINT D$;"APPEND SEQUENTIAL.FILE"
30 PRINT D$;"WRITE SEQUENTIAL.FILE"
40 PRINT "USING APPEND WE CAN ADD"
50 PRINT "A FIFTH; SIXTH OR"
60 PRINT "ANY NUMBER OF ITEMS"
70 PRINT D$;"CLOSE SEQUENTIAL.FILE"
```

Rather than tell you what has happened, I'll let you see for yourself in the next section.

Retrieving Data from a Sequential Text File

Now that we've created the file, let's find out how to get our data out! For this example, we'll use the disk commands OPEN, READ and CLOSE.

Listing 23

```
 10 D$ = CHR$(4): REM Control-D
 20 PRINT D$;"OPEN SEQUENTIAL.FILE"
 30 PRINT D$;"READ SEQUENTIAL.FILE"
 40 FOR I = 1 TO 7
 50 INPUT A$(I)
 60 NEXT I
 70 PRINT D$;"CLOSE SEQUENTIAL.FILE"
 80 FOR I = 1 TO 7
 90 PRINT A$(I)
100 NEXT I
110 END
```

Notice the use of the FOR--NEXT loop to INPUT and PRINT the data items. This is a perfectly acceptable procedure and does not interfere with the disk commands in any way. Try entering and running this program to see for yourself. Did you see both the original items and those we appended? Good for you!

Positioning the File Pointer

When you are using text files on disk, a pointer is used to identify items in the file. OPEN automatically sets the file pointer to the first item in a file. If you want to read or write a specific item in the file, POSITION can provide the capability. The format used is POSITION, R(*n*) where *n* represents the desired record relative to the current pointer position. The value of *n* determines how many records forward of the current one will be used in the specified operation. Although we won't dwell at any length on POSITION at this time, its use will be illustrated in some of the later program examples.

Chapter 4 Summary: The Ins and Outs of Storage

Take some time to experiment with the commands we have covered to this point. Try adapting some of the programs presented in the first three chapters for disk storage use, or write some simple programs that will help you understand how disk storage works.

Memory

1. K is an abbreviation for Kilobyte, which commonly refers to 1024 bytes of memory.
2. ROM (read only memory) is permanently programmed memory that is not accessible for user-generated programs or data.
3. RAM (random access memory) is volatile and intended for active program storage and READ and WRITE operations during the execution of a program. It is not suitable for long-term program or data storage. Any power interruption will cause the complete loss of data contained in RAM.

Disk Storage

1. DOS (disk operating system) is a series of software routines used to handle disk read and write procedures and to operate the drive(s). The two disk operating systems for the Apple II are DOS 3.3 and ProDOS. DOS is "booted" from Applesoft using a PR#*s* command (where *s* is the number of the slot that contains the controller card).
2. Blank disks must be formatted before they can be used to store information.
3. To store an Applesoft program on disk, just type SAVE *program name.*
4. Program and data file names must start with a letter. Under DOS 3.3, file names may be up to 30 characters long and may contain any character except a comma. Under ProDOS, file names may be up to 15 characters and may contain only characters, digits, and periods.
5. Files may be renamed using the RENAME *old name*, *new name* command.
6. Files may be erased from a disk using the DELETE *file name* command.
7. LOCKing a file prevents it from being accidentally renamed or deleted. UNLOCK is used to restore normal operations.
8. DOS commands used in deferred execution mode (i.e., from within programs) must be contained in a PRINT string and preceded by Control-D. The format is PRINT CHR$ (4); "*command* ".

9. Usually D$ is defined as Control-D (CHR$ (4)) early in a program (with the statement D$ = CHR$ (4)), allowing disk command PRINT strings to read: PRINT D$; "*command*".

Text Files

1. Two types of text files are commonly used in disk storage systems: Sequential and random access.
2. Text file names are preceded by a "T" in the DOS 3.3 catalog, and are followed by TXT under ProDOS.
3. Items in a sequential text file are stored one after the other, separated by a Return character (ASCII code 13).
4. OPEN tells DOS to set aside file buffer space in memory and begin READ or WRITE operations from the beginning of a file.
5. WRITE directs the output of PRINT commands to the disk file instead of to the screen.
6. PRINT is used to actually place data into a disk file.
7. CLOSE releases the workspace (buffer) in memory and secures a file from further use at the moment.
8. APPEND allows you to add additional data items to a sequential text file. New items are added after the last one presently contained in the file.
9. READ is used to retrieve data from a file. INPUT or GET commands following READ accept data from a disk file instead of from the keyboard.
10. POSITION is used to move the file pointer to a specific record in a file. The format is POSITION,R*n* where *n* is the record (relative to the one currently in use) desired. OPEN sets the current item pointer to the beginning of the first record.

Chapter 5: Beyond the Basics of Storage

Sequential text files consist of a number of records separated by a Return character and stored one after the other on a disk. In order to retrieve one of the pieces of information, it is necessary to read them all up to that point.

Essentially, the operation of a sequential text file is like a tape cassette containing several pieces of music. You may listen to each of the songs in order, or any one of the songs by locating the end of the previous selection. It is important for you to locate the proper point on the tape since each song may be of any length.

For some types of data, the sequential text file is ideal. Word processing text files, for example, are almost always sequential files because they may range from a few to several thousand characters in length. There are other types of data, however, that would be more useful if faster access to selected portions of the file could be obtained. For this purpose, random access file capabilities are included in your Apple's disk operating system. Random access files offer the advantage of rapid access to and efficient management of the types of data to which they are best suited.

BIG DEAL

One way to visualize the difference between sequential and random access files is to imagine that you have two decks of cards. One is a sequential deck, one is a random access deck.

The cards in the sequential deck can be any size. In other words, there is no fixed length for the information that can be stored on any one card. If you want to access the information on a specific card, say card number 33, in the sequential deck, you have to begin with the first card, and read each card in sequence until you reach the thirty-third card. The only way to access any card in the sequential set is to read all the cards that precede it first.

The cards in the random access deck are all the same size. Each card can contain the same amount of information as all the others, and no more. However, if you want to access the information on card 33 in the random access deck, you don't have to read any of the other cards. You can go directly to card 33 and read the information there.

As you can see, the advantage to sequential files is that records in the file can be of variable length. The disadvantage is that access speed is slow. The advantage to random access files is that access speed is fast. The disadvantage is that each record must be a fixed size.

RANDOM ACCESS FILES

Four basic commands are used to handle random access files. You are already familiar with them, although we will be adding a few things for this application. OPEN serves the same purpose we learned earlier under sequential text files. For random access text files, OPEN must also include a record length parameter. The length specified must be for each individual record and not the length of the total file. The format for this command is as follows (assume D$ has been assigned to Control-D using the statement D$ = CHR$(4)):

```
PRINT D$;"OPEN RAN.FILE,Ln"
```

where *n* is the record length.

When using OPEN with random access files, the name of your file must be followed by the length parameter each time the file is OPENed, since the length is used to calculate the record position.

Calculating Record Length

To set up a random access file that reflects the card data in the earlier example, we need to determine the record length. We have three items of data -- color, suit and number. Let's look at the longest possible length of each of these: BLACK (5 characters), DIAMOND (7 characters) and 13 (2 characters). Since we want to keep all three items in each record, two Return characters are added to separate them. With a final Return character, that gives us 14 characters and 3 Returns, or a total of 17 possible characters per record. Each character will take one byte, so the length parameter for your card data should be 17.

Although this is a simple example, the process is the same for virtually any type of information storage. Just remember to calculate your record length figures on the longest possible data and you won't go too far wrong. As a general rule, it is better to specify too much room than too little. Once the structure is in place, using a random access file is almost as simple as using the sequential variety.

Setting the Record Straight

We have determined how long each record will be in a random access file, OPENed it and are now ready to READ and WRITE data. Complicated? Not at all. Use READ and WRITE just as you do with sequential files -- just add a record number. The record number, as its name implies, designates which one of the group of fixed length records you want to manipulate. Each record may be thought of as a small sequential text file. If we return to our deck of cards for an example, the top card would be record one and the bottom card record 52. Any value between those figures would give you a valid card record number.

CLOSE Is Simple

The good old CLOSE command is exactly the same for random access files as for sequential files. No length record numbers or any other parameters are required. When you are finished using either type of file, a CLOSE *file.name* command will do the job nicely.

Try It, You'll Like It!

Remember the television commercial that used that line? Well, we can say the same thing about random access files. Although they seem difficult to set up and use, the additional versatility afforded by this type of file is well worth the extra effort. Let's set up a random access file containing the playing card data we have been talking about to see how it's done:

Listing 24

```
 10  REM  MAKE.CARDS
 20  D$ =  CHR$ (4): REM  CONTROL-D
 30  HOME : VTAB 7: PRINT "CREATING CARDS FILE..."
 40  PRINT D$;"OPEN CARDS,L17"
 50  FOR N = 1 TO 13
 60  PRINT D$;"WRITE CARDS,R"N
 70  PRINT "RED": PRINT "HEART": PRINT N
 80  NEXT
 90  FOR N = 14 TO 26
 100  PRINT D$;"WRITE CARDS,R"N
 110  PRINT "BLACK": PRINT "SPADE": PRINT N - 13
 120  NEXT
 130  FOR N = 27 TO 39
 140  PRINT D$;"WRITE CARDS,R"N
 150  PRINT "RED": PRINT "DIAMOND": PRINT N - 26
 160  NEXT
 170  FOR N = 40 TO 52
 180  PRINT D$;"WRITE CARDS,R"N
 190  PRINT "BLACK": PRINT "CLUB": PRINT N - 39
 200  NEXT
 210  PRINT D$;"CLOSE CARDS"
 220  HOME : VTAB 7: PRINT "CARDS FILE COMPLETED"
```

If you study this program for a minute, it is apparent that we are simply supplying data here -- no other operations are included. Now that we have the data in a file, let's write a short program to read the file and display the data for any selected card. The following program lets you enter a card number, and displays the data for that card.

Listing 25

```
 10  REM  READ.CARDS
 20  D$ =  CHR$ (4)
 30  TEXT : HOME : NORMAL : VTAB 3
 40  INPUT "ENTER CARD # (1-52 OR Q TO QUIT): ";A$
 50  IF A$ = "Q" THEN  HOME : END
 60  A =  VAL (A$)
 70  IF A < 1 OR A > 52 THEN  GOTO 30
 80  PRINT D$;"OPEN CARDS,L17"
 90  PRINT D$;"READ CARDS,R"A
 100  INPUT C$: INPUT S$: INPUT V
 110  PRINT D$;"CLOSE CARDS"
 120  VTAB 10: PRINT "THE COLOR IS: ";C$
 130  PRINT "THE SUIT IS : ";S$
 140  PRINT "THE VALUE IS: ";V
 150  VTAB 22: INPUT "PRESS RETURN TO CONTINUE";A$
 160  GOTO 30
```

MULTIPLE DISK DRIVES

Until now, we have assumed that you have one disk drive in your Apple system. Many Apples are equipped with two or more drives. For those with more than one drive, the SLOT and DRIVE options may be exercised to handle the additional drives.

If your system has two disk drives, chances are they are both connected to the same controller card in your machine. Usually this card is located in slot #6, although it is not required to be in that particular slot. Changing from one drive to another in this case is simply a matter of adding D1 or D2, preceded by a comma, to some of your disk commands.

By default, the drive from which you boot DOS is automatically selected for subsequent operations until a command that specifies a different drive is encountered. Once D2 is specified, drive 2 becomes the recipient of any disk commands until a different drive is selected. The operational effect is similar to a toggle switch that remains in place until you change it. The drive number selection may be added to almost any disk command. For example:

```
CATALOG,D2
LOAD PROGRAM,D1
DELETE TEXTFILE,D2
INIT HELLO,D1
```

Those with more than two disk drives will also have multiple controller cards to deal with. Each of these cards must be in a different slot and may be connected to either one or two disk drives. Controller card selection is made with the S*n* command. Like the drive number command just discussed, the slot from which DOS is booted remains the default slot until it is changed by user command. The order in which slot or drive number selection appears is unimportant, so long as they are both preceded by a file name. For instance:

```
LOAD GAME,S6,D2
SAVE TEXTFILE,S7,D1
RUN PROGRAM,D2,S6
DELETE OLDFILE,D1,S7
```

PRODOS PATHNAMES

Although you can use the same slot and drive designations with ProDOS, you have the additional option of using pathnames. Under ProDOS, each disk (called a volume) has a name that begins with a slash (/). You can access the volume by specifying its name. For example, if you have a ProDOS volume named /TEST, you can catalog that disk by typing:

```
 CAT /TEST
```

no matter what drive it's in. ProDOS will check each connected drive until it finds the volume named /TEST.

By combining the disk name with a file name (and possibly some directory names) you can create a pathname that describes the location of a file. For example, if there is a file on the /TEST volume named MYFILE, you can access that file by specifying /TEST/MYFILE as the pathname.

ProDOS includes the option of creating directories on a volume. These act like separate, smaller volumes and allow you to subdivide a volume into logical partitions. For example, you can create a directory called UTILITIES in which you store utility programs, or a directory called GRAPHICS in which you store painting programs and pictures. To access a file that is in a directory, you must specify the volume name, the directory name, and the file name. For example, if you have a program called PAINT in the GRAPHICS directory of the /TEST disk, you could run it by typing:

```
RUN /TEST/GRAPHICS/PAINT
```

The hierarchical nature of ProDOS is most useful with large volumes, such as 800K floppies and hard disks. It is of limited use on 5 1/4 inch floppies, because you can't store that much on them anyway.

DISK UTILITIES

Both Apple II disk operating systems for Applesoft (DOS 3.3 and ProDOS) are supplied with utility programs to facilitate disk and file operations. COPYA is one of the most frequently used DOS 3.3 utilities and is the program called for if you are using DOS from Applesoft. This program provides a convenient way to copy the contents of an entire disk on to another. Excellent on-screen prompting is included, so you should have no problems using it. Just be sure to write-protect your original disk (cover the little notch on the upper right side of the disk) to guard against inadvertently losing valuable data.

FID is a versatile program for manipulating files. To use it, type BRUN FID with the DOS 3.3 System Master in the drive. By selecting the proper item from the ones listed in the menu, you will be able to catalog disks; reset the slot and drive; and lock, unlock, verify, delete or copy files. Space remaining on the disk may be displayed by selecting the space option.

But don't you have all these commands at your disposal already? Indeed you do, at least to a point. I think the file copying routines are the most valuable features of FID. Selecting COPY FILES allows you to copy specified files to another disk either automatically or in response to program prompts.

ProDOS users will find all of these features and more in the Filer utilities on the ProDOS User's Disk, or SYSUTIL.SYSTEM on the IIGS System Utilities disk. Both programs let you: catalog disks; list the volumes on-line; duplicate disks; format disks; create directories; rename volumes; and copy, rename, delete, lock and unlock files.

DISK TRICKS

We are not able to discuss all of the fine points of DOS 3.3 or ProDOS. After all, the DOS manuals are rather comprehensive. What we can do, however, is reinforce a few of the points made in the DOS manuals and offer some suggestions for things that may not have been included. Now that you have a basic understanding of what DOS is and does, let's take a look at some of the more useful "tricks" available to the beginning programmer.

Variable Volume

You can specify a volume number between 1 and 254 for each disk you initialize under DOS 3.3 with the INIT command: INIT HELLO,V254. Any disk access command can also contain a volume number specification. The light may be already turning on in your head -- why not use volume numbers to verify the presence of a correct disk in the drive? Good thinking. Adding V*n* to a command tells DOS to check the disk volume number and issue an error message if the number does not match. ProDOS does not use volume numbers because all volumes are named.

Genuine Imitation Chaining

An interesting feature included in some operating systems is the ability of one program to initiate the operation of another program. Most often, variables may be shared and common data used. Some provisions are included on the DOS 3.3 System Master for using this capability in Applesoft, although the process is a little complicated. You may, however, use a less elegant (but entirely adequate) method which consists of typing the desired command as a statement within a program:

```
510 PRINT D$;"RUN program name"
```

As you learned earlier, typing RUN clears variables and data from memory. But if your program does not require a great deal of data, just re-read the associated files in the first part of the program called. The process takes only a few seconds.

The Chief EXEC

One of the most interesting applications of the sequential text file uses the EXEC command. EXEC takes program lines or commands from a text file and treats them as though they were being typed from the keyboard. Spend a few minutes looking over the following explanation -- then be sure to sit down at your own computer and actually try it.

EXEC may be used to control a sequence of events, including the running of programs. As each event is completed, the EXEC command processes the next. To use this capability, you must first create a text file. A short program for that purpose is shown below (If you are using ProDOS, substitute STARTUP for HELLO in **line 40**):

Listing 26

```
10 D$ = CHR$(4): REM Control-D
20 PRINT D$;"OPEN AUTOFILE"
30 PRINT D$;"WRITE AUTOFILE"
40 PRINT "RUN HELLO"
50 PRINT "CATALOG"
60 PRINT "LIST"
70 PRINT "HOME"
80 PRINT D$;"CLOSE AUTOFILE"
```

After the program has finished its operations, examine the catalog. Do you see the AUTOFILE file on your disk? So far, so good. If you type EXEC AUTOFILE, it will initiate the following sequence of events:

1. RUN the HELLO (or STARTUP) program on your disk.
2. CATALOG the disk.
3. LIST the program.
4. Clear the screen (HOME).

Your disk should contain a HELLO program; if not, rename some other program HELLO. Now type EXEC AUTOFILE, sit back and watch.

EXEC is somewhat different from LOAD or RUN. Each time a program is RUN or LOADed, EXEC waits until the program is finished and then continues its operations. That means, of course, that the EXEC file remains in memory despite the fact that RUN or LOAD normally clears whatever else is there.

Aside from the multiple operation capabilities that you have just seen, EXEC holds some exciting features for programmers. Using this command, you can make a text file out of a file containing a program listing. Once a program listing has been transformed into a text file, it may be inserted into another program, edited with a text editor, etc. As the text file is EXECed, the effect is the same as if the lines were being typed from the keyboard.

To place a program into a text file, a short routine must be added to the front of the program. This takes only one line and may use line zero, which is rarely used. For example:

```
0 D$=CHR$ (4) :PRINT D$"OPEN LISTFILE" :PRINT D$"WRITE
     LISTFILE" :LIST 1,63999: PRINT D$"CLOSE": END
```

After you have added this line to your program, type RUN. Instead of your program running, a text file that contains all the lines of the program is created, called LISTFILE. You may rename it to something else if you like. Delete line zero when you're finished, if you want to run your program again.

Try setting up some EXEC files and using them. The opportunities for programmer creativity should become immediately apparent.

Chapter 5 Summary: Beyond the Basics of Storage

We could devote a lot more space and still not cover DOS and storage procedures as thoroughly as possible. It would be virtually impossible to explain all the subtle situations and circumstances necessary to handle every problem that might arise during storage operations. Try each of the procedures discussed. Experiment enough to get comfortable with the process. Make a few mistakes! See how far you can go before problems surface. You can't damage anything in the Apple, so "have at it" and learn by doing.

Sequential Text Files

1. Sequential text files consist of any number of records separated by a carriage return and stored one after the other.
2. Records contained in a sequential text file may be of virtually any length.
3. Data is read and retrieved sequentially, i.e., items are read from the beginning of the file to the end.
4. Sequential files are well suited to data that contains records of variable length, is not often updated, and is rarely needed in isolation.

Random Access Files

1. Random access files consist of a number of records of fixed length. Each record in a random access file is essentially a small sequential text file.
2. Data best suited for random access application is of fixed length, frequently updated, and part of a specified number of items.
3. Commands used in the management of random access files are the same as those used for other disk files., with some additional parameters.
4. OPEN, for random access purposes, must be accompanied by a parameter that indicates the length of each individual record in the file. The length designated must be identical each time the file is OPENed.

5. Record length may be calculated by totaling the number of characters in the longest possible data string and whatever separators will be needed.
6. READ and WRITE are used to retrieve and add information to random access files. A record number must be designated for each READ and WRITE requested.
7. CLOSE functions in exactly the same way as it does for other types of disk files. No additional parameters are needed.

Multiple Disk Drive Use

1. Each disk controller card in your Apple is capable of handling more than one disk drive. Switching back and forth between the drives on a single controller card is simply a matter of adding ,D1 or ,D2 to the end of most disk commands.
2. The drive used for booting DOS is the default drive and will be used for all subsequent disk operations until another drive is selected.
3. If you have more than one controller card, both the slot in which the desired card is located and the drive number on that card must be specified. To select another controller card, add S*n* to one of your disk commands.
4. The order of the slot and drive specifiers is not important as long as they are preceded by the name of the disk file desired.
5. ProDOS uses pathnames to specify volumes, directories, and files.

Disk Utilities

1. Several utility programs are included on the DOS 3.3 System Master and ProDOS User's Disk.
2. To copy the entire contents of a disk onto another disk under DOS 3.3, use COPYA. Follow the on-screen prompts for copying operations. Be sure to write-protect (cover up the slot in the upper right corner of the disk) your original System Master to guard against disaster when copying. ProDOS users should use the Volume commands in the Filer utilities.
3. File Developer (FID) provides a set of routines for DOS 3.3 that allow you to catalog disks; reset the slot and drive; LOCK, UNLOCK, VERIFY, DELETE, or copy selected files onto another disk. Space remaining on a disk may also be displayed. FILER and SYSUTILS provide similar functions for ProDOS.

Disk Tricks

1. A program to be RUN automatically when a disk is booted may be loaded and saved as either HELLO or STARTUP, depending on the operating system in use.
2. Volume number, when used in a disk command under DOS 3.3, requires the presence of the correct disk in your drive.
3. A pseudo-chaining capability may be obtained by including a RUN command in a statement within a program.

EXEC

1. EXEC reads a text file containing program lines and treats them as if they were commands being typed in from the keyboard.
2. EXEC may be used to control a sequence of events, including the RUNning of several programs. As each program's operations are concluded, control of the computer returns to the EXEC file.
3. EXEC, unlike RUN or LOAD, does not clear the program currently in memory.
4. You may create a text file consisting of the listing of a program. Once in this state, program lines may be manipulated, inserted into other programs, edited with a text editor, etc.

Commands Discussed

Some of these commands may be familiar, since we have discussed a few of their applications earlier. This chapter has added new dimensions to the commands listed. Since this is a very short list, you should review the text dealing with any commands you still find unclear.

CLOSE
,D1 and ,D2
EXEC
OPEN, L*n*
READ, R*n*
,S*n*
,V*n*
WRITE, R*n*

Chapter 6: Editing and Debugging

Our programming efforts thus far have been primarily devoted to problem solving. We have attempted to identify problems and solve them with simple computer programs that illustrate some of the commands being discussed. This chapter will deal with other sorts of problems -- those created while trying to write programs to solve problems! Call them errors, mistakes, bugs or boo-boos, whatever you decide to call them, they are those troublesome little creatures that seem to elude your best efforts at detecting them.

A few errors, like typing mistakes, can be readily located and corrected. Others are less obvious. I have had some obscure errors in programs that took hours of digging to find! Am I the only programmer who makes errors on a fairly regular basis? Not likely!

The title of this chapter is Editing and Debugging. Applesoft has provisions for both detection and correction of many programming errors. Error messages and editing features help to make the process of modifying programs easier. Debugging refers to the process of locating and eliminating errors (more commonly referred to as bugs) in your programs. Editing, on the other hand, refers to actually changing program lines so that they will operate as intended.

THREE STEPS TO BECOMING A SLIPUP SLEUTH

In order to become a super slipup sleuth, you need to be able to identify the source of an error, determine its nature and formulate a solution. If Watson had had such a simple formula, you might never have heard of Holmes!

Sources of Errors

The source of an error is, of course, a program line. The trick is finding which line. If you recall the discussion of LIST, you know one way of locating errors. To review, here are some of the uses of the LIST command.

LIST displays the program currently in memory, starting with the lowest numbered line and proceeding toward the highest. To examine a portion of a program, type LIST followed by two numbers (separated by commas) representing the lowest and highest line numbers for display. For instance:

```
LIST 0,20
```

displays the program **lines 0-20** on the screen. To examine a single line, type LIST followed by the number of the line. A listing may be terminated by typing Control-C at any time. You may want to review the information about using the LIST command at the end of chapter 3.

Error Messages

Error messages are the primary method of determining the nature of problems noted in program lines. Error messages appear when a statement is executed -- not when you type it in. This is a distinct inconvenience at times, since you must repeatedly attempt to RUN a program in order to locate and correct any errors that it may contain.

Applesoft error messages are identified by a question mark preceding them. The format is ?*XX* ERROR for immediate execution lines and ?*XX* ERROR IN *line number* for program lines. In both cases, *XX* is an error description given by Applesoft. Although the messages are not exactly prize-winning prose, they contain enough information to determine the nature of an error.

Error messages for deferred execution (program) lines indicate the line number in which the error occurred. Now that we have a method of locating the source of a problem, let's take a look at what the error messages tell us. (Some of these messages refer to statements that are beyond the scope of this book.)

BAD SUBSCRIPT

This message is displayed if your program attempts to reference an array element outside the dimensions of the array. For example, if you have dimensioned an array as follows:

```
  DIM A$(5)
```

and then write this line in your program:

```
  PRINT A$(6)
```

a BAD SUBSCRIPT error results. The error also occurs if the wrong number of dimensions is used, for example:

```
   A(1,1,1) = 5
```

when A has been defined by:

```
   DIM A(2,2)
```

If you get a BAD SUBSCRIPT error, check that your arrays are properly dimensioned.

CAN'T CONTINUE

This error occurs if the CONT command is executed when no program exists in memory, or after an error, or after a line has been changed, deleted from, or added to the program. Since CONT is rarely used, you won't see this one often.

DIVISION BY ZERO

Because division by zero is mathematically undefined, Applesoft issues an error message if it is attempted. Check your variables. Usually you have attempted to divide by a variable that is set equal to zero.

FORMULA TOO COMPLEX

The Applesoft IF-THEN statement is not intended to be used with strings. If more than two statements of the form:

```
   IF "STRING" THEN
```

are executed (where "STRING" is any quoted string), the FORMULA TOO COMPLEX error is displayed. It is best to avoid this type of construction altogether.

ILLEGAL DIRECT
Some Applesoft commands cannot be given in immediate mode. They are:

```
DEF FN
GET
INPUT
ONERR GOTO
READ
RESUME
```

If you enter any of these at the keyboard, you'll get an ILLEGAL DIRECT error.

ILLEGAL QUANTITY
This error message indicates that the argument supplied to a statement was out of range. This error can be caused by a negative array subscript (for example, A(-1) = 5), LOG with a negative or zero argument, SQR with a negative argument, A ^ B with A negative and B not an integer, or by using LEFT$, MID$,RIGHT$, WAIT, PEEK, POKE, CALL, TAB, SPC, ON...GOTO, ON...GOSUB, or any of the graphics statements or functions with an improper argument. That certainly covers a lot of ground, doesn't it? If you get this error, the first thing to check is the value of any variables in the statement. Often, you'll find that a variable has a value you didn't expect it to have, caused by a logical error somewhere in your program.

NEXT WITHOUT FOR
This message is displayed when the variable named in a NEXT statement (for example, NEXT X) does not agree with the variable in the corresponding FOR statement. It also occurs when a nameless NEXT is executed when no FOR statement is in effect. Most of the time, it just means you forgot the FOR statement or mistyped the variable in the NEXT statement. Another common cause is accidently branching into the middle of the FOR-NEXT loop.

OUT OF DATA
This is an easy one. It means that a READ statement was executed after all the DATA statements in the program had already be read. Some common causes are omitting one or more DATA statements, or issuing the READ command too many times (usually inside a loop). You may also have omitted a RESTORE statement in a program that reads the data more than once.

OVERFLOW
The size of a number that Applesoft can handle is limited. The largest number is 1E+38, or 1 with 38 zeros after it. If your program manages to exceed this limit, an OVERFLOW error results.

REDIM'D ARRAY
An array may only be dimensioned once in a program. If an attempt is made to define it twice, this error message is displayed. The most common cause for this error is using the array before it is dimensioned. For example, if your program has these lines:

```
A(3) = 3
DIM A(20)
```

you'll get a REDIM'D ARRAY error in the second line. That's because when Applesoft interpreted the first line, it assigned the default dimension of 10 to the previously undefined array A(). Be sure to dimension all your arrays before you use them. In fact, it's a good practice to dimension all arrays at the beginning of your program.

RETURN WITHOUT GOSUB
This is another easy one. It means that a RETURN statement was encountered without a corresponding GOSUB. Usually, it occurs when some code that you intended to be a subroutine is executed inadvertently, perhaps via a GOTO to the wrong line number or because of a missing END statement.

STRING TOO LONG
Applesoft allows a maximum string length of 255 characters. If your program attempts to concatenate two strings to create a string longer than that, the STRING TOO LONG error message is displayed.

SYNTAX ERROR
This is the most common error message. There are a multitude of possible causes, such as a missing parenthesis, illegal character, or incorrect punctuation. Most of the time, it results from a simple typing error. If you are not sure of the correct syntax for a particular statement, look it up and check it against your program line. One misplaced character can cause a SYNTAX ERROR, so be careful when entering programs.

TYPE MISMATCH
Another easy one, it means that the left side of an assignment statement is a numeric variable and the right side is a string, or vice versa. For example:

```
    A = "HELLO"
```

Often, it is caused by leaving off the dollar sign ($) on a string variable.

UNDEF'D FUNCTION
Applesoft allows you to define your own specialized functions, using the DEF FN statement. Naturally, you can't use one of them until you've defined it. If you try anyway, the UNDEF'D FUNCTION results.

UNDEF'D STATEMENT
This message indicates that a nonexistent line number was referenced, either by GOTO, GOSUB, or IF-THEN. You may have deleted a line accidentally or changed a line number without updating all the references to it.

As you can see, Applesoft provides you with quite a range of diagnostic material. However, all the error messages are limited in scope, and Applesoft can only recognize errors related to the functioning of the program. It cannot help you find logical errors in your programs, or mistakes in your approach to a certain task. If it could, you could just instruct it to write the program itself!

LINE EDITING

So far, we've discussed the process of locating and eliminating program errors (debugging). In order to actually do the debugging, you need to revise or replace statements in a program. This is where editing comes into play.

After you've LISTed a line and examined it on the screen, you have two choices -- retype the line or correct the error in the existing line. To retype the line (not a bad idea for short, simple lines) just type the line number followed by the corrected statement and press Return. The line in memory will be replaced with the one just typed. To check the new line, type LIST *line number.*

For more complicated lines, it may be preferable to correct just the error rather than retype the whole line. Applesoft has editing functions to enable you to do this. Each time the Left-Arrow key is pressed it erases the characters at the cursor position. The Right-Arrow key copies the character at the cursor for each move.

To correct an error in a long program line, what you want to be able to do is save most of the line and just change the incorrect part. You can accomplish this by using a combination of the Escape key, the I-J-K-M keys, and the arrow keys. When you press the Escape key, it puts the cursor in "escape mode." While the cursor is in escape mode, you can move it around on the screen without affecting anything else. Try this:

1. Type NEW
2. Type HOME
3. Press the Escape key.
4. Press the M key or Down-Arrow three times; the cursor moves down three lines.
5. Press the K key or Right-Arrow three times; the cursor move to the right three spaces.
6. Press the J key or Left-Arrow three times; the cursor moves to the left three spaces.
7. Press the I key or Up-Arrow three times; the cursor moves up three lines.
8. Press the Space bar to terminate escape mode.

By careful use of the escape mode, you can move the cursor around on the screen and edit existing program lines. Let's try an example. Type in the following short program:

```
10 RAM "THIS LINE HAS AN ERROR"
20 PRONT "SO DOES THIS ONE"
```

Try to RUN the program, and you get a SYNTAX ERROR IN 10 message. Here's how to fix it:

1. Type HOME
2. Type LIST 10
3. Press the Escape key
4. Press the I key or Up-Arrow until the cursor is over the 1 in 10.
5. Press the Space bar to exit escape mode.
6. Press the Right-Arrow four times to copy over the first four characters of the line The cursor is now over the A in RAM. Press the E key to replace the A with an E. Press the Right-Arrow to copy over the rest of the line.
7. Press Return to enter the altered line into memory
8. Type LIST to see the altered program.

See if you can change the word PRONT in line 10 to PRINT using the same method.

Although the escape editing mode provided by Applesoft is useful in fixing some errors, it can be cumbersome to use when you need to delete characters from a line or insert characters into a line. These things can be accomplished by entering escape mode, moving the cursor, exiting escape mode, copying part of the line, entering escape mode again, moving the cursor somewhere else, exiting escape mode, copying another part of the line or typing in some characters, and on and on until you have what you think is the right line in memory and then pressing Return.

Fortunately, there are several excellent Applesoft program editors on the market that make things much easier by giving you word processor-like features for editing Applesoft lines. Applesoft Turbo Editor, a full screen editor that was published in *Nibble* magazine (Vol. 8/No. 6), is included on the *Hands On Applesoft* disk. For a description of the program and a sample editing session, see the appendix. Other popular commercial editors are listed at the end of this chapter.

THIS IS ONLY A TEST

Can you locate the errors in the lines listed below? Type them into your Apple and see if it does as well.

```
10 PRIMT "THIS IS A PROGRAM LINE;"
20 A(12) = 4 * 7
30 A$ = A + B
40 PRINT TAB(10); THIS LINE HAS A PROBLEM
50 A = 3 * (4 + 7)7/3)
```

Of course, those few were easy! Spend some time writing lines that contain errors and see how your Apple reacts. Learn just how each error affects the computer and what is needed to correct it. This time will be well spent when actual program errors are encountered.

Chapter 6 Summary: Editing and Debugging

Up to this point, our discussion of editing and debugging has been limited to language related errors. To be a successful programmer, the requirements of the language must be satisfied before your programs will execute correctly. Once you master that, we will look a little further at finding errors in program logic and design.

Editing and Debugging

1. Editing refers to the actual changing of program lines so they operate as intended.
2. Debugging is the process of locating and eliminating program errors, more commonly known as bugs.

Becoming a Slipup Sleuth

1. Three factors for handling program errors are identifying the source of an error, determining its nature and formulating a solution to the problem.
2. The source of a program error is a program line. The secret, of course, is determining which program line is creating the problem!
3. LIST is used to examine program lines for possible problems. Error messages that refer to a specific line are helpful and indicate which line should be examined using the LIST command.

4. Applesoft lines are evaluated for possible errors when they attempt to execute, not when they are typed. Almost any statement preceded by a line number is readily accepted during the input phase of program construction. Typing RUN will evaluate each line for any possible errors.
5. Error messages describing the nature of an error sometimes indicate the offending program line. Applesoft error messages are preceded by a question mark.
6. Applesoft does not detect errors in program logic or design.

Line Editing

1. LIST is used to display the contents of a program line prior to its being edited.
2. Editing refers to the process of actually changing the contents of a program line.
3. To correct the contents of a short line, merely retype the line and press Return. This replaces the line in memory with that which has just been typed.
4. The Left-Arrow key erases characters from memory with each cursor move.
5. The Right-Arrow key copies the character at the cursor with each cursor move.
6. Pressing Escape followed by the I, J, K or M keys lets you move the cursor without altering the contents of the display or program memory. The Left- and Right-Arrow keys may be used in lieu of the J and K keys, respectively. Apple IIe, IIc and IIGS owners can use the Up- and Down-Arrow keys instead of the I and M keys if desired.
7. To add or delete characters from a program line, the escape sequences must be made to position the cursor properly. Each use of the Left- or Right-Arrow key when not in escape mode alters the contents of both the screen display and memory.

Applesoft Program Editors:

PROGRAM WRITER
THE SOFTWARE TOUCH
9842 Hilbert St. #192
San Diego, CA 92131
(619) 549-3091

GPLE
Beagle Bros
3990 Old Town Ave.
San Diego, CA 92110
1-800-345-1750

GALE
MicroSPARC, Inc.
52 Domino Dr.
Concord, MA 01742
(617) 371-1660

Chapter 7: More Bugs

Ready for another bug hunting expedition? Ideally, we would never have to locate and kill program bugs. In the real world, however, bugs are an integral part of the life of any computer programmer. The disk operating system (DOS 3.3 or ProDOS) is quite communicative and its error messages, although still not exactly welcomed by programmers, do a pretty good job of pointing out the problems.

DISK ERRORS

Let's consider a group of errors that can most kindly be called careless. These are the "operator errors" that can be traced to not satisfying the basic requirements of the operating system. In other words, you are the source! DOS 3.3 or ProDOS error messages tell you the nature of the problem, just as the Applesoft error messages do.

WRITE PROTECTED appears when DOS attempts to store information on a disk that has been locked. A 5 1/4 inch disk is locked by covering the write protect notch with an adhesive strip. A 3 1/2 inch disk is locked by sliding the write protect switch so that the write protect hole is open.

If you attempt to access a disk but forgot to insert one in the drive selected, the I/O ERROR message is displayed. There are other conditions that result in the I/O ERROR message -- the door on the disk drive left open, the disk in the drive not formatted correctly, etc.

As in Applesoft, if you attempt to use a command in immediate execution mode (from the keyboard) that isn't intended for such, NOT DIRECT COMMAND reminds you of the fact.

By their nature, operator errors are relatively easy to correct. The solution to most of the errors noted here is simply to remedy the condition and try again. Although I'd like to be able to tell you it gets better with experience, I still leave the disk drive door open, insert disks upside-down, forget to format disks and as a result, periodically get to see all the error messages we have just discussed.

Play it Again, Sam!

Even though we have heard of a SYNTAX ERROR before, we'll encounter it again in disk operations. Each time you make a grammatical error in a disk command, the ever-popular SYNTAX ERROR appears.

When a disk command specifies a file that is not on a particular disk catalog, a FILE NOT FOUND error is generated. If you fail to type the name of a file exactly as it appears in the catalog, this error message reminds you of that. Of course, you may have accidentally deleted the file or changed disks.

Write operations are not permitted to disk files that are locked. Locked files are identified by an asterisk (*) preceding their name on the disk catalog. If you try to WRITE, RENAME, SAVE, BSAVE or DELETE a locked file, the DOS error trapping mechanism will generate a FILE LOCKED message.

If your disk has no more room but you attempt to store additional data, you will see a DISK FULL error message. This creates an interesting problem in that a portion of a file may be saved before this message is displayed. Cataloging the disk shows the file name,

Chapter 7: More Bugs

Ready for another bug hunting expedition? Ideally, we would never have to locate and kill program bugs. In the real world, however, bugs are an integral part of the life of any computer programmer. The disk operating system (DOS 3.3 or ProDOS) is quite communicative and its error messages, although still not exactly welcomed by programmers, do a pretty good job of pointing out the problems.

DISK ERRORS

Let's consider a group of errors that can most kindly be called careless. These are the "operator errors" that can be traced to not satisfying the basic requirements of the operating system. In other words, you are the source! DOS 3.3 or ProDOS error messages tell you the nature of the problem, just as the Applesoft error messages do.

WRITE PROTECTED appears when DOS attempts to store information on a disk that has been locked. A 5 1/4 inch disk is locked by covering the write protect notch with an adhesive strip. A 3 1/2 inch disk is locked by sliding the write protect switch so that the write protect hole is open.

If you attempt to access a disk but forgot to insert one in the drive selected, the I/O ERROR message is displayed. There are other conditions that result in the I/O ERROR message -- the door on the disk drive left open, the disk in the drive not formatted correctly, etc.

As in Applesoft, if you attempt to use a command in immediate execution mode (from the keyboard) that isn't intended for such, NOT DIRECT COMMAND reminds you of the fact.

By their nature, operator errors are relatively easy to correct. The solution to most of the errors noted here is simply to remedy the condition and try again. Although I'd like to be able to tell you it gets better with experience, I still leave the disk drive door open, insert disks upside-down, forget to format disks and as a result, periodically get to see all the error messages we have just discussed.

Play it Again, Sam!

Even though we have heard of a SYNTAX ERROR before, we'll encounter it again in disk operations. Each time you make a grammatical error in a disk command, the ever-popular SYNTAX ERROR appears.

When a disk command specifies a file that is not on a particular disk catalog, a FILE NOT FOUND error is generated. If you fail to type the name of a file exactly as it appears in the catalog, this error message reminds you of that. Of course, you may have accidentally deleted the file or changed disks.

Write operations are not permitted to disk files that are locked. Locked files are identified by an asterisk (*) preceding their name on the disk catalog. If you try to WRITE, RENAME, SAVE, BSAVE or DELETE a locked file, the DOS error trapping mechanism will generate a FILE LOCKED message.

If your disk has no more room but you attempt to store additional data, you will see a DISK FULL error message. This creates an interesting problem in that a portion of a file may be saved before this message is displayed. Cataloging the disk shows the file name, but not all the information is included in the file. Before attempting further operations, DELETE the incomplete file and save another copy of the original on another disk with more room.

Ridin' the Range

If your Apple could strum a guitar and yodel (we could call it Slim Bitman!), you might hear something like this coming from the computer some night:

Don't tell me a slot where a disk drive is not,
Or a volume that doesn't match up!
Watch your values of B, R, L and A,
And my disks will read and write OK!

Although the lyrics may be corny, the advice is valid. The slot number, for example, must be from 1 to 7 and the disk drive number either 1 or 2. Specifying a slot or drive outside the permissible range results in the RANGE ERROR message. It is possible to get into trouble with a perfectly legal command (at least in terms of range), but we'll cover that state of affairs in the next chapter.

Values of B (byte), R (relative or absolute field for sequential access files, record number for random access files) and L (record length for random access files or number of bytes for binary files) must not exceed 32767. Except for record length (which is 1), the minimum acceptable value is 0. A, the starting address for a binary file, may range from 0 to 65535. (B and A have specialized uses that are beyond the scope of this book.)

Using DOS 3.3, volume numbers from 1 to 254 may be assigned to disks during the initialization process. If the volume specified in a disk command does not match the volume assigned to a specific disk, the VOLUME MISMATCH message appears. Volume numbers outside the 1 to 254 range usually result in the RANGE ERROR message.

Now for the exceptions (you knew this was coming, didn't you?). Specifying a slot number from 8 to 16 in a program line will result in the SYNTAX ERROR message, rather than the RANGE ERROR you would expect. Similarly, any disk command parameter or command quantity that is less than 0 or greater than 65535 triggers the SYNTAX ERROR message. Oh well, nobody said this was going to be easy!

AND MORE BUGS

As you may have guessed, we have not yet exhausted the full list of disk error messages. They have been listed in decreasing order of their appearance in the normal course of programming operations. Bear with it a while longer, however, while we discuss the rest of the little rascals.

END OF DATA occurs when you try to retrieve more information from a file than was stored there initially. This message is displayed after an INPUT or GET statement and can often be the result of not resetting the file pointer.

NO BUFFERS AVAILABLE is displayed when another file buffer required for input or output operations is not available. (A buffer is simply an area of memory that is set aside to hold data as it is transferred to and from the disk.) It normally occurs in a program when you have forgotten to CLOSE a series of files after OPENing them.

FILE TYPE MISMATCH results from trying to use an incorrect combination of a disk command and existing file type. You won't get away with trying to BRUN an Applesoft program file or LOAD a binary file. Check the disk catalog for the letters that identify the file type to avoid this problem altogether.

PROGRAM TOO LARGE is displayed when a disk command attempts to READ a file that is too large for the available memory.

SUCCESSFUL BUG SQUASHING

The errors we've discussed here can be eliminated using much the same process as outlined last time. First, determine the source and nature of the problem. Error messages are designed to give you at least a rudimentary clue as to what is wrong. In the vast majority of cases, examining the line or statement in question should readily show the source of the error.

Some problems, such as DISK FULL, are not directly related to program lines but rather reflect hardware considerations that need to be kept in mind during program operation. The key to eliminating hardware-related problems is anticipating them. Change the disk before it gets full, be sure the drive is set up properly, etc. Although you can correct a problem after an error message is displayed, anticipating conditions that may precipitate errors is far more efficient (and satisfying).

If you have done your homework during the process of learning storage operations, you will have learned to "'dot your i's and cross your t's." Once you know the correct format and syntax requirements of specific commands, spotting errors becomes a relatively simple matter. A key to avoiding a great many errors is careful proofreading.

Another key to avoiding errors is experience. After a while, most of the common problems should become quite familiar. You may find, for instance, that your most troublesome statements are contained in the creation and handling of random access files. If so, it would seem logical to spend some extra time carefully examining and troubleshooting those types of statements. Experience points out both strengths and weaknesses. Learn to eliminate your programming weaknesses, and errors will disappear like magic.

Chapter 7 Summary: More Bugs

The solutions to most of the problems we have discussed so far are quite simple. If they're so simple, you say, why have we spent two chapters discussing bugs? Let's take a minute to talk about it.

We discussed language related errors in chapter 6. Together with the disk errors, they form a group of bugs we could loosely classify as "pilot error." If you have ever heard reports of plane crashes, you know that pilot error is used to cover a multitude of sins not directly attributable to the aircraft. In other words, if the engines ran properly and the radios transmitted, all else must be the fault of the pilot. Computer programming is not really all that different. If the hardware works correctly, any other problems are the fault of the programmer.

Before you get a giant-sized complex, let's make it clear that not all errors are "pilot error" -- only the vast majority. Another whole group, called process errors, are much more subtle and difficult to detect. But enough bad news for now. I am reminded of the cartoon character that was popular a few years ago -- do you remember the fellow with the black cloud over his head in "Lil Abner"? I know just how he felt!

Disk Error Messages:
WRITE PROTECTED
NOT DIRECT COMMAND
SYNTAX ERROR
FILE NOT FOUND
FILE LOCKED
DISK FULL

RANGE ERROR
VOLUME MISMATCH
PATH NOT FOUND
END OF DATA
NO BUFFERS AVAILABLE
FILE TYPE MISMATCH
PROGRAM TOO LARGE

The Keys to Bug Squashing

1. Anticipation -- you must learn to anticipate and make provisions for hardware-related potential problems such as disk full, etc.
2. Careful proofreading -- since most errors are typographical, proofreading helps reduce their number significantly.
3. Experience -- there's no substitute for it.. You must learn which areas are likely to give you problems and monitor them carefully.

Chapter 8: Errors Without Messages

Now you're ready for a graduate course in software sleuthing! As anyone who has written a program will tell you, there are times when things do not happen as intended. No error messages appear in spite of the fact that processes or file manipulations do not do what you thought they would. Under other circumstances, error messages can be made to work for you, instead of against you

Eliminating and controlling errors is the topic of this chapter. We examine error trapping and outline some of the procedures for using it effectively. Next, we discuss error handling. We also look at some of the commands and procedures necessary to check your programs for proper operation. Finally, a few of the more common programming tricks that help you avoid process errors are outlined.

ERROR TRAPPING

The most obvious way to solve error problems is to simply not let them occur. While that may be easier said than done, there are some programming techniques that can help prevent errors from sneaking in.

Let's say, for instance, that you are using an INPUT statement to solicit a number from 1 to 10 from the keyboard. How can you prevent someone from entering 11? Easy -- watch:

Listing 27

```
10 PRINT "ENTER A NUMBER FROM 1 TO 10"
20 INPUT A
30 IF A > 10 THEN PRINT "THAT'S MORE THAN 10":  PRINT "TRY IT
     AGAIN.": GOTO 10
40 PRINT "THANKS, I NEEDED THAT!"
50 END
```

The program doesn't do anything except thank you for a proper value and chide you if the number input is greater than 10. But you just know some wise guy will try typing zero or a negative number. As it stands, the program will say thanks and go on. Let's expand the error trapping a little and try to catch the zeros and negative numbers:

Listing 28

```
10 PRINT "ENTER A NUMBER FROM 1 TO 10"
20 INPUT  A
30 IF A > 10 THEN PRINT "THAT'S MORE THAN 10": PRINT "TRY IT
     AGAIN.":  GOTO 10
40 IF A < 1 THEN PRINT "THAT'S LESS THAN 1": PRINT "TRY IT
     AGAIN.": GOTO 10
50 PRINT "THANKS, I NEEDED THAT!"
60 END
```

Try typing and running these short examples. Do they trap the potential errors? Can you think of any other possibilities for input that the error trapping doesn't cover? Let's try to streamline the error trapping:

Listing 29

```
10 PRINT "ENTER A NUMBER FROM 1 TO 10"
20 INPUT A
30 IF A < 1 OR A > 10 THEN PRINT "THAT'S NOT BETWEEN 1 AND
     10.": PRINT  "TRY IT AGAIN.": GOTO 10
40 PRINT "THANKS, I NEEDED THAT!"
50 END
```

The preceding routine does the same job but eliminates one line by checking for values of A that are either too low or too high in a single line. Note that **line 30** will execute only if the value of A is outside the acceptable limits; otherwise, the program proceeds directly to **line 40**.

What if the same wise guy tried to input a letter or control character instead of a number? Applesoft will come to your rescue here since A must be numeric, so there's no need to trap for those errors. Granted, this is a simple example, but the principle holds true no matter how complicated your specific application.

There are many situations in which you want to check the accuracy of a keyboard response. By properly structuring the error-trapping routine, you can check for almost any possible combination of responses. Simple error trapping in the INPUT or GET routine often saves hours of grief later in the program's operation (or lack of operation, as the case may be).

Let's consider another example. Suppose your program allows for the manipulation of a particular string data field of up to 10 characters. Simple error trapping must be used in the input to be sure the string does not exceed that length.

Listing 30

```
10 PRINT "TYPE YOUR FIRST NAME."
20 PRINT "(10 CHARACTERS OR LESS)"
30 INPUT N$
40 IF LEN(N$) > 10 THEN GOTO 20
50 REM BALANCE OF PROGRAM
```

This routine ensures that any response longer than 10 characters results in another "(10 CHARACTERS OR LESS)" prompt. Notice that we are recycling the original length prompt rather than using a separate message. Setting up original program prompts for dual use is a very efficient use of program lines. You could add a separate error message to direct the user to TRY AGAIN or ABBREVIATE IF NECESSARY.

Error trapping is used in many other situations. Limiting the range of values for calculations and checking for typographical errors in data fields are just two examples. The goal of any error-trapping operation is to catch potential problems before they are allowed to occur. Error trapping might be called the proverbial ounce of prevention, and error handling the pound of cure. The process of error trapping isn't really all that complicated -- you just need to think of all possible erroneous responses and anticipate them in your programming!

While trapping every possible error in a program is a worthy goal, it is rarely achieved. By thinking of error possibilities and testing your programs with inaccurate responses, you will go a long way toward reducing the frustration of having to handle the same errors time after time. Try to bomb your programs with inappropriate responses to input prompts. For each successful bomb of the program, try to add some error trapping that eliminates the problem. Both your programming skills and programs will benefit greatly from this process.

USING ERROR TRAPPING EFFECTIVELY

Here is an outline of a few of the principles of effective error trapping. No doubt you'll be able to add some more that have worked for you.

1. In order to trap a routine for errors, you must be thoroughly familiar with the type of response expected by the program.
2. Error-trapping procedures must account for the full range of possible responses that can reasonably be expected from the user. For example: If the program prompts the user for his/her name, then the input command should not accept a number as a response.
3. Messages generated by error-trapping procedures should give the user a clear picture of why a specific response was incorrect and offer another chance to respond appropriately.
4. The result of error-trapping procedures is the prevention of program or process errors that result in incorrect operation.
5. Error-trapping routines should be invisible to the program user unless an inappropriate condition is encountered.
6. Error-trapping routines should be as simple and efficient as possible. You cannot cover every possible contingency; be content to handle the vast majority!

ERROR HANDLING

If your error-trapping procedures are up to par, there will not be many unintentional errors in your programs. What do I mean by unintentional? Error conditions often indicate the lack of completion of a process or the program's inability to function because of a un-resolvable problem. Normally, error conditions evoke one of the standard error messages. Once the error message is printed, all program operations cease.

If we interrupt the normal process of getting an error message just prior to program termination, some alternate method must be provided to resolve the problem before execution can continue. This alternate method is called error handling, which is exactly what we are doing. Error handling provides some means of resolving simple problems without loss of data.

PEEK AND POKE

To handle errors without interrupting the program, you'll need to know two new Applesoft statements: PEEK and POKE. These statements are used to read and write directly to your Apple's memory. For example, the statement, Z = PEEK(222), reads the 222nd byte of memory and assigns the value at that byte to the variable Z. Since your Apple's memory is numbered from 0 to 65535, any number within that range is valid in a PEEK statement. Similarly, since a byte may only hold a value from 0 to 255, a PEEK statement will only return a value in that range.

The POKE statement is used to actually write to a byte in memory. For example, POKE 768,13 will write the value 13 in the 768th byte of memory. Again, the only valid values for the first POKE parameter are valid memory addresses in the range from 0 to 65535. The value of the second parameter must be a valid byte value in the range from 0 to 255.

Since memory above 53248 is read only memory (ROM), POKEing values into this range will have no effect.

While you may safely PEEK any location in memory, be careful how you use POKE. You are allowed to POKE anywhere in random access memory (RAM), but you may get strange results if you POKE into the middle of the disk operating system code or into the middle of your Applesoft program.

VERIFY

Another statement that is useful in trapping errors without interrupting the program is a disk command: VERIFY. Under ProDOS, this command simply allows you to find out if a certain file is present on disk. Under DOS 3.3, it will actually read the file and check to see that it is internally consistent. In practice, the command is used under both operating systems to find out if a file is present or not. To use the command, use the statement PRINT CHR$(4)"VERIFY *filename"* , where *filename* is the name of the file for which you are checking. If no error occurs, the file exists.

Before we go on, why don't we take a look at a real live error handling routine. Here's the scenario: Your program is designed to create data files for any of several categories of items. If there is already a file on disk by that name, a new one should not be created. We also want to know when the storage disk is full so that the user can be prompted to insert another data disk in the drive. Any other error should terminate program operation. How do you set this up? Try something like this:

Listing 31

```
10  D$ =  CHR$ (4)
20  ONERR  GOTO 100
30  INPUT "NAME OF DATA FILE ";F$
40  PRINT D$;"VERIFY";F$
60  END
100  Y =  PEEK (222): REM  READ ERROR CODE
110  IF Y = 6 THEN  PRINT "THERE IS NO FILE BY THAT NAME ON":
     PRINT "THIS DISK. OK TO CREATE ONE?": GOTO 200
120  IF Y = 9 THEN  PRINT "THIS DISK IS FULL. PLEASE INSERT
     A": PRINT "FRESH DATA DISK IN THE DRIVE AND": PRINT "TRY
     AGAIN.": GOTO 300
130  POKE 216,0: REM RESET THE ERROR FLAG
140  PRINT "PROGRAM TERMINATED BY ERROR."
150  END
200  PRINT : PRINT "(ROUTINE TO GET RESPONSE AND CONTINUE.)":
     END
300  PRINT : PRINT "(ROUTINE TO DELETE FILE, CHANGE DISK, ":
     PRINT "AND TRY AGAIN.)": END
```

This routine looks considerably different from the error-trapping variety outlined earlier. **Lines 10-60** make up a simple program that asks you to type a file name and checks the disk to see whether or not the file is present. If an error is encountered any time after **line 20** is executed, the program will branch to **line 100** instead of printing an error message.

Error handling may be as simple or as complicated as you like. In this routine, we have provided for only two of the more likely error possibilities: FILE NOT FOUND and DISK FULL.

If error code 6 is detected in memory location 222, the file name typed was not found on the disk. In that case, a routine to either change the name or create a file by that name must be provided. If the disk is found to be full during the file creation portion of the program, a routine to delete the partial file from the current disk and create a complete file on another disk must be provided. In this example, any other error code causes PROGRAM TERMINATED BY ERROR to be displayed and the program ends. See **Table 2** for a list of errors and their associated code numbers.

If you consider the wide range of possible error messages discussed in the last couple of chapters, it should be apparent that you don't want to write routines to handle all the possible errors for a program. Most error-handling routines try to cover a few of the more likely possibilities and let the standard error messages appear for the rest.

CAVEATS FOR USING ERROR-HANDLING ROUTINES

Error handling can be a powerful programming tool if properly used, conversely, it can be a pain in the neck if problems are encountered. This process, like so many others in programming, is best learned by experimentation and experience. Here are a few guidelines to keep you out of serious trouble.

1. An ONERR GOTO statement sets the line number to which the program will branch regardless of where an error occurs. On encountering an error, program execution continues as directed by the most recently executed ONERR GOTO statement.
2. An ONERR GOTO command must be executed prior to an error being encountered.
3. A code corresponding to the nature of the error encountered is contained in memory location 222. To see which of the error codes is there, use PEEK (222). (See **Table 2**.)
4. Error-handling routines for problems encountered in FOR--NEXT loops must restart the loop after the error is resolved. If you attempt to re-enter the loop in progress, a NEXT WITHOUT FOR error will halt program execution.
6. With a great deal of caution, you may use RESUME at the end of an error-handling routine to return program execution to the beginning of the statement where an error occurred. Be careful, however, because in certain cases RESUME can place your program in an infinite loop.
7. End your error-handling routines with POKE 216,0 to restore normal operation of the error message mechanism.
8. Error-handling routines must include provisions to both identify and resolve the problems noted. Program execution must be resumed at an appropriate point to avoid further difficulties.

Do you get the impression that you have just been presented with the universal solvent and been told to come up with a container to hold it? Error-handling routines can be a tremendous help in keeping the program user out of trouble, but they present a unique set of problems of their own. The primary requirement for using error handling is to plan for any contingency during program execution and set up routines accordingly.

TABLE 2: Error Code numbers

Disk errors		Applesoft errors	
1	Language Not Available (d)	0	Next Without For
2	Range Error (d)	16	Syntax Error
3	No Device Connected (p)	22	Return without Gosub
4	Write-Protected	42	Out of Data
5	End of Data	53	Illegal Quantity
6	File (d) or Path (p) Not Found	69	Overflow
7	Volume Mismatch (d)	77	Out of Memory
8	I/O Error	90	Undef'd Statement
9	Disk Full	107	Bad Subscript
10	File Locked	120	Redim'd Array
11	Syntax Error (d)/Invalid Option (p)	133	Division by Zero
12	No Buffers Available	163	Type Mismatch
13	File Type Mismatch	176	String Too Long
14	Program Too Large	191	Formula Too Complex
15	Not Direct Command	224	Undef'd Function
17	Directory Full (p)	254	Re-enter
18	File Not Open (p)	255	Break (Control-C pressed)
19	Duplicate File Name (p)		
20	File Busy (p)		
21	File Still Open (p)		

(d) DOS 3.3 only
(p) ProDOS only

WHO'S ON FIRST?

Remember the classic comedy routine that asked that question? Although you may not be old enough to remember Abbott and Costello, the confusion created by perfectly logical answers to questions has a lot of application in computer programming. We usually get correct answers to the questions we ask the computer -- it's just that we really don't want to know what we are asking. Right answer, wrong question!

Programmers are often faced with finding out why certain results are produced by a particular program or routine. There may be no errors in the program lines, yet unsatisfactory answers are being obtained. That's about the time we begin to ask, "Who's on first?"

TRACE AND SPEED

Applesoft incorporates several commands to help you see the operations of a program as they are taking place in order to solve problems. TRACE, which can be used in either immediate or deferred execution mode, causes the number of each line to be displayed on the screen as it is executed. Although this may mess up some of your pretty screen formats, knowing the line number where an error occurs can be a big help. To turn off the TRACE provision, type NOTRACE.

Remember the SPEED command? It is often easier to see what is happening in a program if you slow things down. To use this command, just type SPEED= (while you are in immediate mode) followed by a number from 0 (slowest) to 255 (normal speed). Combining TRACE and SPEED= commands can let you follow the program operations step by step.

MON AND NOMON

Under DOS 3.3, disk operations can be monitored through the use of the MON (short for monitor) and NOMON commands. A little more versatile than TRACE, MON lets you select any combination of three parameters to monitor: C (commands to the disk), I (input from disk files) and O (output to the disk). The format for the commands is MON C, I, O or NOMON C, I, O (if all three parameters are desired). The parameters may be used alone or in conjunction with one another. At least one of the parameters is required or else the command is ignored. If MON were in effect and you were reading five items from DATA FILE, you would see the commands and data on the screen as they were executed.

```
OPEN DATA FILE
READ DATA FILE
7.5
3.25
77.6
57.9
42.1
CLOSE DATA FILE
```

Having both the commands and data displayed is valuable for debugging programs. Used in conjunction with TRACE, the MON command allows you to see exactly what your program is doing during each step. TRACE must be turned off before issuing the MON command in DOS 3.3. In ProDOS, there is no MON command, but TRACE will work during disk activities.

Try these commands with your own programs. What you see may surprise you!

AVOIDING PROCESS ERRORS

Earlier in the chapter, we referred to the prospect of avoiding process errors in the execution of your programs. What are process errors? I like to think of them as little boo-boos rather than major programming problems. They are usually the result of some carelessness on the programmer's part and are almost always difficult to find. Most process errors do not show up with error messages because the statements that contain them are probably correct in terms of syntax and punctuation. These little jewels surface when your program puts out some bizarre results.

The best way to avoid process errors is, of course, not to make any mistakes in writing your programs. Simple? Yes. Possible? Doubtful. I have never met a programmer who has not made at least one error. There are some areas where mistakes are more likely to creep in than others. Here are a few of the process errors I have found in my own programs. You can use this list as a starting point for your error-trapping.

1. Count the left and right parentheses in mathematical expressions to be sure their numbers are equal. It helps to try evaluating some of your expressions by hand to be sure the desired results are obtained.
2. A quirk in Applesoft adds a .000001 to the rounded results of some calculations. There are even more quirks in my techniques for rounding numbers!
3. Arrays can be confusing. It is often easier to confine your arrays to one or two dimensions.
4. Pay attention to variable assignments so that they don't get inadvertently reassigned during program execution. Do you use FOR I = 0 to 10 to set up a FOR--NEXT loop? Even though I try not to use the same variable twice, every once in a while I find the same variable (I) used somewhere else in the same loop (most often in conjunction with array elements).

5. Really bizarre results can be obtained if you intrude on the list of reserved words for your variable names. The best way to avoid this problem? Study the list of reserved words in chapter 2 and avoid them like the plague.
6. Disk commands must be preceded by a carriage return. Ignoring this requirement can give you fits if you try to use disk commands after a GET prompt. The usual response is for the program to print the disk command on the screen. Solution: precede each series of disk commands by PRINT: For example, PRINT: PRINT CHR$(4); "RUN DEMO".
7. Punctuation and syntax are particularly important in disk commands. You can get some very interesting results if you combine several disk commands on a line and don't use the proper combination of quotation marks.

We could go on with this list, but I think you get the point. Every programmer, whether beginner or expert, needs to compile a mental list of areas where problems are likely to surface. Every item on that list should receive extra attention during the debugging of a new program. If you are aware of potential problem areas, the debugging and editing process becomes much simpler.

Chapter 8 Summary: Errors Without Messages

We have discussed some of the possible errors that may be eliminated or rectified without the loss of program function or data. Although avoiding errors altogether is certainly a worthwhile goal, it is rarely achieved in most programming efforts. Knowing how to trap errors, handle the more common ones within a program, and outline areas where you are likely to make errors will save you countless hours of debugging.

Error Trapping
1. Error trapping is most often used to detect inappropriate responses to INPUT or GET statements.
2. To successfully error trap your programs, try to think of all the possible erroneous responses and provide for them in your program.
3. One of the best ways to sharpen your error-trapping skills is to intentionally try to bomb your own programs, and then eliminate the error with some simple error-trapping procedures.

Using Error Trapping Effectively
1. You must be familiar with the type of response expected by the program.
2. Error-trapping procedures should account for the full range of possible responses that can reasonably be expected from the user.
3. Messages generated by error-trapping procedures should be clear and self-explanatory to the user.
4. Error traps should remain invisible during program execution unless an error condition is encountered.
5. Keep your routines for error trapping as simple and efficient as possible.

Error Handling
1. Error handling provides a means of identifying and rectifying errors that occur during program execution.
2. It interrupts the normal process used by Applesoft to handle errors.

3. It should cover the most likely error possibilities and provide for the rest as a group.

Using Error Handling Effectively

1. An ONERR GOTO statement must be executed prior to an error being encountered if error handling, rather than program termination, is to take place.
2. When an error occurs, program execution branches to the line number specified by the most recently executed ONERR GOTO statement.
3. A code corresponding to the nature of the error is contained in decimal memory location 222. This code may be examined through the use of a PEEK(222) statement.
4. Error handling that occurs in the process of a FOR--NEXT loop must restart the loop.
5. The RESUME command returns program execution to the beginning of the statement where an error occurred. It should be used with appropriate caution.
6. To reset the normal error-handling control to Applesoft, use POKE 216,0.
7. Error handling must be able to identify and resolve any problems noted, and then return program execution to the proper point to avoid further errors.

Watching Your Program Operate

1. Line numbers of Applesoft programs may be displayed during program operation through the use of the TRACE command.
2. TRACE is disabled by typing NOTRACE. Either command may be used in deferred or immediate execution modes.
3. The SPEED command may be used to slow down the display processes of the program. This will, in effect, slow down program operation enough for you to see where problems may be occurring.
4. DOS 3.3 disk operations may be monitored with the MON command. The parameters C (disk commands), I (input from the disk) and O (output to the disk) may be specified.
5. The MON capabilities selected may be disabled individually or collectively by using the NOMON command with the same parameters.

Process Errors

1. Process errors are errors that do not surface during the debugging of a program because they have correct syntax and punctuation.
2. Often the first indication of a process error is some bizarre result coming from the program.
3. Programmers need to compile a mental list of areas where they are likely to make process errors and check those areas carefully in their programs.
4. Some process error problem areas include parenthetical grouping for mathematical operations, rounding numbers, arrays, variable assignments, reserved words in variable names, disk commands, punctuation and syntax.

Chapter 9: Playing the Slots

Most Apple users have at least one peripheral device attached to their machines, while some have many different types of devices. Although disk drives are the most common, peripherals run the gamut from printers to graphics tablets. With each new peripheral comes a whole new set of operating instructions and programming requirements.

Although we will not be able to provide details about more than just a few of the common peripheral devices, the principles discussed will apply to the management of many. Some of the more common Applesoft commands and slot management techniques for peripherals are outlined. In addition, we will explore a few ways to use the game paddle ports.

SLOTS

Those who have visited Las Vegas or Atlantic City may recognize this term as the name for machines that typically swallow your money (input) with very little tangible reward (output). Fortunately, the slots on your Apple are a lot more generous; they give as well as take. If you have an Apple II Plus, IIe or IIGS, remove the cover and look toward the back of your computer. The rather long connectors with numerous connections running their length are the slots. (If you want to get technical, those slots provide access to the computer's bus. For those of us who don't know a bus from a car and have no burning desire to understand the details, let's just say that these slots provide one of the means by which the Apple communicates with other devices.)

Note that the slots are numbered 1-7 (from left to right). Your computer may have one or more of the slot connectors filled with printed circuit boards with edge connectors (commonly referred to as "cards"). Your disk drive's printed circuit card contains the electrical components necessary to operate the drive, and a multiple connector cable runs to the drive itself. **Table 3** shows typical slot assignments. Though you can vary these assignments, bear in mind that many programs are designed around them. (The Apple II Plus also has a slot zero. This slot is typically used to hold a card which adds memory to the II Plus.)

Table 3: Typical Slot Assignments

Slot	Card
1	Printer Interface card
2	Serial Device interface card
3	80-column card
4	Mouse Interface card
5	Disk Drive controller card (secondary)
6	Disk Dive controller card (primary)
7	Hard disk drive controller card

So now that you know what a slot and controller card look like, put the cover back on your Apple and let's see how they work.

PR#*n* : AVENUE TO THE SLOTS

Typing PR# followed by a valid slot number directs the output of the computer to whatever device is connected to the slot specified.

The one exception to this rule is slot 0. Regardless of what device is in slot 0 or whether it even exists in your Apple, using the command PR#0 returns output to the screen, *not* to the device in that slot.

Getting the Stream from Your Paddle

A surprisingly underutilized input/output capability has been incorporated in the hand controller port built into your Apple. The controller port on the Apple II/II Plus is located inside the computer on the motherboard, and is called the game I/O socket. On the IIe, IIc and IIGS, the port is a D-shaped connector on the back of the computer. While the location and configuration of the hand controller ports vary from model to model, they function nearly identically. We have all used the paddles or joysticks connected to the this port, but have probably not explored some of its other capabilities.

The only access to game paddles in Applesoft is provided by the PDL command. The command sets a variable equal to the value of any one of four game controls that the Apple can accommodate; for example:

```
X = PDL(0)
```

sets X equal to the current setting of game control 0, a number between 0 and 255.

A word of caution here -- if you are using two or more PDL statements in a program, be sure to separate them by a few program lines to keep the setting of one from interfering with the others. You can cause some strange things to happen by specifying a value less than 0 or more than 8 for the PDL command, so be careful that your range is acceptable.

In order to use some of the other capabilities built into the hand controller ports, PEEKs and POKEs will have to be utilized. While not Applesoft commands, you may find these few PEEKs and POKEs of interest.

The Apple is capable of handling three button switches. Since these are simple pushbuttons, there are only two possible states: pushed or not pushed (closed or open, if you want to get electronic about it). The statement:

```
X = PEEK (-16287)
```

examines the button on game paddle 0. If the button is being pressed, X is greater than 127. Conversely, if the button is not being pushed, X has a value of 0 through 126. Similar examination of button 1 (PEEK (-16286)), button 2 (PEEK (-16285)) is possible. Each of the button readings should be assigned to separate variables if you are using more than one.

MANAGING DISK CONTROLLERS AND DRIVES

Each Apple II disk controller card is capable of handling two 5 1/4 inch disk drives, designated drive 1 or drive 2. If you wanted to, seven controller cards and a total of fourteen drives could be used with your Apple (although by that time, perhaps, a hard disk drive is really what you need). Each controller card is known by its slot number and contains drive 1 and/or drive 2. More likely, you have one or two drives in your system connected to a single controller card located in slot 6. Even if you have a single drive and controller card, you may want to write your programs so that they have the capability of accessing two drives.

If you recall our discussion of disk storage, specifying either slot or drive is necessary only for those systems where more than one of each is used. One controller card and drive are designated as primary and used to boot DOS on startup of the system. Once you have booted DOS on a particular drive, that slot and drive are used for all other disk operations until a change is made via a slot or drive command.

Disk operations sometimes involve more than one slot, drive or disk volume. Provisions to include these parameters are found in Apple's DOS commands. You may want to review the chapter on disk storage before using these parameters. Just remember that slot and drive need to be specified *only* when you wish to change them, not every time you issue a disk command from your program.

PRINTERS -- THE PRIMA DONNAS OF PERIPHERALS?

Though printers sometimes seem to be among the most fickle and difficult to operate peripherals, they are really quite easy to handle if you understand a few of the basics. Let's assume, for the purpose of this discussion, that you have a printer interface card connected to your Apple in slot 1 (and, of course, a printer connected to the interface card). Furthermore, let's assume that you know how to turn the printer on, load paper and generally get things ready for operation.

If you are working on a program and would like to get a hardcopy listing of what you've done, simply turn on your printer and type:

```
PR#1
```

to direct output to the printer, then type LIST and the program listing will be printed out on your printer as well as shown on the screen. To disable the printer, and return the output to your screen, simply type:

```
PR#0
```

Your printer is probably capable of printing lines more than 40 characters wide. Let's try printing a listing of your program that is 80 characters (80 columns) wide. Type:

```
PR#1
Control-I 80N
```

The PR#1 directs output to the printer, which promptly proceeds to print SYNTAX ERROR after you type Control-I 80N (you type Control-I by holding down the Control key while pressing I). Now type LIST and see your program printed in 80-column format.

If your printer is capable of condensed printing, listings up to 255 columns wide may be printed using the Control-I *width* N sequence. Each time you enter this sequence, of course, you will get a SYNTAX ERROR message, since the line is incorrect Applesoft syntax. The message does get through to the interface card, however. To avoid the SYNTAX ERROR message, you may elect to print CHR$(9);"80N" from within a program line. Many interface cards also have other printer functions selected by control character sequences.

To return output to the screen, just type PR#0. You may, of course, use all of these commands from within a program any time hardcopy output is desired. When redirecting output from within a program, you should use the format:

```
10 PRINT CHR$(4); "PR#1"
```

This ensures that the disk operating system will remain connected. This format is required under ProDOS.

Printer Options

Many printers allow you to select numerous options from within a program. The ImageWriter, for example, allows you to select print size, fixed or proportional spacing, enhanced or normal printing, graphics, justification, horizontal and vertical tabs, etc. To set these options, a control character or two is sent to the interface card. Again using the ImageWriter as an example, Escape-N or CHR$(27);"N" selects 10 character per inch printing, CHR$(12) performs a form feed, etc. If you would like to set these options from within programs, check your printer and interface card manuals for specific instructions.

OTHER PERIPHERALS

Some of the more popular peripherals include modems, 80-column display cards, memory expansion boards, language cards, light pens, music synthesizers and graphics tablets. Each of these usually takes up at least one of the available slots or ports on the IIc, so some people find themselves running short of places to put all these peripherals! As a matter of fact, at least one manufacturer of peripherals has offered an extension chassis with several more slots.

Well-designed peripherals do not require totally unique programming to manage them, but follow the conventions established for most other peripheral devices. The single best way to find more specific information about any of your peripherals is in its manual. Just as you did with your Apple when you first got it, you must experiment with input and output devices to discover their subtle capabilities. Don't be afraid to experiment. The chance of hurting anything is very slight, while the opportunity for increasing your computer sophistication is overwhelming.

Chapter 9 Summary: Playing the Slots

Avenue to the Slots

1. Use PR#*n* , where n is a valid slot number, to direct output to a specific slot.
2. Use PR#0 to return output to your screen.

Reading Paddles

1. Use the statement X = PDL(n) to read the value of paddle (n). A number between 1 and 255 will be returned in X.
2. To read the paddle pushbuttons, use the statement X = PEEK(-16287). This will read the pushbutton on paddle 0. To read pushbutton 1 use PEEK (-16286), and to read pushbutton 2, use PEEK (-16285). These statements will return values greater than 127 if the button is being pushed.

Managing Disk Controllers

1. Once you have booted DOS (or ProDOS) from a particular slot and drive, that slot and drive will be used for all future operations until changed.

Printers

1. To obtain a hardcopy of a listing, direct output to the printer with a PR#*n* command, where *n* is the slot that holds your printer interface card. (On the IIc, this is assumed to be slot 1.) Then type the command LIST.
2. To print 80 columns wide, you may have to send the string Control-I80N to your printer. This can be done from within a program line with the statement PRINT CHR$ (9) "80N".

Chapter 10: Getting a Line on Graphics

If you have followed along with the commands and features as they have been presented, your programming efforts should be getting progressively more sophisticated. Once you are able to handle "straight" programming, the fun can begin in earnest.

The Apple is capable of producing some very interesting graphics displays. You have, no doubt, seen examples of graphics in commercial programs. Games, in particular, exploit the best aspects of the Apple's graphics capabilities.

If graphics seems a little intimidating, relax! I won't try to make you into a graphics wizard in one chapter. Rather, we'll concentrate on the basics and then cover a few techniques for those who want to get fancy. After learning some simple concepts, you can experiment until you are comfortable with the commands.

GRAPHICS AND TEXT

There are two primary modes for the Apple screen display. So far, we have used the text mode exclusively. As the name implies, the standard text mode displays only alphanumeric information in a 24-line by 40-character (or 24-line by 80-character) format. As we have learned, there are many ways to vary the display of straight text. For many programs, the text mode is an entirely adequate method of displaying the output.

Some long-forgotten sage once observed that a picture is worth a thousand words. Obviously he didn't try to make a living as a writer! It is true, however, that some circumstances dictate the use of some form of display other than text. That's where graphics come in.

Every Apple II supports at least two graphics modes. Low resolution (Lo-Res) graphics divides the screen into 40 lines of 40 blocks, leaving four lines for text at the bottom of the screen, or 48 lines of 40 blocks with no room for text. High resolution (Hi-Res) graphics divides the screen into 280 by 160 pixels (short for picture elements) with four lines for text at the bottom, or 280 by 192 pixels if the text lines are eliminated. The Apple IIc, IIGS and IIe with 128K of memory are also capable of displaying Double High Resolution graphics, and the IIGS can display Super High Resolution graphics. Since neither of these two graphics modes are directly accessible from Applesoft, their use is beyond the scope of this book.

LOW RESOLUTION GRAPHICS

One way to visualize the low resolution graphics display capability is to think of 1,600 blocks. Each of these blocks is unique, however, in that it has 16 sides, each painted a different color. After you have mentally collected all of these blocks, build a square frame that will hold 40 rows of 40 blocks. Mark each possible block location so that it can be readily located.

Everything assembled and ready to go? Now put a block in each of the possible locations. Remember, any block placed in one of the locations can display any one of its sixteen colored sides. If you like, all 1,600 blocks could be in place and showing the same color.

As a matter of fact, let's assume that those blocks showing their black side represent a blank area. Turn a few blocks to a different color. Very nice looking horizontal or vertical lines can be constructed by making adjacent blocks the same color.

If you really get into playing with blocks, all sorts of shapes could be constructed. Straight horizontal and vertical lines are the easiest and best-looking. Diagonal lines, circular shapes and other deviations are a little rougher since they must be made from relatively large, square blocks.

Apple Blocks

Let's turn on the computer and try an electronic version of our block game. Ready? Type GR and press Return. No parameters, nothing fancy, just GR. Notice that the screen cleared and nothing else happened. You have entered the Lo-Res graphics mode and are ready to try your wings. Now type COLOR= followed by a number from 1 to 15.

Actually, 16 colors are available (see **Table 4**), but 0 is black, which doesn't show up very well on the screen. Isn't this exciting? So far, we have typed two commands and nothing has happened!

Now type PLOT 0,0. See the colored block in the upper-left corner of the screen? That's location 0,0. The bottom right of the screen is 39,39, which you can see for yourself by typing PLOT 39,39. As long as you keep your parameters within the screen, you can keep typing PLOT *X,Y* all day, where *X* corresponds to the horizontal coordinate (0-39) and *Y* refers to the vertical coordinate (0-39).

To change the color of the next block plotted, type COLOR= followed by a different number. After you have experimented with these commands for a while, type COLOR=0, and try a few more PLOT commands. What happens? Black (color 0) is used to erase a previously plotted block color.

TABLE 4: Lo-Res Colors

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

More Blocks

Would you like to plot more than one block at a time? There are two Applesoft commands for doing just that: HLIN and VLIN draw horizontal and vertical lines, respectively. HLIN draws a horizontal line from location A to location B at vertical location C. Locations A, B and C must be in the range of 0 to 39. For example, you could draw a horizontal line across the top of the screen by typing the command:

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

or the bottom of the screen by typing:

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

How about a vertical line? Same idea. VLIN draws a line from vertical location A to vertical location B at horizontal location C as long as your locations are within the proper range.

To draw a line from the top to the bottom of the left side of the screen, the command:

```
VLIN 0,39 AT 0
```

does the job. To complete the screen frame by drawing a line up the right side, the command is:

```
VLIN 0,39 AT 39
```

Using a combination of PLOT, HLIN and VLIN commands enables you to construct almost any type of figure in Lo-Res graphics. Don't forget that each command is executed in the color specified by the most recent COLOR statement, and GR by itself simply specifies the low resolution graphics mode and clears the screen.

And Still More Blocks

Let's construct a program that fills the screen with different colored blocks:

Listing 32

```
10 GR: COLOR = 1
20 FOR X = 0 TO 39
30 FOR Y = 0 TO 39
40 PLOT X,Y
50 C = C + 1
60 COLOR = C
70 IF C > 15 THEN C = 0
80 NEXT Y
90 NEXT X
100 END
```

Try typing and running this program. Did it fill the screen with different colored blocks? Do you think you can make it work a little differently? Take some time to experiment with Lo-Res graphics. Use the demonstration program as a starting point and see what you can come up with.

Reading the Screen

If you would like to check the color status of any block on the low resolution screen, use the SCRN statement as follows:

```
Z = SCRN (X,Y)
PRINT Z
```

These two statements read the color of location *X*,*Y* and print the number on the screen. There are a few quirks to the SCRN command, but you'll never see them as long as you use it only with the Lo-Res mode. Remember to keep *X* in the range 0-39 and keep *Y* in the range 0-39 (or 47 if the full screen is being used).

Full-Screen Graphics

If you would like to take advantage of full-screen graphics, an additional eight graphics lines can be gained at the bottom of the screen by typing:

```
POKE -16302,0
```

after executing the GR command. Then the vertical range of the screen is 0-47 instead of 0-39. The four-line text area is no longer present.

Back to Text

The graphics modes may be exited by typing TEXT. No parameters are necessary. TEXT sets the screen to full-screen text and disables the current graphics mode.

HIGH RESOLUTION GRAPHICS

Let's go back to playing blocks again for a little while. This time, I'd like you to gather 44,800 blocks with different colors on each of 7 sides. (Actually, the blocks may be a little cheaper if you buy 50,000!) Now build a rectangular frame that will hold 280 blocks in each of 160 horizontal rows. Still with me? OK, put a block in each possible space and turn the black side out.

You can immediately see some differences between this and the last frame we built. By turning different colored sides out, much more complex and detailed figures may be constructed. Diagonal lines look much more like straight lines than they did in the Lo-Res frame.

We have created an approximation of one of the Hi-Res graphics screens. (Remember, there are two.) Now, let's turn to the real thing and see how to use the Hi-Res mode.

Setting Hi-Res

The command to enter high resolution graphics mode and clear the contents of Hi-Res page 1 is HGR. Hi-Res page 2 is cleared and displayed by typing HGR2. Let's stick with page 1 for now. Like GR, HGR sets the screen into combined graphics and text mode so that there are four lines of text available at the bottom of the screen. If those lines are not needed, type POKE -16302,0 to convert the display to full screen (280 x 192) graphics.

If you have typed HGR, your screen should be blank. Now let's select a color. Hi-Res mode offers a much more limited variety of colors than does Lo-Res, but has substantially increased screen resolution. As you can see in **Table 5,** the Hi-Res colors are black, green, violet, white, orange and blue.

The numbering system for these is a little confusing, since there are two blacks and two whites. The color displayed depends on the location being plotted and the color of the adjacent pixels. If you don't yet understand what we're talking about, don't despair! Remember, a picture is worth a thousand words and we'll be doing some pictures soon. The statement HCOLOR is the command used to specify a color between 0 and 7 (see **Table 5**). For the moment, let's just say HCOLOR=1 (green).

TABLE 5: Hi-Res Colors

0	black-1	4	black-2
1	green	5	orange
2	violet	6	blue
3	white-1	7	white-2

HPLOT

A single statement replaces the PLOT, VLIN and HLIN statements used in Lo-Res graphics. This all encompassing statement is HPLOT and it works like this: HPLOT *X,Y* plots a single high resolution dot at the location specified by the X and Y coordinates. The color of the dot is determined by the most recently executed HCOLOR= statement.

HPLOT TO *X,Y* causes a line to be drawn between the last point plotted and the new location of *X,Y* specified in the statement. The color of the line will be the same as the last dot plotted, regardless of the current HCOLOR status.

```
HPLOT X1,Y1 TO X2,Y2 TO Xn,Yn
```

HPLOT can be extended almost indefinitely as long as you stay within the confines of the screen and do not exceed the 239 character limit for a single line. In each instance, a line is plotted from the previous X,Y position to the current one. Some rather interesting displays can be created by using just a single (although somewhat lengthy) statement. For instance, to plot a rectangular border around the screen, all you need is the statement:

```
HPLOT 0,0 TO 279,0 TO 279,159 TO 0,159 TO 0,0
```

You know the best way to learn about HPLOT? Sit down at your computer and type HGR, HCOLOR = 1 and HPLOT to your heart's content. Change colors and plot some more. Be creative!

SHAPES AND SUCH

In addition to plotting points and drawing lines, you can also draw complete shapes using a set of special Applesoft statements. With Applesoft shapes you can create pictures such as Hi-Res text characters, aliens for a space game, the signs of the Zodiac, or whatever your graphics program requires.

It might help to think of Applesoft shapes as dot-to-dot drawings in which the entire drawing is defined by a sequence of numbers. Imagine a series of numbered dot-to-dot pictures in collection and you've got the idea of a shape table. The encoding of a shape table can be done by hand, but it is so tedious that it is best left to the Apple. Most programmers use a commercial software package called a shape table editor to build them. One such editor is the Designer and Illustrator package available from MicroSPARC.

Using the Shape Table

To put your shape on the screen, you must first make sure that your shape table is in memory and that Applesoft knows where it is. Since a shape table consists of machine readable numbers, it can either be BLOADed from a disk file or POKEd directly into memory. To let Applesoft know its location, its starting address is POKEd into locations 232 and 233. Then you simply issue the HGR command, set the color with the HCOLOR statement and use the DRAW statement to put your shape on the screen.

DRAW

DRAW is the command used to display shapes from your table on the Hi-Res screen. The format is:

```
DRAW shape AT X,Y
```

You may also use an abbreviated version of the command:

`DRAW` *`shape`*

In this instance, the shape specified will be drawn at the last point plotted by an HPLOT.

The shape number should be within the range of shapes specified in your table. X and Y, as in the other commands, are the screen coordinates where the shape is to be drawn and must be kept within the permissible ranges. Before attempting to execute a DRAW command, color, scale, and rotation must be specified. Scale and rotation are discussed below.

The following short program POKEs a shape table containing a single shape (an arrow) into memory, and DRAWs the shape on the screen.

Listing 33

```
 10  GOSUB 140: REM  POKE SHAPE TABLE
 20  HOME
 30  HGR
 40  HCOLOR= 3
 50  SCALE= 1
 60  ROT= 0
 70  X = 10
 80  Y = 50
 90  DRAW 1 AT X,Y
 100  VTAB 22: PRINT "PRESS RETURN TO QUIT"
 110  GET K$
 120  TEXT : HOME : END
 130  REM  POKE IN SHAPE TABLE
 140  FOR I = 0 TO 34: READ ML: POKE 768 + I,ML: NEXT I
 150  DATA
     1,0,4,0,36,36,36,60,63,63,8,12,12,12,12,12,12,12,21,21,2
     1,21,21,21,21,62
 160  DATA  63,55,54,54,54,63,63,39,0
 170  POKE 232,0: POKE 233,3: REM  POKE SHAPE TABLE POINTERS
 180  RETURN
```

Sharp-eyed programmers will notice that this program contains two new statements, SCALE and ROT.

SCALE= sets the size scale for the shape to be drawn and must be specified before any attempt is made to draw the shape. Values may range from 1 to 255, with 1 being the original size saved in the shape table. Each shape is expanded by the number specified following SCALE=.

ROT= is the command that sets the angular rotation of a shape to be drawn on the screen; this must also be specified before any attempt is made to draw a shape. Values between 0 and 255 are acceptable, although the entire rotation range can be covered by values of 1 through 63 (64 repeats the rotation process exactly the same as 0).

If you have specified SCALE=1, four rotation values are recognized. ROT=16, 32 or 48 will rotate the shape 90, 180 or 270 degrees clockwise, respectively. You remember, of course, that 0 and 64 will cause the shape to be drawn as it was defined.

If you specify SCALE=2, eight rotation values are recognized, filling in the 45 degree rotation points. Similarly, larger scale values will cause more rotation values (always between 0 and 64, however) to be recognized. If, by the way, you specify a value for

ROT= that is not recognized, the shape will be drawn with the next smaller recognized rotation. Try experimenting with rotation and scale by changing the values in **lines 50 and 60.**

If XDRAW is specified instead of DRAW, the color used to draw the shape will be the complement of that already existing at the screen location. In other words, if that area of the screen is black, the shape will be drawn in white. Similarly, blue and green are complements, as are orange and violet. Repeatedly executing XDRAW statements is a good way to draw a shape and then erase it. The following program animates an arrow shape across the screen using XDRAW.

Listing 34

```
 10  GOSUB 200: REM  POKE SHAPE TABLE
 20  HOME
 30  HGR
 40  HCOLOR= 3
 50  SCALE= 1
 60  ROT= 16: REM   POINT RIGHT
 70  X = 10
 80  Y = 50
 90  XDRAW 1 AT X,Y
 100  FOR I = 1 TO 24
 110  FOR T = 1 TO 150: NEXT T: REM  SHORT DELAY
 120  XDRAW 1 AT X,Y
 130  X = X + 10
 140  XDRAW 1 AT X,Y
 150  NEXT I
 160  VTAB 22: PRINT "PRESS RETURN TO QUIT"
 170  GET K$
 180  TEXT : HOME : END
 190  REM  POKE IN SHAPE TABLE
 200  FOR I = 0 TO 34: READ ML: POKE 768 + I,ML: NEXT I
 210  DATA
     1,0,4,0,36,36,36,60,63,63,8,12,12,12,12,12,12,12,21,21,2
     1,21,21,21,21,62
 220  DATA  63,55,54,54,54,63,63,39,0
 230  POKE 232,0: POKE 233,3: REM  POKE SHAPE TABLE POINTERS
 240  RETURN
```

You have delved into the topic of graphics, albeit in a rather cursory fashion. If you take the time to experiment with both graphic modes, some interesting additions to your programming repertoire may be forthcoming. However, fully understanding graphics involves considerably more explanation and experience. There are several excellent books devoted solely to graphics -- check your local computer store or technical bookstore.

Chapter 10 Summary: Getting a Line on Graphics

Modes of Display

1. The standard text mode displays alphanumeric information in a 24-line by 40-character format.
2. Low resolution graphics mode divides the screen into 40 lines of 40 characters, leaving room for four lines of text at the bottom. If the text area is eliminated, there is room for 48 lines of 40 characters each.
3. High resolution graphics mode divides the screen into 280 by 160 possible positions with room for four lines of text at the bottom. Full screen graphics, without the text lines, offers 280 by 192 positions.

Low Resolution Graphics

1. The standard low resolution screen offers 1,600 positions for plotting (40 rows of 40 characters).
2. Eliminating the text area (four lines at the bottom of the screen) increases the number of rows to 48.
3. Each position on the low resolution screen can be plotted in one of sixteen colors.
4. Lo-Res mode is entered by typing GR. No other parameters are necessary.
5. Color selection is made using the COLOR= statement. The color used for plotting will be that of the most recently executed COLOR= statement.
6. PLOT *X*,*Y* is the statement used to place a colored block in any desired position on the screen. Both *X* and *Y* must be within the permissible range 0-39.
7. Black (COLOR=0) is used to erase a previously plotted screen location.
8. HLIN draws a horizontal line from location A to location B at vertical location C.
9. VLIN will draw a line from vertical location A to vertical location B at horizontal location C.
10. SCRN (*X*,*Y*) returns the color value of the specified point on the screen.
11. Lo-Res graphics mode is exited by typing TEXT. No other parameters are necessary.

High Resolution Graphics

1. Hi-Res graphics offer a maximum of 44,800 screen positions (280 by 160) for each of the memory pages (or 53,760 if the full 280 by 192 screen is used).
2. HGR is the statement used to enter high resolution graphics mode and display the contents of memory page 1. HGR2 displays the contents of memory page 2.
3. POKE-16302,0 converts the Hi-Res screen to full graphics, eliminating the four text lines at the bottom. The number of screen positions is then 280 by 192.
4. Only seven colors are available in Hi-Res graphics mode. HCOLOR= is the command used to select the color to be used by subsequent commands.
5. HPLOT *X*,*Y* plots a single dot at the screen location specified by *X* and *Y*. The color of the dot is determined by the most recently executed HCOLOR= statement.
6. HPLOT TO *X*,*Y* causes a line to be drawn between the last point plotted and the location specified by *X* and *Y*. The color used will be the same as the last dot plotted, regardless of the setting of HCOLOR.
7. HPLOT *X*,*Y* TO *X1*,*Y1* TO *Xn*,*Yn* . The HPLOT command can be expanded almost indefinitely as long as *X* and *Y* are valid screen locations and the total length of the command does not exceed 239 characters.

Shape Tables

1. Shape tables are used in high resolution graphics to specify the characters for DRAW and XDRAW statements. Each shape must be defined in a shape table before it can be used in one of these commands.
2. The beginning address of the shape table must be specified. You can use POKE commands from Applesoft to handle this chore.
3. SCALE= sets the size scale for the shape to be drawn on the screen. SCALE=1 is original size, SCALE=255 is the largest size permitted.
4. ROT= determines the angular rotation of the shape to be drawn. Values up to 255 are permitted, although the full range of rotation is 0-63. ROT=64 is exactly the same as ROT=0.
5. The value of SCALE determines how many rotation points will be recognized. Four points are recognized at SCALE=1, eight at SCALE=2, etc.
6. DRAW displays shapes from the table on the high resolution graphics screen. DRAW *shapenumber* AT *X,Y* displays the specified shape at screen locations *X* and *Y*. COLOR=, SCALE=, and ROT= must be executed before DRAW is attempted.
7. DRAW *shape* displays the shape specified at the location last plotted by the HPLOT, DRAW or XDRAW commands.
8. XDRAW works exactly the same as DRAW except the color used will be the complement of that already in that location on the screen. XDRAW is commonly used to erase a shape that has already been drawn.

Appendix

INTRODUCTION

The programs described in this Appendix, Applesoft Tutor and Applesoft Turbo Editor, were originally published in *Nibble* magazine. Because they are invaluable tools for the beginning Applesoft programmer, they have been included on the Hands On Applesoft program disk. The *Hands On Applesoft* disk also includes all of the numbered listings in this book. It is available for $14.95 from MicroSPARC, Inc., 52 Domino Drive, Concord, MA 01742, (617) 371-1660. See the ordering card in the beginning of this book for details.

Applesoft Tutor

by Jan Harrington, Ph.D.

Deciphering Applesoft's fifteen cryptic error messages is one of the most difficult tasks facing the novice programmer. This system provides explanatory error messages for a hundred different errors that will help pinpoint the actual problem. To use it, you will need an Apple with at least 64K of memory and DOS 3.3.

Imagine this: you have just keyed your very first Applesoft program into your Apple. Nervously, you type RUN and press the Return key. The result of your efforts is a loud beep and the message ?SYNTAX ERROR IN 20. You LIST line 20 and, though you look at it carefully, you can't figure out what in the world is wrong; the SYNTAX ERROR message told you nothing except that you have a problem.

Applesoft's few error messages are notoriously cryptic (there are only 15 error messages to describe all the programming errors the interpreter can detect). This is not a major problem for experienced programmers who have already learned how to find and correct their errors. Novice programmers, on the other hand, are confronted with a Catch-22. It takes practice to develop debugging strategies, but how can you learn these tricks when you keep getting messages like SYNTAX ERROR?

Applesoft Tutor can help you learn to program in Applesoft by providing you with more specific error messages. This is a teaching system — it does not support all of Applesoft. By the time you outgrow it, though, you'll be well on your way to becoming an expert program debugger.

USING THE PROGRAM

Applesoft Tutor is very easy to use. The program loaded by a short Applesoft program called TUTOR.BOOT. To install it, simply type the command RUN TUTOR.BOOT. Then you can begin to program. After you have entered your program, RUN it as usual. If an error is encountered, the Applesoft Tutor system will replace the normal, cryptic error message with a more detailed message that tries to pinpoint the problem.

Applesoft Tutor imposes some limitations on the programs you use with it. Most importantly, it severely limits the size of the programs you can write. Why? The program and its table of error messages are very large, and they consume a great deal of main memory. Therefore, you have only 5K for your program.

This space problem sheds light on the reasons that the Applesoft interpreter doesn't provide better error trapping. The Apple is a multi-purpose machine. If memory were set

This space problem sheds light on the reasons that the Applesoft interpreter doesn't provide better error trapping. The Apple is a multi-purpose machine. If memory were set aside for very specific error messages, you would lose not just space for programs, but for Hi-Res graphics as well.

Limited memory is also the reason that Applesoft Tutor does not provide enhanced messages for all of Applesoft. Nevertheless, if you cause an error using an unsupported keyword, the system won't crash; instead, it displays the regular Applesoft error message.

The keywords for which you will receive enhanced error messages appear in **Table 1**. The system does not support multiple statements with a single line number (such lines are much harder to debug when you're learning!). If you do use a line containing more than one statement and that line contains an error, you will not receive the correct error message.

TABLE 1: Keywords for Which Applesoft Tutor Provides Enhanced Error Messages

FOR	LEN	SPC
GOTO	LET	STEP
HTAB	MID$	TAB
IF-GOTO	NEXT	TO
IF-THEN	ON-GOTO	VTAB
INPUT	PRINT	
LEFT$	RIGHT$	

Notes:

1. The program provides enhanced messages for assignment statements that begin with LET as well as those that do not (e.g., LET X = 6 is equivalent to X = 6 as far as both Applesoft and Applesoft Tutor are concerned).
2. The program provides enhanced messages for functions (LEFT$, MID$, RIGHT$, LEN) when they are used in PRINT statements. It will also handle errors in arithmetic expressions contained in PRINT statements.
3. Applesoft Tutor analyzes all types of errors that the above keywords could cause, not just SYNTAX ERRORs (e.g., ILLEGAL QUANTITY ERRORs caused by out-of-range HTAB and VTAB arguments).

Applesoft Tutor does not support graphics or game paddle commands. If you attempt to use Hi-Res graphics commands, the program will crash, since it overlays the parts of main memory that are used to display high resolution graphics. Most DOS commands are unaffected by the presence of Applesoft Tutor. You can SAVE, LOAD, RUN and DELETE files as you normally would.

Because the system rearranges the Apple's memory, you should turn off the Apple before running any program that is not designed to be used with Applesoft Tutor. In other words, do a cold start rather than typing RUN. The cold start will restore the Apple's original memory configuration.

One caution is in order. Don't press Reset while using Applesoft Tutor. If you do, you will revert to normal Applesoft error messages. To reinitialize the Applesoft Tutor system after an accidental reset, LOAD TUTOR.BOOT and RUN 50.

Applesoft Turbo Editor

by Claude Leyo

Although every Apple II has the BASIC language in ROM to make programming easy, the available editing features are rudimentary at best and frustrating at worst. Applesoft Turbo Editor enhances the utility of BASIC by adding several powerful function keys that make editing programs a breeze. Turbo Editor is a full-screen editor that lets you make changes to your Applesoft program anywhere on the screen. Simply move the cursor to the instruction you want to change, and make the correction directly where the instruction is displayed.

Think of Applesoft Turbo Editor as a word processor for your BASIC programs. All of the features you need, such as insert, delete, and erase to end of line, are available through function keys. You don't have to retype the line or move the cursor over all the characters in a line to copy them into memory; just change the part that needs fixing and press the Return key.

The entire line is recorded in the program as it appears on the screen. You can also scroll forward and backward through your program. During a list, this keeps the pertinent section of code on the display. There are no complicated instructions to learn — the most useful features are available at your fingertips via a reduced set of function keys, and the way you use disk commands is unchanged.

APPLESOFT TURBO EDITOR TUTORIAL

To install Applesoft Turbo Editor , type RUN INSTALL.TURBO. This program determines which operating system is being used (DOS 3.3 or ProDOS), then loads and activates the editor.

You will know that TURBO.EDITOR has been correctly installed when you see the initial TURBO.EDITOR command display and the Applesoft prompt at the bottom of the screen. The editor is now ready to assist you with writing and modifying BASIC programs. If for some reason (use of the PR#3 command, for example) the editor is deactivated, press Control-Reset to reactivate it.

Now that TURBO.EDITOR is active, you can either load or create your own program. Let's walk through its features using a short test program. Type NEW and enter the following short listing:

```
10 HOME
20 PRINT "HOW MANY TIMES ";
30 INPUT A$
40 IF VAL(A$) = 0 THEN 20
50 FOR X = 1 TO VAL(A$)
60 PRINT "NIBBLE"
70 NEXT X
80 END
```

When done, type SAVE TURBO.TEST to make sure you have a backup on disk.

To see if you entered TURBO.TEST correctly, type LIST and the program will appear on your monitor. If you notice any errors, don't worry; you can go back and fix them easily after you're more familiar with TURBO.EDITOR. Normally, you must type PR#1 and then LIST to get a printout of your program. TURBO.EDITOR simplifies this process. Simply type LISTP and the program listing is printed on the device connected to slot 1.

Now that you have a copy of the program for reference, it's easy to see that the space after the word TIMES in **line 20** is not needed. To remove it, first position the cursor between the word TIMES and the ending quotation mark using the arrow keys (II and II Plus owners must use Control-K instead of the Up-Arrow and Control-J instead of the Down-Arrow). When the cursor is in position, press Control-D to delete the space. Note that everything to the right of the cursor has now shifted one character position to the left. When you press Return, the current line is entered into your program just as displayed on the screen. It doesn't matter where the cursor is when you press Return.

If you are unfamiliar with Applesoft, the lack of the command GOTO in **line 40** of TURBO.TEST might be confusing. To insert it, simply position the cursor on top of the "2" in **line 40** and press Control-I to enter the insert mode. As you type GOTO, notice that everything to the right of the cursor shifts to the right to make room. Press Return to enter the new version of the line into memory.

Now position the cursor on the beginning of the word Nibble in **line 10** and press Control-E. This command erases the remainder of the line and lets you begin typing whatever you like. Now type in your name; but before you press Return, press Control-O and then the letter G. Control-O allows you to insert a control character, such as the bell character. When you typed Control-G, did it appear on the screen in inverse video? It should have. TURBO.EDITOR automatically displays all control characters in inverse video so they stand out in your programs.

When writing long programs, it's often difficult to remember if you've already used a certain variable. With TURBO.EDITOR it is easy to check. The search function, invoked with Control-S, locates the first occurrence of any character string. Let's say we want to find the variable A$ in our test program. Press Control-S and the SEARCH: prompt will appear at the bottom of the screen, waiting for you to enter the search string. Type A$ and then press Return; TURBO.EDITOR will respond by displaying **line 30**, with the cursor on the first letter in the search string. If you wish, you can edit the line using other TURBO.EDITOR commands. To find the next occurrence of the same search string, press Control-N. Use Control-S again to enter a new search string. For a list of all Applesoft Turbo Editor commands and their functions, see **Table 2.**

TABLE 2: Applesoft Turbo Editor Commands

Command	**Function**
Up-Arrow or Control-K	Moves the edit cursor up.
Down-Arrow or Control-J	Moves the edit cursor down.
Left-Arrow and Right-Arrow	Move the edit cursor left and right, respectively.
Control-D (Delete on IIGS, IIe and IIc)	Deletes the character under the cursor. Moves all characters that are to the right of the cursor one position to the left.

Control-I (Tab on IIGS, IIe and IIc)	Enters insert mode, allowing you to input characters at the cursor position. Everything to the right of the cursor is shifted automatically to make room for the inserted material.
Control-E	Erases everything from the cursor position to the end of the program line.
Control-O *X*	Inserts any control character *X*, which is subsequently displayed in inverse video. For example, to enter Control-D character, press Control-O, then press Control-D.
Control-S	Searches for a character string. The prompt SEARCH: appears at the bottom of the screen. Enter the string you choose to locate and press Return. The editor will display the first line in the program that contains the specified search string, with the cursor positioned at the beginning of the string.
Control-N	Finds and displays the next occurrence of the previously specified search string.
Control-B	Lists forward. The list starts from the last line displayed on the screen, or from the beginning of the program if no lines are currently displayed. Stop the list by pressing any key. To start the list at a given line, enter the line number and then press Control-B. Control-B functions like the Applesoft LIST command, but the format differs slightly as follows. •The full width of the screen is used. •Control characters are displayed in inverse video. •Blanks in arithmetic expressions are not displayed.
Control-T	Lists backward. The list starts from the first line displayed at the top of the screen, or from the end of the program if no lines are currently displayed.

Control-P	Activates 80-column mode. (Note: the PR#3 command deactivates the editor; use Control-P instead.)
Control-Q	Disconnects the editor. It is unnecessary to disconnect the editor when running most programs that use the standard DOS, ProDOS and Applesoft commands. You may, however, run into trouble using the editor in conjunction with certain programs. In this case, use Control-Q to disconnect the editor. To reconnect the editor, type CALL 37888 under DOS 3.3, or CALL 37120 if running ProDOS.
LISTP	Prints a complete list of the program on the device connected to slot 1. LISTP has the same syntax as the Applesoft LIST command and may be followed by one or two optional parameters (the first and last lines to list). (Note: the PR#1 command deactivates the editor, so use LISTP.)

Index

Hands On Applesoft

by Leslie R. Schmeltz

Learn Applesoft at your own keyboard! *Hands On Applesoft* shows you how. Here is a practical, tutorial approach for absolute beginners. It teaches the art of creating Applesoft programs, from start to finish. You'll learn the functions of BASIC's simple but powerful commands. Dozens of examples show you exactly how to write, modify and debug your own programs.

Disk operations are an integral but often-misunderstood part of most Applesoft programs. *Hands On Applesoft* includes detailed, easy to understand instructions for mastering disk storage techniques.

Bugs! Despite your best efforts, you'll undoubtedly run into program errors. The chapters on debugging describe how to untangle the snarls, exterminate the bugs and turn your programs into polished gems.

Whether your goal is to tinker, to gain more control over your Apple, or to become a professional programmer, *Hands On Applesoft* is the place to start.

Optional Programmer's Diskette

All of the example programs from *Hands On Applesoft*, plus two high-powered programming utilities, are available on optional diskette. **Turbo Editor** lets you scroll through your program forward or backward, insert characters and perform searches. **Applesoft Tutor** is a debugging aid that gives you detailed error messages and suggests remedies. See the bound-in card for ordering information.

0 74470 19728 7 01

$12.95