Apple ///

Apple Writer ///
Word Processing Language

Customer Satisfaction

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Apple dealer. Return any item to be replaced with proof of purchase to Apple or an authorized Apple dealer.

Limitation on Warranties and Liability

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is," and you the purchaser are assuming the entire risk as to their quality and performance. In no event will Apple or its software suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Copyright

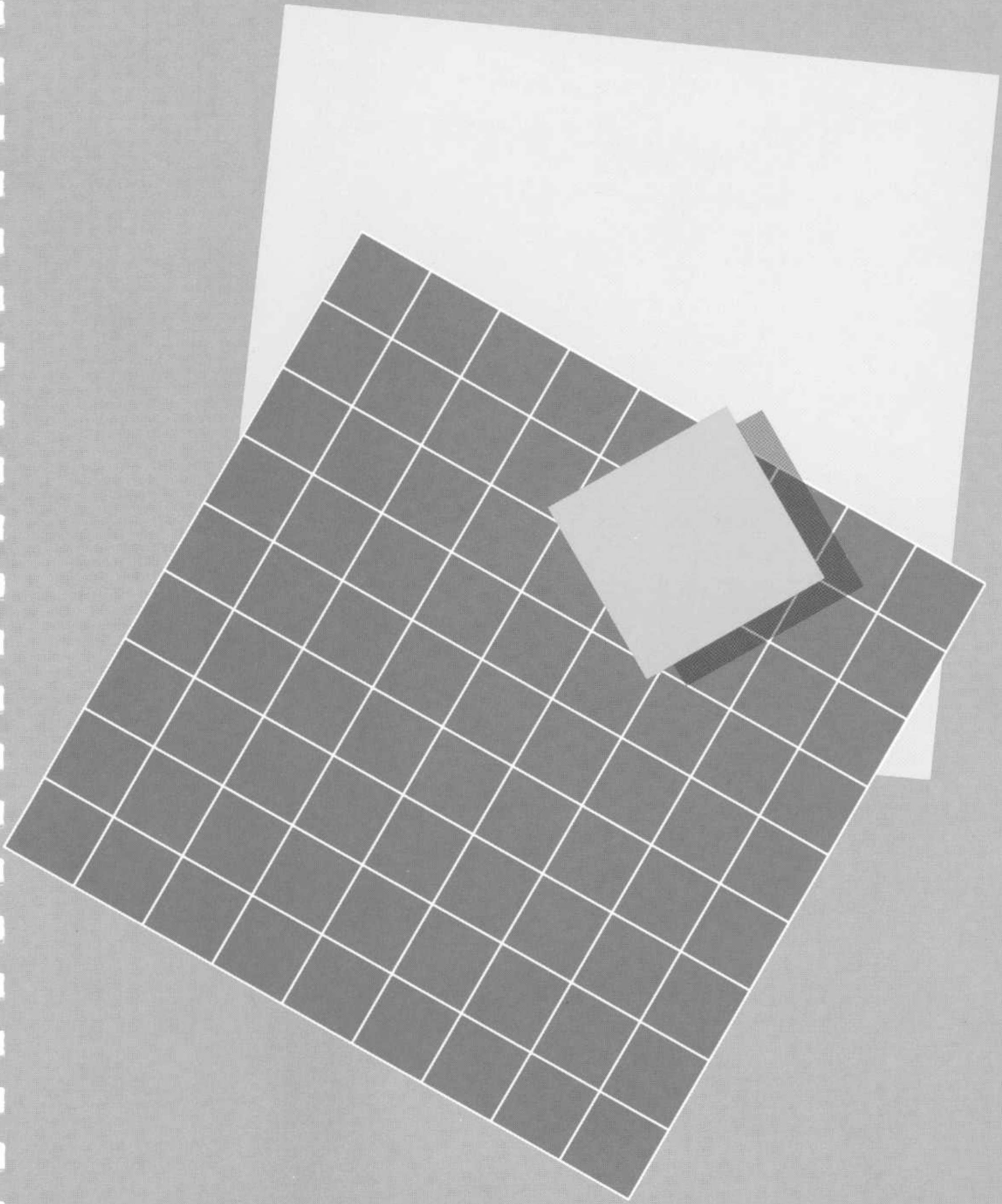
Product Revisions

Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should periodically check with your authorized Apple Dealer.

© Apple Computer, Inc. 1982
20525 Mariani Avenue
Cupertino, California 95014

Apple ///

Apple Writer ///
Word Processing
Language



Contents

6

Using Numeric Variables 65

7

Advanced Techniques 79

C Summary of WPL Commands by Function 121

D Error Messages and Debugging Hints 129

List of Figures and Tables

Preface

IMPORTANT INFORMATION: If haven't yet mastered the basic functions of Apple Writer, you might find yourself confused by what you read in this manual. That's only natural: reading about WPL without having learned Apple Writer is like trying to bake brownies when you've never used a stove before! Save yourself from the frustration of burned brownies: don't pick up this book with any serious intent until you feel comfortable with Apple Writer.

This manual gives you the power to automate word processing by writing programs in a Word Processing Language (WPL). WPL is a special feature of Apple Writer III. Before reading further, please be sure you are familiar with the contents of the *Apple Writer III* manual (which will be referred to from now on as the "Apple Writer manual"). You might want to read Appendix B in the Apple Writer manual right now. It takes you through an example of how to use a WPL program; in this manual you will learn how to write your own programs.

How to Use This Manual

If you already know how to program in another computer language (such as BASIC, FORTRAN, Pascal, or assembly language), you will be able to learn WPL very rapidly. In fact, you may not need to read all of this manual. To determine whether you need to read the manual, turn to the appendixes. There you will find the rules for writing WPL statements and the rules for using each WPL command. The appendixes may provide all you need to know in order to begin using WPL. On the other hand, if you'd like more detailed information, read or skim the manual to see how WPL differs from the computer language you already know.

If you are new to computer programming, you will discover that WPL is a very easy language to learn. Begin with Chapter 1 and read through the whole manual. Right away you will learn programming terminology and many fundamental programming concepts. You will be able to automate simple word processing functions by the time you finish Chapter 2. As you read further in the manual, you will become familiar with more WPL commands until you have mastered the entire language.

The best way to master automated word processing is to work with the sample programs in this manual. As you read each chapter, study the sample programs line by line so that you understand what each statement does. Type the programs exactly as they appear in the manual—you'll learn a lot about WPL statements that way—and save them. Then run the sample programs and experiment with them. The more you do, the sooner you'll be speaking WPL like a native.

Here is an annotated list of the chapters and appendixes in this manual:

Chapter 1 **Introduction to WPL**: Provides a conceptual overview of WPL.

Chapter 2 **How to Write in WPL**: Explains the rules for writing WPL statements.

Chapter 3 **Controlling Execution**: Describes various ways to tell WPL which statement to execute next. Shows how a statement may be executed or not depending on some condition.

Chapter 4 **Output**: Shows how to send output to disk, screen, and printer from a WPL program.

Chapter 5 **Using String Variables**: Teaches how to use string variables—special symbols that stand for text.

Chapter 6 **Using Numeric Variables**: Teaches how to use numeric variables—special symbols that stand for numbers.

Chapter 7 **Advanced Techniques**: Describes how to write very large programs; explains how a WPL program can execute another WPL program; tells how to make and use a STARTUP program; teaches how to load the catalog into memory.

Chapter 8 **Enhancing WPL Programs**: Shows how to modify the standard WPL programs that come with Apple Writer.

Appendix A **Syntax of WPL Statements**: Summarizes the syntax of WPL statements.

Appendix B **List of WPL Commands**: Lists WPL commands in alphabetical order.

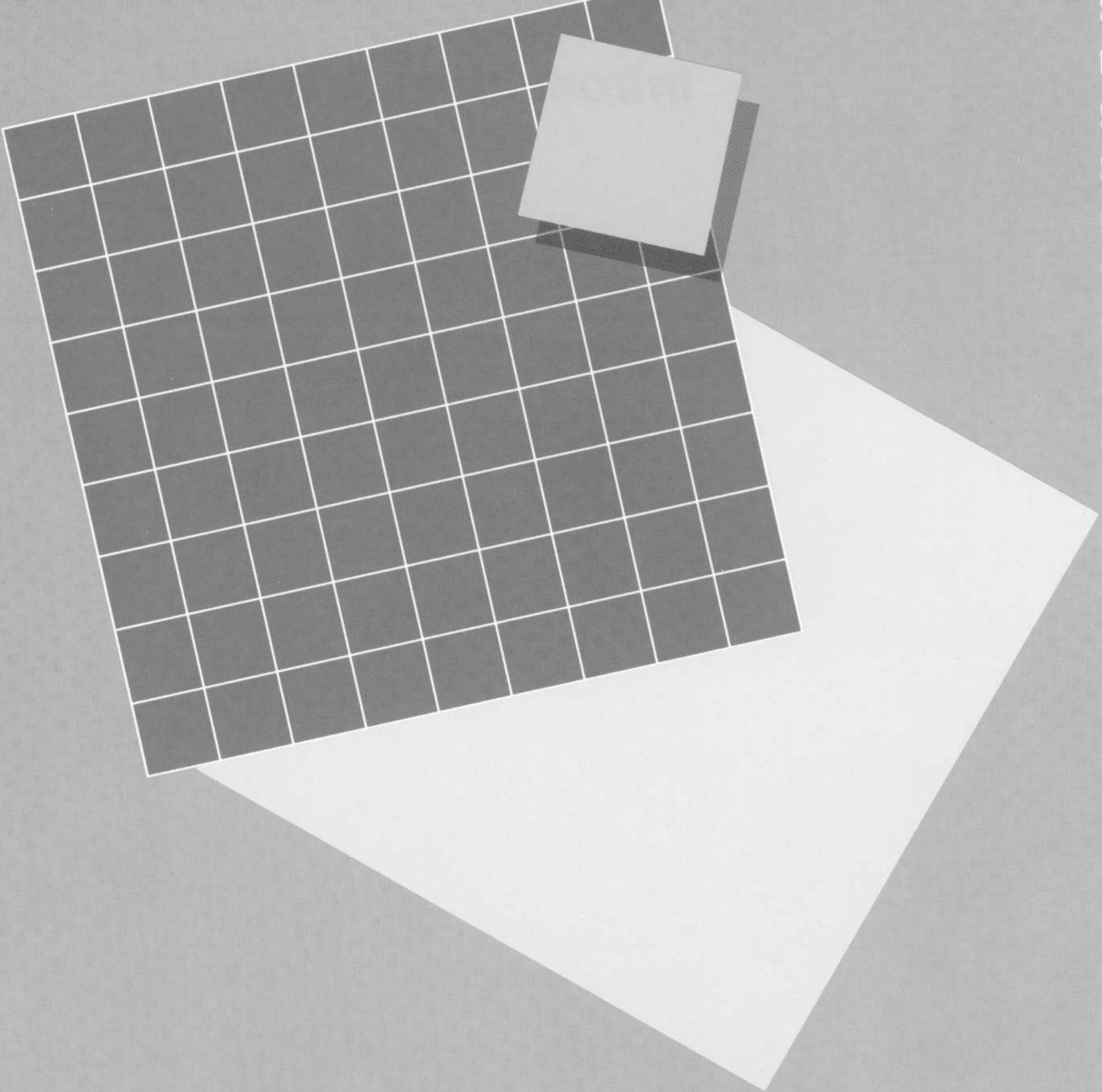
Appendix C **Summary of WPL Commands by Function**: Lists WPL commands by function; gives syntax of each command and examples.

Appendix D **Error Messages and Debugging Hints**: Lists and explains WPL error messages; provides debugging (error identification) hints.

Appendix E **Answers to Programming Questions**: Contains answers to the programming problems presented in this manual.

Appendix F **WPL Programs in This Manual**: Lists and describes all of the sample programs that appear in this manual.

Introduction to WPL



Introduction to WPL

WPL gives you a way of doing many exciting things with Apple Writer; you can

- create custom reports,
- write individualized form letters,
- do arithmetic calculations,
- perform repetitive Apple Writer functions,
- create your own menu programs,

and much more.

This chapter tells you about the relationship between Apple Writer and WPL. It also shows you how much you already know.

You already know how to write a WPL program!

WPL is a feature that makes Apple Writer more powerful and easier to use. WPL has some special commands that you'll learn about in this manual. But you could write an entire program right now, using just the Apple Writer commands you've already learned to use. Most of the commands you learned in Apple Writer can be used in WPL programs.

An Example of a WPL Program

Here's an example of a program you already know how to write. It's called `WPL.MEMOPRT` and it's on your UTILITIES disk.

> Before you do anything else, it's a good idea to make an extra copy of your UTILITIES disk. Once you have a backup copy, put the original away in a safe place and use the backup copy for the exercises in this manual.

WPL.MEMOPRT automatically prints the heading and body of a memo. Be sure your printer is turned on. Then start up your computer with the MASTER disk, insert the UTILITIES disk, and run the program by pressing [P] and then typing

```
do .d1/wpl.memoprt
```

The program is shown below in the lefthand column. The middle column contains the command name as you are used to seeing it in Apple Writer. Explanations of what each command does appear in the righthand column.

program	**Apple Writer command**	**what it does**
`MEMOPRT NY`	[N]Y	clears text buffer
`L .D1/HEADING`	[L]	loads document
`PNP`	[P]NP	prints document
`NY`	[N]Y	clears text buffer
`L .D1/BODY`	[L]	loads second document
`PCP`	[P]CP	prints second document

You'll notice that although you recognize all the commands in this program, they look somewhat different from the format you're used to seeing them in. Chapter 2 explains how to write any Apple Writer command you already know in this new format so you can use it in a WPL program.

What you need to be a programmer

There's one thing you need to be clear on to become a successful WPL programmer: that is, programming is easy! All you need is to be able to follow instructions, think through a problem step by step, pay attention to details, and learn from your mistakes.

How WPL Is Related to Apple Writer

Apple Writer and WPL are really both part of the same overall word processing system. WPL builds on commands and procedures you already know from Apple Writer, but WPL has some differences and many additional features.

Immediate, Embedded, and Deferred Commands

In Apple Writer you learned about two kinds of commands: immediate and embedded. In WPL there are immediate and embedded commands and a third kind of command as well: deferred.

Immediate commands cause something to happen right away. For instance, as soon as you type this immediate command

```
[S] filename
```

the computer immediately saves your file.

Table 1-1 shows all of the immediate commands you've learned in the Apple Writer manual as they appear in Apple Writer and as they appear in a WPL program.

Table 1-1. Immediate Commands

In Apple Writer	Meaning	In WPL
[B]	Beginning	B
[C]	Case Change mode	C
[D]	Direction arrow	D
[E]	End	E
[F]	Find and replace	F
[G]	Glossary	G
[L]	Load	L
[N]	New	N
[O]	SOS Command Menu	O
[P]?	Print/Program Menu	P?
[Q]	Additional Functions Menu	Q
[S]	Save	S
[T]	Tabs	T
[W]	Word delete/retrieve	W
[X]	Paragraph delete/retrieve	X
[Y]	Splits screen	Y
[Z]	Word wraparound	Z

The immediate commands [R] (Replace) and [V] (Control Character Insertion mode) cannot be used in a WPL program.

Embedded commands are those that you insert in your document. They aren't executed right away. Apple Writer pays no attention to them when you put them in your document. When you print the document, Apple Writer finds and executes the embedded commands in the course of printing.

In general, embedded commands have to do with determining how your document is printed. The Input command, an especially useful embedded command, stops the printing of a document and waits for your input. For instance, if you want Apple Writer to remind you to tighten the platen before a subscript or superscript command, you embed this command in your document:

```
      .IN Tighten the platen!
```

When Apple Writer comes to this command, it will display

```
Tighten the platen!
```

on the screen and wait for you to press RETURN before continuing to print.

Table 1-2 shows all of the embedded commands you've learned in the Apple Writer manual as they appear in Apple Writer and as they appear in a WPL program. As you may recall from the Apple Writer manual, most of the commands in Table 1-2 can also be used as immediate commands (that is, without the leading period).

Table 1-2. Embedded Commands

In Apple Writer	Meaning	In WPL
.LM	Left Margin	PLM
.PM	Paragraph Margin	PPM
.RM	Right Margin	PRM
.TM	Top Margin	PTM
.BM	Bottom Margin	PBM
.PN	Page Number	PPN
.PL	Printed Lines	PPL
.PI	Page Interval	PPI
.SP	Single Page	PSP
.CR	Carriage Return	PCR
.UT	Underline Token	PUT
.LJ	Left Justify	PLJ
.FJ	Fill Justify	PFJ
.CJ	Center Justify	PCJ
.RJ	Right Justify	PRJ
.TL	Top Line	PTL
.BL	Bottom Line	PBL
.IN	Input	PIN

The embedded commands `.FF` (Formfeed), and `.EP` (Enable Print) cannot be used in a WPL program.

A *deferred command* is one that is in a program; it is not executed until the program is run. Almost all of the commands that you learned in the Apple Writer manual can be used as deferred commands in a WPL program when they are written in a slightly different way. Here's what the [S], .IN , and LM commands look like when written as deferred commands:

```
s filename
pin Tighten the platen!
plm15
```

You write a WPL program by making a text file of deferred commands like those above; then to run the program you press [P], then type

```
do .d2/programname
```

and press RETURN.

Using Apple Writer "Hands Off"

WPL gives you a way to use Apple Writer "hands off," just like playing a player piano. You can run WPL programs that someone else wrote just as you can use a player piano roll to play a song that someone else recorded.

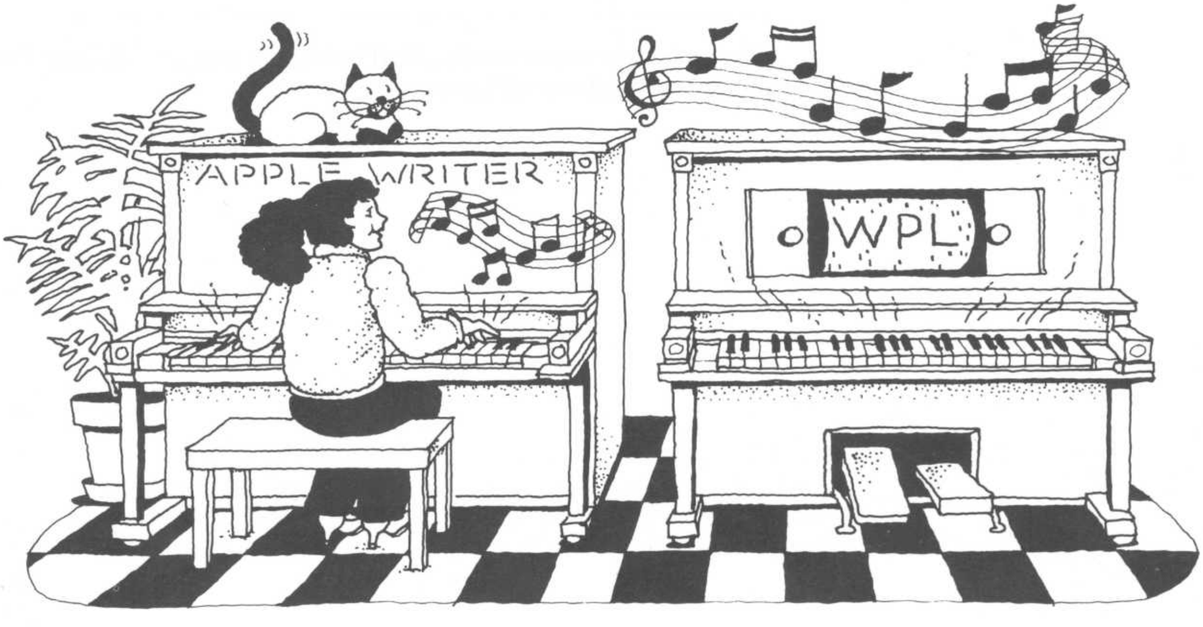


Figure 1-1. Music by WPL

You can "play" a WPL program over and over again and it never gets tired. And your hands don't have to be on the keyboard—you can be busy doing other tasks (or resting from your labors) while WPL plays merrily away.

Let's look at this a little more closely. To create a player piano roll, you have to play the piano. Everything you play, wrong notes and all, is recorded exactly as you played it. To create a WPL player piano roll, otherwise known as a program, you type the Apple Writer commands almost as you would it you wanted them to be executed immediately. (Apple Writer knows that they're not immediate commands because you don't use the CONTROL key.) You complete the recording by [S]aving the commands in a file called, say, PROGRAM, and you play it back by pressing [P], then typing

```
do .d1/program
```

and pressing RETURN.

There are a few special rules for typing commands that are part of a WPL program. The rules are called syntax and are covered in Chapter 2.

The [P]rint/Program Option

Some of the commands you've learned in Apple Writer are really WPL commands. Here's how to tell the difference. If you have to press [P] before issuing the command, it's a WPL command. Remember that [P] gets you the [P]rint/Program option of Apple Writer; the *print* part of this option has to do with text formatting, and the *program* part has to do with WPL. The WPL commands you know are

Commands that are preceded by [P] are WPL commands.

DO	Do (execute or run a WPL program)
NP	New Print (print the first or only file of a document)
CP	Continue Print (print the next file of a document)

Now look again at the WPL.MEMOPRT sample program earlier in this chapter. Notice that the NP and CP commands have a P in front of them. This stands for CONTROL - P (which you are used to seeing represented as [P] in the Apple Writer manual).

You will learn more about specific syntax requirements in Chapter 2. Right now the important thing to remember is that WPL commands are connected to Apple Writer through the [P]rint/Program option.

How WPL Shares Resources With Apple Writer

Apple Writer is a program that can talk to other programs. For instance, Apple Writer passes your messages to the operating system every time you [L]oad or [S]ave a file. Apple Writer also receives messages from any WPL program you run. Therefore Apple Writer and the WPL program must be in the memory of the computer at the same time. You might think that things get fairly complicated inside the computer, but in fact Apple Writer has "a place for everything and everything in its place." That's what the rest of this chapter is about.

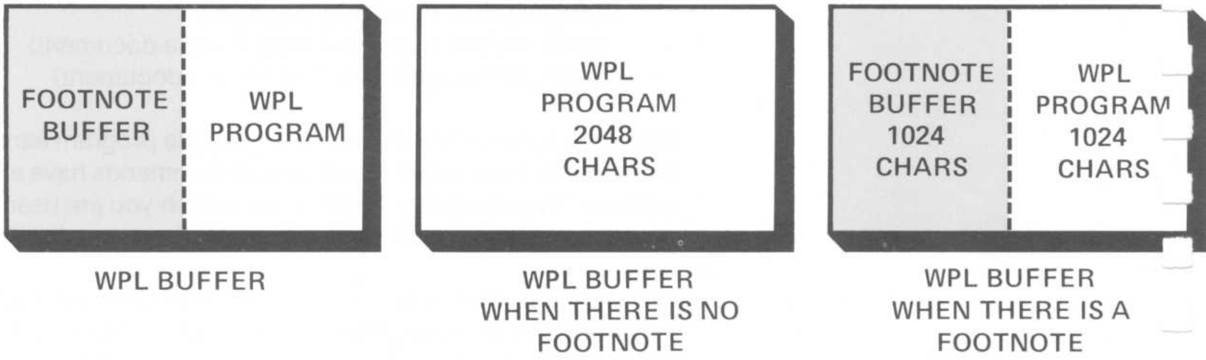
How WPL and Apple Writer Share Memory

Apple Writer and WPL each stay in special sections of memory assigned to them; these sections of memory are called *buffers*. The buffer assigned to WPL is 2,048 characters long. That's long enough to write quite a large WPL program, but it's not long enough to write any possible program, so there's a method called chaining that lets you connect several programs together. Chaining is described in Chapter 7.

WPL programs and the footnotes in a document share the same buffer. So if you're using a WPL program to print a document that contains footnotes, you won't have as much room available for your program as you otherwise would. The room available for the WPL program will be reduced to 1,024 characters; the rest of the buffer is used to hold the footnotes.

All of the other buffers that were described in the Apple Writer manual continue to exist when you run a WPL program. Figure 1-2 shows how WPL shares memory.

Figure 1-2. How WPL Shares Memory


FOOTNOTE BUFFER
WPL PROGRAM
WPL BUFFER
WPL PROGRAM 2048 CHARS
WPL BUFFER WHEN THERE IS NO FOOTNOTE
FOOTNOTE BUFFER 1024 CHARS
WPL PROGRAM 1024 CHARS
WPL BUFFER WHEN THERE IS A FOOTNOTE

The screen buffer assumes special significance in WPL, so we'll look at that next.

How WPL and Apple Writer Share the Screen

Remember that player piano? When you're playing the piano yourself—using Apple Writer in immediate mode—you usually want the screen to display all or part of the document in the text buffer. Occasionally you may load a document without putting it in the text buffer—that is, you display it temporarily on the screen. When you're using a player piano roll—a WPL program—you will scarcely ever want to display the document on your screen. There are three reasons for this:

Turn the text display off in most WPL programs.

- first, your WPL program will run up to five times faster if you don't display the document as it's being processed;
- second, you don't need to look at the text as it's being processed—you already know how it will be processed, because you designed the program; and
- third, there are other, more useful ways to use the screen.

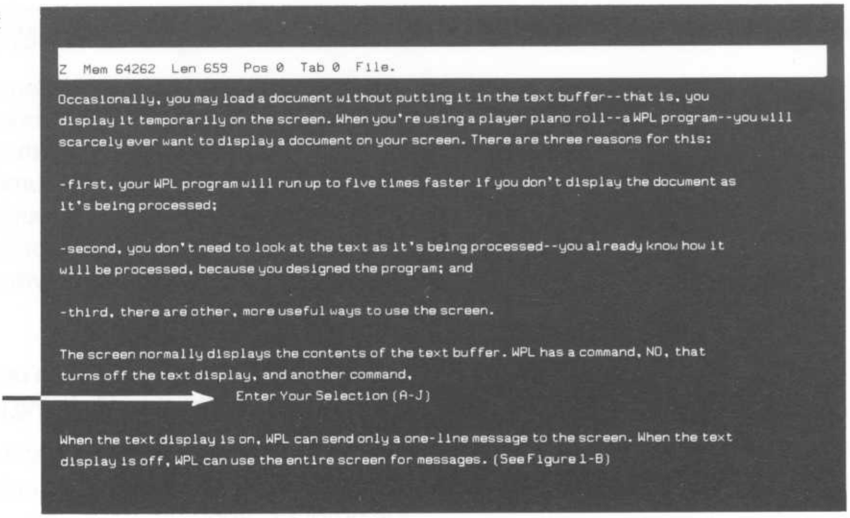
For a translation of the two-character name of any WPL command, see Appendix B. For a summary of how to use any WPL command, see Appendix C.

The screen normally displays the contents of the text buffer. WPL has a command, `ND`, that turns off the text display, and another command, `YD`, that turns it back on.

When the text display is on, WPL can send only a one-line message to the screen. When the text display is off, WPL can use the entire screen for messages. (See Figure 1-3.)

Figure 1-3. The Screen With and Without Text Display.

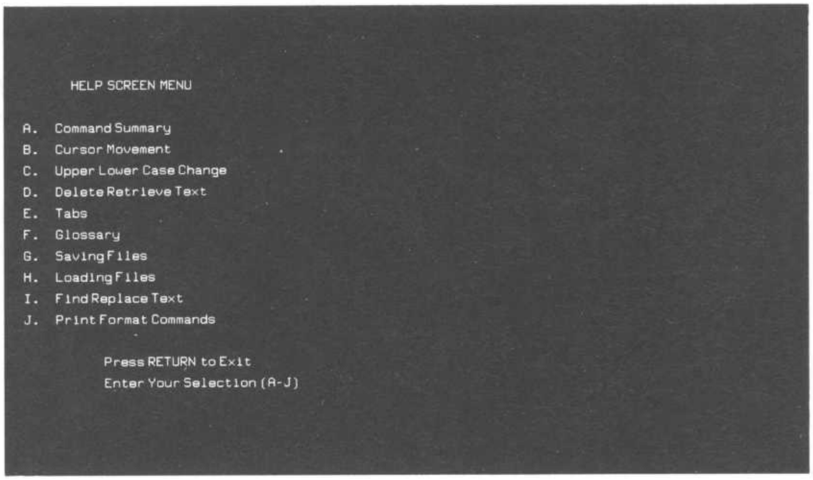
With text display (YD), you see only one line of the message, and it's often hard to see.



```
Z  Mem 64262  Len 659  Pos 0  Tab 0  File.

Occasionally, you may load a document without putting it in the text buffer--that is, you
display it temporarily on the screen. When you're using a player piano roll--a WPL program--you will
scarcely ever want to display a document on your screen. There are three reasons for this:

-first, your WPL program will run up to five times faster if you don't display the document as
it's being processed;

-second, you don't need to look at the text as it's being processed--you already know how it
will be processed, because you designed the program; and

-third, there are other, more useful ways to use the screen.

The screen normally displays the contents of the text buffer. WPL has a command, NO, that
turns off the text display, and another command,
                     Enter Your Selection (A-J)

When the text display is on, WPL can send only a one-line message to the screen. When the text
display is off, WPL can use the entire screen for messages. (See Figure 1-B)
```

Figure 1-3 A.

Without text display (ND), you see messages on the whole screen. (The Help Screen Menu is a message from a WPL program.)



```
      HELP SCREEN MENU

A.  Command Summary
B.  Cursor Movement
C.  Upper Lower Case Change
D.  Delete Retrieve Text
E.  Tabs
F.  Glossary
G.  Saving Files
H.  Loading Files
I.  Find Replace Text
J.  Print Format Commands

          Press RETURN to Exit
          Enter Your Selection (A-J)
```

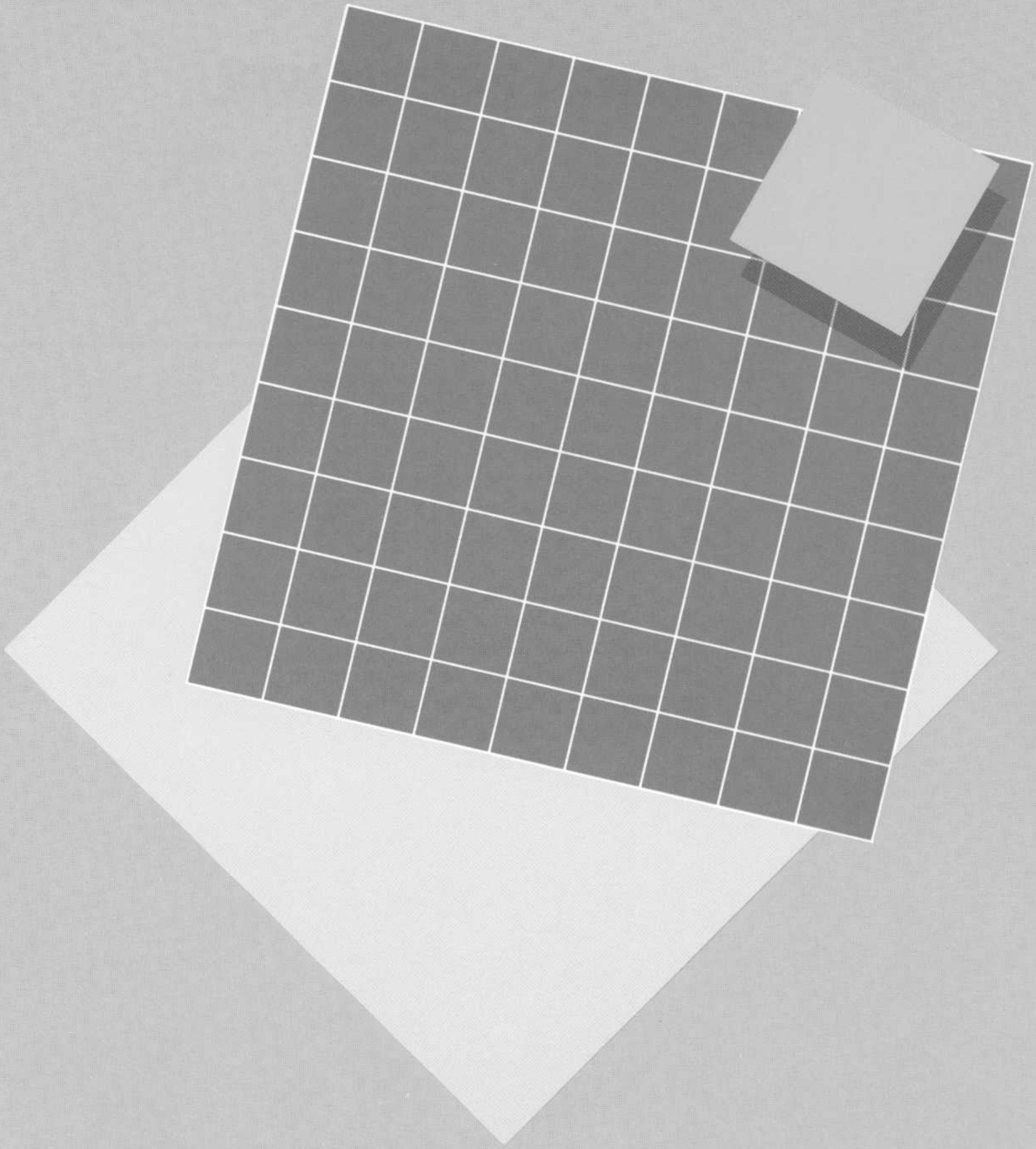
Figure 1-3 B.

Use your screen for **debugging**.

By turning off the text display, you can create menus like those in Apple Writer or you can display the results of computations performed by your WPL program. You can also display information that will help you *debug* your program. Debugging is a popular computer term. It means removing the errors, or bugs, from your program. See Appendix D for more information on program errors and on using the screen to debug programs.

Now that you know how WPL and Apple Writer are related, you are ready to begin actually writing in WPL. Just turn the page.

How to Write in WPL



How to Write in WPL

A WPL program is a series of statements. Whether we refer to them as *WPL statements* or just plain *statements*, we mean exactly the same thing. Each statement represents a complete thought, just as an English sentence does. A statement usually occupies one line of text (that is, it begins and ends with a RETURN), although certain Apple Writer commands such as [F]ind may require a two-line statement.

First you will learn what the parts of a statement are called and what they're for. Then you'll be shown how to make statements out of all the Apple Writer commands and WPL commands you already know.

Syntax—the rules for fitting parts together

Syntax is the name for the collection of rules that describes how words in a sentence fit together. For instance, in English syntax the following order is usual:

SUBJECT	VERB	OBJECT
I	threw	the ball.
The ball	hit	the fence.

The only way to tell whether the ball is a subject or object in the second sentence is to study the word order. This chapter teaches you the simple syntax rules that apply to WPL statements. (See Appendix A for a summary of the syntax rules.) The rules help Apple Writer interpret your program correctly, without ambiguity; they also help you remember what you meant when you wrote it.

Before we look at the details, though, let's look at the big picture.

Overview of WPL

This section

- defines some terms that are used in this manual;
- introduces you to two programming concepts: labels and comments;
- gives you some general rules about when to use upper- and lowercase;
- reminds you how to save and use a WPL program.

Commands and Statements

Command. This word means just what it does in the Apple Writer manual: an order given to the computer, like [F]ind or [P]NP (New Print).

Apple Writer Command. This is a command that you learned to enter as an immediate command by pressing the CONTROL key while typing the command letter. Apple Writer commands in a WPL statement are entered without using the CONTROL key. The command names are always one character long. Here are some Apple Writer commands:

As Immediate Commands	As Deferred Commands in a WPL Program
[B]	B
[L]	L
[N]	N

WPL Command. This is a command that you learned to enter as an immediate command by pressing [P] before typing the command. In WPL, these commands are entered by preceding them with the letter P. The command names themselves are always two characters long. Here are some WPL commands:

As Immediate Commands	As Deferred Commands in a WPL Program
[P]NP	PNP
[P]CP	PCP
[P]DO	PDO

Statement. A statement, sometimes called a WPL statement, is like a sentence; it is a complete thought. The term statement generally refers to a statement that contains a command, but a statement may also be a comment, which is defined below. A command statement includes the command name, an optional label, and any other information supplied with the command.

Labels

Giving names to commands

Labels are names. They tell Apple Writer which WPL statement in your program you're referring to. You've never had to do that before because you haven't needed to talk about a command—you could just execute it. But sometimes in a program you want to tell Apple Writer to go back and do a set of commands that it has already done. The way to do that is to give the first statement in the set a name; for example in the following program segment, `NEWDOC` is a label.

```
NEWDOC L .D2/NEWTEXT
       . . .
       . . .
       PGO NEWDOC
```

Comments

Internal documentation

A comment is a note to yourself written in WPL syntax and included in a program. Comments, like labels, are things you haven't had any use for before. They're special statements that Apple Writer ignores when it executes your WPL program. Comments are there for the sole purpose of helping you to remember how your program works. Then if you haven't looked at the program for six months you can read the comments to see what it does. The following statement is a comment:

```
   P Note that the latest version of the file must be loaded.
```

(A comment always begins with a `P`. See the end of this chapter for more information on how to write comments.)

Uppercase and Lowercase

Some of the WPL statements in this manual are printed in uppercase and others are printed in lowercase. Most of the time Apple Writer doesn't care which you use. `PNP` is exactly the same command as `pnp` or `Pnp` or even `pNp`.

Upper- and lowercase matter in labels.

There are two places where upper- and lowercase definitely do make a difference, however. The first has to do with *labels*: a label must be typed exactly the same way wherever it is used. If you use uppercase for a label in one place in your program and lowercase in another, Apple Writer will think you are talking about two different labels. This can lead to very strange program results.

The argument follows the command and gives more information about it.

The other place where upper- and lowercase make a difference is in the *argument* of a WPL statement—the portion of the statement that follows the command and gives additional information about it. (To find out more about arguments, see the section called "The Parts of a WPL Statement" later in this chapter.) A part of the argument may be intended to have a precise value in which upper- and lowercase are significant. For instance, if you ask Apple Writer to [`F`]ind `/apple/` (the fruit), that's not the same as finding `/Apple/` (the company). Here's the rule of thumb to follow in writing Apple Writer and WPL arguments:

- If the argument must be a specific value, such as `Y` or `N`, it can be entered in either upper- or lowercase.
- If the argument may be any value, such as a string of text to be found or printed, then the argument is taken exactly as you typed it.

Editing and Saving a WPL Program

A WPL program is a document that has been saved in a file. It is created the same way any Apple Writer document is created.

To **create** a new WPL program, clear memory with the Apple Writer [`N`]ew command, type the program, and save it in a file using the Apple Writer [`S`]ave command:

```
[S]ave: .d1/programname
```

To **edit** an existing WPL program, use the Apple Writer [L]oad command to load the file:

```
[L]oad: .d1/programname
```

Then edit it and save it.

To **use** a WPL program that you have written and saved on a disk, press [P] and then type the WPL Do command followed by the name of the file you saved the program in:

```
[P]rint/Program: do .d1/programname
```

See Appendix F for a list of the programs on the UTILITIES disk.

All the sample programs used in this manual are found on the UTILITIES disk; the program names all begin with WPL.

Important:
It's a good idea to start the names of all your WPL programs with WPL., like this

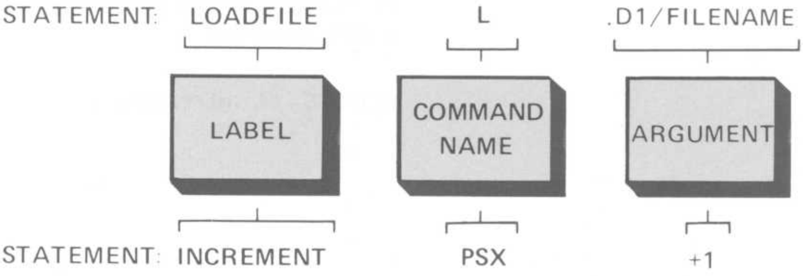
```
wpl.programname
```

so that you can easily distinguish WPL programs from the rest of your files.

The Parts of a WPL Statement

A statement always contains a command name (unless it is a comment statement); it may also contain a label (the name of the statement) and an argument (the additional information that is needed in order to execute the command). Figure 2-1 shows the syntax of a statement.

Figure 2-1. Syntax of a WPL Statement.



The Label

The label is a name for the statement. Labels are optional, but you must give a name to a statement if you refer to it in another statement. The commands in WPL that refer to another statement are GO and Subroutine Call (SR).

You may also give any statement a name in order to help you remember what the statement does. This is known as *internal documentation*—information about a program that is contained in the program itself. (Another way to document your program is to include comment statements. You will learn how to write comments later in this chapter.)

Here are the rules for writing a label:

- It must begin in the leftmost position of the line (that is, at Tab position 0 on your screen).
- It may contain any character except the space, including upper- and lowercase letters.
- It may be any length.
- It must be followed by at least one space.

When you refer to a label in a WPL statement, you must spell it **exactly** the way you did when you wrote the label. The following list contains three **different** labels:

```
MYLABEL
mylabel
MyLabel
```

Here's what a label (GETFILE) looks like as part of a statement in a WPL program:

```
GETFILE   L .dl/filename
```

The Command

The command is like a verb; it's the part of a statement that tells the computer what to do. Most commands contain two parts: the command name (like [F]ind or DO) and an argument (described in the next section). Some commands do not take arguments. Here are the rules for writing a command:

- Leave one or more spaces before the command, **whether or not it is preceded by a label**.
- If the command is an Apple Writer command, type the command name by itself, without using the CONTROL key; for instance, type [L]oad or [S]ave as L or S.
- If the command is a WPL command like NP or DO, place the letter P immediately before the command name: PNP or PDO. (The P stands for the Apple Writer [P]rint/Program command.)

The Argument

Argument is a mathematical term that programmers use. The argument is the part of a command that gives additional information about it. In other words, it's a modifier that tells Apple Writer precisely how to execute the command. Some commands (like [E]nd) work all by themselves without an argument. Others have an argument that consists of one, two, or more parts. As you are introduced to each new command in this manual, you'll be told exactly how to write the argument if the command takes one.

The argument is the last part of a statement, and it is always followed by RETURN. This means that the next statement in the program always starts on a new line. In the following statement, .d1/autoletter is the argument:

```
PDO .d1/autoletter
```

How to Write a Statement

Let's begin with a question of style. Some programming languages require you to write certain parts of a statement in particular positions on the line. (These positions are often called *columns* because they originally referred to the columns of a keypunch card.) WPL, on the other hand, is quite flexible in its format. Nevertheless you may find that your program is a lot easier to read and work with if you follow a set pattern in writing statements. Here is the one we recommend:

- Decide how long your longest label will be. Then begin every command two positions beyond that point, whether or not the command has a label. That way the commands will be lined up and easy to read.
- When spaces between the command name and the argument are optional, use them. Apple Writer won't care, and you'll be able to read your program more easily.
- If you line up the commands as we suggest, a statement will use the same number of characters in its label area whether it contains a label or just spaces because spaces are counted as characters. Therefore use labels liberally to remind you what the statements do. In the `LOADFILE` program below, the first statement takes no more room than the last statement.

```
LOADFILE  L .D1/LETTER
REPLACE   F/1981/1983/A
PRINT     PNP
          S .D1/LETTER
```

- If your program is long, you'll save room by keeping your labels short.

Writing an Apple Writer Command

Here are the rules for writing an Apple Writer command in a WPL program:

- Type the one-letter command name without pressing the CONTROL key. If you want [`F`]ind, type `F`, if you want [`G`]lossary, type `G`, and so on.
- For file commands ([`L`]oad, [`S`]ave), spaces between the command name and the argument are optional.

- For all other commands, spaces between the command name and the argument are **not allowed**.

Here are the rules for writing the argument of an Apple Writer command in a WPL program:

- Type the argument **exactly** the way you would type it if you were using the command as an immediate command. That is, if you must press CONTROL or RETURN when you enter the argument in Apple Writer, press CONTROL or RETURN when you enter the argument in your WPL program. CONTROL should always be preceded and followed by [V].
- Always end the statement with RETURN. If the statement has no argument, press RETURN after the command name. If there's an argument, press RETURN at the end of the argument.
- If, when you type the command in Apple Writer, you need to press RETURN twice (for instance, to exit from the DOS Command Menu) then you must also press RETURN twice when you use the command in a WPL program. However, in the WPL program you must put one or more spaces between the two RETURN's. This is because the second RETURN cannot be in the leftmost position of the line. If it were in the leftmost postion, it would look like a label and be ignored. This in turn would mean that the RETURN at the end of the following line would be used to exit from the menu and the command on that line would be ignored.

Now let's look at three examples of Apple Writer commands written as program statements. Each statement begins on a new line because the preceding statement ends with RETURN.

Apple Writer Command	Description
1. `B`	Move cursor to beginning of file.
2. `NY`	Clear memory.
3. `F/June/July/` `Y?`	Find and replace text.

Example 1. B

Example: move cursor to beginning of file

[B] is an Apple Writer command that doesn't need an argument. It moves the cursor to the beginning of the document in the memory buffer, and it sets the direction to >. In a WPL program, you use this command if, for example, you want to search for a marker from the beginning of a document to the end. For instance

```
L .d1/budget.annual      Load file.
B                        Move cursor.
F/1982/1983/A            Find, replace all dates.
```

This little program loads a file called BUDGET.ANNUAL into the text buffer from the disk in drive 1. It moves the cursor to the beginning of the document in memory and then searches through the document, replacing every occurrence of 1982 with 1983.

Now that you know the syntax rules for Apple Writer commands in a WPL program, you should be able to read this sample program without difficulty. If you're having trouble remembering what the Apple Writer commands do, please look them up in your Apple Writer manual right now.

Even if you're feeling pretty comfortable with Apple Writer, you may find it helpful to review all the features of each command before you begin programming. That's because some features are especially useful when used in a program, and you might have missed their significance the first time around.

Example 2: NY

Example: clear memory

[N]ew is an Apple Writer command that requires an argument. The command clears memory. When you press [N], Apple Writer responds by asking you if you really want to erase memory, and you may answer either Y or N. Because you are prompted for an answer when you use [N] as an immediate command, you must respond as if you were prompted when you issue this command in a WPL program. Remember, WPL is like a player piano—the same notes are played whether or not the pianist is at the keyboard.

You want your program to answer yes, of course. If you didn't, you wouldn't have written a [N]ew statement in the first place. So type the response, which is this command's argument, just the way you would if you wanted the command to be executed immediately. Notice that there aren't any spaces between the N and the Y. You wouldn't have typed a space at the keyboard, so Apple Writer doesn't expect you to put one in your program.

Example 3: `F/June/July/`
`Y?`

Example: replace a word

[F] is an Apple Writer command whose argument has more than one part. You can use this command to find text (that is, position the cursor), or to find and replace the next occurrence of the text (which is what this example does), or to find and replace all occurrences automatically.

The [F]ind statement requires **two lines** in your WPL program. That's because you would have to press RETURN after typing `/June/July/` if you were entering the [F]ind command at the keyboard for immediate execution. If you want to see for yourself, type some text containing the word `June` and then do the following:

Press [B]	(Move cursor to start.)
Press [F]	(Find/replace command.)
Type `/June/July/`	(Replace June with July.)

Nothing happened, right? That's because you haven't pressed RETURN yet. Do that right now, and you'll see the next prompt:

```
[F]ind:RETURN = Proceed / Y = Replace
```

Type `Y` and the replacement will be made. You'll also get the final prompt, without having to press RETURN:

```
[F]ind:RETURN = Proceed
```

Now type

```
?
```

The prompt disappears and the cursor is positioned after the text that was replaced. (The question mark is explained below.)

If you look at the statement that contains this Apple Writer command, you'll see that the statement does exactly what you did at the keyboard. Here it is again:

```
    F/June/July/
    Y?
```

Where you had to press RETURN, the statement contains a RETURN and continues on the next line. Where you didn't press RETURN after the `Y`, the statement continues on the same line. That's the way **all** Apple Writer commands are written in WPL—just the way they're entered "live at the keyboard."

It's probably all clear to you now except for that question mark. What in the world is that? You should be able to see from the syntax you've learned that it's a response to the final prompt:

```
[F]ind:RETURN = Proceed
```

What's confusing is that you're used to responding to that prompt with a space if you don't want to proceed. However, if you go back to your Apple Writer manual, you'll see that any character except RETURN will have the same effect as a space.

How to exit from the [F]ind command in a program

Since a space doesn't print, you can't see whether it's there or not. Your WPL program can read it, but you can't. Therefore most WPL programmers use the question mark by convention to exit from the [F]ind command.

Writing a WPL Command

Here are the rules for writing a WPL command in a program:

- Precede the 2-letter command name with the letter P. For instance, NP becomes PNP, CP becomes PCP, and so on.
- For a few commands—Assign String, Input, and Print—spaces between the command name and the argument are considered to be part of the argument.
- For all other WPL commands, spaces between the command name and the argument are optional and may be used to make your program more readable.
- Always end the statement with RETURN. If the statement has no argument, press RETURN after the command name. If there's an argument, press RETURN at the end of the argument.

Here are the WPL commands you learned in the Apple Writer manual, first as they are written in WPL and then as they are entered as immediate commands:

In a Program	From the Keyboard
`PDO .d2/filename`	Press [P] Type `do .d2/filename`
`PNP`	Press [P] Type `NP`
`PCP`	Press [P] Type `CP`

The rest of this manual will teach you how to write programs and how to use the 14 other WPL commands.

Writing Comments

Comments are statements that annotate your program. The format of a comment statement is

`{label}` space(s) `P` space(s) `text of comment`

Braces ({ }) mean that the label is an optional part of the comment statement.

The comment format is a useful convention. If, while executing a program, Apple Writer finds a command it doesn't recognize, it throws it away—that is, the unrecognized command is ignored. Apple Writer interprets the comment format as an unrecognized WPL command because the first character after the `P` is a space.

Comments help you remember how your program works.

You cannot believe how easy it is to forget how your program works or even how to read it until it happens to you. Therefore when you write a program it's important to document it by inserting comments that explain the design of the program. Here are some other documentation techniques you might want to consider:

- Keep a list of all the files on each disk, with a one-line statement of the nature of each file.
- Write a paragraph about each program you write, with an explanation of what the program does and how to run it.
- Make a master list of all programs and files that work together to accomplish a particular job.

Note that you can write all your documentation with Apple Writer!

Here is an example of a commented program called STAR. The first statement is a comment. Its label is the program name and the comment itself is the program description. The convention of labeling the first statement with the program name is used throughout this manual.

Example: filling memory with stars

```
STAR  p THIS PROGRAM FILLS MEMORY WITH STARS  (comment)
      ny
      p INSERT A STAR INTO MEMORY             (comment)
      f//*/
      y?
loop  e
      p LOAD MORE STARS                       (comment)
      L#
      pgo loop
```

You know all the commands in this program except for the very last, pgo loop, which says, "Go to the statement labeled loop and begin executing instructions." You'll learn more about the Go command in the next chapter.

By the way: The STAR program illustrates a handy way to insert text into the document in memory. Just use the [F]ind commmand to find "nothing" and replace it with text, like this:

```
F//text/
```

(The WPL equivalent of "nothing" is two delimiters with nothing in between.)

The text is inserted at the current cursor position. You can also delete text by replacing it with nothing, like this:

```
F/text//
```

Here's a chance for you to practice editing and running a WPL program. Clear memory and type the sample program in with Apple Writer. Notice that Apple Writer doesn't do anything except edit text—it doesn't try to execute any of the commands because you haven't entered them in immediate mode.

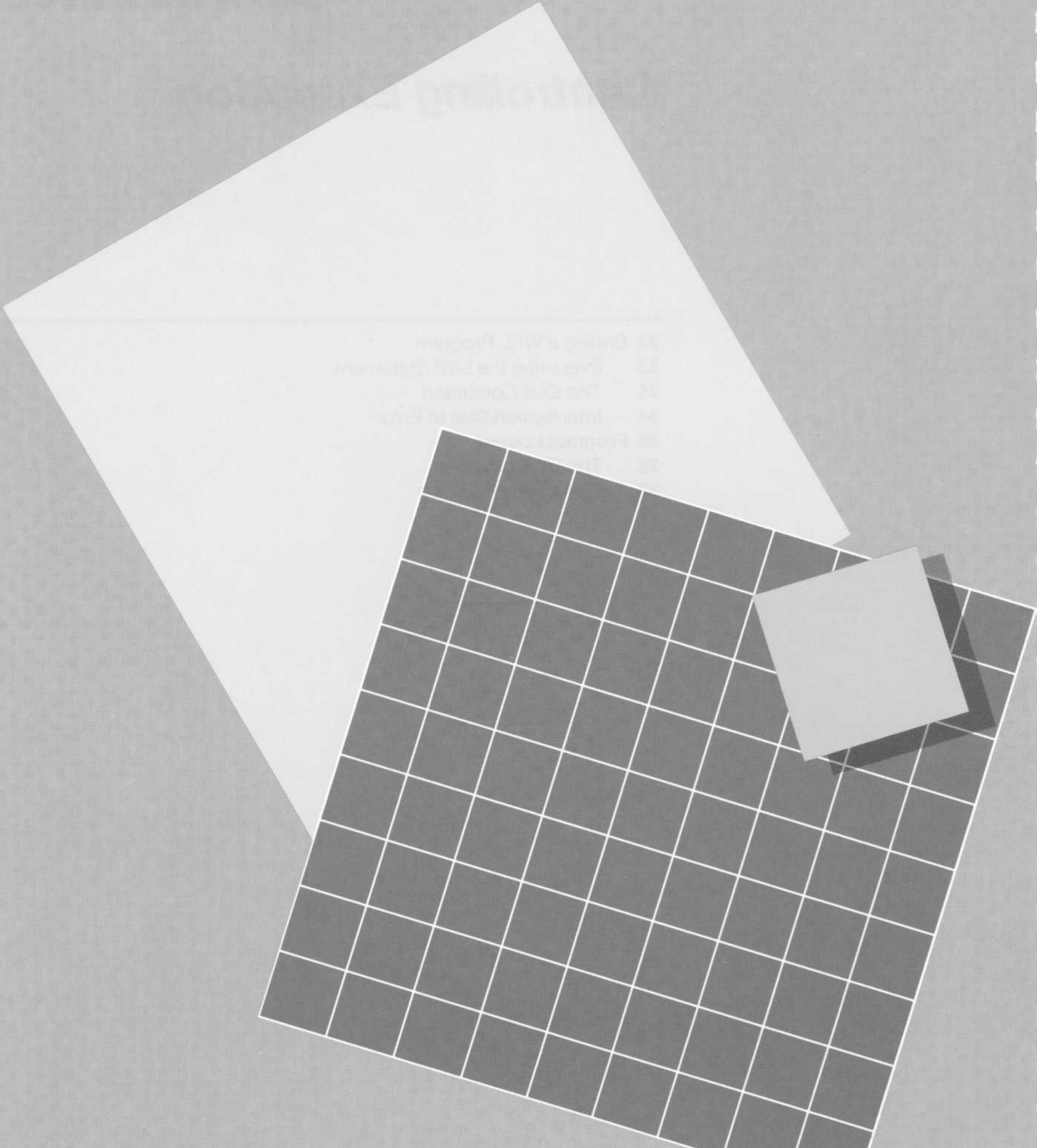
When you're finished typing, save the program and run it. Watch the screen fill up with stars and see the length value (Len:) on the Data Line increase. Soon the computer will beep and the cursor will begin flashing again. That means the program has filled all of memory and has halted.

Now clear memory and load the program so you can edit it. Change it so that it places `*#$` in memory. Save the new version and run it. You can stop the execution of the program at any time by pressing (ESCAPE). (Look in Appendix E to see if you modified the `STAR` program correctly. Appendix E contains the answers to all of the programming questions in this manual.)

Try "playing computer" by reading the program and writing down the results of each statement so that you can see for yourself why the program works the way it does. See if you can change the program so it fills memory faster or makes a more interesting pattern on the screen.

When you're ready to learn some new WPL commands, go on to the next chapter.

Controlling Execution



Chapter 3

Controlling Execution

In this chapter you will learn

- how to end a WPL program.
- how to perform repetitive functions in a WPL program. (You saw an example of this in Chapter 2, when you ran the STAR program which fills memory with stars.)

Ending a WPL Program

The four ways to stop a program

You already know one way to stop a running WPL program: press ESCAPE and it will immediately stop. That's a little like stopping a car by driving it into a brick wall. There are two ways that you can program a graceful ending: by executing the last statement or by using the Quit command. Apple Writer also stops your program when it finds an error in it.

Executing the Last Statement

In a simple WPL program, the statements are executed in sequence until the last statement is encountered. When there are no more statements to execute, the program stops and Apple Writer returns you to its edit function. The MEMOPRT program at the beginning of Chapter 1 is an example of halting by executing the last statement. Here's another example, the APPEND program:

Example: appending files

(For a more general version, see APPEND2 in Chapter 5.)

```
APPEND   P THIS PROGRAM MAKES THREE FILES INTO ONE
         NY
         L .D2/JANUARY
         L .D2/FEBRUARY
         L .D2/MARCH
         S .D1/QUARTER1
```

The APPEND program doesn't need a special command to tell Apple Writer when it's done. More complex programs, however, may be written so that the last statement in the program isn't necessarily the last one to be executed. In that case, the Quit command is used.

The Quit Command

The format of the Quit command is

```
      PQT
```

When Apple Writer encounters the Quit command in the course of running a WPL program, it stops the program and puts you back in the Apple Writer editor. Whatever was in the text buffer while the program was running is still there.

You may use the Quit command more than once in a program. The MESSAGE program, below, contains two Quit statements.

Example: Printing a daily message

```
MESSAGE L .D1/MSG.DAILY/JUNE 07/JUNE 08/N
        PGO PRINT
        PQT
PRINT   PNP
        PQT
```

The first Quit statement causes the program to stop only if the JUNE 07 marker is not found. (We'll tell you more about why some commands work only under certain conditions in the section "Conditional Execution" in Chapter 5.) The second Quit statement causes the program to stop after printing the text between the markers.

Interruption Due to Error

When you run a WPL program, Apple Writer sometimes finds a condition that keeps it from proceeding further. For instance, you may have told it that a certain file is on your disk, but Apple Writer can't find the file because you misspelled the name in your program. When Apple Writer recognizes an error condition, it says in effect, "I give up. I can't continue to run this program. Here's the problem I found, displayed on the screen."

Appendix D lists and explains the five WPL error conditions that can cause Apple Writer to stop running a program. If the error message displayed on the screen is not covered in Appendix D, it's a SOS message and you will find an explanation of it in the Apple Writer manual. (Errors associated with files are usually SOS errors.) Please look at Appendix D now to familiarize yourself with the format of the error messages and the types of errors that can occur. You may not understand these messages right now, but you will shortly.

If your program stops with an error message, look up the message in Appendix D to find an explanation of the message and some suggestions as to what might have caused the error. When you find the problem, load and edit your program, save the corrected version, and run the program again.

Program Loops

How to do things over and over again

Program loops are a means of doing repetitive tasks. If you want a certain function in your program to be performed over and over again, you don't want to have to write out a complete set of instructions for each repetition. In WPL there's a way to say, "Go back and do that again." In fact, the command is called `GO`.

Here's how program loops work. Let's say that there's a circular track near a jogger's house. Every day the jogger walks to the track, runs around it five times, and then walks to the office. Figure 3-1 shows what the jogger's path looks like:

Figure 3-1. A Path Containing a Loop

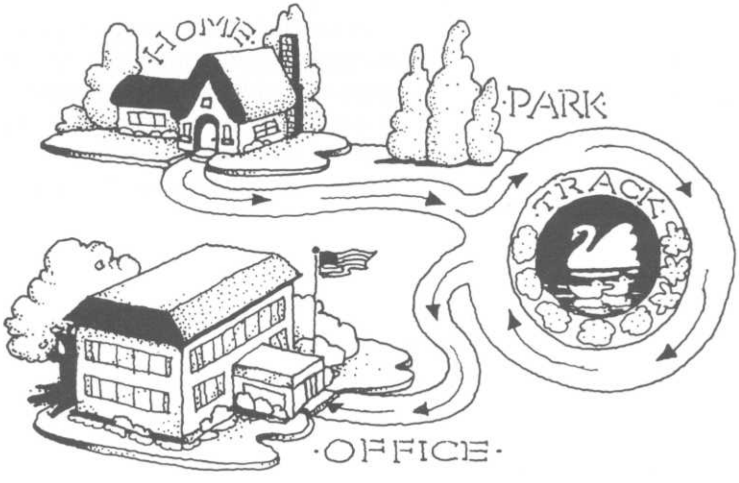
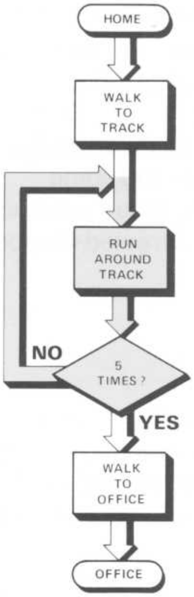


Figure 3-2. Program Logic of the Jogger's Path



The loop in the path is the part that is repeated. It's called a loop because the end joins the beginning to make a circle. A program, on the other hand, is a (more or less) straight path. It has a beginning, a middle, and an end. But it may contain any number of loops. If you were to write a program to describe the jogger's path, its logic would look like the drawing in Figure 3-2.

The Go Command

The Go command makes program loops possible in WPL. The format of the command is

```
PGO statementlabel
```

Normally Apple Writer executes WPL statements consecutively. The Go command, however, causes execution to continue at the statement whose label is named in the Go command. For example

```
loop1   ...
        ...
        {if condition 1 skip next statement}
        pgo loop1
        ...
loop2   ...
        ...
        {if condition 2 skip next statement}
        pgo loop2
        ...
```

This example shows two loops in a program, equivalent to having our jogger run around the track a few times (`loop1`) and then run around the office building a few times (`loop2`). The first loop is executed until condition 1 occurs; then the Go statement is skipped and the program proceeds to loop 2. The second loop is executed until condition 2 occurs. Chapter 5 tells you how to define conditions.

It is also possible to *nest* loops so that one falls inside another:

A nest of loops

```
loop3   ...
        ...
loop4   ...
        ...
        {if condition 4 skip next statement}
        pgo loop4
        ...
        {if condition 3 skip next statement}
        pgo loop3
        ...
```

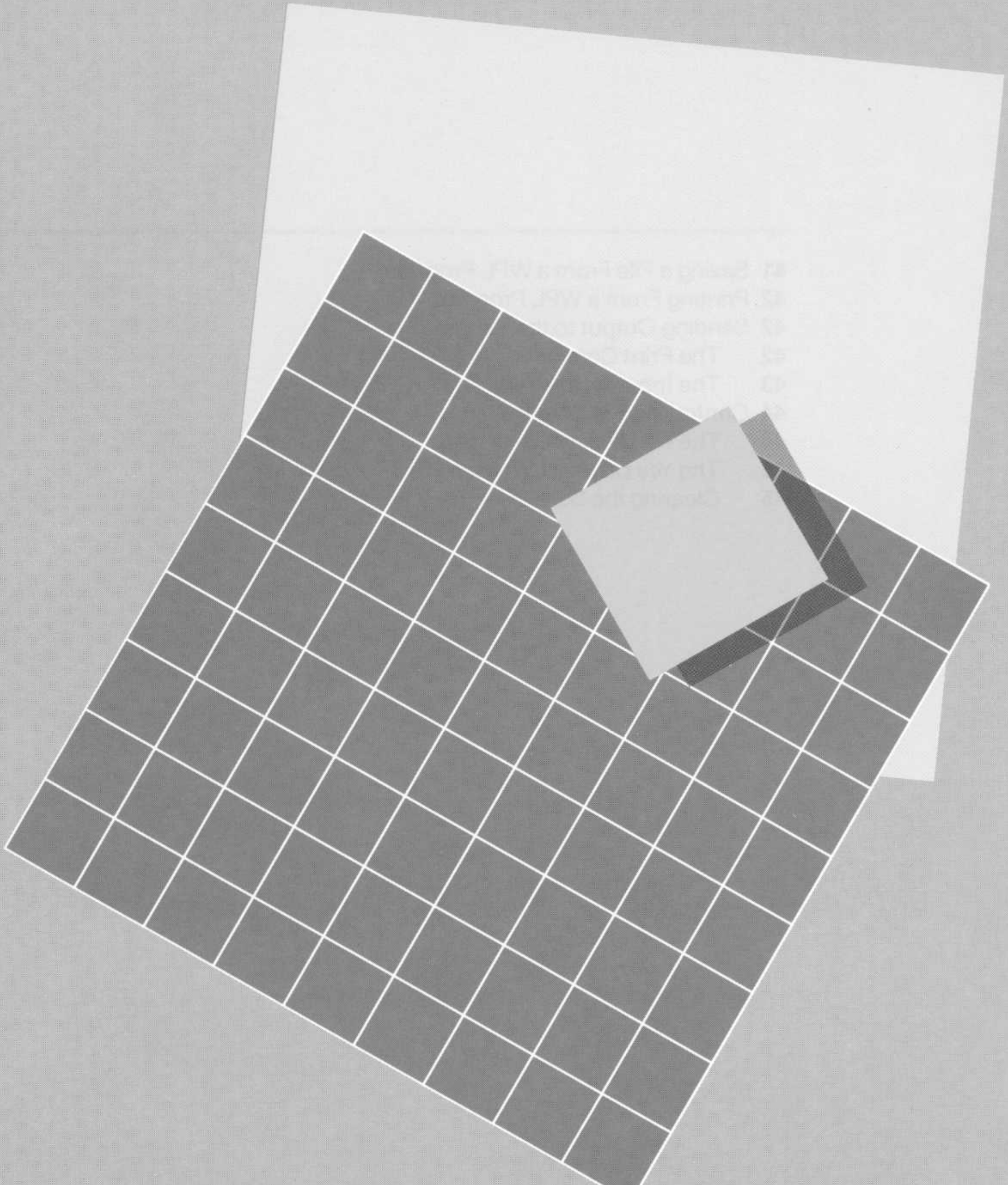
Exiting From a Loop

A never-ending loop is not usually a good idea, for a jogger or for a program. The STAR program in Chapter 2 contained an endless loop, but it was the kind that Apple Writer would eventually find out about. Sometimes Apple Writer doesn't know or care if your loop goes on forever. If you happen to be printing in the loop, you can chew up vast forests worth of paper before you discover the problem.

There are three ways of exiting from a loop:

- You can use the Go command to go (or *branch*) to a statement outside the loop.
- You can use the Quit command to end the program.
- You can take advantage of conditional execution, a feature of WPL in which certain commands may cause the next statement in the program to be bypassed. Chapter 5 tells you how to use conditional execution.

Output



Chapter 4

Output

This chapter covers WPL's facilities for

- writing files,
- sending messages to the screen,
- receiving input from the keyboard,
- and printing text.

Saving a File From a WPL Program

You can write a program that creates a file or modifies an existing file and saves the results on a disk. You have already seen an example of this. The APPEND program in Chapter 3 created a quarterly file by loading three monthly files into the text buffer and then issuing a [S]ave command. This program saved the entire document in the QUARTER1 file.

Refer to the Apple Writer manual for more ways to save and load files.

You can also use the special features of the Apple Writer [S]ave command to **save part** of a document in a new file or to **add part or all** of a document to an existing file. A WPL program that saves part of a document in a new file is

```
savepart L .d2/oldfile
           b
           f/firstword/
           ?
           s .d2/newfile!lastword!
```

A WPL program that adds a document to the end of an existing file is

```
addon      L .d1/newdata
           s .d1/oldfile+
```

Printing From a WPL Program

When you learned Apple Writer, you learned how to use the NP (New Print) and CP (Continue Print) WPL commands as immediate commands. You also saw these commands used in the MEMOPRT program in Chapter 1 of this manual. In Chapter 5, you will discover how to create a menu program that uses these commands to print any number of files. For a summary of the New Print and Continue Print commands, see Appendix C.

Sending Output to the Screen

One of the most useful enhancements WPL makes to Apple Writer is the capability of sending and receiving screen messages, allowing you to interact with a running WPL program. You have already used programs that do this: the Help screens in Apple Writer are created using WPL programs to display menus and receive your input. This chapter introduces you to various screen output commands and techniques.

The Print Command

The Print command displays text on the screen. You can use it to display up to 128 characters of text, but we recommend that you limit your text to that which will fit on one line of your display.

The format of the Print command is

```
PPR text
```

Any spaces between the command and the first printable character will be treated as part of the text and placed on the screen. For example, the following command displays a centered heading on an 80-column display:

```
PPR                     DAPPLED SAPLINGS TOPPLED
```

The text portion of the Print command may contain any combination of keyboard characters. A **blank line** is displayed if the command is entered without an argument:

```
PPR
```

The Input Command

The format of the Input command is

```
     PIN text
```

Additional features of the Input command are described in Chapter 5.

This command displays a line of output on your screen, just as the Print command does. But Input also causes the program to stop and wait for you to press RETURN. Here's an example:

Example: getting your program to stop temporarily.

```
HALT PND
     PPR [G]
     PPR ===== Insert special forms in printer =====
     PIN           Then press RETURN
     ...
```

In the first line, PND, the No Display command, allocates the entire screen to messages rather than to text. The first Print command rings the bell to get the user's attention. ([G] causes the bell to ring.) The second Print command displays a message line on the screen. The Input command displays another message line and causes the program to stop and wait. When the user signals by pressing RETURN that the proper paper has been loaded, the program continues processing.

Try the HALT routine now to see how it works.

- Type the four program statements using Apple Writer. You must press [V] before and after [G] (to enter and leave Control Character Insertion mode). See the section "Clearing the Screen," below, for more about [V].
- Save the program in a file—**do not** save it on the MASTER disk.
- Run it using the Do command.

You'll see the two messages displayed on the screen. Now press RETURN and watch the messages disappear. (The screen now contains whatever was there before you ran the HALT program. Whenever a WPL program ends, the text buffer display is restored to the screen.)

Here's another example of a program that uses the Print and Input commands. The section shown below displays a logo for the program LETTER.

Example: logo display

```
LETTER PND
       PPR [\]
       PPR
       PPR
       PPR ************************************************
       PPR *                                              *
       PPR *            WIDGET INDUSTRIES, INC.           *
       PPR *             Form Letter Generator            *
       PPR *                                              *
       PPR ************************************************
       PPR
       PIN              Press return to begin...
       ...
```

The statement PPR [\] (which clears the screen) is explained below.

The Input command, in addition to stopping and starting a WPL program, can be used to receive information from the user and deliver it to the program during execution. You will learn how to do this in Chapter 5.

Controlling the Screen Display

Apple Writer usually reserves only one line on the screen for output messages from a program. That's because the rest of the screen is kept for display of the document in memory. But you can control the use of the screen by using the WPL commands ND (No Display) and YD (Yes Display) to turn the document display off and on.

The No Display Command

Your program will run up to five times faster when the document in memory is not displayed.

The ND (No Display) command turns off the text buffer display so that the entire screen may be used for message output. No Display is usually the first command in a WPL program.

The format of the command, which does not take an argument, is

```
PND
```

If you issue a series of Print commands in your program and forget to turn off the text display with ND, the output messages will be displayed one at a time in the one-line space provided for them...at computer speed. All you'll see is a fast flicker. Therefore if you want to display several lines of information, such as a menu, you **must** use the No Display command. You must also put an Input command (IN) at the end of the lines of information so that they continue to be displayed until the user presses RETURN.

The Yes Display Command

The YD (Yes Display) command turns the text buffer display back on so that you can display the document your WPL program is working on. This can be helpful while you're developing a program because it lets you see the effect the program statements are having on the text.

Document display is the state that Apple Writer is in unless you tell it otherwise. The format of the command, which does not take an argument, is

```
        PYD
```

Clearing the Screen

Before you load a file, you normally clear memory. Similarly, before sending messages from a program to the screen, it is customary to erase the screen.

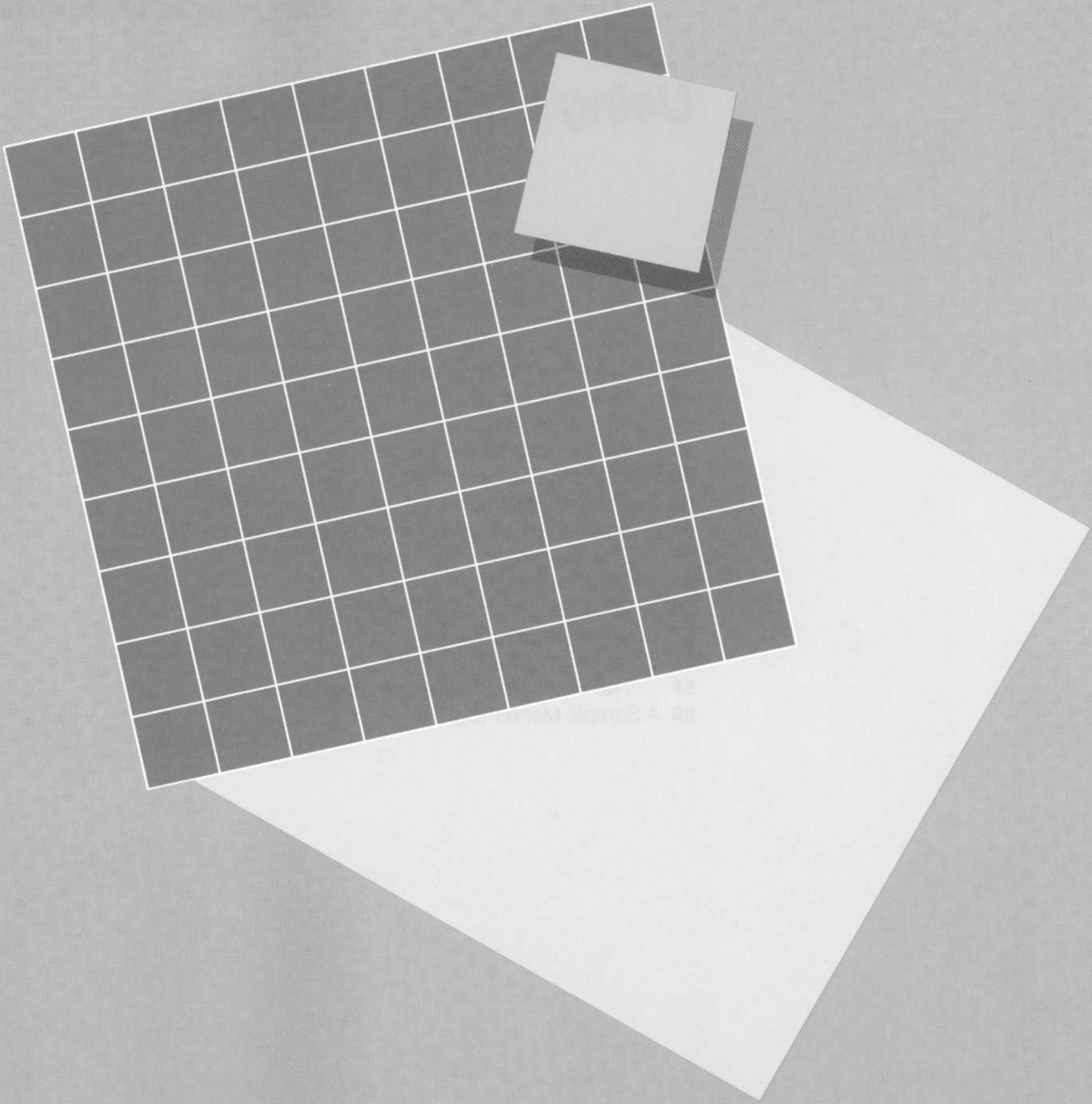
The Print command that clears the screen is shown in this manual as follows:

```
        PPR [\]
```

Inserting control characters in a WPL program

This means "Enter the command by typing PPR and then pressing [V] [\] [V]." The first [V] tells Apple Writer to enter Control Character Insertion mode so that [\] or any other control character is entered as text. (The [\] is a clear screen character.) The second [V] causes Apple Writer to exit from Control Character Insertion mode and enter Text mode again. When the program is run, Apple Writer executes the statement PPR [\] by clearing the screen.

Using String Variables



Using String Variables

This chapter explains the terms *string* and *variable*. It shows you how to use string variables to

- create and manipulate strings of text;
- compare a string to a constant or to another string;
- use strings in place of constants in Apple Writer commands;
- modify a program at execution time by using string variables;
- write a menu program.

Introduction to String Variables

In this section you will learn what a string variable is and why it is such a powerful programming tool.

What Is a String?

A string is text.

Simply stated, a string is text. Each word in this sentence may be thought of as a string—a sequence of characters strung together. In WPL, a string may contain up to 64 letters, numbers, and any special characters you can enter from your keyboard, including control characters and space characters. Each line below is an example of a valid string:

```
A
.dl/filename
***************************************
[G]
#$a!*&
Cupertino, CA 95014
10
Fourscore and seven years ago
```

What Is a Constant?

A constant is a string that always has the same value. Its value does not change while the program is running. All of the strings shown in the previous section are constants.

What Is a Variable?

In algebra, a variable is a letter or symbol that represents an unknown number. Sometimes the letter represents a specific number, as in the expression

$x = 10 + 2$

Other times the letter may represent an infinite number of possibilities, as in the expression

$x = 10 + y$

In this expression, x is known only when we know what y is.

Both of these uses of variables are found in WPL. That is, a variable may represent a specific value such as "YES" or "10", or it may represent a vast number of possibilities such as "any file name" or "any number from 1 to 1,000". Your program can control how a given variable is used—more about this in the next section.

String variables versus numeric variables

There are two kinds of variables in WPL: string variables and numeric variables. String variables, which represent text, are described in this chapter. Numeric variables, which represent numbers, are described in Chapter 6. The difference between numbers in string variables and numbers in numeric variables is that you can do arithmetic only with a numeric variable.

String Variables in WPL

WPL uses four symbols for string variables: `$A`, `$B`, `$C`, and `$D`. These may be entered in either upper- or lowercase. `$A` and `$a` are the same variable. Each variable may be used over and over again to represent different values of text during the course of a program. Whenever a variable name appears in a program, Apple Writer substitutes the current value of the variable.

You can think of a variable as a bucket that holds whatever you put in it. (See Figure 5-1.) The contents of the bucket will be there until you put something new in the bucket. When you do, the old contents are automatically emptied out before the new contents are put in.

Your program can decide whether a given variable may contain any value at all or whether it must contain a specific value or set of values. In the APPEND2 program, which appears later in this chapter, all of the variables may contain any value—the program doesn't check them. In the CHOICE program (also later in this chapter), the $a variable must contain the answer "YES" or "NO"; otherwise the program asks the question again.

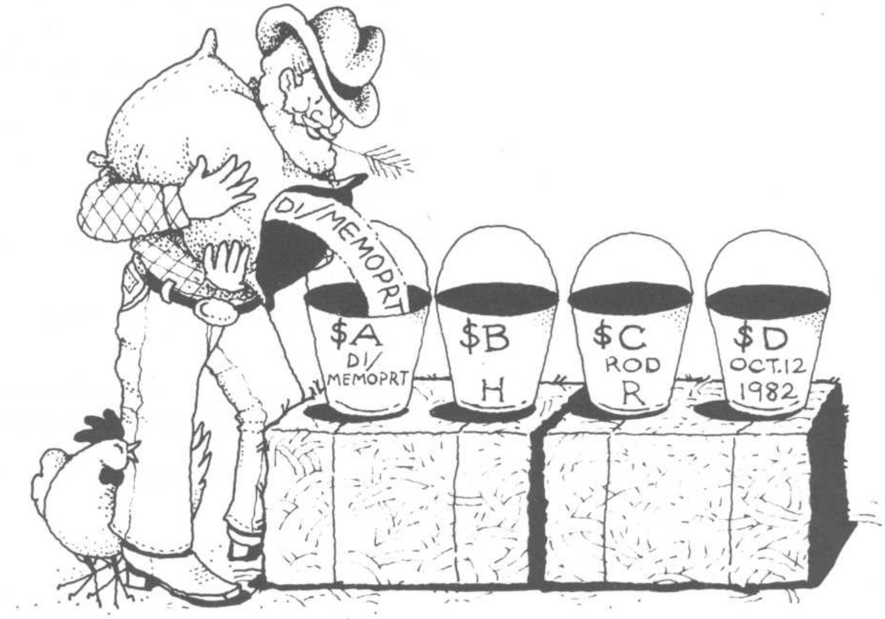


Figure 5-1. A String Variable Bucket

A string variable may be used in any Apple Writer or WPL command instead of text. Here are some examples of string variables used in statements:

1. `ppr The new file for month $a is $b.`
2. `L $D/$A`
3. `f/$c/$b/a`

In example 1, the $a variable represents the name of a month and the $b variable represents a filename. In example 2, the $A variable represents a filename and the $D variable represents a drive designation. In example 3, the $c variable represents the string to be replaced and the $b variable represents the replacement string.

The WPL program that contains these statements assigns each variable its value by means of the Input command, the Assign String command, or the Load String command. These commands are described in the next section. Here is how the three statements might look if Apple Writer replaced the variable names with the current values of the variables:

1. `ppr The new file for month JULY is JUL23`
2. `L .D1/STOCK`
3. `f/nuts/bolts/a`

Setting a String Variable

A string variable keeps its value until its bucket is filled with a different value. Filling the bucket is also called setting the variable. You may set a string variable by

- typing its value while the program is running (in response to the Input command);
- assigning a specific value to it in your program (with the Assign String command);
- loading text from a file into the variable (with the Load String command).

Additional Features of the INPUT Command

The format you learned for the Input command is

```
     PIN text
```

In this format, `text` is displayed and the program halts until you press RETURN. (Remember that the string includes the space before text.) The expanded format of the Input command is

```
     PIN text = $A
```

where `$A` may be any string variable. In the expanded format, `text` is displayed but `= $A` is not displayed. When the program halts, whatever you type before pressing RETURN, up to 64 characters, will be stored in `$A` (that is, in any string variable named in the Input statement).

You can try out a simple program right now that uses the two formats of the Input command. Just type in the following five-line program, [S]ave it in a file, and execute it.

Example: pick a number

```
pick pnd
     ppr [\]
     pin Pick a number from one to ten, then press RETURN. = $A
     pin The number you picked is $A; press RETURN.
     pqt
```

Whatever you type is put in the $A bucket by the first Input statement and displayed by the second Input statement. You can type three or 3 or antidisestablishmentarianism and the program echoes it. Later in this chapter you'll learn how to use the Compare Strings command to find out what was typed and take different paths within the program depending on what's in the bucket.

Note: Variables give you a way to generalize a program. For instance, the APPEND program in Chapter 3 consolidates 3 **specific** input files (JANUARY, FEBRUARY, and MARCH) into a **specific** output file. By substituting string variables for the file names, you can make a program that works with **any** files you specify at execution time:

```
APPEND2 P THIS PGM MAKES ANY THREE FILES INTO ONE
        NY
        PIN WHAT'S THE FIRST FILE? = $a
        L $a
        PIN WHAT'S THE 2ND FILE? = $b
        L $b
        PIN WHAT'S THE 3RD FILE? = $c
        L $c
        PIN WHAT'S THE OUTPUT FILE? = $d
        S $d
```

All of the Apple Writer menus are written in WPL using the Input command. By the end of this chapter you'll be able to write your own menu program.

The Assign String Command

You can fill a string variable bucket from outside the program by using the Input command or from inside the program by using the Assign String command. The Assign String command allows your program to fill any string variable with text.

The format of the Assign String command is

```
PAS text = $A
```

where `$A` represents any string variable and `text` is the value to be assigned to the string variable. The spaces before and after the text are also assigned to the string variable. The variable to the right of the equals sign represents the bucket to be filled. The following are legitimate Assign String statements:

```
pas Thank you for your letter dated = $A
PAS International Industries Inc. = $C
pas .D2/ = $d
```

Note: Variables may appear on both sides of the equals sign so that you can give a string variable the value of any string variable(s) plus text. We'll tell you more about this in the next section.

Concatenation

Combining two or more strings is called concatenation. When concatenating strings, the total length of the new string (the variable on the right of the equals sign) may not be more than 64 characters. Three examples of concatenation are shown below.

Example: putting brackets around a string

In the **first example**, two text strings are concatenated with a string variable (`$a`). The first text string is a blank space and a left bracket (`[`) and the second text string is a right bracket (`]`). The result is placed in the same string variable; the previous contents of `$a` are destroyed.

```
pas [$a] = $a
```

If the value of `$a` is `APPLE` before the Assign String statement is executed, then the value of `$a` after execution is `[APPLE]`. The space between the command and the left bracket is part of the argument.

Example: putting strings together

In the **second example**, three string variables ($D, $C, and $A) are concatenated and placed in $B:

```
PAS$A$D$C = $B
```

If the value of $A is .D1 and the value of $D is WPL. and the value of $C is MEMOPRT then the value of $B after execution is .D1/WPL.MEMOPRT.

Example: adding strings that contain spaces

In the **third example**, text including space characters is concatenated with two string variables; $b represents a first name, $c represents a last name, and $a contains the result after execution:

```
pas Mr. $b $c = $a
```

If the value of $b is John and the value of $c is Doe, then after execution of this Assign String statement, $a will contain the value Mr. John Doe (including the space between pas and Mr.).

Concatenation is a useful way to store the contents of two or more buckets in a single bucket. For instance, here's a routine that asks for three pieces of information and saves them in a single variable for use later in the program:

Example: collecting parts of an address

```
pnd
...
pin What's your city?       = $a
pin What's your state?      = $b
pin What's your zip code?   = $c
pas $a, $b $c = $a
ppr Your mail will be sent to$a.
...
```

> **Notice:** There is a space between the Assign String command and $a in the second-to-last statement; this space is actually part of the argument of the Assign String command. When text is substituted for $a in the last statement, it will begin with a space.
>
> That's why there's no space between to and $a in the last statement. If you had a space in the text and a space in the variable, you would get two spaces between to and the address when the substitution was made. Here is another way to write the last two statements of the routine:
>
> ```
> pas$a, $b $c = $a
> ppr Your mail will be sent to $a.
> ```

After this routine has been executed, $b still contains the state and $c still contains the zip code, but you can now reuse $b and $c by filling them with new values since $a contains the city, state, and zip code formatted with commas and spaces.

The Load String Command

Load String lets you set a string variable from a file. (Input lets you set a string variable from the keyboard. Assign String lets you set a string variable from within the program.) The format of the Load String command is

```
        PLS .d2/filename!start!end!AN =  $A
```

where `start` identifies the beginning of the string to be loaded and `end` identifies the end of the string to be loaded. `$A` represents the string variable being loaded. The Load String command has no effect on the document in memory.

Load String works exactly like [L]oad except that [L]oad places text in the text buffer and Load String places up to 64 characters of text in a string variable. As in [L]oad, you may specify `A` or `N` or both. `A` means load all occurrences of the string. `N` means don't include the start and end markers in the string variable.

Let's look at an example of how this command works. You might want to type in the small program and data file so you can try this out for yourself. The data file, `NAMES`, looks like this:

```
Brown,Mary
Jones,Lee
Kitchen,Chris
Sunn,David
```

The program prompts for a last name, uses the Load String command to get the first name, and displays first and last name on the screen:

Example: finding names in a file

```
GETNAME PND
        PIN Enter last name            = $a
        PLS .d2/NAMES<$a, < ><n = $b
        PIN $a's first name is $b
        PQT
```

As you can see, the Load String command is handy for looking up values in tables and lists. Notice the use of special delimiter characters. The first marker in the Load String statement is a concatenation of a string variable and text (the comma). The second marker is the return character (>).

If the first marker in the Load String command is not found, the next statement in the program is skipped. This is called *conditional execution*—it's one of WPL's most powerful features; you will learn all about it in the next section of this chapter.

Conditional Execution

There are times when you can't write a precise program command because you don't know exactly what the conditions will be when it's executed. For example, you may want to tell Apple Writer to put a message on the screen only if the previous [F]ind found what it was looking for. Since you can't know in advance whether or not the text will be found, you have to provide for both possibilities in your program. If the text is found, Apple Writer executes the next statement; if the text isn't found, the next statement is skipped. This is called conditional execution because a statement is either executed or skipped depending on the outcome of the previous statement. We refer to the condition that causes the next statement to be skipped as the *unsuccessful outcome*.

Figure 5-2. Conditional Execution

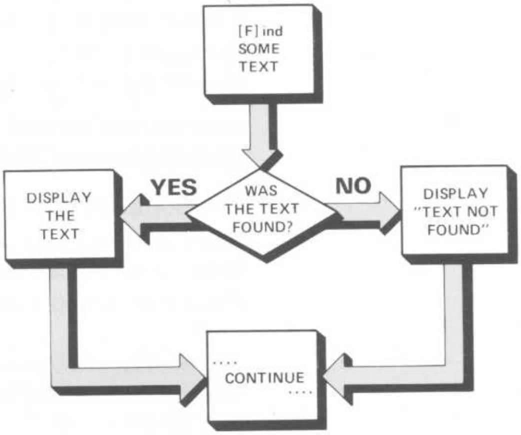


Figure 5-2 shows the logic behind conditional execution. The same picture can be drawn with words:

FIND SOME TEXT
DISPLAY THE TEXT
GO TO NEXT SEARCH

The second statement, DISPLAY THE TEXT, is executed only if the text is found. The third statement, GO TO NEXT SEARCH, is always executed. If the text is not found, the second statement is skipped and Apple Writer goes directly to the third statement.

Conditional execution applies to both WPL commands and Apple Writer commands. As you learn the WPL commands, you will be told which ones cause conditional execution. (You learned earlier in this chapter that the Load String command causes conditional execution.)

Here are the Apple Writer commands that cause conditional execution when written in a WPL program:

Command	**Next statement skipped if...**
[F]ind	Text not found
[L]oad with markers	First marker not found

Comparing Strings

In the GETNAME program earlier in this chapter, a variable is used as a marker in a Load String command. In this particular case, the program doesn't have to know the current value of the variable. Sometimes, however, a program needs to make a choice that depends on the value of the variable. In a menu program, for instance, the keyboard input determines which menu item is selected. The Compare Strings command tests a variable to see if it contains a particular string—for example, a particular menu selection.

The Compare Strings Command

The Compare Strings command determines whether two strings are equal, and causes conditional execution. If the strings are equal, the next statement is executed. If the strings are not equal, the next statement is skipped.

By the Way: Two strings are equal if they are the same length and if both strings contain the same characters in the same order. The following strings are equal:

```
BA135C       =       BA135C
```

The following strings are not equal:

```
BA135C     not =     BA135CD
BA13       not =     B13A
BA 135C    not =     BA135C
```

The format of the Compare Strings command is

```
PCS /string1/string2/
```

String1 and string2 may consist of text, one or more string variables, or a combination of these.

The delimiters for the Compare Strings command are similar to the delimiters for the [F]ind command: the first non-space character typed after the command is the delimiter.

The following are valid Compare Strings statements:

```
pcs/$a/$b/
PCS .YES.$C.
pcs /$d$b/July 1983/
PCS !Ms. $A!$B!
```

The following routine demonstrates the use of string variables and conditional execution in comparing strings:

Example: comparing YES and NO

```
       pnd
       ...
choice ppr Do you want to print another file?
       pin Enter YES or NO; then press RETURN. =  $a
       pcs /$a/YES/
       pgo print
       pcs /$a/NO/
       pgo quit
       ppr You didn't answer the question!
       pgo choice
       ...
```

After each Compare Strings command, the next statement will be executed only if the comparison is equal. Because the next statement in each case is a Go command, the statements after the Go command will be executed only if the strings are not equal. It is also possible to have two Go commands in a row; compare the following routine to the routine above:

Example: print or quit

```
...
pin To print another file type Y, then press RETURN. = $a
pcs /$a/Y/
pgo print
pgo quit
...
```

In this version, any response other than Y will cause the program to execute the quit routine. "Y" and "y" are not the same! How would you rewrite this routine so that the user could type either an uppercase or lowercase response? (See Appendix E for the answer.)

A Sample Menu Program

It's often helpful to print a document on the screen in order to review it before it goes on paper. In order to print on the screen, you must change the print destination setting using the Apple Writer [P]rint/Program command. The MENU program determines whether you want to print on the screen or on the printer; it then changes the print destination setting automatically and prints the file.

Example: a menu program

```
menu      pnd
          ppr [\]
          ppr PRINT OPTIONS MENU:
          ppr
          ppr    (1) Screen
          ppr    (2) Printer
          ppr    (3) Quit
          ppr
select    pin Select 1, 2, or 3:       = $a
          pcs /$a/3/                    (Compare $a to 3 )
          pgo quit                      (Equal: Quit routine )
          pcs /$a/2/                    (Not Equal: Compare to 2 )
          pgo printer                   (Equal: Printer routine )
          pcs /$a/1/                    (Not Equal: Compare to 1 )
          pgo screen                    (Equal: Screen routine )
          pgo select                    (Not Equal: Select routine)
screen    ppd.console
          pgo file
printer   ppd.printer
          pgo file
quit      pqt
file      pin Enter file name:         = $c
          ny
          L $c
          pnp
          pin Press RETURN.
          pgo menu
```

To change several print values, make a print value file.

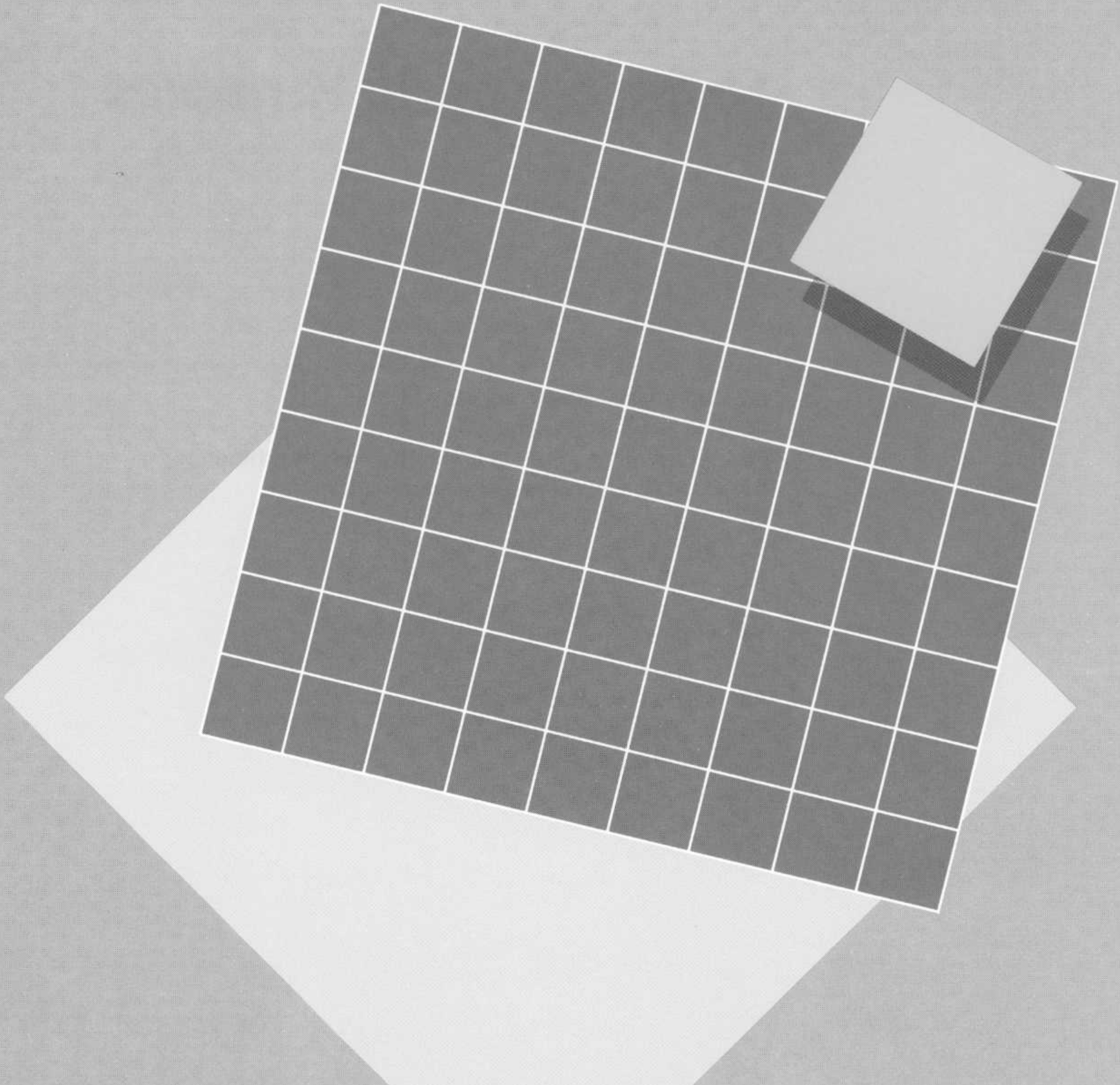
You may want to change not only the print destination but other print values. One way to do this in Apple Writer is to create a print value file. Can you figure out how to modify the MENU program to load a print value file for each option on the menu? Hint—you will need a statement that looks something like this:

```
qCprintvalfile
```

Remember that you must also create print value files with the names your program will be looking for. Can you write a menu program that creates a print value file? See Appendix E for the solution to these programming puzzles.

Chapter 6

Using Numeric Variables



Using Numeric Variables

In this chapter you will learn

- how to convert a number in string form to a numeric variable;
- how to add and subtract in WPL;
- how to use counters and accumulators in a program;
- how to use numeric variables to control a loop;
- how to create and number an address file;
- how to produce personalized form letters.

What Is a Numeric Variable?

Like string variables, numeric variables can be set from the keyboard or from within a program. Unlike string variables, you can do arithmetic with numeric variables. The three WPL numeric variables, `(X)`, `(Y)`, and `(Z)`, may be substituted for text in WPL and Apple Writer commands. There is also a special set of commands associated with them.

A numeric variable has the following characteristics:

- It may represent zero or any positive integer from 1 through 65,535.
- The variable name may be either upper- or lowercase and is always enclosed in parentheses: `(X)`, `(Y)`, or `(Z)`.
- If the value of the variable is increased or decreased so that it becomes zero, conditional execution causes the next statement to be skipped. (In WPL's arithmetic system, 65,535 + 1 = 0. This is known as *overflow*.)
- Setting the variable to zero (as opposed to increasing or decreasing it to zero) does not cause the next statement to be skipped.

The Set X Command

Adding and subtracting with variables

The command that performs arithmetic manipulation of numeric variables is called Set X. (There are actually three commands—Set X, Set Y, and Set Z—one for each variable. They function identically.) The format of the Set X command is

```
PSX number
```

where number may be signed or unsigned. When the number is **unsigned** as in

```
PSX 35
```

the variable is given the value of number—in this case, `(X)` is set to `35`. When the number is **signed** as in

```
PSZ + 10
psy - 200
```

the variable is modified by the value of number according to the direction of the sign—in this case, 10 is added to the current value of `(Z)` and 200 is subtracted from the current value of `(Y)`.

Converting Strings and Performing Arithmetic

The `AGE` program, following, shows how numeric variables are used in WPL. This program figures out your age from information you type in response to Input statements. The information is entered as strings, converted to numeric form, used in arithmetic, and displayed as output text. In the `AGE` program, your birth year and the current year are converted to the variables `(X)` and `(Y)` respectively. `(X)` is then subtracted from `(Y)` to approximate your age. If you have not had a birthday in the current year, 1 is subtracted from the answer.

Example: how old are you?

```
AGE PND
    PPR [\]
    PIN ENTER YOUR BIRTH YEAR                          = $a
    PIN ENTER THE CURRENT YEAR                         = $b
    PIN HAD A BIRTHDAY YET THIS YEAR (Y or N)?         = $c
    PSX $a
    PSY $b
    PSY -(X)
    PCS /$c/N/
    PSY -1
    PIN YOUR AGE RIGHT NOW IS (Y)!
```

The statement

```
        PSX $a
```

takes the value of the $a string and converts it to a numeric variable, (X). As you see in the final statement, a numeric variable may also be used as text.

Using Counters and Accumulators

One advantage of WPL is that it allows you to perform repetitive Apple Writer functions. Sometimes these functions need to be performed a given number of times. Numeric variables allow you to control the number of times a routine or loop is performed. They also serve as accumulators for numeric values collected during the loop.

Let's say that each year on your birthday your company will give you a gift of stock, one share for each year of your life. You can write a WPL program that calculates the total number of shares you'll receive in the next five years. In fact, since the AGE program collects all the necessary data, your stock calculation can be added to the end of that program. Here's what the additional statements would look like:

Example: cumulative addition, counting down

```
        . . .
CALC P    Calculate Total Stock Over 5 Years
     PSZ 5
     PSX 0
LOOP PSY +1
     PSX +(Y)
     PSZ -1
     PGO LOOP
     PIN In 5 years you will have (X) shares of stock
```

The CALC routine uses the (Z) variable to control the number of times the calculation is performed—once for each year, or five times in all. Taking advantage of the conditional execution that occurs when a variable is decreased to 0, CALC begins by setting (Z) to 5. Then it subtracts 1 from (Z) each time the LOOP section is executed. When (Z) becomes 0, the Go command at the bottom of the loop is skipped and the answer is displayed.

The (X) variable contained the birth year in the first part of the program. CALC doesn't need that information anymore, so it reuses (X) as the accumulator—the bucket where it adds each year's number of shares. CALC initially sets (X) to 0 in order to clear out the previous information.

Warning
Always give numeric variables an initial value before adding or subtracting. You can provide an initial value by issuing a Set X command or by converting a string variable to a numeric variable. (Setting a numeric variable to 0 doesn't cause the next statement to be skipped.)

The (Y) variable already contains your current age, which will be increased by 1 each time the program executes the loop. Because you want to start counting with your next birthday, CALC must add 1 to the (Y) variable before doing the stock accumulation. (If you wanted to start counting with the current year, where in the loop would you increase the variable? See Appendix E for the answer.)

Comparing Numeric Variables

You can compare a numeric variable to another numeric variable by using the Compare Strings command. You can also compare a numeric variable to a numeric constant or to a string variable, provided the constant or string consists of an integer between 0 and 65,535. First, though, you must convert the numeric variables to string format using the Assign String command.

Comparing numeric variables gives you a way to end a loop without decreasing a numeric variable until it reaches 0. Let's change the CALC five-year stock calculation routine so that it uses this alternative. (Remember that this routine is an addition to the AGE program which is listed earlier in the chapter.) Here's how NEWCALC looks as it performs the calculation for any number of years:

Example: cumulative addition, counting up

```
        . . .
NEWCALC P Calculate Total Stock Over N Years
        PIN HOW MANY YEARS WILL YOU RECEIVE STOCK? = $a
        PSZ 0
        PSX 0
LOOP    PSY +1
        PSX +(Y)
        PSZ +1
        PAS(Z) = $b
        PCS /$b/$a/
        PGO END
        PGO LOOP
END     PIN In $a years you will have (X) shares of stock
```

In the original stock calculation routine CALC subtracted 1 from (Z) until the value of (Z) was 0. In the new routine NEWCALC begins with 0 and adds 1 until the value of (Z) is equal to the number of years that was input in $a. In order to make the comparison, NEWCALC must convert (Z) to a string variable, $b, which is then compared to the original input value in $a. As long as the two strings are not equal, the PGO END statement is skipped and the program returns to LOOP. When the strings are equal, the loop is exited and the final answer is displayed.

When NEWCALC converts (Z) to $b, the contents of the (Z) bucket are not changed. When the program is executed, Apple Writer just looks at the value of the numeric variable and sets the string variable to that same value.

Creating Form Letters With WPL

You now know all the commands needed to create your own personalized letters. This section contains two sample letter-writing programs. The first program, NUMBER, numbers an address file so that it may be used for form letter input. The second, WRITE, uses the numbered address file and a form letter file to write personalized letters. To begin, let's look at the form letter itself.

A Sample Form Letter

Here's what a very simple form letter looks like. Your own letter, of course, might be several pages long.

```
(Date)

(Address)

Dear (Name):

Your back-ordered merchandise has been received and will be
shipped today by parcel post. Thank you for your order.

Sincerely,

Catalog Sales Distributors, Inc.
```

The date, address, and name are the portions of the form letter that will be customized. You type the date; the address and name are contained in an address file. The WRITE program assumes that this letter is stored in a file called LETTER on the disk in drive 2.

Creating an Address File

The address file, OLDAD, consists of a name and address for each customer. The file is named OLDAD because that's what it's called in the NUMBER program later in this section. An address may contain any number of lines. Each line ends with RETURN.

Marking the beginning of a name

Because WPL has to be able to identify where each address begins, we use the convention of beginning the name line with a pair of angle brackets. A left angle bracket marks the end of the file. The file looks like this:

```
<>Charles Gee
34B Sansome Street
San Francisco, CA 94111
<>Marilyn Bee
12570 Pacific Blvd.
Santa Monica, CA 90002
<>Serendi P. Tee
RFD 3 Box 12
High Point, OR 97567
<
```

There are two steps in creating an address file: (1) type the addresses, and (2) number the addresses. The form letters will be printed in whatever order the addresses are stored in the file.

Note: You don't have to arrange the addresses in any special order such as by zip code or last name , but adding new addresses and finding addresses for corrections is easier if the file is in some order.

Once the address file has been created, the addresses must be consecutively numbered because the WRITE program uses the numbers to step its way through the file. If you add an address in the middle of the file, all the addresses that follow the new one must be renumbered. The NUMBER program saves you from this tedious clerical effort by doing the job automatically.

The NUMBER program, shown below, assigns consecutive numbers to the address file, OLDAD. Each time it finds a <> marker, it inserts the next consecutive number between the angle brackets. The (x) variable is used for numbering the addresses. When all the addresses have been numbered, the numbered address file is saved. The numbered file is given a new name, NEWAD. (That way if your computer loses power during the save operation, you won't lose the original file.)

Example: numbering a list

```
number psx 1
       ny
       L .d2/oldad
       b
loop   f/<>/<(x)>/
       y?
       pgo found
       pgo quit
found  psx +1
       pgo loop
quit   s .d2/newad
```

As the NUMBER program runs, you can watch the numbering process on the screen. If you want to speed up the program, use the No Display command at the beginning of the program to eliminate the screen output.

After the NUMBER program runs, the NEWAD file looks like this:

```
<1>Charles Gee
34B Sansome Street
San Francisco, CA 94111
<2>Marilyn Bee
12570 Pacific Blvd.
Santa Monica, CA 90002
<3>Serendi P. Tee
RFD 3 Box 12
High Point, OR 97567
<
```

A Form Letter Program

The WRITE program creates a form letter for each address in the NEWAD file created by the NUMBER program. A section-by-section explanation of the WRITE program follows; first, here's the entire program:

Example: form letter processing

```
Write  pnd
       ppr [\]
       psx 1
       ppr ***** FORM LETTER PROCESSOR *****
       pin Enter current date:    = $a
Loop   ny
       L .d2/letter
       b
       f!(Date)!$a!
       y?
       f/(Address)//
       y?
       L .d2/newad!< (x) >!< !n
       pgo Name
       pgo Quit
Name   f/(Name)//
       y?
       L .d2/newad!<(x)>! !n
       pnp
       psx +1
       pgo Loop
Quit   ppr [g][g][g]
       psx -1
       pin The number of letters printed was (x).
```

As you read the detailed description of the WRITE program, which follows, study the WPL statements that correspond to the functions described. You will find a syntax summary and examples for each command in Appendix C.

The Write *Section*

Here's the Write section of the WRITE program:

```
Write  pnd
       ppr [\]
       psx 1
       ppr ***** FORM LETTER PROCESSOR *****
       pin Enter current date:    = $a
```

The program begins by turning off text display, clearing the screen, setting the address file counter to 1, displaying the program title on the screen, and prompting for the current date, which is used in every letter. These functions are performed only once; therefore they are not included in the loop.

The Loop **Section**

Here's the Loop section of the WRITE program:

```
Loop   ny
       L .d2/letter
       b
       f!(Date)!$a!
       y?
       f/(Address)//
       y?
       L .d2/newad!<(x)>!<!n
       pgo Name
       pgo Quit
```

The processing loop extends from the beginning of the2 Loop section to the end of the Name section. The text buffer is cleared and a fresh copy of the form letter is brought into memory. The cursor is placed at the beginning of the letter. The first [F]ind command inserts the current date. The second [F]ind command places the cursor at the (Address) tag and deletes the tag.

A Special Note: You might be accustomed to using / as a delimiter, but there are times when it cannot be used. For instance, if you are loading segments of a file, you cannot use the / as your delimiter, since it is an integral part of the file name (i.e., .d2/). You also don't want to use / as a delimiter when it might appear elsewhere in the argument. For instance, the WRITE program uses ! delimiters instead of the usual / delimiters in the [F]ind command for the date because you may want to use the / character as part of a date.

Alternate delimiters are necessary because of the way Apple Writer executes a command that contains a variable: first the variable name is replaced by the current value of its bucket, then Apple Writer executes the command. If the bucket happens to contain characters that match the delimiter, you won't get the results you expect. These two examples show how a poor choice of delimiters can get you into trouble:

1. Command: `f/(Date)/$a/`
   Current Value of $a: `9/17/77`
   The Command as Executed: `f/(Date)/9/17/77/`
   What Happens: `(Date)` is replaced by `9`—the rest of the command is ignored.

2. Command: `F!$B!`
   Current Value of $B: `DECEMBER!!`
   The Command as Executed: `F!DECEMBER!!!`
   What Happens: `DECEMBER` is deleted.

To avoid problems, don't choose a delimiter that might appear elsewhere in the argument. Table 6-1 shows you what delimiters you may use in specific situations.

Table 6-1. Table of Delimiters

Delimiter	Any Length	Carriage Return	Any Character (Wildcard)
/	none	none	none
!	none	none	none
<	=	<	?
#	$	%	&
&	'	(	)
*	+	'	-

The `(x)` variable is used as an index to the address file. `(x)` is set to `1` in the `Write` section and increased by 1 in the `Name` section. The second [`L`]oad command searches the address file for the current address marker and loads the address, not including markers, at the current cursor position. This form of [`L`]oad causes conditional execution because [`L`]oad may or may not find the text it's looking for: if the address marker is found, execution continues at the `Name` label; if not found, execution continues at the `Quit` label.

The Name Section

Here's the Name section of the WRITE program:

```
Name  f/(Name)//
      y?
      L .d2/newad!<(x)>! !n
      pnp
      psx +1
      pgo Loop
```

The [F]ind command places the cursor at the (Name) tag and deletes the tag. Then the same address that was loaded in the Loop section is loaded again. This time, however, the ending marker is a space character, so only the first name of the address is inserted into the document in memory. The program assumes that the [L]oad operation is successful—if the address was found in the Loop section, it will certainly be found again.

The form letter is now complete, so it is printed. The program adds 1 to the address file index and goes to the top of the loop.

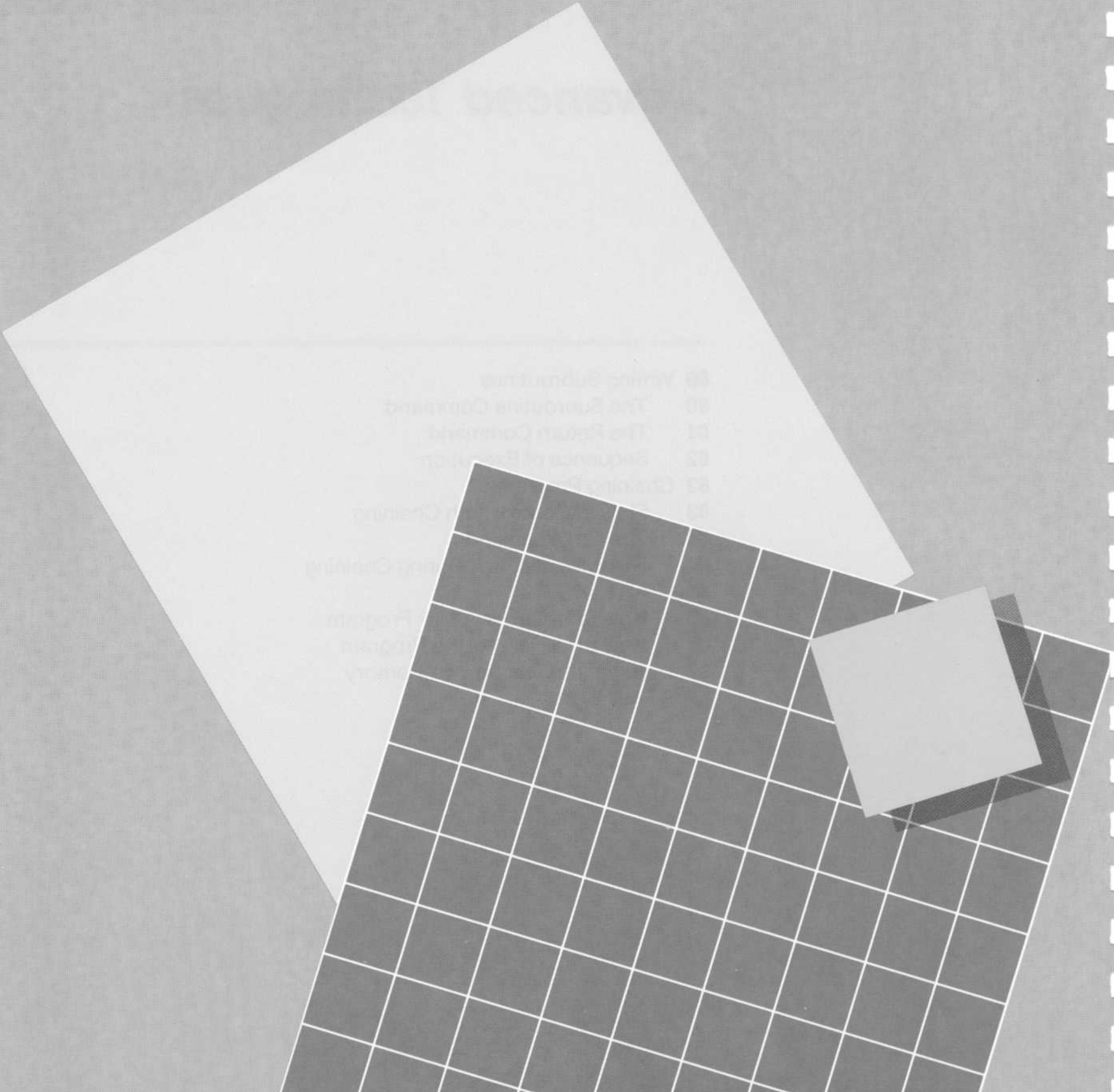
The Quit Section

Here's the Quit section of the WRITE program:

```
Quit  ppr [g][g][g]
      psx -1
      pin The number of letters printed was (x).
```

This section is executed when the value of (x) is one higher than the number of the highest index in the address file. That signals the end of the job. The Quit section rings the bell three times to notify the operator that the printing is complete. Then, in order to display the correct number of letters, 1 is subtracted from (x). Finally, a message is displayed.

Advanced Techniques



Advanced Techniques

This chapter covers four advanced techniques: subroutines, chaining, STARTUP, and loading the catalog into memory. It is possible to make effective use of WPL without ever needing these techniques. You will find that subroutines and chaining are of value, however, if you plan to write programs that are very large or very complex. You will find STARTUP useful if you perform the same commands or run the same program every time you boot Apple Writer. You will find loading the catalog useful if you want to print it or save a copy of it on disk.

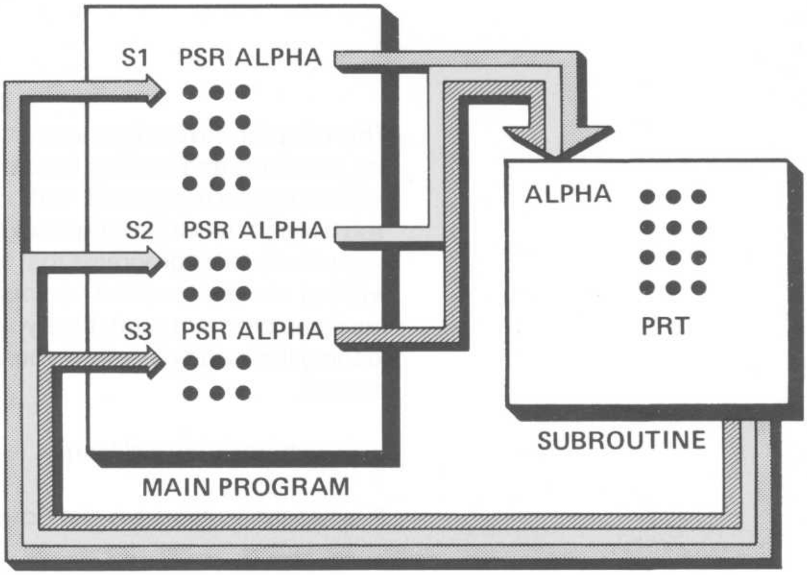
In this chapter you will learn

- what subroutines are and how to write them;
- how to use subroutines to save memory;
- how to write WPL programs larger than the 2,048-character limit;
- how to add a STARTUP program to Apple Writer;
- how to put a copy of the catalog into memory.

Writing Subroutines

In a large program, identical sets of statements are often needed in more than one place. Subroutines provide a means of writing these statements in one location and accessing them from anywhere in the program. Figure 7-1 shows the flow of control from several places in a program to a subroutine and back again.

Figure 7-1. Subroutine Flow of Control.



A subroutine is self-contained: it has only one beginning and one end. It is part of a WPL program but functions as if it were a separate program. Go commands are allowed within a subroutine but may not refer to a label outside the subroutine.

The Subroutine Command

The first statement of a subroutine is a labeled statement. The label is the name of the subroutine. The Subroutine command says, "Go to the labeled statement and begin executing at that point." In this sense the Subroutine command works just as the Go command does. The difference is that the Go command causes a permanent transfer of control to a different place in the program. The Subroutine command causes a temporary transfer of control; when the subroutine has finished executing, control is automatically returned to the statement following the Subroutine command.

The format of the Subroutine command is

```
PSR label
```

Programmers talk about calling subroutines or calling programs. A call is a transfer of control. The Subroutine command calls the routine named in its argument. A subroutine may not call itself. For instance, a subroutine named SUBX may not contain the following statement:

```
PSR SUBX
```

Later in this chapter you will learn how to use the Do command to call another program.

The Return Command

Every subroutine contains a Return command.

Execution of a subroutine is initiated by a Subroutine command and ended by a Return command. The Return command causes the statement after the Subroutine call to be executed; execution then proceeds sequentially.

The Return command must be the last statement executed in any subroutine. (It can appear anywhere within the subroutine, as long as it is the last statement executed.) There may be more than one Return statement in a subroutine, just as there may be more than one Quit statement in a WPL program.

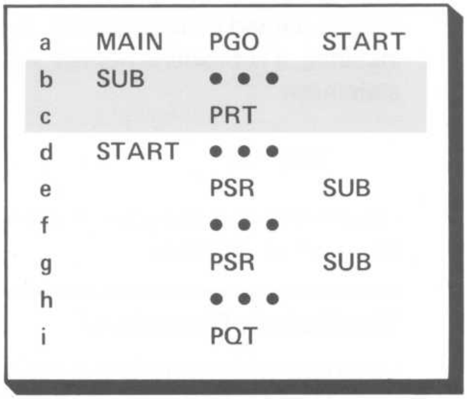
The format of the Return command, which takes no argument, is

```
PRT
```

Sequence of Execution

Figure 7-2 illustrates the order in which statements are executed in a program using a subroutine.

Figure 7-2. Execution Sequence of a Subroutine



```
a   MAIN    PGO   START
b   SUB     • • •
c           PRT
d   START   • • •
e           PSR   SUB
f           • • •
g           PSR   SUB
h           • • •
i           PQT
```

Statements are executed in this order:

a
d
e
b
c
f
g
b
c
h
i

When a subroutine is called, Apple Writer searches for the subroutine label beginning at the first statement. For the sake of efficient execution, subroutines should be placed near the start of the program and the most frequently accessed subroutine should be placed first.

The only valid way to enter a subroutine is by means of a subroutine call; that is, an SR command. You may not Go to a subroutine label or enter a subroutine as the next sequential instruction in your program. If you do, the results are unpredictable. For instance, a program may not begin with a subroutine. (If it did, the Return command would have nowhere to return to.) Use the technique shown in Figure 7-2, where the first statement is a Go command that sends control to the start of the main program. Place all subroutines between the Go command and the main program.

Chaining Programs

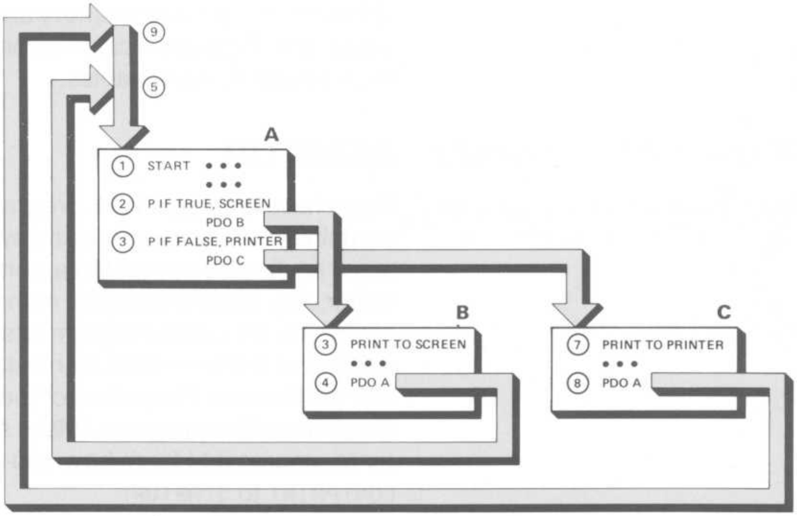
About program length

A WPL program may not be longer than 2,048 characters. Because WPL programs and footnotes share the same buffer, if the program prints a text file containing footnotes then the program is limited to 1,024 characters. Most WPL applications will not be affected by these size restrictions. Occasionally, however, a program exceeds the limit. Chaining is a method of writing such a program by dividing it into two or more smaller programs that are linked together at the time they are run.

Flow of Control With Chaining

Chaining is accomplished when one program calls another using the Do command. Figure 7-3 shows how control flows from one program to another in chaining. A, B, and C are separate WPL programs. Program A can call either B or C. When a program is called, it **replaces** the calling program in the WPL buffer. B and C can return to A only by calling A. In that case A starts over again from the beginning, not from the place where it called B or C. Unlike subroutines, which return to the statement after the Subroutine call, **chaining always begins at the first statement of the called program**.

Figure 7-3. Flow of Control With Chaining.



The Do Command

Chaining is done by issuing a Do command. The command may appear anywhere in the calling program. When it is issued, the called program replaces the calling program in the WPL buffer. Therefore any statements that follow the Do command in the calling program will not be executed. The format of the Do command is

```
    PDO .d2/filename
```

where `filename` is the name of the called program. You are used to using the Do command as an immediate command that executes a WPL program. In chaining, the Do command is used as a deferred command. Exactly the same thing happens in both cases: the file named in the Do command is read into the WPL buffer, and execution begins at the first statement in the buffer.

Variables and Text During Chaining

When one program chains to another, the calling program is no longer available. Variables and most buffers, however, remain unchanged. (The footnote buffer, which is shared with WPL programs, gets cleared when the called program is loaded.) Program `A` can read a document into memory and then call program `B` to modify the document. Program `A` can also prompt for keyboard input and `B` will have access to the variables.

STARTUP

Refer to the Apple Writer manual to learn more about `STARTUP` programs.

If you have created a WPL program named `STARTUP` on the disk that contains your system print and system tab files, `SYS.PRT` and `SYS.TAB`, it will be executed automatically when you boot Apple Writer. Any program may be named or renamed `STARTUP`. If you always run the same program after you boot, name it `STARTUP` so that it is part of the boot itself. For instance, if you always run `CONTPRINT`, use the Rename File option of the SOS Command Menu to change the name of the program. (Make sure the file is unlocked before you try to rename it.) Here's how you would change the name of `CONTPRINT` to `STARTUP`:

1. Press [`O`] to display the SOS Commands Menu.
2. Type `B` to rename a file.
3. Type `CONTPRINT,STARTUP` and press (RETURN).
4. Type `STARTUP` and press (RETURN).

How to Make a STARTUP Program

You may want to write your own STARTUP program to help you start up your system efficiently. This is a good idea if other people also run Apple Writer on your computer. For instance, here's a program that helps the user load the correct glossary file for various tasks without knowing anything about files or glossaries:

Example: a STARTUP program that chooses a glossary

```
startgloss p this is a STARTUP program
           pnd
           ppr[\]
           ppr WELCOME TO APPLE WRITER!
           ppr
           ppr Do you want to run...
           ppr                       a. Text editing?
           ppr                       b. Report printing?
           ppr                       c. Form letters?
          x ppr
           pin Type a, b, or c and then press RETURN. = $a
           pcs/$a/a/
           pgo a
           pcs/$a/b/
           pgo b
           pcs/$a/c/
           pgo c
           pgo x
          a pasEDITGLOSS =  $b
           pgo y
          b pasRPTGLOSS =  $b
           pgo y
          c pasLTRGLOSS =  $b
          y ppr
           ppr Make sure the GLOSSARY disk is in drive 1.
           pin Then press RETURN.
           p load the correct glossary file
           qe.d1/$b
           pqt
```

Type the STARTGLOSS program using Apple Writer and give it the name STARTUP when you save it. You may save it on any disk except your Apple Writer Master Disk. The next section describes how to use a STARTUP program such as this one.

How to Use a `STARTUP` *Program*

The `STARTUP` program may be on the boot disk, or on any other Apple Writer disk. You may use different `STARTUP` programs for different tasks. There is only one restriction: **The STARTUP disk must contain a** `SYS.TAB` file and a `SYS.PRT` file because otherwise you will get a SOS error. You may copy the standard versions of these files from the Apple Writer MASTER disk to your STARTUP disk, or you may create new ones using the Save Tab File and Save Print/Program Value File options of the Additional Functions Menu. To see this menu, press [`Q`]. (For further information about `SYS.TAB` and `SYS.PRT`, see the Apple Writer manual.)

> **Note:** To copy the standard tab file from the Apple Writer MASTER disk to a data disk, follow these instructions:
>
> 1. Put the Apple Writer MASTER disk in drive 1.
> 2. Put the data disk in drive 2.
> 3. Select the Additional Functions Menu by pressing [`Q`].
> 4. To load the system tab file, type
>
>    ```
>    A.D1/SYS
>    ```
>
>   (`A` selects the Load Tab File option; `SYS` is the name of the file.)
> 5. To copy the system tab file, press [`Q`] and type
>
>    ```
>    B.D2/SYS
>    ```
>
>   (`B` selects the Save Tab File option; `SYS` is the name of the file.)
> 6. You can copy the standard print value file in the same way, using the Additional Functions Menu and selecting options C and D.

To use a `STARTUP` program:

1. Put the Apple Writer MASTER disk in drive 1 and boot.
2. When the Apple Writer copyright display appears, put the disk containing the `STARTUP` program and the `SYS.TAB` and `SYS.PRT` files in drive 1.
3. Press (RETURN).

That's all there is to it.

Loading the Catalog Into Memory

In Apple Writer, you can load a disk catalog into memory so that you can print the text, modify it, search it, or save it. (Note: you can modify the copy of the catalog in memory, but that does not modify the disk's catalog.) This feature is also available in a WPL program. Here's how it works.

First, enter the statement

```
OA.D1#
```

This WPL statement consists of the following parts:

O	Opens the SOS Command Menu.
A	Selects option A, the catalog option.
.D1	Reads the catalog from drive 1.
#	Puts the catalog in the text buffer.

Now comes the tricky part. Apple Writer doesn't load the catalog all at once; instead, it loads a screenful and then waits for you to type a space or press RETURN. That's fine when you use `[O]A#` as an immediate command, but when you use it in a program your program doesn't know in advance how many screens the catalog contains. Here are the statements that you must put after `OA.D1#` to make sure you load the entire catalog into memory:

(spaces) `P` (5 spaces) RETURN
(spaces) `P` RETURN

Notice that both statements are in the comment format. The first contains five spaces because the catalog may contain up to five screens of information—and each space causes Apple Writer to load another screen. The RETURN at the end of the first comment brings Apple Writer back to the SOS Command Menu. The RETURN in the second comment causes Apple Writer to exit from the SOS Command Menu and execute the next statement in your program.

The following `CATALOG` program reads a catalog of any length (up to five screens) and prints the catalog:

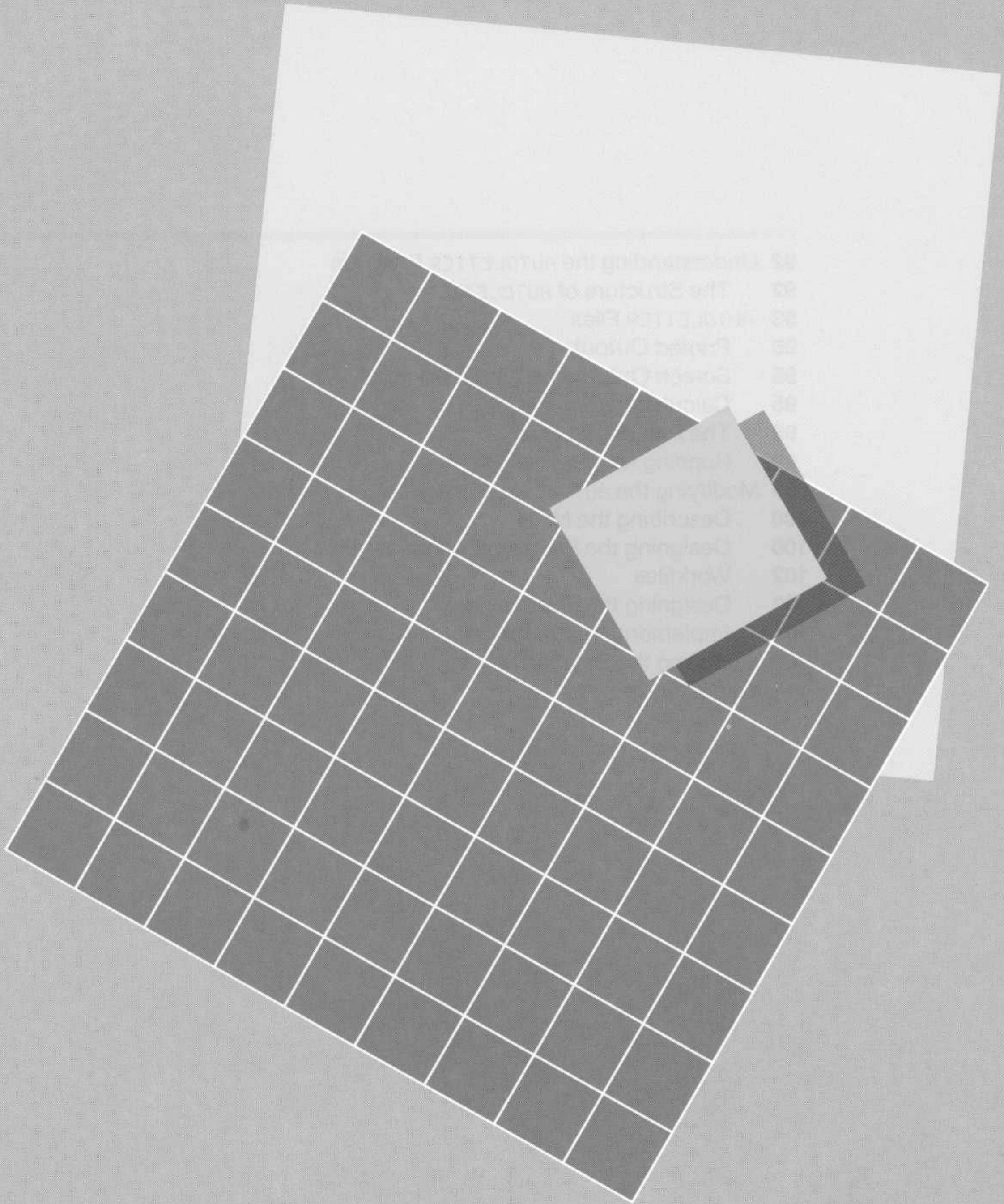
Example: printing your catalog

```
CATALOG NY
        OA.D1#
        P (5 spaces)
        P
        PNP
        PQT
```

A Final Word

As you continue to use Apple Writer and WPL, you'll see how WPL programs can simplify your writing and organizing tasks. Discover the power of WPL by experimenting. One good way to begin is to modify the sample programs written in thise manual. You can also change the programs that are provided on your Apple Writer disk. Chapter 8 explains how to modify programs, using the `AUTOLETTER` program as an example.

Enhancing WPL Programs



Enhancing WPL Programs

The quick way to write a program: revise an existing one.

New programmers quickly learn that the easiest way to write a program is to find one that does **almost** what you want it to do, and then modify it. In this chapter you'll find out how to modify a program that someone else has written, using the AUTOLETTER program as an example. You will improve AUTOLETTER by giving it the ability to use titles (Mr., Ms., and so forth), and the ability to use first names and last names selectively in the body of the form letter.

You will learn

- how to understand a program that you didn't write;
- how to use a workfile;
- how to redesign a program;
- how to test the modified program.

Understanding the AUTOLETTER Program

The AUTOLETTER program, found on your MASTER disk, appears below. The numbers to the left of the WPL statements are not part of the program. We've added them so that you can easily refer to specific statements.

```
 1.  START    PSX 1
 2.  LOOP     NY
 3.           L .D1/FORMLETTER
 4.           B
 5.           F/(Address)//
 6.           Y?
 7.           L .D1/ADDRS!<(X)>!<!N
 8.           PGO FOUND
 9.           PGO QUIT
10.  FOUND    PLS .D1/ADDRS!<(X)>! !N =  $A
11.           B
12.           F/(Name)/$A/A
13.           PNP
14.           PSX +1
15.           PGO LOOP
16.  QUIT     PIN[\] Done at address (X) (press RETURN)
17.           NY
```

The Structure of AUTOLETTER

The first thing to look for is the structure of the program. That is, what is the basic outline of the program logic? To acquire an understanding of any WPL program, answer these questions:

Basic elements of a WPL program

- What files does the program use? What do they look like?
- What printed output does the program produce? What screen output and keyboard input?
- What calculations are performed?
- What is the main processing loop? (What does the program do?)

AUTOLETTER **Files**

Scan the program listing looking for [L]oad, Load String (LS), and [S]ave commands—commands that refer to files. Statements 3, 7, and 10 contain references to the FORMLETTER file and the ADDRS file.

> **Note:** When analyzing a program, print every file the program uses. If a file is very long, print just the beginning and the end of it.

The FORMLETTER file is printed below:

```
(Address)

Dear (Name):

Congratulations on your purchase of an
Apple computer. You and your family
will spend many enjoyable and
instructive hours with your new
personal computer. In today's
fast-paced high-technology world,
(Name), you can't afford to be without
one. And you can rest assured that
when you use an Apple computer, you're
using the best there is.

Best wishes,

The Folks at Apple Computer

.inAddress number (X) (press return)
.FF
```

In order to understand the structure of the AUTOLETTER program, study the structure of its files. Notice the following things about the FORMLETTER file:

- It contains the text of a letter.
- It contains two words in parentheses: (Address) and (Name). (Name) appears throughout the letter.
- It contains an embedded Input command, which refers to address number (X). This implies that the numeric variable (X) is set in the AUTOLETTER program, and also that AUTOLETTER will stop after each letter is printed and wait until RETURN is pressed.
- It ends with an embedded .FF (formfeed) command. When the form letters are printed, .FF causes the printer to skip to the top of the page so that each letter begins on a new page.

Next, look at the ADDRS file. The beginning and end are shown below:

```
<1>John Smith
123 Elm Street
Anytown, U.S.A. 12345
...
<5>Mary Sanders
00000 Null Result
Meander, OH. 54637
<
```

Notice the following things about the ADDRS file:

- It contains a series of four-line addresses consisting of name, street address, city and state, and zip code.
- Each address begins with a number enclosed in angle brackets. The addresses are consecutively numbered.
- The file ends with a left angle bracket.

Printed Output

To find out what the program prints, first look for New Print (NP) and Continue Print (CP) statements. Then see what file is [L]oaded before the print command is issued. It turns out that at statement 13 the AUTOLETTER program prints the letter that was loaded from the FORMLETTER file at statement 3.

Screen Output and Keyboard Input

Screen displays usually provide choices for the user, control over printing, or information about the progress of the program. Screen messages provide clues to how the program works. To locate them in the AUTOLETTER program, look for Print (PR) and Input (IN) commands. (Don't forget the embedded Input command you found in the FORMLETTER file.)

The following is true of AUTOLETTER:

- There are no user choices because there are no Input commands that contain a string variable for input.
- There is one message, at statement 16. It contains a number that is 1 greater than the number of the last address printed by the program.
- The Input statement at the end of the form letter stops the program until you press RETURN. It's there to allow you to change paper in the printer. If your printer uses continuous forms (paper that comes in a roll, like paper towels, or in a box of attached sheets), the embedded Input statement may be removed.

Calculations

Look for commands that set numeric variables—SX, SY, and SZ. One of these commands at the top of the loop (just after the label that begins the loop) or at the bottom of the loop (just before a Go command) probably identifies a calculation that is part of the program structure. A numeric variable is often used to control a loop. In the CALC routine in Chapter 6, for instance, the (Z) variable is decreased by one each time the loop is executed; when (Z) reaches zero, looping ends.

Numeric variables control looping.

Statement 14 in AUTOLETTER adds 1 to the (X) variable. The following statements also use the (X) variable:

- Statement 1 initializes the value of the (X) variable by setting it to 1.
- Statement 7 uses the (X) variable in a [L]oad command to search for a marker in the ADDRS file. Substitute a value for the variable to see what the marker looks like: when (X) is 1, the marker is <1>.
- Statement 10 uses the (X) variable in a Load String command to search for a marker in the ADDRS file. (X) has the same value in statement 10 as it had in statement 7.

Use of the (X) variable in the AUTOLETTER program can be summarized as follows:

- it is initialized to 1,
- it is used to locate text in the ADDRS file,
- it is increased by 1 at the bottom of the loop.

The ADDRS file reveals how the consecutive address numbers are related to the use of the (X) variable in AUTOLETTER.

The Processing Loop

Most programs do some main task over and over. For instance, a program that generates form letters prints the body of a letter over and over again. No matter how complex the processing may be, it can be summarized in a simple statement. Because you have analyzed the inputs, outputs, and calculations of the AUTOLETTER program, you can now define its main task.

Look for Go commands in the program. The Go command determines what statement is executed next. Notice whether it is associated with conditional execution of a previous statement.

There are three Go commands in AUTOLETTER:

- Statements 8 and 9 each contain a Go command. These statements are associated with the preceding [L]oad, which searches for a marker. If the marker exists in the FORMLETTER file, the program proceeds to the FOUND label at statement 10; if not, it proceeds to the QUIT label at statement 16. Statements 8 and 9 together represent a *conditional transfer of control*: the next statement to be executed depends on whether the [L]oad command finds the marker. A conditional transfer of control is like a fork in the road. See Figure 8-1.

  Because there is no Go command after statement 16 sending control back to the processing loop, the program ends after the QUIT section is executed.

- Statement 15 contains the third Go command. This statement represents an *unconditional transfer of control*. It always causes statement 2 to be executed next. An unconditional transfer of control is like a bend in the road. It causes the program to stop executing statements in a straight line, one after another, and go to a different place in the program. See Figure 8-2.

Figure 8-1. Conditional Transfer of Control: A Fork in the Road

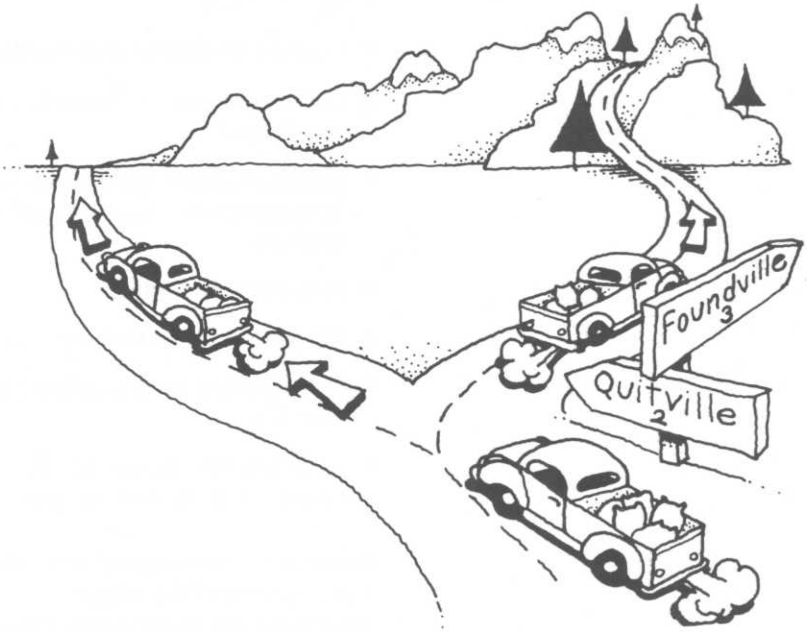
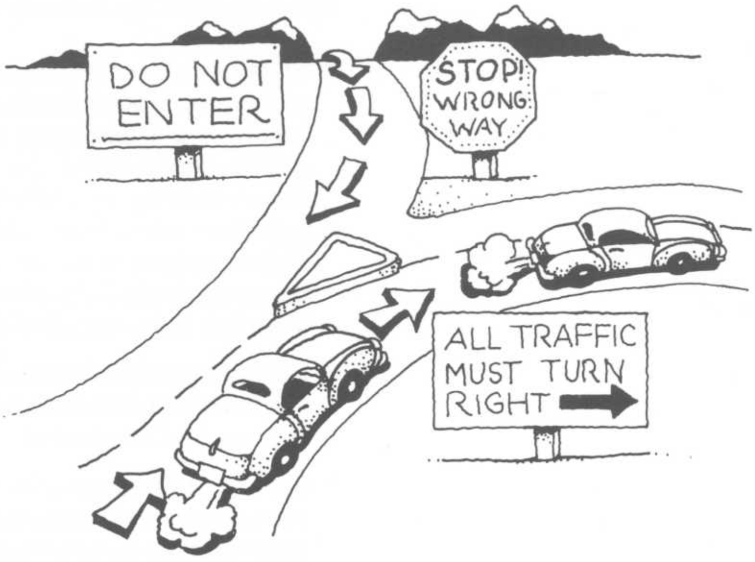


Figure 8-2. Unconditional Transfer of Control: A Bend in the Road



With what you've learned, you can now describe the processing loop. Here's what AUTOLETTER does:

- Sets (X) to 1.
- Loads a form letter into memory.
- Loads address <1> from the ADDRS file. If the address doesn't exist, time to quit.
- Loads a name from address <1> and uses it to replace every occurrence of (Name) in the form letter. (See statements 10 and 12.)
- Prints the form letter.
- Adds 1 to (X), so that the next address will be <2>.
- Goes back to the top of the loop and loads a fresh copy of the form letter.
- Displays a message, clears memory, and ends the program when (X) is 1 more than the highest-numbered address in the ADDRS file.

When you have finished analyzing the program, prepare a brief summary statement like this one: "AUTOLETTER prints a customized form letter for every address in a consecutively numbered address file."

AUTOLETTER is a relatively simple program, but the technique you've just learned will work for any WPL program.

Running AUTOLETTER

The program you're changing may modify the files that it uses.

You may be wondering why you didn't begin this process by running AUTOLETTER. The reason is that sometimes a program modifies files, and if you don't know exactly what it does you don't want to change the files without making a backup copy. To back up the files, you need to find out their names by studying the program—as you just did.

Before going any further, copy the AUTOLETTER program and its two files to a set of files on another disk. Name the copies AUTOLETTER2, FORMLETTER2, and ADDRS2. These are the files that you will modify in the next part of the chapter.

Now run AUTOLETTER and see exactly how it works:

- Make sure the Apple Writer MASTER disk is in drive 1 and that your printer is turned on.
- Press [P]. To print the form letters on a printer, type PD.PRINTER. To display them on the screen, type PD.CONSOLE. Then press RETURN .
- Press [P]. Type DO AUTOLETTER and then press RETURN .
- Press RETURN after each letter is printed.

Modifying the AUTOLETTER Program

Modifying a program is easy ... but only if you do it in a methodical manner. There are four steps to take, no matter what kind of program you want to change or how complex the change is:

Define your need and design a solution.

1. Describe the problem or need. What do you want the program to do?
2. Design a solution. Conceptually, what additions and changes to the program need to be made? What changes must be made to the files? What will the new output look like?
3. Implement the solution. What specific program statements must be added or changed? What specific changes must be made to files used by the program?
4. Test the solution. Does the output of the modified program match the results you defined in step 2?

In the rest of this chapter you will follow these four steps in order to add new features to the AUTOLETTER program.

Describing the Need

AUTOLETTER takes information from an address file and creates personalized form letters. However, it does not provide for titles such as Ms., Mr., Dr., and so on. Therefore you might define what needs to be changed in the following way:

AUTOLETTER should be able to address the recipient of a form letter by title and last name ("Dear Mr. Smith"). It should also include the title in the address portion of the letter ("Mr. John Smith").

Designing the Program Changes

The design phase of a program change consists of two parts: changes to the program and changes to the files. It's best to tackle the parts one at a time.

When you analyze the need statement, you may discover aspects of the situation that weren't obvious at first. For instance, AUTOLETTER can't address a letter to Mr. Smith unless it is able to distinguish among the different parts of the name in the ADDRS file.

Distinguishing parts of the name turns out to be a major stumbling block. Looking at the FORMLETTER document, you can see that only a single form of the name is used. Statement 10 in the AUTOLETTER program shows that the name consists of whatever is between the end of the address marker and the next space character. This, presumably, is the person's first name.

What if you add a title to the beginning of the name line:

```
<1> Mr. John Smith
```

Now the information between the end of the address marker (<1>) and the first space is "Mr.". Good, because you want to be able to isolate that information.

But how will AUTOLETTER get the first name in the name line? Can it say that the first name begins and ends with a space? No, because the first marker in a Load String statement must be unique in the file, and the space character is not unique. Therefore the first name in address <1> will always be used because that's the first occurrence in the file of information beginning and ending with a space.

If you think about it, you will see that this problem occurs no matter what part of the name line you try to isolate. Even if you insert markers into the name, such as

```
<1>Mr.+John$Smith
```

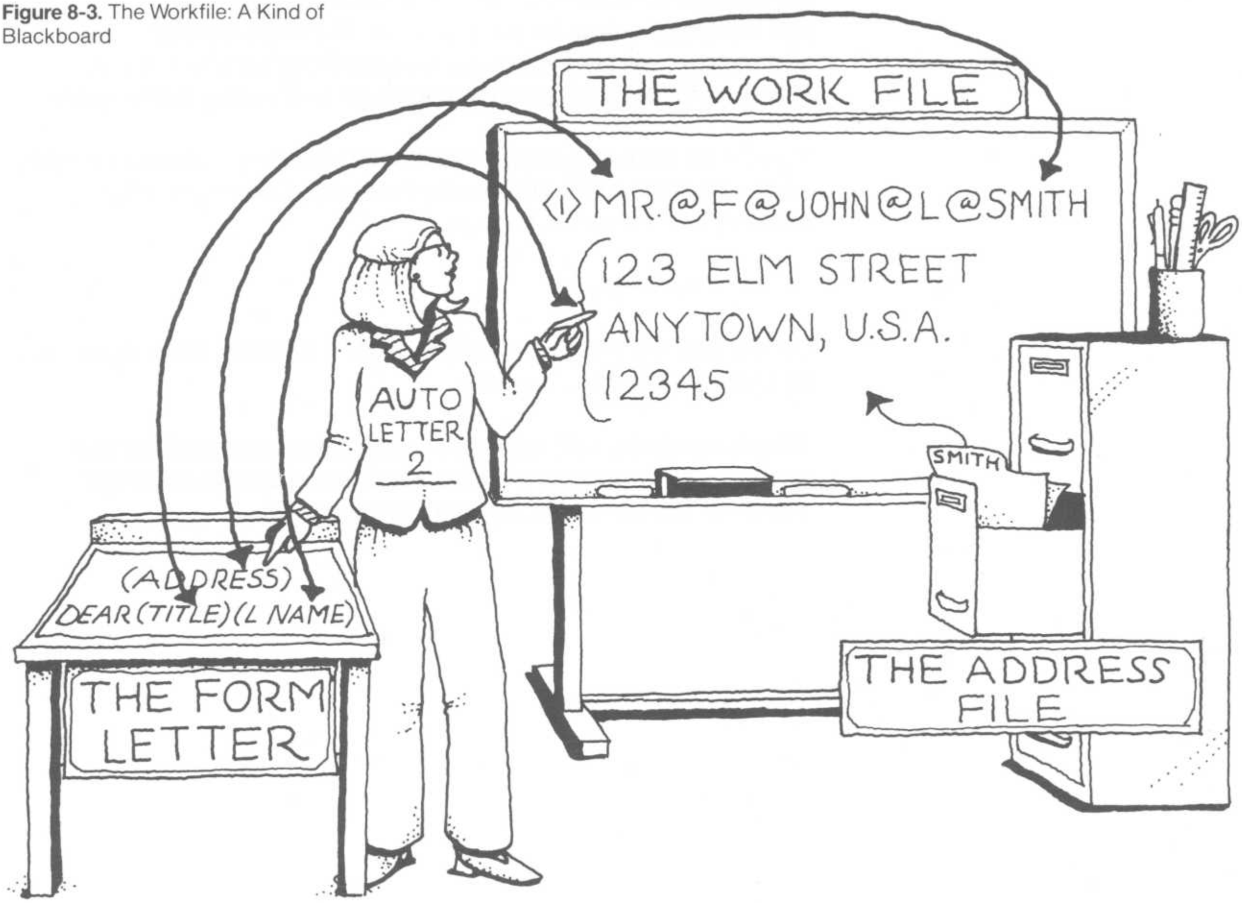
the markers are unique only within a given address. They would not be unique within the ADDRS file.

What's needed is a file containing only the current address that AUTOLETTER is working on. How in the world can we make that happen? The answer is, by creating a workfile.

Workfiles

A workfile is a file that a program uses over and over again. It's something like a blackboard. Let's say you're writing a book on mathematics and you want to prepare a list of answers to the problems in the book. You might work a problem on your blackboard, record the answer, erase the blackboard and work the next problem, and so on. Once you've got the answer, you don't have any reason to preserve the means by which you arrived at it.

Figure 8-3. The Workfile: A Kind of Blackboard



Let's see if AUTOLETTER can make use of such a workfile. We must envision how the workfile would be used and determine whether such a use is possible in AUTOLETTER:

How the Workfile is Used	Why This Is Feasible
Add the current address to the document in memory (the form letter).	AUTOLETTER already does this.
Save the current address in a workfile.	[S]ave with markers saves a portion of a document.
From the workfile, load the title, first name, and last name into separate strings.	Because the workfile contains only the current address, you can designate parts of the name with unique markers.
Insert the title, first name, and last name into the form letter.	[F]ind with Replace option.

Having demonstrated the feasibility of the proposed solution, you can proceed to look at the changes that must be made to the files.

Designing the File Changes

The files AUTOLETTER uses are

```
FORMLETTER
ADDRS
```

During this discussion it will be helpful to have a copy of the FORMLETTER document in front of you, so print a copy or display the document on your screen or refer to the listings earlier in this chapter.

Looking at the FORMLETTER document, you recall that it contains two variable elements, (Name) and (Address). In order to create the salutation "Dear Mr. Jones" it's necessary to identify specific name elements for the title and last name. Because the letter is informal, it would be nice to have the first name available as well. Therefore you decide to replace every occurrence of (Name) with one of the following:

Name Element	**Examples**
(Title)	Mr.,Ms.,Dr.,Rev.,Gen.,Hon.,Sir,Miss,Mrs.,etc.
(Fname)	George, Carol Sue
(Lname)	Smith, Huck-Finn, van den Berg, Lloyd George

You list as many different examples as possible to ensure that the design solution covers all cases.

Next, let's tackle the `ADDRS` file. That's a little trickier. Referring to the list of name elements, you see right away that the space character is no longer useful as a delimiter because it may be part of a first name (Carol Sue) or a last name (van den Berg).

> **By the Way:** The `Fname` element represents the name the person wishes to be addressed by. Carol Sue Davis may be known as either Carol or Carol Sue. It will be the responsibility of the person maintaining the `ADDRS` file to know this information about each addressee.

A worst-case example of a complex name line in the `ADDRS` document is

```
<1>Ms. Carol Sue B. Davis-Robbs
```

Your task now is to design a system, or *algorithm*, for marking the line so that `AUTOLETTER` can distinguish among the name elements in a Load String statement. Let's see what happens if you place the following symbols before each element:

Symbol	Element
`>`	`(Title)` ... part of address marker
`@F@`	`(Fname)`
`@M@`	`Middle name or initial`
`@L@`	`(Lname)` ... end of name is RETURN

The worst-case example would look like this:

`<1>Ms.@F@Carol Sue@M@B.@L@Davis-Robbs` or
`<1>Ms.@F@Carol@M@Sue B.@L@Davis-Robbs`

To determine whether these symbols will perform correctly as markers in a Load String command, check the worst-case examples against a table of markers:

Beginning Marker	Ending Marker	Element
`>`	`@`	`(Title)`
`@F@`	`@`	`(Fname)`
`@L@`	RETURN	`(Lname)`

From the table you can see that each element has a unique set of markers. If you want to test the solution in greater detail, make up some names using the examples in the table of name elements; then insert markers and examine the results to see if they satisfy the requirements you've stated.

▲ **Warning**
When you choose a marker, don't choose a character that appears in your form letter! `AUTOLETTER` will delete all marker characters in the form letter.

Implementing the Solution

Having stated how you're going to change `AUTOLETTER`, you're ready to begin programming. Put the data disk containing your copy of `AUTOLETTER2` in drive 1. Load `AUTOLETTER2` into memory and make the actual changes as we talk about them in the manual.

The first modification to the program is to insert a comment before statement 1 documenting the change:

```
AUTOLETTER2  P MODIFIED VERSION OF AUTOLETTER
```

Use new names for the modified files.

Then change the filenames in statements 3, 7, and 10 to `FORMLETTER2` and `ADDRS2` so that your modified program doesn't try to use the original files that came with Apple Writer.

You're going to use all of AUTOLETTER except statements 10, 11, and 12. These statements load the name from the address file and replace it in the document in memory. Instead, you want to load different name elements from a workfile and replace them in the document.

AUTOLETTER

```
 1.     START PSX 1
 2.     LOOP  NY
 3.           L .D1/FORMLETTER
 4.           B
 5.           F/(Address)//
 6.           Y?
 7.           L .D1/ADDRS!<(X)>!<!N
 8.           PGO FOUND
 9.           PGO QUIT
10.     FOUND PLS .D1/ADDRS!<(X)>! !N = $A
11.           B
12.           F/(Name)/$A/A
13.           PNP
14.           PSX +1
15.           PGO LOOP
16.     QUIT  PIN[\] Done at address (X) (press RETURN)
17.           NY
```

To save the current address in a workfile, precede it with a unique marker such as (). The first section of new statements inserts the () marker at the end of the FORMLETTER document in memory. The current address is then loaded, including markers. The following statements replace statements 10, 11, and 12.

```
FOUND   E
        D
        P INSERT RETURN AND () MARKER
        F< <>()<
        Y?
        P LOAD CURRENT ADDRESS AT END OF LETTER
        L .D1/ADDRS2!<(X)>!<!
```

The next section of the program places the cursor at the () marker and saves the current address in the workfile.

```
B
F/()/()/
Y?
P CREATE WORKFILE WITH CURRENT ADDRESS
S .D1/WORKFILE#%< #
Y
```

Now the name elements can be replaced in the form letter.

```
P INSERT TITLE FROM WORKFILE INTO FORM LETTER
PLS .D1/WORKFILE!>!@!N = $D
B
F/(Title)/$D/A
P INSERT FIRST NAME
PLS .D1/WORKFILE!@F@!@!N = $D
B
F/(Fname)/$D/A
P INSERT LAST NAME
PLS .D1/WORKFILE<@L@<>< N = $D
B
F/(Lname)/$D/A
```

There are just two small housekeeping details left to take care of. First, you need to remove the markers that are embedded in the name.

```
P DELETE MARKERS IN FORM LETTER
B
F/@F@/ /A
B
F/@M@/ /A
B
F/@L@/ /A
```

Next, you must delete the current address at the end of the letter to be printed. From the end of the letter, delete one line at a time until the ( ) marker is deleted. When it's gone, you know you've deleted the entire address.

```
DELTEMP E
        P DELETE CURRENT ADDRESS AT END OF FORM LETTER
        X
        F/()/()/
        Y?
        PGO DELTEMP
```

The finished program looks like this:

AUTOLETTER2

```
       P *** AUTOLETTER2 ****
START  PSX 1
LOOP   NY
       L .D1/FORMLETTER2
       B
       F/(Address)//
       Y?
       L .D1/ADDRS2!<(X)>!<!N
       PGO FOUND
       PGO QUIT
FOUND  E
       D
       P INSERT RETURN AND ( ) MARKER
       F<<>()<
       Y?
       P LOAD CURRENT ADDRESS AT END OF LETTER
       L .D1/ADDRS2!<(X)>!<!
       B
       F/()/()/
       Y?
       P CREATE .D1/WORKFILE WITH CURRENT ADDRESS
       S .D1/WORKFILE#%<#
       Y
       P INSERT TITLE FROM .D1/WORKFILE INTO FORM LETTER
       PLS .D1/WORKFILE!>!@!N = $D
       B
       F/(Title)/$D/A
       P INSERT FIRST NAME
       PLS .D1/WORKFILE!@F@!@!N = $D
       B
       F/(Fname)/$D/A
       P INSERT LAST NAME
       PLS .D1/WORKFILE<@L@<><N =  $D
```

```
         B
         F/(Lname)/$D/A
         P DELETE MARKERS IN FORM LETTER
         B
         F/@F@/ /A
         B
         F/@M@/ /A
         B
         F/@L@/ /A
DELTEMP  E
         P DELETE CURRENT ADDRESS AT END OF FORM LETTER
         X
         F/()/()/
         Y?
         PGO DELTEMP
         PNP
         PSX +1
         PGO LOOP
   QUIT  PIN[\]  Done at address (X) (press RETURN)
         NY
```

Testing the Solution

In order to test the programming changes you've made, it's necessary to change the files AUTOLETTER uses so that they match the new version, AUTOLETTER2.

In the ADDRS2 file, the appropriate markers must be inserted in the name line, according to the table of markers, so that the addresses look like this:

```
<1>Mr.@F@John@L@Smith
123 Elm Street
Anytown, U.S.A. 12345
...
<5>Ms.@F@Mary Alice@M@R.@L@Sanders
00000 Null Result
Meander, OH. 54637
<
```

Warning
Be sure the last address in the ADDRS2 file ends with a < marker on a separate line. If the < marker is missing, the last form letter will not be printed.

Notice that address `<5>` has been changed to test one of the conditions you designed `AUTOLETTER2` to handle: a first name that contains a space character. Check to see if there are other conditions that need to be tested, and add or change addresses in `ADDRS2` so that all program modifications are verified.

You must also change the `FORMLETTER2` document to conform to `AUTOLETTER2`'s algorithm for filling in the title and last name separately. Here's how the changed letter might look:

```
(Address)

Dear (Title) (Lname):

Congratulations on your purchase of an
Apple computer. You and the (Lname)
family will spend many enjoyable and
instructive hours with your new
personal computer. In today's
fast-paced high-technology world,
(Fname), you can't afford to be
without one. And you can rest assured
that when you use an Apple computer,
you're using the best there is.

Best wishes,

The Folks at Apple Computer

.inAddress number (X) (press return)
.FF
```

Now run the enhanced version of `AUTOLETTER`:

Press [`P`]
Type `DO .D1/AUTOLETTER2`
Press (RETURN)

If you like watching the letter being personalized, leave the program as is. If you'd like the program to run much faster and don't need to see the document displayed, insert a No Display `(ND)` statement before the `START` statement.

The final step in testing is to review the results. Was every letter printed correctly? If not, follow the debugging instructions in Appendix D, change the program or files as necessary, and run the test again. The letter produced for address `<5>` should look like this:

```
Ms. Mary Alice R. Sanders
00000 Null Result
Meander, OH. 54637

Dear Ms. Sanders:

Congratulations on your purchase of an
Apple computer. You and the Sanders
family will spend many enjoyable and
instructive hours with your new
personal computer. In today's
fast-paced high-technology world,
Mary Alice, you can't afford to be
without one. And you can rest assured
that when you use an Apple computer,
you're using the best there is.

Best wishes,

The Folks at Apple Computer
```

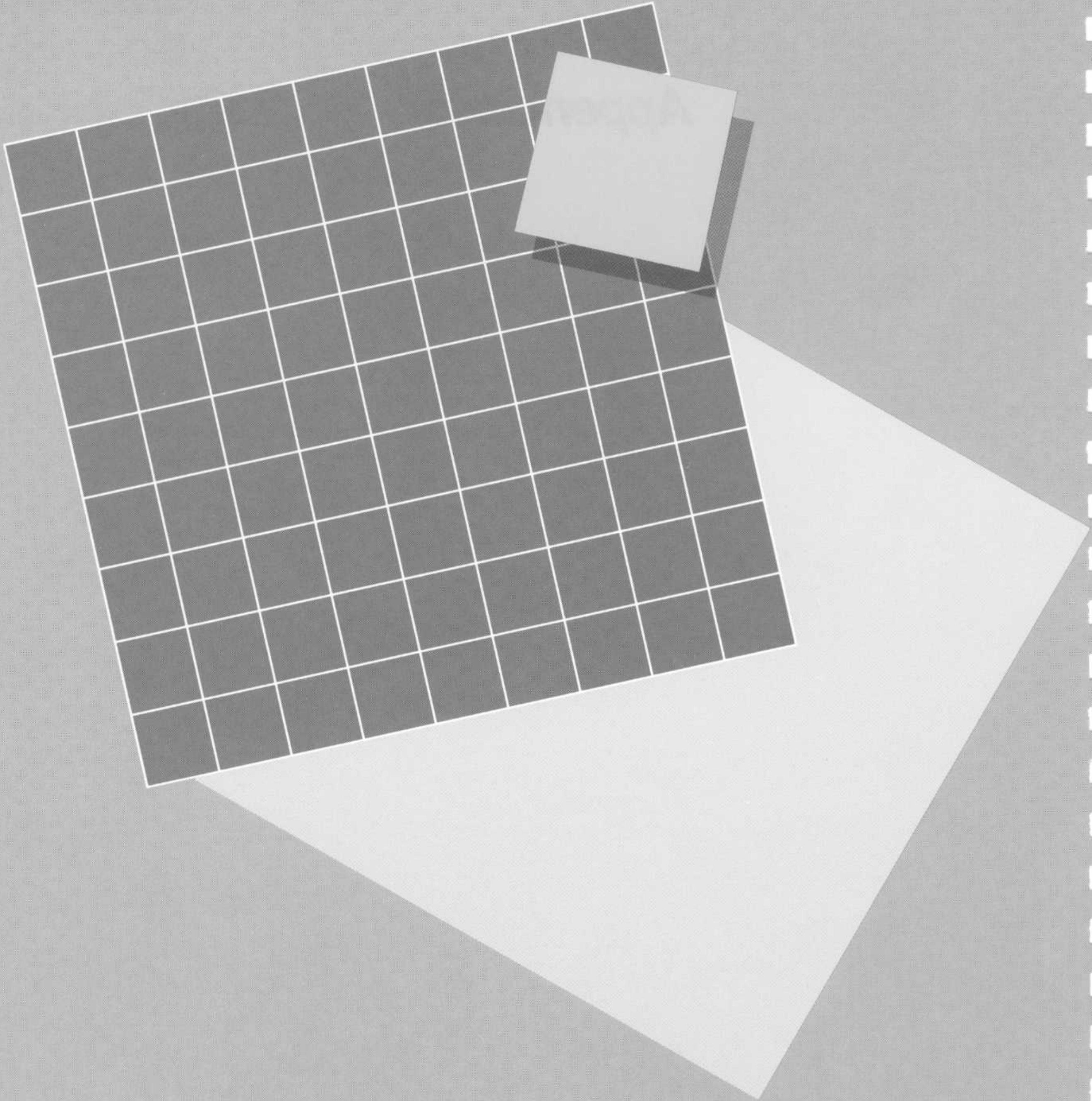
> There are copies of the modified files on the UTILITIES disk (`WPL.AUTOLETTER2`, `FORMLETTER2`, and `ADDRS2`). You can refer to these if you are having any trouble with your modified files.

Summing Up

In this chapter you have learned what steps to take to change a program so that it fits your needs. First you learned how to examine a WPL program and find out what it does. Then you learned how to design, implement, and test your modifications. You became familiar with workfiles, and you saw how using a workfile could expand the capabilities of WPL.

You now have all the tools you need to be an effective, efficient, innovative WPL programmer. But you haven't learned all that WPL can do for you. The power of WPL is that you can tailor-make programs to suit your own Apple Writer requirements. Need a program that sends out customized memos? formats TV and movie scripts? keeps track of all your Apple Writer files? Whatever it is, you're ready to design it, write it, and let the player piano work while you play.

Appendixes



Syntax of WPL Statements

A WPL program contains one or more statements. A statement may contain a WPL command, an Apple Writer command, or a comment. Any statement that does not contain a recognizable command is ignored when the program is executed. Every statement must end with RETURN.

The General Format of a Statement

The format of a statement is

`LABEL` space(s) `COMMAND NAME` space(s) `ARGUMENT` RETURN

The label must start in the leftmost position of the line. One or more spaces are required between the label and the command name. If there is no label, one or more spaces must precede the command. The rules governing spaces between the command name and the argument vary with the command: for Apple Writer commands, use spaces just as you would if you entered the command as an immediate command; for WPL commands, whether or not you need spaces between the command name and the argument depends on which command you're using:

- commands that do not take an argument: `CP`, `ND`, `NP`, `QT`, `RT`, `YD`
- commands in which each space counts as a character: `AS`, `IN`, `PR`
- commands in which spaces are optional and have no effect: `CS`, `DO`, `GO`, `LS`, `SR`, `SX`, `SY`, `SZ`

Upper- and lowercase letters may be used interchangeably except in label references and (in certain cases) arguments.

Labels

A label is a name given to a WPL statement. The label is optional; it may be any length and may contain any characters except spaces. Upper- and lowercase letters in a label are significant: `LABEL` and `label` are not considered the same.

Commands

A command may be an Apple Writer command (see the Apple Writer manual) or a WPL command (see Appendix B).

Apple Writer commands are entered in programs in **exactly** the same way as you would enter them as immediate commands with one important exception: you don't have to press CONTROL. That is, [`L`]oad is entered as `L`, [`F`]ind is entered as `F`, and so on.

WPL commands are always preceded by a `P` when entering them in programs. The `P` stands for [`P`]rint/Program. Thus, the New Print command is entered as `PNP`.

Arguments

Some commands take arguments, others do not. For Apple Writer commands, refer to the Apple Writer manual. For WPL commands, see Appendix C. The argument for an Apple Writer command may contain a RETURN and may therefore appear on more than one line.

RETURN

Every statement must end with RETURN.

If RETURN appears on a line by itself, it is ignored. Therefore if you need two RETURNs in a row—for instance, to exit from a menu—precede the second RETURN with at least one space or other noncommand character.

Variables

There are three numeric variables and four string (text) variables in WPL. Numeric variables represent unsigned whole numbers. String variables represent one or more characters of text.

String Variables

The string variables are represented as $A, $B, $C, and $D. They may be either upper- or lowercase and are always preceded by a dollar sign. Each string variable may contain up to 64 characters of text. A string variable may be used in place of text in any command. The value of the variable at the time the command is executed replaces the variable name.

Numeric Variables

The numeric variables are represented as (X), (Y), and (Z). They may be either upper- or lowercase and are always enclosed in parentheses. They may be increased, decreased, or set to any integer from 0 to 65,535. If the maximum number is increased by 1, the result is 0; this is called *overflow*. When a statement increases or decreases a numeric variable to 0, the next statement is skipped. When a statement sets a numeric variable to 0, the next statement is executed.

Constants

A constant is a string that always has the same value. Its value does not change while the program is running. For example, in a command that adds 1 to a numeric variable, "1" is a constant; in a command that displays HELLO on the screen, "HELLO" is a constant.

Comments

A comment line may be inserted anywhere in your WPL program. By convention, a comment begins with the following characters:

space(s) P space(s)

Any statement that is not recognized as a WPL command or an Apple Writer command is assumed to be a comment and is ignored during execution of the program. A comment may not appear on the same line as a command.

Appendix B

List of WPL Commands

Deferred Commands: Apple Writer does not immediately execute deferred commands when you type a WPL program. They are executed when the program that contains them is run. All of the commands listed below may appear as deferred commands in a WPL program, preceded by the character P. For example, PNP.

Immediate Commands: Immediate commands are executed as soon as they are typed at the keyboard. Commands listed below with an asterisk in column I may be executed immediately from the keyboard, preceded by [P]. For example, press [P] and then type NP followed by (RETURN).

Embedded Commands: Commands embedded in a document are executed when Apple Writer encounters them in the course of printing the document. Commands listed below with an asterisk in column E may be embedded in an Apple Writer document, preceded by a period. For example, .IN.

Command	I	E	Description
AS			ASSIGN STRING (give string variable a value)
CP	*		CONTINUE PRINTing the next file of a document
CS			COMPARE STRINGS
DO	*		DO (execute) a WPL program
GO			GO to a labeled WPL statement and execute it
IN		*	INPUT (display a message and wait for a reply)
LS			LOAD STRING
ND			NO DISPLAY of text on screen
NP	*		NEW PRINT (first or only file of a document)
PR			PRINT (display) a line on the screen
QT			QUIT the program and return to Apple Writer
RT			RETURN from a subroutine
SR			SUBROUTINE call
SX			SET X (change the value of numeric variable X)
SY			SET Y (change the value of numeric variable Y)
SZ			SET Z (change the value of numeric variable Z)
YD			YES DISPLAY text on screen

Summary of WPL Commands by Function

The 17 different WPL commands fall into four functional groupings:

- **Transfer of Control Commands**—DO, GO, QT, SR, RT
  These commands affect the sequence of execution; the next statement to be executed is not the next sequential statement in the WPL program. Commands that cause a WPL program to start and end are included here.
- **Output Commands**—display commands are PR, IN, ND, YD
  —print commands are NP, CP
  These commands allow the program to display messages, print documents, and turn the text buffer display on and off.
- **Numeric Variable Commands**—SX, SY, SZ
  These commands are used to set integer variables and change them by addition and subtraction.
- **String Variable Commands**—AS, CS, LS
  These commands are used to set string variables, compare two strings, and load a specified string from a file.

This appendix tells you how to use each WPL command. The commands are grouped by function and covered in the order shown above.

There is a fifth group that is composed of commands that also belong to other groups:

- **Conditional Commands**—SX, SY, SZ, CS, LS
  These commands affect the sequence of execution. Under certain conditions, the statement following the conditional command is skipped. Conditions that cause a statement to be skipped are described in this appendix in the usage section for each conditional command.

In order to use this appendix, you need to be familiar with the syntax rules of WPL (see Appendix A).

Items in braces ({ }) are optional.

Command Modes

There are three ways (or *modes*) of using WPL commands: deferred, immediate, and embedded. Some commands can be used in more than one of these modes. Every command can be used in deferred mode—that is, in a WPL program. The command's argument is the same regardless of mode, but the command name must be preceded by the character "." in embedded mode or "P" in deferred mode. This section contains a definition of each mode and the command name syntax required for each mode. The command descriptions below give an example for each mode in which a given command can be used. See Appendix B for a table of the modes allowed for each command.

Deferred Mode

Syntax: Command name is preceded by the character P.
Example: `PDO`
Usage: P is the [P] in the Apple Writer [P]rint/Program command, but it is typed without pressing CONTROL when entering a deferred command. Normally [P]rint/Program commands are executed as soon as they are entered. In deferred mode, however, these commands are treated as text; they are then stored in a file as part of a WPL program and are executed only when the program is run.

Immediate Mode

Syntax: Command name is preceded by [P].
Example: `[P]DO`
Usage: From Apple Writer, the command is executed immediately. The command is valid only when typed after pressing [P].

Embedded Mode

Syntax: Command name is preceded by a period.
Example: `.IN`
Usage: When embedded in an Apple Writer document, the command is executed when it is encountered in the course of printing the document.

Transfer of Control Commands

DO—*Execute a WPL Program*

Syntax: `DO .d1/filename`

Examples: `PDO .D2/CHAINPROG`

`[P]DO .D1/WPLPROG`

Usage: In deferred mode, the current WPL program chains to a program called `filename`.

In immediate mode, the WPL program called `filename` is executed immediately.

GO—*Execute a Labeled Statement*

Syntax: `GO statementlabel`

Example: `PGO LOOP`

Usage: Causes the statement named `statementlabel` to be the next statement executed, regardless of where in the program the `statementlabel` statement appears. Execution then proceeds normally.

QT—*Quit*

Syntax: `QT`

Example: `PQT`

Usage: When `QT` is encountered, the WPL program is exited and control is returned to Apple Writer.

SR—*Subroutine Call*

Syntax: `SR subroutinelabel`

Example: `PSR PRINT`

Usage: Causes the statement named `subroutinelabel` to be the next statement executed. Execution then proceeds normally until a Return command is encountered. Note: a subroutine may contain a Subroutine Call. This is called *nesting*. Subroutines may be nested up to 32 levels.

A subroutine is not allowed to call itself.

`RT`—*Return From Subroutine*

Syntax: `RT`
Example: `PRT`
Usage: Causes control to be transferred to the statement following the Subroutine Call command that was executed last.

Output Commands

`PR`, `IN`, `ND`, and `YD` govern screen output. `NP` and `CP` govern printer output.

`PR`—*Print a Line*

Syntax: `PR {message-text}`
Example: `PPR ** Welcome to the Print Menu Program **`
Usage: Causes a message line of up to 128 characters to be displayed on the screen. If no message text is provided, a blank line is displayed.

`IN`—*Input a Line*

Syntax: `{message-text} {$A}`
Examples: `PIN Name of document to be printed? = $A`
`.IN  Please insert special forms in printer`
Usage: Causes a message line of up to 128 characters to be displayed on the screen. If no message text is provided, a blank line is displayed.

Causes the program to wait until RETURN is pressed. A response may be typed before pressing RETURN. In deferred mode, the program may test the response for a particular value. In embedded mode, any response followed by RETURN causes the program to resume execution.

Every character before the `=` sign is displayed, including space characters. The string variable (`$A`) is not displayed. Any of the WPL string variables may be specified. The response is stored in the specified string variable.

ND—*No Display of Text Buffer*

Syntax: ND
Example: PND
Usage: Prevents document in memory from being displayed on the screen. Use this command for faster processing of a WPL program.

Provides full use of screen for PR and IN commands. If ND is not specified, only a single line on the screen is available for output messages.

YD—*Yes, Display Text Buffer*

Syntax: YD
Example: PYD
Usage: Causes document in memory to be displayed on the screen.

NP—*New Print*

Syntax: NP
Examples: PNP
[P]NP
Usage: Causes the document in memory to be printed. The first page is numbered according to the current value of Page Number (PN), which is described in the section on printing in the Apple Writer manual.

CP—*Continue Printing*

Syntax: CP
Examples: PCP
[P]CP
Usage: Causes the document in memory to be printed. Page numbering continues from previous document.

Numeric Variable Commands

SX sets numeric variable (X), SY sets numeric variable (Y), and SZ sets numeric variable (Z). The three numeric variable commands function identically; therefore, only SX is shown here.

SX—*Set X*

Syntax: SX n or SX +n or SX -n
SX (Y)
SX $A

Example: PSX +2

Usage: n is a whole number between 0 and 65,535 inclusive. If n is not preceded by a sign, the variable (X) is set to n. If n is preceded by a plus sign (+n), the value of n is added to (X). If n is preceded by a minus sign (-n), the value of n is subtracted from (X).

n may be a numeric variable. For instance, the statement SX (Y) takes the current value of variable (Y) and places it in variable (X). (Y) remains unchanged.

n may be a string variable. For instance, the statement SX $A takes the current value of variable $A, converts it to numeric format, and places it in variable (X). $A must represent a whole number between 0 and 65,535.

If n is signed and the new value of (X) is 0, the next statement is skipped.

String Variable Commands

The four string variables are $A, $B, $C, and $D. In the syntax of the string variable commands shown below, the source string is on the left of the equals sign and the receiving string is on the right. The receiving string is shown by convention as $A but may be any of the string variables. A string variable may contain up to 64 characters.

`AS`—*Assign String*

Syntax: `AS text-or-variable = $A`

Example: `PAS .D1/ = $D`

Usage: The string variable on the right of the equals sign is given the value of the text, variable(s), or concatenation of text and variables, on the left side of the equals sign. A string variable may appear on both sides of the equals sign; in this case its original value is lost after the Assign String command is executed. In all other cases, the values of any variables on the left side of the equals sign remain unchanged.

`CS`—*Compare Strings*

Syntax: `CS /text-or-variable/text-or-variable/`

Example: `PCS /$A/yes/`

Usage: A slash (/) is usually used as the delimiter, but you may use any character that does not appear in the text. (See the discussion on delimiters in Chapter 6). The comparison results in one of two possible outcomes: EQUAL or NOT EQUAL. If the comparison is equal, no action is taken. If the comparison is not equal, the next statement is skipped.

`LS`—*Load String*

Syntax: `LS filename!string-start!string-end!{n}{a} = $A`

Example: `PLS .D1/GETTYSBURG!Fourscore!ago! = $A`

Usage: The file is searched for the first occurrence of `string-start`. Up to 64 characters are loaded from the file into the string variable. Loading ceases when `string-end` is found. If `string-start` is not found, the next statement is skipped.

The Load String command is similar to the Apple Writer [`L`]oad command and uses the same options: `N` means "do not include the markers in the string"; `A` means "load all occurrences."

Error Messages and Debugging Hints

This appendix contains two sections. The first section explains every error message that WPL provides. The second contains hints on how to figure out why your program is not doing what you expected it to do.

Error Messages

If, while running a WPL program, Apple Writer comes to a condition that prevents it from executing the current statement, it stops and displays a message on the screen. This condition is known as an *execution error*. To leave the WPL program (which can proceed no further) and return to Apple Writer, press RETURN

Here is a list of WPL execution error messages. When the message appears on the screen, it is preceded by the words `WPL Error` and followed by the words `(Press RETURN)`.

```
Label not found -->   xxxxx
```

A Go command contained the label argument xxxxx, but no such label exists in the program.

- Have you spelled the label correctly in the Go statement?
- Did you type letter I for number 1? letter O for number 0?
- Is the use of upper- and lowercase identical in the label and the label argument?

'RT' without 'SR'

A Return (RT) command was encountered, but there was no subroutine to return from.

- Did you enter the subroutine without calling it with a Subroutine (SR) command? For instance, did you enter the subroutine by Go-ing to it?

Program > 2048 chars

The WPL program is longer than 2,048 characters. (That is, the size of the WPL program exceeds the space allotted for it in Apple Writer.)

- Refer to Chapter 7 for suggestions on using subroutines and dividing long programs.

More than 32 'SR'

More than 32 Subroutine (SR) commands were executed without a Return (RT) command.

- Does a Go command in a subroutine refer to a label outside the subroutine?
- Was the Return command bypassed? (Can the statement just before the Return statement cause conditional execution?)
- Does the subroutine call itself? For example

```
AAA    PSR  AAA or

BBB    P..
       PSR  BBB
       P..
       PRT
```

Footnote Overflow

The WPL program tried to print more than 1,024 characters of footnote text on a single page, or a single line of footnotes contains more than 128 characters.

- Is a footnote missing the end-of-footnote identifier ">]"?
- If you need help in reducing the amount of footnote material on a page, see the Apple Writer manual.

Debugging Hints

In addition to errors listed above that cause the WPL program to stop, there are others—known as *logic errors*—that do not stop the program but that produce unexpected results. These errors typically occur when

- a command name or argument is mistyped
- a WPL command is not preceded by P
- duplicate labels exist and are referred to by a Go command
- a syntax error occurs (see Appendix A)
- a file does not contain the information your program expects
- the program contains a design error
- conditional execution occurs and a statement is unintentionally skipped

The following sections describe techniques for correcting logic errors.

Desk Checking

Desk checking is pencil-and-paper work. It means taking a printed copy of your WPL program and a printed copy of the textfile, if any, and playing computer—reading each statement and writing down what happens when it is executed. Each time the value of a variable changes, write down the new value. Is it what you expected? After a command that causes conditional execution, which statement is executed next?

This is the time to make sure the program syntax is correct. This is also the time to look up in the manuals any command whose format or use you are unsure of. When you find an error, check to see if you have made the same error in more than one place in the program. After you have corrected all the errors you can find by methodical desk checking, run the program again. You may have to go through this process a number of times if you are unfamiliar with Apple Writer or WPL.

Trace

A trace is a kind of debugging diary displayed on the screen by your program. When a program is producing unexpected results and desk checking doesn't reveal where the error is, insert trace statements to display

- the sequence of execution of program statements
- changes in variables

After your program is operating properly, you can remove the trace statements.

Displaying the progress of a program

You create a trace statement by inserting an ordinary WPL Print or Input statement in your program to display the progress of the program as it is executed. The Input command allows you to stop the program as well as print out the value of any variables you are using. A good place to put a trace is at the beginning of a loop or after a change in a variable. Here is a sample program with traces inserted. (Explanations are in the righthand column.)

```
              ...
              PSX 1                           Initialize X
LABELA        F/string//
              Y?
              PGO LABELB                      Go if string found
              PQT                             Quit if not
LABELB        PSY (X)
              PSY +1                          Y = X + 1
trace1        PIN LABELB, X is (X), Y is (Y)
              L .D2/TEXTFILE!(X)!(Y)!N        Load text from marker
                                              X to Y at cursor
              PSX +1                          X = X + 1
trace2        PIN After load, X is (X)
              PGO LABELA
              ...
```

The trace statements are labeled `trace1` and `trace2`. `Trace1` lets you know that the program is in the `LABELB` routine and tells you the current values of `(X)` and `(Y)`. `Trace2` lets you know that text has been loaded and shows you the new value of `(X)`.

When you run the program with traces, the following messages will be displayed:

```
LABELB, X is 1, Y is 2
After load, X is 2
LABELB, X is 2, Y is 3
After load, X is 3
```

(and so forth)

You may also find it helpful during debugging to turn on the display of the document in memory by inserting a Yes Display (YD) command or deleting a No Display (ND) command.

Answers to Programming Questions

Throughout the manual are programming puzzles and questions. Try working out the answers by yourself before you look them up in this appendix.

Chapter 2 Answers

The STAR Program

The STAR program fills memory with stars by inserting the first star and then loading from memory. The STAR program looks like this:

```
STAR  p THIS PROGRAM FILLS MEMORY WITH STARS
      ny
      p INSERT A STAR INTO MEMORY
      f//*/
      y?
loop  e
      p LOAD MORE STARS
      L#
      pgo loop
```

To change the program so that it places *#$ in memory, change the [F]ind statement

from: `f//*/`   to: `f//*#$`

Chapter 5 Answers

Uppercase and Lowercase Responses

The following routine goes to the QUIT label if any response except uppercase "Y" is typed:

```
...
pin To print another file type Y, then press RETURN. = $a
pcs /$a/Y/
pgo print
pgo quit
...
```

To change the routine so that it accepts lowercase "y" as well, insert the following two statements after the Compare Strings statement:

```
pgo print
pcs /$a/y/
```

The modified routine looks like this:

```
...
pin To print another file type Y, then press RETURN. = $a
pcs /$a/Y/
pgo print
pcs /$a/y/
pgo print
pgo quit
...
```

Modifying the MENU *Program*

The MENU program, shown below, displays a menu that allows you to select an output destination for your document:

```
menu     pnd
         ppr [\]
         ppr PRINT OPTIONS MENU:
         ppr
         ppr     (1) Screen
         ppr     (2) Printer
         ppr     (3) Quit
         ppr
select   pin Select 1, 2, or 3:  = $a
         pcs /$a/3/
         pgo quit
         pcs /$a/2/
         pgo printer
         pcs /$a/1/
         pgo screen
         pgo select
screen   ppd. console
         pgo file
printer  ppd. printer
         pgo file
quit     pqt
file     pin Enter file name:  = $c
         ny
         L $c
         pnp
         pin Press RETURN
         pgo menu
```

To change the MENU program so that it loads a print value file for each print option (screen or printer), you must change the SCREEN routine and the PRINTER routine so that they load the appropriate files.

Print value files are loaded by means of Apple Writer's Additional Functions Menu. To use this menu, type [Q] followed by the letter of the option you have selected. If you choose Option C (Load Print/Program Value File) or Option D (Save Print/Program Value File), Apple Writer asks you for the name of the file.

The print value file that contains the values appropriate to screen display is called PVSCRN. The print value file for the printer is called PVPRTR. (The next section of this appendix tells you how to write a menu program that creates these print value files.) Replace the SCREEN and PRINTER routines in the MENU program with the following:

```
screen        qc.D2/pvscrn
              pgo file
printer       qc.D2/pvprtr
              pgo file
```

qc.D2.pvscrn means

q	Open the Additional Functions Menu
c	Choose Option C: Load Print/Program Value File
.D2/pvscrn	the file and drive (followed by RETURN)

A Menu Program That Creates a Print Value File

The followingPRTVAL program creates a print value file tailored to user specifications. The program shows how to create a print value file from a menu program. It is included here to illustrate a technique and therefore modifies only a few of the available print options. PSP, PPD, and PPL are the WPL equivalents of embedded Apple Writer print commands .SP, .PD,and .PL.

```
PRTVAL   P CREATE A PRINT VALUE FILE
         PND
         PPR [\]
         PPR PRTVAL creates a print value file for
         PPR             (1) Screen
         PPR             (2) Printer
WHICH    PIN Select 1 or 2    = $A
         P BEGIN WITH STANDARD SYSTEM PRINT VALUES IN MEMORY
         PPR Put Apple Writer master disk in drive 1,
         PIN      then press RETURN.
         QC.D1/SYS
         PCS /$A/1/
         PGO SCREEN
         PCS /$A/2/
         PGO PRINT
         PGO WHICH
SCREEN   PPD.Console
         PSP1
         PIN How many lines long is the screen display? = $B
         PPL $B
         QD .D2/PVPRTR
         PGO QUIT
PRINT    PPD.Printer
         PSP0
         PIN How many printed lines per page? = $B
         PPL $B
         QD .D2/PVSCRN
QUIT     PPR *** The print value file was created. ***
         PPR To verify the new values, press RETURN and then
         PIN     type CONTROL-P followed by ?
         PQT
```

The PRTVAL program loads the standard system print value file, PRT.SYS. Any values that are not modified by PRTVAL have the standard system setting in the new print value file.

Chapter 6 Answers

Starting the Stock Calculation With the Current Year

The stock calculation routine looks like this:

```
        ...
CALC    P Calculate Total Stock Over 5 Years
        PSZ 5
        PSX 0
LOOP    PSY +1
        PSX +(Y)
        PSZ -1
        PGO LOOP
        PIN In 5 years you will have (X) shares of stock
```

Each year on your birthday you receive your age in stock. The (Y) variable represents your age each year. When the CALC routine is entered, (Y) is your present age. At the LOOP label, (Y) is immediately increased by 1 and the result is then added to accumulator (X). That is, CALC is designed to calculate the total stock you will receive on your next five birthdays.

To redesign CALC so that it begins calculating with your **present** age, not your age at your next birthday, add your age to the accumulator before adding 1 to your age. You can do this by switching the order of the two commands following the LOOP label:

```
LOOP  PSX +(Y)
      PSY +1
      ...
```

WPL Programs in This Manual

This appendix contains a list of all the programs and routines that appear in the manual. Some of the programs are on the Apple Writer UTILITIES disk with the prefix WPL.; these are marked with an asterisk (*).

*MEMOPRT (Page 4)
    Prints a document from two files.
APPEND (Page 33)
    Makes three specific files into one.
MESSAGE (Page 34)
    Prints a selected portion of a file.
SAVEPART (Page 41)
    Saves part of a document in a new file.
ADDON (Page 41)
    Adds a document to an existing file.
PICK (Page 53)
    Echoes whatever you type.
APPEND2 (Page 53)
    Makes any three files into one.
GETNAME (Page 56)
    Prompts for a value, searches a file, and displays the result.
CHOICE (Page 59)
    Prompts for an answer, compares it to valid responses.
*MENU (Page 60, 137)
    Prints a file on the printer or displays it on the screen.
*AGE (Page 66)
    Calculates current age from birth year.
*CALC (Page 67, 140)
    Calculates a 5-year stock option. (Includes AGE program.)
NEWCALC (Page 68)
    Another way of calculating a stock option.
*NUMBER (Page 71)
    Assigns consecutive numbers to addresses in a file.

*WRITE (Page 72)

Creates a form letter.

STARTGLOSS (Page 85)

A Startup program that loads a choice of glossaries.

AUTOLETTER (Page (92)

Prints form letters, inserting names and addresses from an address file.

AUTOLETTER2 (Page 108)

Prints form letters, inserting names and addresses from an address file.

PRTVAL (Page 139)

Tailors a print value file to specifications.

Index

apple computer®
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576
030-0601-A