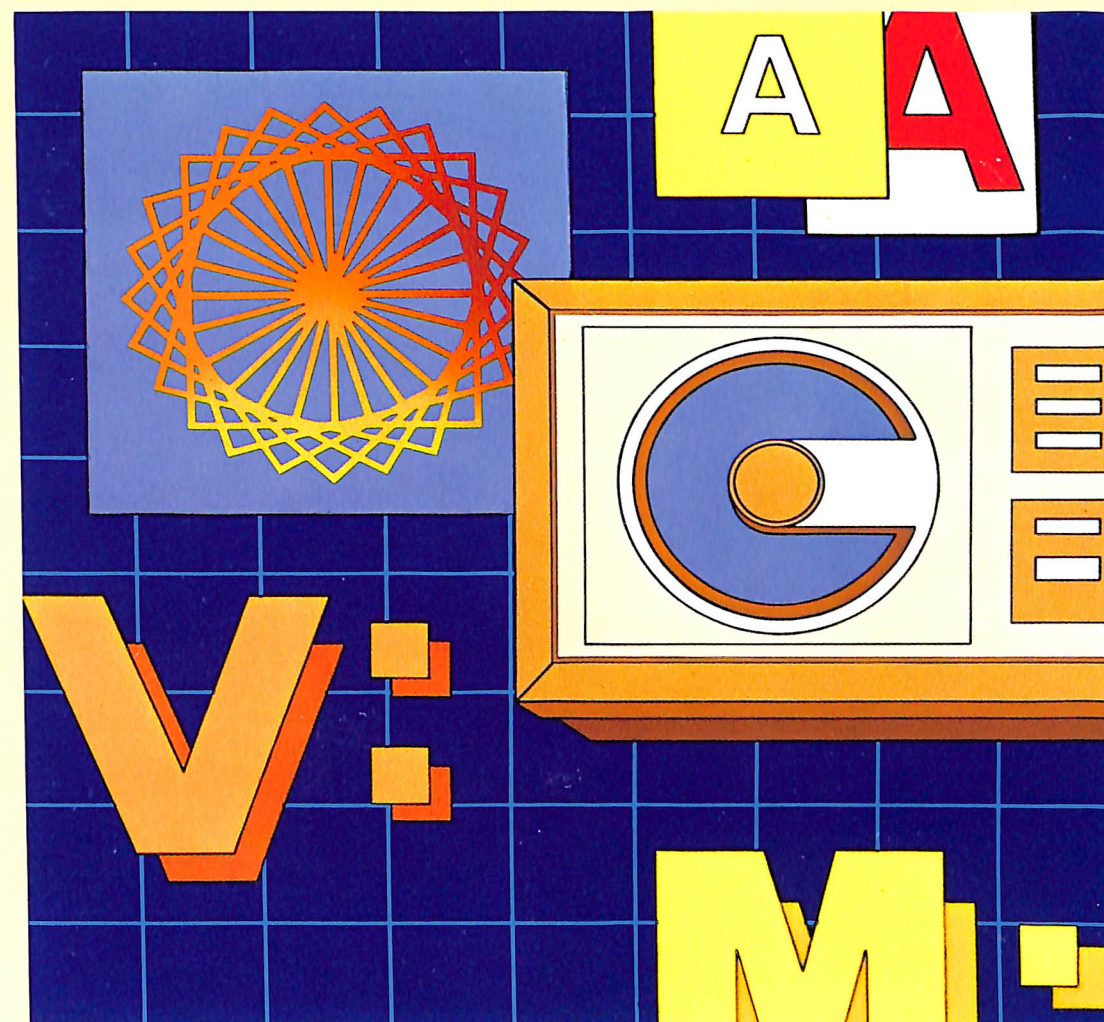




Apple II

# Apple SuperPILOT

Language Reference Manual



20525 Mariani Avenue  
Cupertino, California 95014  
(408) 996-1010  
TLX 171-576

030-0386-A

## **Notice**

---

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

## **Disclaimer of All Warranties And Liabilities**

---

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

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

© 1980, 1982 by Apple Computer, Inc.  
20525 Mariani Avenue  
Cupertino, California 95014  
(408) 996-1010

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

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

Reorder Apple Product Number A3D0051



Apple II

---

# Apple SuperPILOT

---

Language Reference Manual





# Table of Contents

## Preface

vii

- vii Where Do I Start?
- viii How This Manual is Organized

## Chapter 1

### Overview of the Language

1

- 2 What is SuperPILOT?
- 2 Introduction
- 5 Correct Form for Instructions
- 8 The Parts of an Instruction
- 28 General Information
- 29 Running a Programmed Lesson

## Chapter 2

### Text Instructions

35

- 36 R: Remark
- 37 T: Type

## Chapter 3

### Response Instructions

47

- 48 PR: Problem
- 53 A: Accept
- 63 M: Match

## Chapter 4

### Control Instructions

71

- 72 J: Jump
- 74 U: Use

77 E: End  
83 L: Link  
88 XI: Execute Indirect  
91 W: Wait

## Chapter 5

# Computation Instructions

93

94 D: Dimension  
98 C: Compute

## Chapter 6

# Special Effects Instructions

112

114 G: Graphics  
129 GX: Execute Graphics File  
131 TS: Type Specify  
151 TX: Execute Character Set File  
155 S: Sound  
158 SX: Execute Sound File  
160 AP: Accept Point  
165 V: Audio/Visual Device Control

## Chapter 7

# File Handling Instructions

167

168 K: Keeping Student Records  
174 General File Information: FOX: , FIX: , FO: , FI:  
176 FOX: Create and Open New File  
181 FIX: Open Existing File  
184 FO: Output to Open File  
188 FI: Input from Open File

## Chapter 8

# Execution-Time Commands

193

194 Goto Command  
197 @ (Escape) Command  
200 CTRL-C  
200 CTRL-I  
201 RESET



## Chapter 9

### Hints for Beginners

203

- 204 Using the Student's Name
- 206 Using Numbers
- 207 Counting Answers
- 209 An Example

## Chapter 10

### Advanced Programming

213

- 214 Constants
- 215 Variables
- 229 Functions
- 242 Operators
- 246 Expressions

## Appendix A

### ASCII Character Codes

251

- 252 Screen Command Characters
- 255 Screen Modes
- 258 Printing Characters

## Appendix B

### Using More Disk Drives

259

- 260 Diskette Names
- 261 Disk Drive Names
- 262 Lesson Mode
- 265 Author Mode

## Appendix C

### Error Messages

267

- 268 Language Error Messages
- 272 System Execution Errors
- 275 System Failure Errors

## Appendix D

# SuperPILOT Language Summary 277

---

- 278 Expressions
- 282 The Parts of an Instruction
- 284 The SuperPILOT Instruction Set

## Appendix E

# Differences from Apple PILOT 289

---

- 290 Language Extensions
- 292 Operating System Extensions
- 293 Converting Apple Pilot Lessons

## Appendix F

# Apple II Colors 295

---

- 296 Natural Colors and Apple II Colors
- 297 Where Colors Are Displayed on the Screen
- 299 Where the Colors Come From

# Index 303

---



# Preface

This is one of two manuals that come with your SuperPILOT system. The purpose of this Apple II SuperPILOT Language Reference Manual is to describe the SuperPILOT language completely. The Apple II SuperPILOT Editors Manual explains the use of the four editors, which are used for creating SuperPILOT lessons. Both of these manuals provide information that you will find important for successful and efficient lesson-writing. You should therefore familiarize yourself with the contents of each manual, regardless of the amount of previous programming experience or PILOT knowledge you may have.

## Where Do I Start?

---

The answer to this question depends a great deal on where you are, so to speak. Here are some suggestions:

**IF YOU ARE A COMPUTER NOVICE:** Get to know your computer first. Read the manuals that describe how to set up the Apple II, how to use the keyboard, disk drives, and other parts of the system. Both SuperPILOT manuals assume that you know your way around the system.

**IF YOU ARE UNFAMILIAR WITH PILOT:** Start by running the SuperCO-PILOT tutorial diskette. Chapter 1 of the Apple II Editors Manual is a good place to go next. It gives an overview of the lesson-writing process and refers you to demonstration lessons that allow you to see SuperPILOT at work. Then read Chapter 1 of a this language reference manual, which gives you an overview of the language itself. You can then proceed through either manual, or jump back and forth, as you wish.

**IF YOU KNOW PILOT ALREADY:** This manual's Appendices D and E and Appendix D of the Apple II SuperPILOT Editors Manual are for you. They summarize the syntax of the language and describe the differences between SuperPILOT and Apple PILOT. They will save you much time that you might otherwise spend reading the manuals just to find out what's new and/or different.

## How This Manual is Organized

---

The Apple II SuperPILOT Language Reference Manual is designed to serve two different kinds of readers. Some readers, especially those who are new to PILOT or even new to computing, will want to read these chapters in order, building up their knowledge and skill one step at a time. When they become proficient SuperPILOT programmers they will join with a second group of readers, who will use this manual in a "random-access" fashion, as a dictionary or encyclopedia, to look up something they have forgotten or to check on a technical point.

If you are among the first, less experienced, group of readers, you will find that the features of the language are generally presented

beginning with the simplest and progressing to the more difficult. Explanations in the earlier chapters tend to be slower paced, with more frequent reinforcement. Whenever possible, lesson examples avoid using instructions that have not been introduced in that chapter or previous chapters. On the other hand, each example tries to use as much as possible of the language already described in earlier chapters, so the effect is a cumulative one. Finally, each example contains running annotations and many are followed by notes on how the lesson operates. These may be helpful to you in following the logic of the lesson.

If you just need this manual as a reference, however, you should find the information you want easily accessible. Each chapter and major section follows a predictable pattern, beginning with the syntax of the instruction and ending with an example. Programming Notes sections point out special "tricks" or identify common problem areas. Lists and tables have been included wherever they might help to get the answer to you faster. Appendix D and the Quick Reference Card bring all the language elements together in a compact easy-to-use form.

One possible way of approaching the material in a given chapter is to begin at the end. That is, study the example lesson program that is given at the end of each main chapter. By seeing new instructions in the context of those you have already learned, you may cut short the time you need to understand the chapter explanations. But whatever method of study you find best for your needs, you should find that the information in this manual is easy to find and easy to put to use.

Note: Throughout this manual, an extra space has been used to set off the beginning and end of certain words or phrases in a sentence: the names of lessons and variables, actual SuperPILOT instructions, literal text to be typed on the keyboard, etc. This method has been adopted in place of the more conventional method of enclosing the word or phrase in quotation marks, because quotation marks can often be part of the text, thereby causing confusion. For example, in this sentence:

Now type "Hi there!" and press the RETURN key.

it might not be clear whether the quotation marks are part of the sentence or part of the text to be typed. This same sentence would appear in this manual as follows:

Now type Hi there! and press the RETURN key.



## Chapter 1

# Overview of the Language

- 2 What is SuperPILOT?
- 2 Introduction
- 5 Correct Form for Instructions
- 8 The Parts of an Instruction
  - 8 Labels
  - 9 Unlabeled Destinations
- 10 Instruction Names
- 12 Modifiers
- 13 Conditioners
  - 14 Yes and No Conditioners
  - 15 Answer-Count Conditioner
  - 17 Error Conditioner
  - 21 Expression Conditioners
  - 24 Last-Expression Conditioner
- 25 The Colon
  - 25 The Continuation Colon
- 26 The Object
- 27 Variables
- 28 General Information
  - 28 Upper- and Lowercase Letters
  - 29 Instruction Length
  - 29 Blank Lines
- 29 Running a Programmed Lesson
  - 29 Running in Author Mode
  - 30 Running in Lesson Mode
  - 31 Immediate Execution Mode
    - 31 From a Running Lesson
    - 32 From the Editor Menu
    - 32 From Lesson Mode
  - 32 Repeating Instructions
- 33 Execution-Time Commands
  - 33 Goto
  - 33 @ (Escape)
  - 33 CTRL-C
  - 33 CTRL-I
  - 34 RESET

# What is SuperPILOT?

---

Every educator and training director has wished for the ability to provide unlimited, personal attention to each student in a one-to-one interaction tailored specifically to that student's abilities and needs. Computer-Assisted Instruction (sometimes called CAI) is one attempt to help grant that wish. Using a program written by the instructor, a student may receive individualized tutoring, practice, and testing, with frequent interaction between the student and the program.

Computer-assisted teaching lessons can be written using any programming language, but instructors often know much more about their subject matter than about computer programming. The original PILOT language was developed in the early 1970's in an attempt to solve this problem. The language had only about eight instructions and was designed to let an instructor write teaching programs after just a short time learning the language.

Since that time, PILOT has been revised and extended many times. Today, it is both more complex and much more powerful than the original language. The Apple SuperPILOT programming language is based on Apple PILOT, which had its origin in Common PILOT. Common PILOT was developed at Western Washington University. Any program written in Common PILOT or Apple PILOT should run correctly on the Apple SuperPILOT system. Apple SuperPILOT includes the features of both these ancestors, plus many more improvements that make use of the Apple's advanced graphics and sound capabilities and incorporate suggestions and recommendations of Apple PILOT users.

## Introduction

---

The Apple SuperPILOT programming language consists of twenty-six instructions, of which you will use only about nine for writing the largest portions of a lesson. Each instruction consists of two main parts: the instruction itself and its object. Each of the twenty-six instructions themselves is simple, but many of them can operate on a wide variety of objects. Furthermore, you can use modifying elements with an instruction to change its operation, and conditioning elements that determine whether or not the instruction will be executed under various conditions. Finally, you can develop more complicated programs and instructional strategies by combining the instructions into complex sequences.

Most instructions begin with a single-letter instruction name, followed by a colon. For example, one of the most common instructions is the Type instruction, which causes a question or any other text to be



displayed (or "Typed") on the student's screen. The Type instruction is written

T:  
or  
t:

It makes no difference to SuperPILOT if you capitalize the instruction name or type it lowercase: both will be executed the same way. In this manual we will use the lowercase consistently for instruction names in lesson examples because you will most likely discover this to be the simplest and fastest to type.

If the instruction has an object, it follows the instruction name and colon. In our example, the Type instruction's colon would be followed by the text to be displayed:

t:Are you male or female?

Most of the remaining common instructions are designed to let the student "talk" with the program. The Accept instruction tells the computer to "listen" while the student types something on the Apple's keyboard:

a:

The Match instruction then compares the student's response to the word following the Match instruction:

m:female

If the Apple finds the Match word in the student's response, it stores the answer "Yes;" otherwise it stores the answer "No." This answer can then be used as a condition for executing further instructions. For example, if you want the program to skip over one or more instructions and Jump to a program portion labelled women, dealing with health questions for women, you could use the instruction:

j:women

If you want the Jump to occur only when the Apple stores a "Yes" answer to the previous Match instruction, you can add the Yes conditioner to the Jump instruction:

.jy:women

This instruction will be skipped if the Apple stores a "No" answer to the previous Match, and the program will go on to the next instruction. This is only one example of the many ways a simple Apple SuperPILOT instruction can be modified for more complex tasks.

An Apple SuperPILOT program is just a sequence of SuperPILOT instructions. The execution of an Apple SuperPILOT program begins with the first instruction and, unless otherwise directed, continues by

executing the remaining instructions in their order of appearance. Several instructions cause execution to jump out of this normal sequence. Also, you can cause individual instructions to be skipped over if they contain conditions that are not met at the time they are to be executed.

Here is a little program that uses the four most common instructions, plus the End instruction, which ends a lesson. The comments to the right of the program lines (in parentheses) do not appear in a real lesson; we have added them to explain the action caused by each line.

```
t:Let's talk about your health.      (Display text on screen.)

*gender                               (Label for this section,
                                     identified by the *.)
t:Are you male or female?           (Display this text.)
a:                                   (Accept student's response.)
m:female                             (Did student type female ?)
jy:women                             (If Yes, Jump ahead to the
                                     section labelled women .)

m:male                               (Did student type male ?)
jy:men                               (If Yes, Jump ahead to the
                                     section labelled men .)

t:You must be one or the other.     (If the program gets to this
                                     line, student typed neither
                                     female nor male , so
                                     display this text.)

t:Let's try again.                  (Display this text.)
j:gender                             (Jump back to the section
                                     labelled gender .)

*men                                  (Label for men's section.)
t:OK, you are a man.                 (Display this text.)
t:Now, what is your weight?         (Display this text.)
...
...
e:                                   (End of lesson.)

*women                                (Label for women's section.)
t:OK, you are a woman.              (Display this text.)
t:Now, what is your weight?         (Display this text.)
...
...
e:                                   (End of lesson.)
```

The main activity in this program is the evaluation of the Match instructions and the route program execution takes if the Match is successful or unsuccessful. This simplified program contains only two Match possibilities, which could result in an unsuccessful Match even if the student gave an appropriate response, such as "I am a girl." This is because the Apple will store a "Yes" answer to a Match instruction only if the Match word is absolutely identical to a word in the student's last typed response.



For example, if the Match word is female , the Apple will say "Yes" only if it finds the word female in the student's response; a response of Female or FEMALE or FEmale won't do. For this reason, you can tell the Apple II to convert all response letters to capital letters, or all to small letters. See the discussion of the PROblem instruction in the Response Instructions chapter for details on how to make this happen. Most of the examples in this manual include an instruction that tells the Apple II to convert all student responses to capital letters.

On the Apple II keyboard, holding down the "control" key (CTRL) while typing the letter Z causes the next letter to be shifted to uppercase. That is, after a CTRL-Z, the first letter typed will be a capital letter, after which the letters will revert to lowercase. CTRL-A acts like a typewriter's locking shift key. After CTRL-A, all letters typed will be capital letters, until the next CTRL-A shifts back to lowercase. CTRL-Z and CTRL-A affect letters only; the Apple II's SHIFT key works normally for all other characters.

The issue of uppercase or lowercase letters is important for only three commonly used instructions: the text to be displayed by the Type instruction, the text to be compared by the Match instruction, and the program data to be stored on a diskette by the Keep instruction. All the rest of your lesson can be in uppercase, lowercase, or any mixture; SuperPILOT is indifferent. Since lowercase is often easier to read, you may wish to type most of your lesson in lowercase, saving true uppercase for specific capital letters in the text displayed by Type instructions (proper nouns, etc.).

The remaining chapters of this manual explain the Apple SuperPILOT instructions in detail, with small examples showing their use. In this manual, however, we can only hint at the power of the SuperPILOT programming language. More complete examples are provided in the demonstration programs that you received with your Apple SuperPILOT system and on the CO-PILOT tutorial. We also urge you to seek out and study examples of SuperPILOT programs from a variety of authors to learn how the features of the language can be combined. From there, your own imagination will be your guide.

## Correct Form for Instructions

---

Like sentences in the English language, instructions in the Apple SuperPILOT programming language must follow a certain form to be understood correctly. In their simplest form, most Apple SuperPILOT instructions have only three basic elements: the instruction name, a colon, and then the object of the instruction. The object may be text, a calculation, the name of a place to store something, a more detailed command to the Apple II, or it may be missing altogether, depending on the instruction. Here is the general format for an Apple SuperPILOT instruction in its simplest form (square brackets indicate that the object is optional):

```
instruction name:[object]
```

A common example is the simple form of the Type instruction, which causes the entire object to be displayed literally on the student's screen:

```
t:Hello. What is your name?
```

The longest line you can see in the Lesson Text Editor is 39 characters, since the width of the screen is 40 characters and one space is needed to display the cursor when a line is full. To make longer instructions, Apple SuperPILOT lets you "continue" an instruction on subsequent lines (up to 250 characters in all), by starting each continuation line with another colon:

```
t:Four score and seven years ago our
:fathers brought forth, upon this
:continent, a new nation, conceived in
:Liberty and dedicated to the
:proposition that all men are created
:equal.
```

Any SuperPILOT instruction can be extended this way. When the instruction is executed, the Apple just ignores the breaks and extra colons, treating the instruction as one long, unbroken line.

There are two ways to alter the execution of particular instructions. Each requires the addition of another element between the instruction name and the colon. These two elements are called modifiers and conditioners.

A modifier changes some detail of how an instruction works. Most modifiers work on only one or two instructions. For example, the Type instruction normally puts its object text on the screen, and then jumps to the beginning of the next line. If the student then types a response, it appears on that next line. But a Type instruction with the Hang modifier stops ("Hangs") at the end of the displayed text, so the student's response will appear right there, instead of on the next line:

```
th: 7 + 6 =
```

A conditioner causes an instruction to be executed or skipped, depending on a particular internal test. Conditioners can be used with almost all instructions. An instruction used with the Yes conditioner, for example, will be executed only if the previous Match word was found in the student's typed response:

```
ty:Good, that is the correct answer.
```

An expression conditioner is a conditioner that allows you to define your own test. The expression performs a calculation or comparison whose result causes the instruction to be executed or skipped. An instruction used with the expression (  $n > 13$  ), for example, will be

executed only if the number-storage place named `n` contains a number that is greater than thirteen at that moment:

```
t(n>13):No, your answer is too large.
```

In addition to these two kinds of instruction-changers, an element called a label, preceded by an asterisk, can appear alone or before an instruction name. A label identifies the beginning of a program section, and can be used as a destination by instructions that need to Jump to that section:

```
*test
```

```
t:OK, let's try some problems.
```

or

```
*test t:OK, let's try some problems.
```

A little later in this chapter, you will find a detailed discussion of each possible element in an instruction. When more than one of these elements is used in an instruction, the elements must appear in a certain order. Here is the general format showing the correct order for all the possible elements of an Apple SuperPILOT instruction (square brackets surround optional elements):

```
[*label] instruction name[modifiers&conditioners][(expression)]:[object]
```

An instruction need not contain all the possible elements, but any elements included must be in the order specified. Here is another way to see the order of possible elements in an instruction:

<u>Order</u>	<u>Element</u>	<u>Comment</u>
1.	*label	Optional; may be on a separate line.
2.	instruction name	Required; one, two, or three letters.
3.	modifiers	Optional; usually just one; one letter.
	conditioners	Optional; 4 allowed; one character each.
4.	(expression conditioner)	Optional; one allowed; in parentheses.
5.	:	Required colon.
6.	object	Depends on the instruction.
7.	RETURN	Required at the end of every instruction.

Note: In Common PILOT, in Apple PILOT, and in Apple SuperPILOT, any number of modifiers and conditioners may appear in any order (except the expression conditioner, which must appear last), as long as they immediately follow the instruction name. Instructions that do not follow the correct form may be executed anyway, but with unpredictable results.

In the following section, the elements of an Apple SuperPILOT instruction are discussed in detail.

# The Parts of an Instruction

## Labels

A label is a word that identifies an instruction or program section so that other instructions in the lesson can refer to that place. A label is an optional element that may precede any Apple SuperPILOT instruction or stand alone on a line (although such a line is not considered an instruction).

When the label is placed in a program, it usually stands alone on a line. More than one label may appear on a line, separated by at least one space, but no other instruction may precede a label on the same line. An asterisk ( \* ) as the first character of a line (or the first character after an earlier label and a space) indicates to SuperPILOT that the element that follows is a label. The label itself is a name up to thirty-five characters in length, of which only the first six characters are significant. This means that the words `beginner` and `beginning`, both valid as labels, will be considered by SuperPILOT to be the same label: `beginn`. Labels should be meaningful to you, the programmer. Labels such as `startofcarbohydratesprogram` and `3rdwrongspellremedial` impart useful information; incomprehensible labels, such as `label4`, `q`, or that most dreadful scourge, the invisible label (no characters at all) are to be discouraged.

Any printing character may be used for any of the characters in a label name, but a label must not contain spaces in the middle of the name. Spaces between the asterisk and the label are allowed. It does not matter whether you type capital letters or small letters (or any mixture), nor do you have to be consistent in that regard from one use of the label name to the next. If the label is on the same line as an instruction, the label must be separated from the instruction name by at least one space.

For example, you can add a label before the instruction name of a Type instruction in either of these two equivalent ways:

```
*startofprogram t:Hello, what is your name?
```

or

```
*startofprogram  
t:Hello, what is your name?
```

Other instructions can cause program execution to branch to (or "go to") the labelled instruction. When a label name appears in the object field of an instruction to specify the destination for a branching operation, an asterisk does NOT precede the label name. For instance, you might wish the program to Jump to the instruction labelled in the example above:

```
j:startofprogram
```

The instruction `j:STARTOFFPROGRAM` or even `J:StaRto` would also do.



Label names can also be used by both author and student, when the lesson is running, to choose certain sections for testing or review and to skip other sections. One label name has a special meaning in this context: if the Escape option has been enabled by the last PROblem instruction, any response beginning with the character @ will cause an automatic branch to the label sysx . See the chapter Execution-Time Commands for details.

To avoid ambiguity, a particular label name should be placed at only one location within a lesson (but see the Wipe-labels option of the PROblem instruction, in the Response Instructions chapter, for exceptional cases). You will find it helpful to insert labels in your program according to a coherent lesson plan, thought out in advance. This will make the program more readable, helping you and others to understand and modify your program in the future. The following is an example of a possible labelling scheme:

```
*startofprogram
...
...
*introduction
...
...
*beastsofburden
...
...
*steam
...
...
*electricity
...
...
*review
...
...
*test
...
...
*end
```

Apple SuperPILOT stores the lesson's labels (up to 50) and their locations in a reference table (called the label table), so program execution can Jump to those places in the lesson quickly. Because 50 labels is quite a lot for even a large lesson, you will rarely have to think about how SuperPILOT keeps track of your lessons' labels. An explanation of the label table is included in the discussion of the Wipe-labels command of the PROblem instruction in the Response Instructions chapter. Refer to this explanation in the unlikely event that your lesson must include more than 50 labels.

## Unlabeled Destinations

In addition to the instructions that you have labelled in your lessons, certain instructions can be used as destinations for branching

operations, even though they are not explicitly labelled. To specify a branch to one of these instructions, the following notation is used:

<u>Notation</u>	<u>Branch Destination Specified</u>
@A or @a	the last Accept instruction previously executed
@M or @m	the next Match instruction
@P or @p	the next PProblem instruction

These unlabelled relative branch destinations can be used with the instructions Jump, Use, and End, as well as by the Goto execution-time command (if enabled).

Using the unlabelled destinations helps to reduce the number of labels your lesson needs. As up to 50 labels can be used without slowing down your lesson, this is rarely a limitation. However, these destinations can be very useful for other purposes, too. For example, if each independent section of a lesson begins with a PProblem instruction, you can branch from any section to the next section just by using the instruction:

```
j:@p
```

With this system, you can rearrange the sections, remove sections, or add new ones, and this branch will still work. If you were to remove a section identified by a specific label, all instructions that branched to that section by its label name would have to be changed.

Another common use of unlabelled destinations is to write a lesson section that handles any error in a student's typed response. After handling the error, perhaps just by telling the student to "Try again" or "Type a number", the program can jump back to the unlabelled destination @a. This error-handling section can be used by every Accept instruction in a lesson, and it will always return to the last Accept instruction that accepted a response.

When you use this method, however, be aware that repeated failures by the student could cause the original question to be scrolled off the screen. This problem, along with a suggested solution, is discussed in the section explaining the End instruction, in the Control Instructions chapter.

## Instruction Names

The instruction name is a required element; it specifies the particular kind of action to take when the Apple executes that line in the program.

There are only two kinds of lines that do not include an instruction name: a line that consists of a stand-alone label and a line that starts with a colon. When an instruction (such as Type or Remark) is continued beyond a single thirty-nine character line, the subsequent lines of the same instruction begin with a colon. In this case, the colon implies that the new line is really still part of the previous instruction line. When SuperPILOT executes the instruction, it

ignores any such breaks and colons in the instruction line, treating the whole instruction as one long line.

The instruction name is usually just one or two letters, which are usually an abbreviation for the action specified. For example, the instruction name for the Type instruction, which displays or "types" information on the screen, is T.

The instruction name must be separated from any preceding label on the same line by at least one space, so that SuperPILOT can tell where the label ends and the instruction name begins. If there is no preceding label, the instruction name begins the line. There may be any number of spaces between the instruction name and the element that follows it (which may be a modifier, conditioner, expression, or colon).

Here are the Apple SuperPILOT instruction names, presented in the order in which they are discussed in the chapters following this one. (Uppercase letters are used for the instruction names in the table, but you may use--and may prefer--lowercase letters in your programs.)

<u>Chapter Title</u>	<u>Name</u>	<u>Full Name</u>	<u>Description</u>
Text Instructions	R	Remark	Remarks to the author; not executed.
	T	Type	Displays text on the student's screen.
Response Instructions	PR	PRoblem	Controls the handling of typed responses.
	A	Accept	Accepts the student's typed response.
	M	Match	Determines if student typed certain words.
Control Instructions	J	Jump	Starts doing another part of the lesson.
	U	Use	Like Jump, except it "remembers" to return here when the other part is done.
	E	End	Ends a lesson or part.
	L	Link	Starts a new lesson.
	XI	eXecute Indirect	Executes stored words as an instruction.
	W	Wait	Waits for a set time.
Computation Instructions	D	Dimension	Reserves spaces for storing words or groups of numbers.
	C	Compute	Does calculations and stores the results.

Special Effects Instructions	G	Graphics	Draws lines and dots on the screen.
	GX	eXecute Graphics	Draws a diskette-stored image on the screen.
	TS	Type Specify	Allows you to determine how text will appear.
	TX	eXecute Characters	Changes the character set used for text displays.
	S	Sound	Plays musical notes on Apple's speaker.
	SX	eXecute Sound	Plays a diskette-stored sound effect.
	AP	Accept Point	Accepts the student's pointing response.
File Handling Instructions	V	Video control	Communicates with video devices, like videotape and videodisk, as well as other devices, like speech synthesizers and modems.
	K	Keep	Captures program data and stores it in a special file.
	FOX	Open New File	Creates and opens a new diskette file.
	FIX	Open Old File	Opens an existing file on the diskette.
	FO	File Output	Stores information in the open file.
	FI	File Input	Retrieves information from the open file.

## Modifiers

A modifier is a one-letter optional element that changes some detail of the way an instruction works. Any number of modifiers, in any order, may be used with any particular instruction, but it rarely makes sense to use more than one modifier. The preceding element may be the instruction name, another modifier, or a conditioner. The element that follows may be another modifier, a conditioner, an expression, or a colon.

Each modifier acts successfully on only one kind of instruction name and will be ignored if it is used with other instruction names. For example, after a normal Type instruction displays its text, the cursor automatically jumps to the beginning of the next line. To make the cursor "hang" at the end of the displayed text (so the student's response will appear on the same line) you can add the H modifier to the Type instruction name:

th: 5 + 13 =



The H modifier has no effect when used with any other kind of instruction name.

SuperPILOT has eight special modifiers for use with its instructions, each of which will work with only one type of instruction. Some of them have the same first letter and therefore look the same when you use them, but their meanings depend entirely upon the instruction names they modify. The modifiers, appended to the names of the instructions they affect, are:

th:	(Hang modifies Type)	Stops carriage return after a Type.
ax:	(eXact modifies Accept)	Accepts exact response, with no editing.
as:	(Single modifies Accept)	Accepts a single-character response.
ms:	(Spell modifies Match)	Allows one-letter misspelling on a Match.
mj:	(Jump modifies Match)	Jumps to the next Match if this Match is unsuccessful.
lx:	(Erase modifies Link)	Starts a new lesson without preserving the old lesson's variables.
lp:	(Pascal modifies Link)	Starts a new lesson, written in Pascal.)
ks:	(Save modifies Keep)	Causes an immediate updating of the special recordkeeping file.

In a few cases, you can use more than one modifier sensibly. For instance, the instruction:

```
msj:APPLE
```

looks for the word APPLE in the student's response, allowing certain kinds of misspelling, and also jumps ahead to the next Match instruction if the Match word is not found in the response. However, the nonsense instruction

```
aspx:
```

functions just like an ap: instruction.

For examples and more details about the various modifiers, see the discussions of Text, Accept, Match, Link, and Keep, in the chapters following this one.

## Conditioners

A conditioner is an optional element; it establishes a condition that must be true or the rest of the instruction will not be executed. This is a very important point to consider when testing your lessons: unless you evaluate how the lesson will operate under both true and false conditions for each conditioner, you may not discover errors in your instructions because those instructions were never executed during testing.

An expression may be used as a conditioner; all other conditioners are either a single letter or a one- or two-digit number. The instruction name must immediately precede the list of modifiers and conditioners, with no intervening spaces.

Any number of conditioners, in any combination and in any order, may be used in an Apple SuperPILOT instruction, except that the expression conditioner (if one is used) must be the last in the list. However, a maximum of five conditioners (one each of the five different kinds) may be usefully employed in one instruction. If more than one conditioner is used in an instruction, the instruction will be executed only if all of the specified conditions are met at execution time. Unlike modifiers, which work only with certain instructions, any conditioner can be used effectively with any Apple SuperPILOT instruction.

There are five different kinds of conditioners, each of which can determine under what conditions an instruction will be executed. They are the Yes-No conditioner, the Answer-Count conditioner, the Error conditioner, the Expression conditioner, and the Last-Expression conditioner. The form of each is given in the table below:

<u>As Typed</u>	<u>Name</u>	<u>Condition Established</u>
Y	(Yes conditioner)	Execute if last Match was successful.
N	(No conditioner)	Execute if last Match was unsuccessful.
1 to 99	(Answer-Count conditioner)	Execute if the conditioner number equals the number in the Answer-Count, which counts how many times the student has tried to answer the same question.
E	(Error conditioner)	Execute if the Error Flag has been raised by trouble in executing any instruction.
(expr)	(Expression conditioner)	Execute if the value of the expression is true (non-zero).
C	(Last-Expression conditioner)	Execute if the last evaluated instruction-modifying expression was True (non-zero).

In the following sections, each of these conditioners is discussed in more detail.

## Yes and No Conditioners

The Yes conditioner is always Y, and the No conditioner is always N. These conditioners test the success of the last Match instruction executed. The Y conditioner allows an instruction to be executed only if the previous Match was successful (a "Yes" result); that is, the Match word was found somewhere in the student's response. The N conditioner allows an instruction to be executed only if the previous Match was not successful (a "No" result).

These conditioners are central to the basic interaction between an Apple SuperPILOT lesson and the student. They let the program respond differently, depending on the student's responses.

Example:

*test	(Label for this section.)
t:What is the highest	(Display this text,
: mountain in the world?	and this text.)
a:	(Accept student's response.)
m:Everest!everest	(Did student type Everest
	or everest ?)
ty:That's right.	(If Yes, display this text.)
tn:No, try again.	(If No, display this text.)
jn:test	(If No, Jump back to test .)

Example:

*day	(Label for this section.)
t:Is today sunny or cloudy?	(Display this text.)
a:	(Accept student's response.)
m:sunny	(Did student type sunny ?)
ty:How nice!	(If Yes, display this text.)
jy:next	(If Yes, Jump to label next .)
m:cloudy	(Did student type cloudy ?)
ty:Too bad.	(If Yes, display this text.)
tn:I don't understand.	(If No, display this text
: Please try again.	and also this text.)
jn:day	(If No, Jump back to day .)
*next	(Label for next section.)

## Answer-Count Conditioner

This conditioner is always a number from 1 through 99. It tests the current value of the Answer-Count. The Answer-Count indicates how many times in a row the most recent Accept instruction has been encountered, with no other Accept instruction intervening. It allows an instruction to be executed only if the conditioner number is the same as the current value of the Answer-Count. (If more than two numeric digits appear in the list of modifiers and conditioners for an instruction, only the rightmost two digits are used.)

This conditioner lets the program respond differently, depending on how many times the student has attempted to answer the same question. On the third try, for instance, you might want to provide the answer to the question, or you might branch to a review section. Without this conditioner, as the examples in the previous section show, the student could be stuck on one question indefinitely. For more information about the Answer-Count, see the Accept instruction, in the Response Instructions chapter, or the section on system variables in the Advanced Programming chapter.

Example:

*test	(Label for this section.)
t:What is the highest	(Display this text,
: mountain in the world?	and this text.)
a:	(Accept student's response.)
m:Everest!everest	(Did student type Everest
	or everest ?)
ty:That's right.	(If Yes, display this text.)
tyl:You got it the first try!	(If Yes and first answer,
	display this text.)
tn4:No, Mount Everest is the	(If No and fourth answer,
: world's highest mountain.	display this text.)
jn4:next	(If No and fourth answer,
	Jump to label next .)
tn:No, try again.	(If No, display this text.)
jn:test	(If No, Jump back to test .)
*next	(Label for next section.)

In the example above, the student will be told to "try again" after the first three wrong answers. If the student's fourth answer is still wrong, the program shows the correct answer and goes on to the next question ( \*next ).

Example:

*dukbil	(Label for this section.)
t:What mammal has a bill	(Display this text,
: and lays eggs?	and this text.)
a:	(Accept student's response.)
m:platypus	(Did student type platypus ?)
tyl:Right, on the first try!	(If Yes and answer #1,
	display this text.)
ty2:Right, on the second try.	(If Yes and answer #2,
	display this text.)
jy:koala	(If Yes, Jump to koala .)
tl:No, try again. Hint:	(If wrong answer #1, display
: it lives in Australia.	this text.)
jl:dukbil	(If wrong answer #1, Jump
	back to label test .)
t:I think you should	(Second wrong answer:
: review the following.	display this text, and
j:review	Jump to label review .)
*koala	(Label for next section.)

The Answer-Count is kept in the system variable (one of Apple SuperPILOT's number-storage places) named %a . For a different kind of example, which tests the %a Answer-Count in an expression, see the Expression Conditioners section later in this chapter.

## Error Conditioner

This conditioner is always an E . It tests the current state of the Error Flag. The Error Flag is an indicator in the Apple II that is "raised" any time Apple SuperPILOT has trouble executing one of your instructions. The error conditioner allows an instruction to be executed only if the Error Flag is currently raised. This lets you write instructions designed to handle certain predictable errors that may arise when the lesson is running.

When you run a lesson from the Lesson Text Editor in Author Mode, most problems with instructions cause error messages to be displayed on the screen (the section Running a Programmed Lesson, later in this chapter, discusses error messages). These problems, which may be the result of a mis-typed instruction, a division by zero in a calculation, etc., also raise the Error Flag at the same time. When the student is running a lesson in Lesson Mode, however, no error messages are displayed: the Error Flag being raised is the only indication that an instruction has run into difficulty.

In addition to the many kinds of problems that cause a message to be displayed (in Author Mode) while raising the Error Flag (in either mode), there are two situations that raise the Error Flag without causing an error message in either Author Mode or Lesson Mode:

1. If an Accept instruction is supposed to store a student's typed number in a specified number-storage place, but no number is found in the student's response, no new number is stored in the number-storage place, and the Error Flag is raised. For example, this instruction:

a:#n

(Accept the student's response  
and store the first typed  
number in the number-storage  
place named n.)

will raise the Error Flag if the student types only letters, or presses the RETURN key without typing anything. The value of n will not be changed from what it was before the a:#n instruction was executed. To avoid this undesirable situation, you should always test for the Error Flag immediately after any Accept instruction that expects a numeric response.

2. If a FIX: (open an existing diskette file) instruction does not find a file with the specified name already stored on the diskette, no file is opened, and the Error Flag is raised. For example, this instruction:



fix:lØ,s\$

(Find an existing diskette file whose name is the word stored in the word-storage place named s\$, and open it for storing and retrieving data in records numbered Ø-1Ø.)

will raise the Error Flag if it finds no file with the given file name.

The Error Flag is always down when a lesson starts, and stays down until something raises the flag. Once the Error Flag has been raised, it stays up until something lowers it. There are many kinds of problems that can cause the Error Flag to be raised, but only five things can lower the Error Flag while a lesson is running:

1. The Error Flag is lowered just before evaluating any mathematical or string expression. Thus, successfully evaluating any expression conditioner (discussed later in this chapter) or any expression appearing in an instruction's object field lowers the Error Flag. For example, either of these instructions will lower the Error Flag if it is up:

t(n=12):That's right.

(If the execution-modifying expression (n=12) is true, display this text.)

c:x=5+3

(Evaluate the expression 5+3 and store the result in the number-storage place x.)

However, this instruction lowers and then re-raises the Error Flag:

c:x=5/Ø

(The illegal expression 5/Ø causes an error message in Author Mode and raises the Error Flag in Author Mode and Lesson Mode.)

2. The Error Flag is lowered just before executing the contents of the word-storage place whose name appears in the object field of an XI: instruction. Thus, successfully executing an XI: instruction lowers the Error Flag. For example, this instruction will lower the Error Flag if the word-storage place a\$ contains a correct, executable SuperPILOT instruction:

xi:a\$

(Execute the character string stored in the word-storage place a\$ as a SuperPILOT instruction.)

3. If an Accept instruction successfully stores a student's typed number in a specified number-storage place, the Error Flag is lowered. For example, this instruction will lower the Error Flag if the student's typed response includes a number:

a:#n

(Accept the student's response, and store the response's first number in the number-storage place n.)

4. Successfully executing any PProblem instruction lowers the Error Flag. For example:

pr:

(Mark a new section without changing any options.)

5. Any LX: instruction, which starts a new lesson running, forgetting everything about the old lesson, lowers the Error Flag. For example:

lx:lesson2

(Start running the lesson named lesson2, without keeping old stored words and numbers.)

Flag-lowering item number 1 provides a way to lower the Error Flag intentionally whenever you want to do so. For example, consider the following instructions:

t:What is your first name?

(Display this text.)

a:\$n\$

(Accept student's response; store it in string variable n\$. The first dollar sign is required by SuperPILOT.)  
(Try to open an existing disk file whose name is the name stored in n\$, for storing and retrieving records 0 through 10.)

fix:10,n\$

te:Please type your first name  
: exactly as it appears on the  
: class roster.

(If the Error Flag is up, no old file was found that had the name typed by student, so display this text.)

je(1):@a

(If the Error Flag is up, lower it by evaluating the expression (1), then Jump back to the last Accept instruction.)

The simple expression (1) in the last instruction lowers the Error Flag before execution Jumps back to the Accept instruction. If the Error Flag were allowed to remain up, the last two instructions would

be repeated over and over, even if the student gave a valid name on a subsequent try.

When you use the error conditioner to force a numeric response to an a:#n instruction, you normally will not need to lower the Error Flag if you Jump back to the same Accept instruction:

t:How old are you?	(Display this text.)
a:#n	(Accept student's response; store number in number- storage place n .)
te:Please type a number.	(If Error Flag is up, student did not type any number, so display this text.)
je:@a	(If Error Flag is up, Jump back to last Accept.)

When the student correctly types a numeric response, the Error Flag is automatically lowered.

Most of the other problems that raise the Error Flag can be traced to incorrectly typed instructions or other errors that you clear up in the testing process, while running the lesson from Author Mode.

Example:

d:q\$(10)	(Create a word-storage place named q\$ , for storing words up to 10 characters long.)
t:Please type your first name.	(Display this text.)
a:\$q\$	(Accept student's response and store it in the word-storage place named q\$ .)
fix:9,q\$	(Try to open an existing disk file whose name is the name stored in q\$ , for storing and retrieving records 0 through 9.)
foxe(1):9,q\$	(If the Error Flag is up, no old file was found that had the same name as the student, so create and open a new 10-record disk file with the name stored in q\$ , after lowering the Error Flag.)
*main	(Label for main lesson.)

The example above attempts to open a previously-stored file named after the student. If no previous file by that name is found, a new file by that name is created.

## Expression Conditioners

The expression conditioner is an optional element that performs a calculation or makes an assertion. It causes the instruction to be executed or skipped, depending on the result of the calculation or the truth of the assertion. If the expression, when evaluated, is true (non-zero), the instruction can be executed. If the evaluated expression is false (zero), the instruction is skipped.

The expression conditioner is a single mathematical calculation or a single assertion, enclosed in parentheses. Only one expression may modify an instruction, but that expression can contain any valid combination of sub-expressions. You will have to read the Advanced Programming chapter to discover all the rules and all the powers of expressions. Any expression can be used with any Apple SuperPILOT instruction.

The instruction name, its list of modifiers, and any other conditioners must immediately precede the expression conditioner. The next non-space character after the expression must be the colon. Spaces are permitted before, after, and within the expression's parentheses.

Any other conditioners used must precede the expression conditioner and are tested first. If any previous conditioner causes the instruction to be skipped, the expression is NOT EVALUATED. In such a case, when a subsequent instruction uses the C (last-expression) conditioner, it will not refer to the expression, but to the most recent expression conditioner that WAS evaluated.

The Error flag is automatically lowered at any time that SuperPILOT encounters an expression. If, during the subsequent evaluation of that expression, SuperPILOT encounters an error, the Error flag will be raised again. The error must be mathematical or syntactical: while the expression  $(3 > 135)$  may not be true, it is not a mathematical error, whereas  $(3/\emptyset)$  is.

Even a simple expression such as  $(1)$  or  $(\emptyset)$  can be used to lower the Error Flag. The existence of the expression in the instruction lowers the Error Flag, and since the expression contains no mathematical or syntactical error, the Flag will remain down.

Another way to think of the expression conditioner is as an assertion that, if true, allows the instruction to be executed. If the assertion is false, the instruction is skipped. Here are some sample assertion expressions:

<u>Expression</u>	<u>Assertion</u>
(5<2)	"5 is less than 2."
(n>4)	"The number now stored in the number-storage place named n is greater than 4."
(x<>2)	"The number now stored in the number-storage place named x is not equal to 2."
(a&b) (a and b)	"The number stored in number-storage place a is not zero and the number stored in b is also not zero."
(a!b) (a or b)	"The number stored in number-storage place a is not zero, or the number stored in b is not zero, or both are not zero."
(^p) (not p)	"The number stored in number-storage place p is not non-zero." (The expression is true if p=∅.)
((%a<4)&(x=y))	"The assertion 'the number in %a is less than 4' is true and the assertion 'the number in x equals the number in y' is also true."

**Example:**

t(x>14):That's too large.

(If the assertion "the number stored in number-storage place named x is greater than 14" is true, display this text.)

An expression conditioner is eventually reduced to a number. If that number is not zero, the instruction is executed. If the number is zero, the instruction is skipped. Here are some sample calculation expressions:

<u>Expression</u>	<u>Resulting Number</u>
(7)	7
(12-(2*6))	∅
(q-13)	13 subtracted from the number now stored in the number-storage place named q .
((3*a)-(4/b))	4 divided by the number in b , subtracted from 3 times the number stored in a .

Example:

t(g+5):No, the answer is -5 .

(If the result of adding 5 to the number stored in number-storage place named g is not zero, display this text.)

The rules for writing expressions are discussed fully in the Advanced Programming chapter.

Example:

Note: This example uses the C (last-expression) conditioner, which is discussed in the section following the example.

\*additionexam

t:How much is 5 plus 7 ?

a:#n

(Label for this section.)

(Display this text.)

(Accept student's response; store the response's first number in storage-place n .)

te:Please type a number.

(If the Error Flag is up, no number was typed, so display text.)

je:@a

(If Error Flag is up, Jump back to the last Accept instruction executed.)

t(n=12):Yes, 12 is correct.

(If the assertion "the number in n equals 12" is true, display this text.)

jc:subtractionexam

(If the assertion "the number in n equals 12" is true, Jump to subtractionexam.)

t(n>12):No, #n is too large.

(If the assertion "the number in n is greater than 12" is true, display this text, substituting the number in n for #n and the space that immediately follows.)

t(n<12):No, #n is too little.

(If the assertion "the number in n is less than 12" is true, display this text, substituting the number in n for #n and the space that immediately follows.)

t(%a<3):Please, try again.

(If the assertion "the Answer Count in %a is less than 3" is true, display text.)

jc:additionexam

(If the assertion "the Answer Count in %a is less than 3" is true, Jump back to additionexam.)



t:You need some review.  
j:review

(Third wrong answer: display  
text and Jump to review.)

\*subtractionexam

(Label for next section.)

When the two middle Type instructions are executed, SuperPILOT replaces the symbol #n , including the first space after, with the number stored in the number-storage place named n . See the discussion of the Type instruction, in the Text Instructions chapter, for details.

## Last-Expression Conditioner

This conditioner is always a C (do not confuse this with the name of the Compute instruction, which is also C ). It tests whether the last evaluated instruction-modifying expression was True (non-zero) or False (zero). It acts as an exact substitute for the expression, except that it need not immediately precede the colon; instead, it may appear before modifiers and other conditioners in the same instruction.

The use of the last-expression conditioner simplifies the use of expressions as instruction conditioners. Where the same expression conditioner applies to several instructions in a series, the C conditioner can be substituted for the expression in all the instructions after the first one. Only the original expression is actually evaluated; the result of that evaluation is then applied automatically to the subsequent C conditioned instructions. An instruction using the C conditioner will be executed only if the most recently evaluated instruction-modifying expression (in a previous instruction) was True (non-zero). See the Expressions section next in this chapter, and the Advanced Programming chapter for more examples of expressions.

Note: The C conditioner refers to the last expression conditioner that was actually evaluated. If an instruction is not executed (because of a conditioner or a previous Jump instruction, for instance), any expression conditioner in that instruction is NOT EVALUATED. So a subsequent C conditioner will NOT refer to that instruction's expression.

Example:

*state	(Label for this section.)
t:Which state has the longest days?	(Display this text.)
a:	(Accept student's response.)
m:Alaska!alaska	(Did student type Alaska or alaska ?)
ty:Yes, that's right.	(If Yes, display this text.)
jy:next	(If Yes, Jump to next.)
m(%a<3):Washington!Idaho!Montana	(If the Answer Count stored in the number-storage place named %a is less than 3,
!:Dakota!Minnesota	did student type any of these words?)

tyc:Good guess, but it's	(If Yes and Answer Count is
: still farther north.	less than 3, display text.)
tnc:No, it's farther north.	(If No and Answer Count is
	less than 3, display text.)
tc:Try again.	(If Answer Count %a is less
	than 3, display this text.)
jc:state	(If Answer Count is less than
	3, Jump back to state .)
t:No, Alaska has the longest days.	(Third bad answer: display
	text and then go on.)
*next	(Label for next section.)
t:Some days the sun does not set	(Display this text.)
: at all in northern Alaska.	

The example above tells the student to "try again" after the first two wrong answers, and provides the correct answer after the third wrong answer. Remember that %a is the name of the number-storage place where the Apple II stores the current Answer Count. In the Match instructions, the or symbol ( ! ) separates multiple Match words (see the Match instruction for details).

## The Colon

Every instruction line must contain a colon. It is usually preceded by the instruction name, list of modifiers and conditioners (if any), and expression conditioner (if any). Most instructions ignore spaces between the colon and the object (if there is an object), but the Type instruction will display those spaces.

## The Continuation Colon

If the colon is the first non-space character on a line, the line is a continuation of the previous instruction, and also continues that instruction's modifiers, conditioners, and expression (if any).

Note: Do not allow a blank line (by pressing RETURN twice in a row) to separate a continuation of the instruction from the previous portion of that instruction.

The Lesson Text Editor automatically breaks up any instruction that exceeds 39 characters as you type it in, and continues it (beginning with the continuation colon) on the next line. You can continue any instruction, or add another piece in the middle, by adding lines that begin with continuation colons. When the Apple executes an instruction continued on extra lines beyond the line starting with the instruction name, it ignores the breaks and the continuation colons, treating the instruction as if it were written on a single long line.

Only the first 250 characters of an instruction are used when that instruction is executed. This is about six full lines of text. (The continuation colons are not counted in the 250-character limit, but all other characters are counted.) If any portion of an instruction extends beyond the 250-character limit, that portion of the instruction

is ignored. You will likely want to display more than 250 characters of continuous text in some of your programs, however. This can be done by dividing the text into separate T: instructions of less than 250 characters each. See the Text Instructions chapter for further explanation.

Remark instructions can be continued indefinitely, because they are not executed.

Example:

```
r:THE FOLLOWING EXAMPLES
: SHOW SOME OF THE USES
: AND EFFECTS OF THE
: CONTINUATION COLON.
```

(Remark instruction is not executed, so it can be continued indefinitely.)

```
ty2:Now is the thyme for all
: good beadles to come to
: the aid of their crumhorn,
: and lilacs to graze on the
: summer's daze, and bats to
: their belfries fly, come morn.
```

(If conditioners allow this instruction to be executed, the words in these lines of text will be rearranged into a solid paragraph 40 characters wide on the student's screen. Type treats extended text as one long line, ignoring the continuation colons; 250 characters maximum.)

```
m:Mercury!Venus!Earth!Ma
:rs!Jupiter!Saturn!Uranu
:s!Neptune!Pluto
```

(Match treats extended list as one long line, ignoring the continuation colons; 250 characters maximum.)

```
g:v0,39,10,10;es0;c6;p20,2
:0;d540,20;d540,200;d20,20
:0;d20,20;m200,40;c9;d200,
:21;p30,30;c5;p40,40
```

(Graphics treats extended list as one long line, ignoring the continuation colons; 250 characters maximum.)

```
s:12,12;2,12;8,48;2,12;8,
:12;12,48;2,12;8,12;12,12
:;2,12;8,12;12,12;2,12;8,
:12;12,48
```

(Sound treats an extended list as one long line, ignoring the continuation colons; 250 characters maximum.)

The examples above have been formatted somewhat arbitrarily, to fit on this page with our comments. The lesson Text Editor will automatically break your instructions as you type them, to stay within 39 character spaces.

## The Object

Not every instruction has an object, and the object is often an optional element. It must follow the instruction's colon. Most instructions ignore spaces between the colon and the object, but Type will display all such spaces.

The object may be text, a number, the name of a word-storage place, the name of a number-storage place, a diskette file name, the name of a place in the lesson to jump to, a list of more detailed commands to carry out, control-options that set how things work in SuperPILOT, etc., depending on the instruction.

## Variables

Variables will be an essential part of virtually every SuperPILOT lesson you write. Without them, your lessons would have to rely on fixed routines with little variety or student interaction. So understanding how and where variables can be used will greatly increase your lesson-writing power.

A variable represents a place in the Apple's memory where a number or string of characters is stored. It is called a variable because you can vary, or change, the numbers or words in it whenever you wish.

Variables are very useful. You give one a name, store a number or word in it, and from then on you can use the name of that variable in place of the number or word stored there. If you store a different value in that variable, the new value is used from then on, everywhere that variable's name appears.

The names you give to variables are always one letter or one letter followed by an integer from 0 through 9. Up to 50 different variable names may be used in a lesson.

Several different kinds of variables may be used. Each of the different kinds can be classified as a numeric variable or a string variable or a system variable.

The most commonly used numeric variable stores just one number and is called a simple numeric variable. It has a name like `n`, or `bl`, or `x`, or `t3`. You can create one by storing a number in it with a Compute instruction such as `c:t3=5` or by having the program store the user's response in it with an Accept instruction such as `a:#t3` (the equals sign in `t3=5` means, "is assigned the value." So a statement like `t3=t3+27` means, "take the current value in `t3`, add 27 to it, then assign the new resultant value to the storage location `t3`.")

For some purposes, it may be handy to have a single name refer to a contiguous sequence of number-storage places. For example, in a course about our solar system, you might create an array of nine number-storage places, one per planet, and hold within each the gravity of its planet, relative to the earth. This is called a numeric variable array. It has a name like `y`, `p7`, or `w2`. A subscript number (or pair of numbers) will be attached to the name of a numeric variable array to indicate which of the number-storage places in that set you need at the moment. These individual elements of the array have names such as `y(3)`, `p(4,5)`, or `w2(13)`. You use a Dimension instruction such as `d:y(18)` to create a numeric-variable array and reserve enough space for all of its elements.

A variable that stores a word (or any collection or string of characters) is called a string variable. A string-variable name always ends with a dollar sign, such as g\$ or r5\$. You use a Dimension instruction such as d:s\$(30) to create a string variable and reserve enough space for the string with the most number of characters you expect to store there. (In the above example, 30 characters.) It is also possible to use parts of a string variable; these parts are called substring variables (see the Advanced Programming chapter).

Finally, there are number-storage and word-storage places created and used by Apple SuperPILOT itself, but accessible to you, too. These places are called system variables. There are ten of these: %a (the Answer-Count), %b (stores the last response), %x and %y (the graphics cursor coordinates, %c and %r (the text cursor coordinates), %s (stores the current relative angle), %o, %v, and %w, used in connection with external devices, such as touch screens, video disk players, and video cassette recorders. You don't usually store things in the system variables yourself, but you can use what Apple SuperPILOT has stored there. A description of each of the system variables may be found in the Advanced Programming chapter. Further uses of %a and %b are explained in the discussion of the Accept instruction in the Response Instructions chapter.

## General Information

---

### Upper- and Lowercase Letters

Apple SuperPILOT does not care whether the letters you type in a lesson are uppercase (capitals), lowercase, or any mixture, with the following exceptions:

The Type instruction's object text must be typed exactly as you want it to be displayed on the student's screen.

The Keep instruction's object text must be typed exactly as you want it to be stored in the recordkeeping file.

The Match instruction's object text must be typed exactly as you want it to be compared to the student's Accepted response. See the PProblem and Accept instructions for details about converting all response letters to uppercase or all to lowercase, for easier Matching.

Remember, CTRL-Z shifts only the next letter to uppercase (all future letters will be lowercase); CTRL-A changes all subsequent letters from uppercase to lowercase, or vice versa. Since the actual instructions of the lesson are seen only in the Lesson Text Editor, you may wish to type everything in lowercase except an occasional capital letter in a Type, Keep, or Match instruction's object text.

## Instruction Length

The Lesson Text Editor automatically breaks up any instruction that exceeds 39 characters as you type it and then continues it (beginning with the continuation colon) on the next line. These breaks and continuation colons are ignored by SuperPILOT when executing the instruction, and the instruction is treated as if it were written on one long line.

In the Lesson Text Editor, nothing stops you from continuing an instruction for dozens of lines. When an instruction is executed, however, only the first 250 characters of an instruction are used. Any part of an instruction beyond the 250th character is simply ignored. To display continuous text of more than 250 characters, break it up into consecutive T: instructions of less than 250 characters each. See the Text Instructions chapter for further explanation.

The Remark instruction is an exception. You may safely continue Remark instructions as long as you wish, because they are not executed. All other instructions are limited to 250 characters total (a little over six lines), not counting the continuation colons themselves, which are ignored during execution.

## Blank Lines

Blank lines, containing no instruction or continuation colon, are acceptable between different instructions. These can help make a lesson more readable when you look at it in the Lesson Text Editor. However, do not put blank lines between one portion of an instruction and a continuation colon of the same instruction.

## Running a Programmed Lesson

---

Once you have written an Apple SuperPILOT lesson, you will want to test it and you will want your students to use it. There are two ways to run a programmed lesson: from Author Mode and from Lesson Mode.

### Running in Author Mode

When you are testing and working on a new lesson, you will want to run the lesson from Author Mode. In this mode, error messages are displayed on the screen along with the offending line from your lesson. This helps you find problems such as mistyped instructions, or numbers that are too small or too large. Later, when the student runs the lesson in Lesson Mode, the messages are not put on the screen. (See the Error Messages appendix for an explanation of the various error messages.)

To run a lesson from Author Mode, you select the Run option from the menu of the Lesson Text Editor, and then type the name of the lesson you wish to run. This option runs the specified lesson on the Lesson diskette in drive 2. When the lesson ends, the system restarts itself, and returns you to the Main Menu of Author Mode.

For testing, you may wish to insert a few lines at the end of the lesson that allow you to repeat the lesson over again. For example:

```
t:Type "again" to repeat lesson.      (Display this text.)
a:                                     (Accept your response.)
m:AGAIN!again                          (Did you type AGAIN or
                                         again ?)
jy:start                                (If Yes, Jump back to the label
                                         start , near top of lesson.)
e:                                       (No, so end the lesson.)
```

You may also wish to use the PProblem instruction to select one or both of the execution-time command options, Goto and Escape. These commands let you jump to known portions of your program during execution. See the section Execution-Time Commands later in this chapter.

## Running in Lesson Mode

When your student uses the lesson, the system is started up--or booted, to use the computer term--in Lesson Mode, with the Lesson diskette in drive 1. If you have a lesson called hello on the diskette, that lesson will be run immediately. If not, a menu of the available lessons will be presented to the student on the diskette's title page, and the student will be allowed to select one of the lessons.

If you do not want the student to choose from all of the lessons on the diskette, you can name a lesson hello and structure that lesson in several ways. First, you could make the hello lesson simply Link to a different lesson on the diskette. For example, this one line instruction in a hello lesson:

```
l:butterfly
```

would cause the lesson called butterfly to be the first lesson the student sees when the system is booted in Lesson Mode. Of course, you could accomplish the same thing by saving the butterfly lesson under the name hello when you finish writing it.

Second, you could use the hello lesson to display a menu of lessons for the student to select from. This option can be handy in making certain lessons on the diskette unavailable to the student (because he or she doesn't know their names). It can also allow you to include the names of lessons on another diskette, if you have a multiple-drive student system, particularly if you are using a non-bootable Resource diskette.

The hello lesson has the power to turn the Apple II into a turnkey system. With the lesson diskette in the primary drive, the student has to know only how to turn on the computer (like turning a key in a lock) to make a program run. The student does not have to make any lesson selections or interact with SuperPILOT in any way. Through the lesson you have named hello, you have complete control over what the student does.



For further information on the hello lesson, see the discussion of the Link instruction in the Control Instructions chapter. See the appendix Using More Disk Drives for information on multiple-drive systems and Resource diskettes.

Before giving the new lesson to your students, you may wish to remove any special testing aids, such as the lines shown above, which let you repeat the lesson. You may also wish to change any PProblem instruction that lets you use the execution-time commands Goto and Escape during testing.

The error messages displayed on the screen when running a lesson from Author Mode are not displayed in Lesson Mode. Any problem that would cause an error message and raise the Error Flag in Author Mode raises the Error Flag without any message in Lesson Mode. Your instructions can use the Error conditioner to test the state of the Error Flag at any point. This lets your lesson's error-handling instructions take care of correcting problems that arise in the course of a normal "conversation" with the lesson.

## Immediate Execution Mode

SuperPILOT has an Immediate Execution Mode so that you may type in instructions directly from the keyboard and have them carried out at once. Immediate Mode is available to you in Author Mode when the Lesson Text Editor Menu is on the screen and also when you are running a lesson. It is also available in Lesson Mode, under certain conditions.

### From a Running Lesson

When test-running a lesson in Author Mode, you can enter Immediate Execution Mode by typing a CTRL-I (type I while you are holding down the CTRL key) in response to any request for input (any Accept instruction, except an Accept Point instruction). The immediate prompt, a "greater-than" sign ( > ), will appear (assuming you have not changed the standard ASCII character set). You may then type in any legal SuperPILOT instruction (except a label), up to 250 characters long, and it will be carried out at once.

While you have Immediate Execution Mode in effect, you may examine the current contents of any variables or open and close any files. This gives you the power to find out what is happening at any given point in your program. You may Use any subroutine in the program, returning to Immediate Execution Mode when the subroutine has been executed. As your programming skills increase, this ability to find out exactly what you told the computer to do--as opposed to what you thought you told the computer to do--becomes increasingly more valuable.

You may then leave Immediate Mode in either of two ways: First, press CTRL-I again, which will put you back in the lesson at the instruction immediately following the Accept instruction that you used to enter Immediate Mode. Second, type a Jump, Use, End, or Link instruction. Jump, Use, and Link will take you to the destination you specify in the

object of the instruction; End will terminate the lesson and return you to the Editor Menu (unless a subroutine is active, in which case the subroutine will be terminated without leaving Immediate Mode).

## From the Editor Menu

Immediate Mode may be useful to you even when you are not running a lesson. You can use it to experiment with various SuperPILOT commands and instructions. Select the Run option from the Lesson Text Editor, then type CTRL-I instead of the name of a lesson. You will be placed in Immediate Mode, where you may use any SuperPILOT instruction except a label, Jump, or Use. To leave Immediate Mode after you have entered from the Lesson Text Editor Menu, type CTRL-C or e: .

By using Immediate Mode for experimentation with the SuperPILOT system, you save yourself the time of creating new lessons for the sole purpose of trying out various operations. You are encouraged to keep your Apple II in Immediate Mode while reading this manual, so that you can enter example instructions yourself, to see how they work.

## From Lesson Mode

Immediate Execution Mode is also available from Lesson Mode. It can be entered only by pressing CTRL-I at the title page, which appears when the lesson diskette is first booted and there is no lesson called hello on the diskette. CTRL-C or e: returns the student to the title page again. Immediate Mode cannot be entered from Lesson Mode while a lesson is running, as it can from Author Mode.

## Repeating Instructions

Each time an instruction is terminated in Immediate Mode (by pressing the RETURN key), the instruction is temporarily stored in the Apple II's memory. When the instruction is executed and SuperPILOT is ready for your next instruction, you can use the right-arrow key to redisplay that instruction, character by character. For example, suppose you enter the following instruction in Immediate Mode:

```
t:I'm writing a sentence.
```

When you press RETURN, the characters after the colon are printed on the screen and the > prompt appears at the beginning of the next line. If you then press the right-arrow key repeatedly (use the REPT key, if you like), the complete instruction will reappear, ready to be executed again when you press RETURN.

You do not have to re-display all the characters, of course, and you can add new characters in-between the old ones. So, for example, you could press the right-arrow key until the a appeared, then type the new characters nother , then press the right-arrow key again to reveal the remaining old characters. The new instruction

```
t:I'm writing another sentence.
```

is ready to be executed when you press the RETURN key, and it will be stored in the Apple II's memory for you to redisplay the next time, if you wish. You'll find this technique to be very valuable when you are experimenting with long, complex instructions and want to re-execute them with only minor modifications.

## Execution-Time Commands

In addition to the Apple SuperPILOT instructions that you program into a lesson when you are in the Lesson Text Editor, there are two special commands that you or your student can use when the program is running:

### Goto

The Goto command is enabled by the G control option in a PROblem instruction. After pr:g , any Accepted response that begins with the letters goto causes a jump to the destination following the goto . The destination may be a label, @m (next Match instruction), or @p (next PROblem instruction).

### @ (Escape)

The Escape command is enabled by the E control option in a PROblem instruction. After pr:e , any Accepted response that begins with the symbol @ causes a jump to the subroutine labelled sysx . This is equivalent to the Use instruction u:sysx . You must have written a subroutine labelled sysx into your lesson.

For details about selecting these command options, see the description of the PROblem instruction in the Response Instructions chapter. For information about using the commands, see the Execution-Time Commands chapter.

### CTRL-C

When a CTRL-C is typed in response to any Accept instruction, the lesson is terminated and the system restarts itself. (To type CTRL-C, type C while holding down the CTRL key.) This is a handy way to end any lesson in either Author Mode or Lesson Mode.

### CTRL-I

CTRL-I is used to enter Immediate Execution Mode. At any Accept instruction during the running of a program in Author Mode, you may type CTRL-I to leave the program and enter Immediate Mode. This will allow you to examine or even change the contents of your variables at any point in the program, use a subroutine, or experiment with a new approach. When you want to return to the program, simply type another CTRL-I, which sends you back to the statement immediately following the Accept command where you left. (If, instead, you want to re-execute that last Accept instruction, you can type j:@a instead of CTRL-I.)

If you are doing extensive testing of a particular type, you may find it cumbersome to list all the variables you want to examine each time

you reach an Accept instruction and enter Immediate Mode. In this case, you should consider adding a sysx routine to your program, which could, for example, automatically tell you the value of any variables when you reach an Accept instruction and type @ and then RETURN . See the Execution-Time Commands chapter for a more detailed explanation of this option.

You may also provide a way for students to enter Immediate Execution Mode. First, be sure that the lesson diskette does not contain a hello lesson, so the diskette's title page will appear when the student boots the diskette. When such a diskette is booted, SuperPILOT will display a catalog, ask "Run which lesson?" and wait for the lesson name to be typed. If CTRL-I is pressed instead, the student will enter Immediate Execution Mode.

## RESET

Any time the RESET key on the Apple II is pressed, the system restarts itself and any data in the Apple II's memory will be lost. (On newer Apple IIs, you must hold down the CTRL key while pressing RESET.) To protect the programs and data on your SuperPILOT diskettes, you should never press the RESET key when the disk drive's "In Use" light is on.

## Chapter 2

# Text Instructions

- 36 R: Remark
- 37 T: Type
  - 37 In the Lesson Text Editor
  - 38 On the Student's Screen
  - 40 The Text Window
  - 41 Variables
  - 43 Expressions
  - 44 Modifiers
  - 46 Example

## R: Remark

---

r:any text

The Remark instruction allows you to place remarks or comments to yourself, or to anyone who might read your lesson in the Lesson Text Editor, into the program text of your SuperPILOT lessons. These comments are not executed during the running of the lesson, and are never seen by the student. For example, if you want to leave yourself a note about when you did the last revision of a lesson, you could insert the instruction:

```
r:THIS IS VERSION #3, FINISHED 3/12/82
```

When you are typing a Remark instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation colon) on the next line. Your remark is broken only at the spaces or hyphens between words, and may be continued indefinitely. You may also break the Remark line at any point, yourself, by pressing the RETURN key. If you then wish to continue the Remark, you must begin the new line with an r: or a continuation colon.

You can use remarks to remind yourself how a lesson section works, or what labels and variables are used, or what day you wrote this version of the lesson: anything you or another author might want to know later on, when looking at your lesson in the Lesson Text Editor or on paper. A well-Remarked lesson is much easier to understand. This is important if you wish to share your lessons with other authors, but it is also important for you, when you wish to reread or modify your own lessons, perhaps months after writing them.

Occasional use of blank Remark lines (lines with no text after the colon or after the continuation colon) will help to increase readability of your lessons by letting you indicate breaks in the material. To call attention to new sections or important remarks, you might set them off with Remark lines that are rows of asterisks.

Note: You may also use blank lines with no instruction or continuation colon on them to make a visual break between instructions, but such blank lines must not separate portions of a continued Remark instruction.



Example:

```
r:*****
r:*   LESSON AUSTRALIA   *
r:*  VERSION 3 : 23 FEB 82 *
r:*****
:
r:TELLS ABOUT THE ABORIGINES
: OF AUSTRALIA, AND NATIVE
: ANIMALS.  REQUIRES GRAPHICS
: FILES "KOALA" AND "DUCKBILL,"
: AND SOUND FILE "BOOMERANG."
: LINKS TO LESSON "AUST-TEST."

*intro
t:Halfway around the world ...
...

r:SECTION ON THE KOALA BEAR
*koala
...
r:compute test score, based
: on number of guesses (%a).
c:s=11-%a

t:You get #s  points.
```

(The first 11 lines are just comments to the author. They give information that might be useful to know, but might take a while to find by reading the entire lesson program. These lines of text are never seen by the student when the lesson is running.)  
(Blank line inserted for easier reading in Editor.)  
(Label for this section.)  
(First text student sees.)  
(Remark identifies section.)  
(Remark tells method of scoring a question.)  
(Compute score s by subtracting the Answer-Count %a from 11.)  
(Display text, substituting the score s for #s and one space .)

## T: Type

t:any text or spaces, including simple numeric- or string-variable names

When the Type instruction is executed, the text following the instruction's colon is displayed on the student's screen. Variable names, when properly identified, are replaced by their values. A Type instruction displays its object text just as it appears in the Lesson Text Editor except that it is reformatted to fit the screen, breaking the displayed text at the spaces between words. After the text has been displayed, the cursor jumps to the beginning of the next line on the screen, unless you use the H (Hang) modifier described later in this chapter.

## In the Lesson Text Editor

When you are typing a Type instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation colon) on the next line. Your text is broken only at the spaces or hyphens

between words and may be continued indefinitely. You may also break the Type line at any point yourself, by pressing the RETURN key. If you then wish to continue the same Type instruction, you must begin the new line with a continuation colon. In the Lesson Text Editor, blank lines (with no instruction or continuation colon) must not separate continued lines of a Type instruction.

Note: The text that appears on the student's screen may or may not be formatted like the text that appears in the Lesson Text Editor. All text displayed by the Type instruction is automatically reformatted when it is displayed on the student's screen, according to the rules of display formatting discussed in the next section.

When Apple SuperPILOT executes a Type instruction that occupies more than one line in the Lesson Text Editor, it ignores all breaks and continuation colons in the instruction, treating the instruction as if it had been typed on one long line. Only the first 250 characters (about six full lines in the Lesson Text Editor) of a Type instruction are used, not counting the continuation colons. Any part of an instruction beyond the 250th character is simply ignored.

If you want more than 250 characters of text to be formatted into one paragraph, break the text into a sequence of Type instructions and use the Hang modifier on each Type instruction until the last one whose text belongs in that paragraph. The Hang modifier is discussed later in this chapter.

## On the Student's Screen

If the Type instruction occupies a single line in the Lesson Text Editor, and the displayed text fits on one line of the student's screen, the text appears on the screen just as you typed it in the Lesson Text Editor. For example, these instructions:

```
t:           Hi there!  
t:  
t:How       are       you       today?
```

would cause this display on the student's screen, when the lesson containing the instructions is run:

```
           Hi there!  
  
How       are       you       today?
```

Note: You can display a blank line on the screen by using a one-line T: instruction with no text after the colon.

If the Type instruction is colon-continued onto additional lines in the Lesson Text Editor (or whenever the text displayed by a single Type instruction exceeds the width of the student's screen) the text is automatically formatted and arranged into continuous lines to make a "paragraph" as it is put on the student's screen. The appearance of the text as displayed on the student's screen may be very different

from the appearance of the Type instruction in the Lesson Text Editor. For example, consider the following two versions of the same ditty:

```
r:ditty 1
t:When I was but a little boy,
: My father said to me,
: "The letter that comes after A
: Is, sure, the letter B."

r:ditty 2
t:When I was but a little boy, My
:father said to me, "The letter that
:comes after A Is, sure, the letter B."
```

Ditty 1 was typed with breaks in the poem's lines: the author placed a RETURN at the end of each line and began each following line with a colon and a space. Ditty 2, however, was typed as one continuous line: SuperPILOT automatically put a word on a new line if it would have extended beyond the 39th character space, supplying a continuation colon at the beginning of the new line, but not a space. Both versions, when run as SuperPILOT lessons, produce the same result:

```
When I was but a little boy, My father
said to me, "The letter that comes after
A Is, sure, the letter B."
```

What happens on the student's screen is this: the Apple II displays each word and space of the Type instructions' text, ignoring the breaks and continuation colons, until it comes to the first word that won't fit on the current line of the screen. Then it ends the current line at the last space, and puts the next word at the beginning of the next line. The author's line breaks in ditty 1 are ignored; only a new T: instruction will instruct SuperPILOT to begin a new line.

If a word is so long that there is no space in an entire line on the screen, the Apple II simply breaks the word at the end of the line. If you have typed multiple spaces in your Type instruction's text, they will be displayed faithfully unless they would have been displayed at the beginning of a line on the student's screen. When a single Type instruction is displaying its text, every screen line after the first one will start with a non-space text character. For example, this instruction:

```
t:          Oh the cows fly          up,
:          and the flies cow down
:upontheirtinykneesandshuffleofftoBuffa
:lo
```

will cause this display on the student's screen:

```
Oh the cows fly          up,
and the flies cow
downupontheirtinykneesandshuffleofftoBuf
falo
```

In this example, you can see that the spaces before the text on the first line of the instruction are displayed faithfully on the screen. This lets you indent the first line of a paragraph, if you wish. It also allows you to place any amount of text in the middle of the screen, for instance, just by sticking to one-line Type instructions.

The spaces between the last two words in the first line were also displayed just as typed, because they happened to fall in the middle of a screen line. The next spaces, at the start of the second line in the Lesson Text Editor, would have fallen at the start of a new line on the screen, so those spaces were not retained in the display. In the text displayed by a single Type instruction, every screen line after the first must begin with a non-space character.

Finally, we forgot to type a space after the word "down" or before "upon" (either place would do, to separate the two words), so "down" became part of the extremely long word that followed. That word was too long to fit on the current (second) screen line, so it was displayed starting on the next line. When it completely filled that line, there were no spaces at which to break, so the word itself was broken at the end of the screen line.

## The Text Window

The Type instruction always formats its object text to appear on the screen within the currently set text window. If a line of displayed text exceeds the width of the text window, the text is automatically broken just before the first word that won't entirely fit and continued at the beginning of the next line. If the next line would be beyond the bottom line of the text window, the text in the window is scrolled up to accommodate the new line at the bottom, while the old top line disappears.

The default (the way it is set unless you change it) text window is the full screen: 40 characters wide by 24 lines high. See the discussion of the TS: instruction's Viewport command for details about changing the text window. This command lets you do graphics on one part of the screen, while restricting text to a window on another part of the screen. Your Type instructions will automatically reformat the displayed text to fit in the new text window, without any work on your part. For example, if a single Type instruction displays its object text like this in the normal, full-screen text window:

```
You were roaming in the gloaming, making  
loans but never owning: who could guess  
that you'd be moaning when the clone at  
home stopped phoning?
```

the text will be reformatted for you to look like this, in a narrower text window extending from character position 10 to character position 30 on the screen:

You were roaming in  
the gloaming, making  
loans but never  
owning: who could  
guess that you'd be  
moaning when the  
clone at home stopped  
phoning?

Note: This reformatting takes place automatically, without changing anything in your Type instruction.

## Variables

Any number of string-variable names and numeric-variable names may be placed in the object text of a Type instruction. When the instruction is executed, SuperPILOT does not display the variable's name on the screen, but displays the current value of that variable instead. You can use this feature to personalize your lessons by incorporating the student's name, responses, scores, and other personal or changeable data into the displayed messages. For example, the single instruction:

```
t:Well, $n$, you now have #r right.
```

could produce an infinite number of different messages on the screen, including these:

```
Well, Sally, you now have 173 right.  
Well, George, you now have 0 right.  
Well, genius, you now have -7.E19 right.
```

The message that appears depends entirely on what the values are for  $n$  and  $r$  at the time the instruction is executed.

However, you will sometimes want the variable name to appear in a Type instruction, rather than its value. To let the Type instruction distinguish between these two cases, a variable's name destined for replacement by its value must be placed in the object text according to certain strict rules:

1. Before it can be replaced by its stored value in a Type instruction:
  - a) A string variable must be created using a Dimension instruction, and then a Compute, Accept, or File Input instruction must store a name, word, or other string of characters in that variable. If a string variable is not Dimensioned, its name will be displayed literally by a Type instruction. If it is Dimensioned, but no character is stored, the string variable in the instruction is ignored.
  - b) A simple numeric variable must be created by storing a number in it, using a Compute or Accept instruction. If no value has been assigned to it, no replacement will take place and the name of the variable will appear on the screen just as you typed it.

- c) A numeric variable array must be created using a Dimension instruction, and then a Compute or Accept instruction must store a number in the element to be displayed. If no value has been assigned to an element, no replacement will take place and the name of the variable array will appear on the screen just as you typed it.

For example, to make the string variable f\$ and the simple numeric variable k ready for use in a Type instruction, you could use these instructions:

```
d:f$(30)           (Create string variable f$
                   with enough reserved space to
                   store up to 30 characters.)
c:f$="miles per hour" (Store string miles per hour
                       in string variable f$.)
c:k=55             (Create simple numeric variable
                   k and store the number 55
                   in it.)
```

2. When it appears for replacement by a value in a Type instruction:
- a) a string-variable name must be immediately preceded by a dollar sign (for example, the variable s\$ would appear as \$\$ );
  - b) a simple numeric-variable name must be immediately preceded by a pound sign (for example, the variable n would appear as #n );
  - c) variables that have parentheses as part of their names (such as numeric variable array elements, string pseudo-variables, or substring variables) must be enclosed entirely in new parentheses, with the appropriate pound sign or dollar sign immediately before the left parenthesis (for example, the numeric variable array element x(3,2) would appear as #(x(3,2)) and the substring variable s\$(5) would appear as \$(s\$(5)) in the object field.
  - d) each variable name must be immediately followed by a space unless the last character of the variable name is the last character in that Type instruction. The required space after the variable name is NOT displayed on the screen. If you want a space to follow the substituted value on the screen, the variable name should have TWO spaces after it.

For instance, if we continue the previous example:

```
t:The speed limit is #k $f$ . (Display this text, replacing
                               #k and first following space
                               with the number now stored
                               in simple numeric variable
                               k , and $f$ plus the first
                               following space with string
                               now in string variable f$.)
```

This instruction, when the complete example is run, will cause this display on the student's screen:

```
The speed limit is 55 miles per hour.
```

The following example uses a substring variable in the object field of a Text instruction:

```
t:Well, $n$ , the fourth letter (Display this text, substituting
: in your name is $(n$(4)) .      the name stored in n$ for
                                   $n$ and the following space,
                                   and the fourth letter in that
                                   name for $(n$(4)) and the
                                   following space.)
```

Any variable names that you place in the Type object text, but which do not strictly follow the rules above, will be displayed literally, without substituting their values. For example, the instructions that follow will display their object text exactly as it appears, without substitution:

```
t:Welcome, n$ !                    (The string-variable name n$
                                   is not preceded by a $ .)
t:What is x + y ?                  (The numeric variables x and
                                   y are not preceded by # .)
t:I'll add #s(3,2) to your score. (The numeric variable-array
                                   name s(3,2) is not enclosed
                                   in parentheses.)
t:Another question, $n$?           (There is no space immediately
                                   following the string-variable
                                   name n$ .)
```

## Expressions

In addition to variable names, you may include expressions in your Type instructions for replacement by their value. Any combination of elements that SuperPILOT can evaluate as a single numeric value may be used in this way. Simply enclose the elements in parentheses and be sure a pound sign immediately precedes it and a space immediately follows it. For example:

```
t:Only #(20-x) questions left!    (Display text, substituting a
                                   number equal to 20 minus the
                                   current value of x for the
                                   #(20-x) and following space.)
```

Note: The example above contains two spaces between the expression and the next word; one space is required by SuperPILOT when you include an expression for replacement, and the other space is intended to be displayed on the screen. If we were to use an expression in a Type instruction such as

```
t:You got #(100*r/n)% right.
```

and the values for  $r$  and  $n$  were, say, 8 and 20, the following would appear on the screen:

```
You got 40 right.
```



The percent sign, which was intended to be part of the literally displayed text, disappeared because it immediately followed the right parenthesis of the expression, and SuperPILOT uses that character position in its evaluation process. If we had included a space between the right parenthesis and the percent sign, the screen would have displayed the following:

You got 40% right.

**Note:** There is an important difference between the way SuperPILOT handles variable names and expressions in a Type instruction. When you use a variable name in the object text, SuperPILOT requires that a space follow the name; if any other character appears there, the variable's value will not be substituted and the variable's name will be displayed on the screen exactly as you typed it. However, when you use a legal expression in the object text, the expression will be evaluated and the result will be displayed, even if the character place following the expression is not a space. Anything you type in that place will not appear on the screen, but there will be no effect on how the expression is handled.

## Modifiers

**TH:** (Type Hang) The Type instruction normally displays its object text, and then moves the cursor to the beginning of the next line on the screen. The cursor determines where the next character will be put on the screen, even though the cursor will only be visible when waiting for a response to an Accept instruction.

```
t:What is your name?           (Display this text.)
a:                               (Accept student's response.)
```

will result in this display, after the student responds with his name, Mac:

```
What is your name?
Mac
```

When the Type instruction is modified by the Hang modifier, however, the cursor remains at the first character position following the displayed text, rather than advancing to the next line. Thus, the next text displayed by a Type instruction, or typed in a response by the student, will appear immediately following the text displayed by the Hang-modified Type instruction. Here is the previous example, this time using the Hang modifier:

```
th:What is your name?         (Display text, leaving cursor
                               at the end of the text.)
a:                               (Accept student's response.)
```

And the result, when Mac types in his name, will be:

```
What is your name?Mac
```

To separate the question from the answer, just type a space or two after the question mark in the Type instruction. The space will be correctly displayed, between the question mark and the student's response.

What is your name? Mac

The Hang modifier is useful any time you want the student's reply to a question to appear on the same line with the question. You can also use this modifier to join several Typed portions of a sentence into one, continuous sentence, with different portions used under different conditions. See the example at the end of this section.

A sequence of unmodified T: instructions will display its text as a sequence of short paragraphs, each paragraph limited to a maximum of 248 characters (250, including the T: itself). If you want a sequence of Type instructions to format its text into one long, continuous paragraph, use the Hang modifier on all but the last Type instruction in the sequence. Each TH: instruction is limited to 250 characters, of course, but the combined text displayed by the sequence of instructions may be much longer.

TS: (Type Specify) Form and content are said to be twin daughters of good expression: both are important considerations in making sure your message is understood. SuperPILOT can't help you much with the content of your Type instructions, but there is a wide assortment of options at your disposal for altering the form your message takes. There are 17 options in all, and each is called by TS: plus a single letter after the colon, specifying the option desired. Unlike other Type instructions, TS: does not display its object text on the student's screen. Instead, it determines how the object text of the following Type instructions will appear. You can change the size of the characters you display, specify multiple spacing between lines of text, make characters appear in color, create animated effects, divert text to a printer, and a great deal more.

The TS: commands are covered in depth in the chapter on Special Effects Instructions.

TX: (Execute Character Set) When the Type instruction is modified by the eXecute modifier, and a diskette file name appears in the object field, the character set stored in that diskette file is read into the Apple. From then on, Type instructions and student responses use this new character set instead of the usual one. The normal character set is resumed when another TX: instruction is issued, this time with no file name in the object field. The eXecute Character Set instruction is more fully discussed in the Special Effects Instructions chapter.

## Example

r: 'THREE DIGIT ADDITION' LESSON (Remarks for author's reference.  
r: VERSION 3: SEPT 14, 1981 Will not print in Lesson Mode.)

d:al\$(20)	(Reserve space for storing up to 20 characters in string variable al\$ .)
c:al\$="Marion"	(Store the string Marion in string variable al\$ .)
t:Now, let's test your : ability to add : two numbers together.	(Display this text, formatted to fill the screen lines.)
t:	(Display blank line on screen.)
t:Complete the following:	(Display this line of text.)
*probl	(Label for this section.)
th: 237 + 49 =	(Display this text, leaving the cursor after the equals sign.)
a:#n	(Accept the student's response, displayed on the screen after the = , and store the first number of the response in simple numeric variable n .)
t(n<>286):No, \$al\$ , try again.	(If response number stored in n is not equal to 286, display this text: "No, Marion, try again." Note required space after \$al\$ .)
jc:probl	(If n is not equal to 286 ( c is the last-expression conditioner), go back to the label probl and execute the following instruction again.)
c:x=%a	(Retrieve the number of guesses from the Answer Count variable %a , and store it in simple numeric variable x .)
th:Yes, \$al\$ , 286 is	(Sets bold face type style.)
ts:t2	("Yes, Marion, 286 is right.")
t:right.	(Sets normal type style.)
ts:t1	(These eight lines will print one of several different lines, depending on how many guesses the student needed:
th:You needed	"You needed only one try, Marion.
th(x=1): only one try,	That's very good."
th(x>1): #x tries,	"You needed 3 tries, Marion.
t: \$al\$ .	That's not very good.")
th:That's	
th(x>2): not	
th(x<>2): very	
t: good.	

## Chapter 3

# Response Instructions

- 48 PR: Problem
- 48    The Control Options
- 49      Response-Editing Options
- 49      Response-Timing Option
- 50      Execution-Time Command Options
- 50      Wipe-Labels Option
- 52    Programming Notes
- 52    Example
  
- 53 A: Accept
- 54    Response Timing
- 55    Response Editing
- 56    Answer Counting
- 57    Variables
- 60    Modifiers
- 61    Programming Notes
- 61    Example
  
- 63 M: Match
- 64    Controller Characters
- 65    Editing of Match Text and Responses
- 66    Modifiers
- 67    Programming Notes
- 67    Example

# PR: PProblem

---

pr:[list of all control options selected]

The PProblem instruction may specify any of seven different control options which change certain aspects of lesson execution starting at that point. All but one of these options change how the Accept instruction handles student responses.

The other main use for PProblem instructions is to mark the beginning of new sections of the lesson. Branching instructions may then use these PProblem instructions as unlabelled destinations, whenever you want execution to jump to the beginning of the next section.

The instruction PR: , with no object field, marks the beginning of a new lesson section without changing any control options selected by a previous PProblem instruction. Because a PProblem instruction with no object has no effect on the control options previously set, you can easily use this instruction anywhere in a lesson as a section marker and unlabelled destination for branching instructions.

If the PProblem instruction includes an object field, it must list all of the control options that will be active in the next portion of the lesson. Any control options not respecified are disabled: a PProblem instruction that selects any new option must also reselect any old options that are to remain in effect.

## The Control Options

Each of the seven control options is specified by a single letter (for the Time option, the letter is followed by an integer). Any number of these option specifiers may appear in the PProblem instruction's object field. They may be typed in any order, with or without spaces between them.

<u>Option</u>	<u>Specifier</u>	<u>Effect on the Lesson</u>
Lowercase	L	Converts all response letters to lowercase.
Uppercase	U	Converts all response letters to uppercase.
Spaces	S	Removes all spaces from responses.
Time	Tn	Sets a maximum time of n seconds for responses, where n is an integer from 1 through 32767 (t0 = default: unlimited time). If the response has not been completed after n seconds, the portion typed so far (or the current position of the target of the hand controllers) is accepted anyway.

Goto	G	Any response of the form goto destination causes a j:destination . Lets you or the student use the Goto execution-time command to Jump to other parts of the lesson.
Escape	E	Any response beginning with @ causes a u:sysx . Lets you or the student use the Escape execution-time command, which Uses your subroutine labelled sysx .
Wipe-labels	W	"Forgets" all previously established labels, from this point forward. This lets you use the same label names in the next section as you used in the last section, without confusion.

To select the Goto, Lowercase, Spaces, and Time options, you could use an instruction such as

```
pr:glstl0
or
pr:ltl0gs
```

where the l0 following the t represents a ten second response time. The order in which the options are listed is unimportant; these are just two examples of many possible equivalent instructions.

To change the response time subsequently to 20 seconds, without changing the other options, you might use an instruction such as

```
pr:slgt20
```

## Response-Editing Options

The Lowercase, Uppercase, and Spaces options can be used to remove some of the the variability between different students' responses, for easier Matching. Using either the L option (converts all response letters to lowercase) or the U option (converts all response letters to capital letters) lets you successfully Match responses whether they are typed in uppercase, lowercase, or any mixture. If both the L and the U control options are specified in one PProblem instruction, response letters are converted to Uppercase. The S option lets you Match a response whether the student types it as one word or as two words separated by a space. See the discussions of Accept and Match, in this chapter, for more information on using these options. Also, see the string-editing option of the Compute instruction for more ways to edit student responses.

## Response-Timing Option

You can use the Time control option to encourage rapid answers or to prevent a student from spending excessive time on one problem. This option specifies a maximum number of seconds for completing any Accept instruction. If the student has not completed the response within the specified time, the response as it exists at that moment is Accepted.



The Time option can be used with the TIM function (see the Advanced Programming chapter) to evaluate the speed of student responses.

The T specifier should be followed by a number in the range 0 through 32767, or by the name of a simple numeric variable that currently holds a value in that range. Only the integer portion of this number is used, discarding any portion to the right of the decimal point. If the specified number is less than or equal to 0, no time limit is imposed. If more than one Time control option appears in a PProblem instruction, only the rightmost option is used. The time option is discussed more fully with the Accept instruction.

## Execution-Time Command Options

The Goto and Escape control options let you or the student issue certain commands when the lesson is running, in place of the usual response to any Accept instruction. If the Goto option has been selected, an Accepted response of the form

```
goto destination
```

causes a Jump to the specified destination (usually a label).

If the Escape option has been selected, any Accepted response that starts with the character @ causes a u:sysx, which Uses an author-supplied lesson section or subroutine labelled sysx. The Control Instructions chapter discusses the Jump instruction, and the use of subroutines with instructions Use and End.

For details about the Goto and Escape options, with examples of their use, see the Execution-Time Commands chapter.

## Wipe-Labels Option

Since labels are so handy for marking program sections and branching locations, and since labels don't otherwise affect the execution of your program, it might seem that there is never a reason to remove them, even if they are no longer useful to the program. But there are at least two occasions when you may want to clear your program's labels from the Apple II's memory.

First, the Wipe-labels option, W, can be useful when you copy a portion of an old lesson directly into your new lesson, using the Lesson Text Editor's Copy-from-File command. If the old lesson portion uses some of the same label names as your new lesson, you can avoid having to give the duplicate labels new names by starting the inserted portion with a pr:w instruction.

Second, you can use the Wipe-labels option if you have close to fifty labels in your lesson, to insure that SuperPILOT will be able to find a label specified in a Jump or Use instruction. It is wise to limit the number of labels in your lesson to 50 or less, whenever possible. If you do, you will never need to use the Wipe-labels option for this



second purpose, and you will have no need for the information in the remainder of this section.

If you must use more than 50 labels in a lesson, however, read on. The following discussion describes how SuperPILOT keeps track of labels in a lesson--information you will need if you are to avoid problems with lesson execution.

SuperPILOT maintains a label table: a list of the name and location of each label it encounters during the running of a lesson, up to a maximum of 50 labels. As SuperPILOT moves through your lesson's instructions, it stores labels in the table whenever it reaches them, even if the labels are in sections of the lesson that SuperPILOT is jumping over. If a lesson portion containing a label is re-executed, however, the label is not entered in the table again. At any given point in the running of the lesson, the contents of the label table will be a sequential listing of all the labels in your lesson, from the beginning of the lesson to the furthest point in the lesson that has been reached.

SuperPILOT consults this table first whenever it encounters a Jump or Use instruction. It looks through the table from the most recent entry backward to the first entry until it finds the label destination specified by the Jump or Use instruction. Then it reads the location of the label and goes directly to that point in the lesson. If the specified destination is not in the table, SuperPILOT moves to the furthest point in the lesson it has been before and begins searching forward for the label.

This procedure allows SuperPILOT to move quickly to destinations you specify in Jump and Use instructions, because it does not have to read instructions one by one each time it conducts a search. However, if a new label is encountered when the label table is full, that label will be invisible to SuperPILOT for the remainder of the lesson. If a later Jump or Use instruction gives that label as the intended destination, SuperPILOT will not find it in the table and will not find it in the remainder of the lesson either, because the search begins at some point already past the label's position.

To avoid this problem in lessons containing greater than 50 labels, you can use the Wipe-labels option of the PROblem instruction to erase the contents of the label table. Simply place the pr:w instruction at any point where the preceding labels in the lesson will never need to be reached again. For example, if your lesson began with a diagnostic test to determine what later part of the lesson should be studied, you could place a pr:w at the end of that test. The label table would be wiped clean at that point, so that SuperPILOT could store up to 50 new labels from the later sections. (If other PROblem instruction options are active at the point where you place the pr:w instruction, be sure to respecify these options as well, or they will be cancelled.)

Once a pr:w instruction is executed, later branching instructions will not be able to reach the labels prior to the pr:w instruction.

But you can jump to any label in the lesson, on either side of the pr:w instruction, BEFORE the pr:w instruction is actually executed. You might, for instance, put conditions on the pr:w instruction so that it is not executed until, say, the third time through a certain section of the lesson, making it impossible to return to that section a fourth time.

## Programming Notes

A PProblem instruction may be used as an unlabelled branch destination for Jump, Use, and End instructions, as well as for the Goto execution-time command (if enabled). When one of those instructions contains the destination @p , program execution jumps to the next PProblem instruction. If you start each major section with a PProblem instruction, the goto @p execution-time command will let you or the student jump to the next section at any Accept instruction, even if you don't know any specific label names. Also, branching instructions in the lesson can easily jump to the next section without referring to a particular label name. This lets you rearrange lesson sections freely, without altering any branching instructions. Note that when the PProblem instruction is used as an unlabelled destination, you type @p , not @pr.

## Example

pr:u	(Convert response letters to uppercase.)
t:What color is the sky?	(Display this text.)
a:	(Accept student's response.)
m:BLUE	(Did student type BLUE ?)
ty:Yes, on a clear day it is blue.	(If Yes, display this text.)
jy:@p	(If Yes, Jump to the next PProblem instruction.)
tn:Well, on a clear day it is usually blue, isn't it?	(If No, display this text, and this text.)
...	
...	(More of this section.)
pr:	(Mark new section, retaining conversion of response letters to capitals.)
...	
...	

By using the pr:u instruction, you are able to match all the correct student responses: BLUE, blue, Blue, bLue, etc. Following the pr:u instruction, all such responses would be converted to BLUE .

Note: In order to use PProblem instructions as unlabelled branch destinations, you should be consistent in starting each section with a PProblem instruction. You must also avoid using a PProblem instruction within a section following a j:@p instruction. Otherwise, the j:@p will fail to jump to the desired PProblem instruction at the start of the next section.

## A: Accept

a:[simple numeric- and/or string-variable names]

The Accept instruction displays a small white square called the text cursor, then waits for the student's response, except when a pr:t instruction is in effect. The student may type up to eighty characters. Each character in the response is displayed on the screen as typed by the student, beginning at the location of the text cursor. The response is broken if it reaches the right edge of the screen and continues on the next line. After eighty characters are typed, further typing by the student is not Accepted, and the cursor remains to the right of the eightieth character until the RETURN key is pressed. SuperPILOT then edits the response according to certain rules and stores the edited response in the system variable named %b, where the response may be compared to possible answers written in subsequent Match instructions, or used for other purposes.

If any typed response includes a CTRL-C (type C while holding down the CTRL key), the lesson stops. If in Lesson Mode, the lesson is terminated, and the system restarts itself, returning to the lesson diskette's hello lesson or to the lesson diskette's title page. This provides a handy way for a student to end a lesson at the response to any Accept instruction. In Author Mode, a CTRL-I will put the system into Immediate Execution Mode, at which point you may examine the contents of variables, or carry out any other valid Apple SuperPILOT commands.

The usual sequence of instructions includes a Type instruction that poses a question or problem, or otherwise elicits a response, followed by an Accept instruction that receives the student's typed response, followed by Match instructions that determine whether or not the student's response contains certain predicted key words. For example:

t:How much is 2 + 2 ?	(Display this text.)
a:	(Accept the student's typed response, and store it in system variable %b .)
m:4!four!FOUR	(Did student type 4 or four or FOUR ?)
ty:Very good.	(If Yes, display this text.)
tn:No, 2 + 2 = 4 .	(If No, display this text.)

The response can also be stored at the same time in various simple numeric variables and string variables designated by the author, for later use in the lesson. For example:

t:Please type your age.	(Display this text.)
a:#q	(Accept student's response, and store the response's first number in simple numeric variable q .)

Using Accept to store responses in variables is discussed later in this section.

## Response Timing

Normally, the Accept instruction will wait indefinitely for the student to type a response. Execution of the Accept instruction resumes only when the RETURN key is pressed. However, if the response-timing option has been selected by a PProblem instruction, pr:tn , the Accept instruction waits for a maximum of n seconds. The number of seconds n should be in the range 1 through 32767. Decimal places are ignored: only the integer portion will be evaluated. If you assign a  $\emptyset$  or negative quantity to n , the response time is unlimited. If the student has not terminated the response (usually by pressing the RETURN key) after n seconds, execution of the lesson continues, just as if RETURN had been pressed. In this case, the Accepted response consists of any characters the student had typed before the allotted time ran out, even though the student did not press the RETURN key. For example, if you want the student to respond within five seconds, you could use these instructions:

pr:t5	(Set maximum time of five seconds for Accepting responses.)
t:You have five seconds to : tell me the sum of 5 + 8 .	(Display this text.)
a:	(Accept student's answer, waiting a maximum of five seconds.)
m:13!THIRTEEN!thirteen	(Did the student type 13 , THIRTEEN , or thirteen ?
ty:Right you are! Try this one.	(If yes, display this text, and Jump to label prob2 .)
ty:prob2	
t(TIM(1)= $\emptyset$ ):I'm sorry, : your time is up.	(If the TIM function returns a number equal to $\emptyset$ , the student failed to respond in five seconds, so display this text.)
tn:The correct answer is 13. : Let's try another problem.	(If wrong answer or out of time, display this text.)
*prob2	(Label for next section.)

Note: In this example, the match instruction is placed before the response-timing instruction. With this arrangement, the student can forget to press RETURN or not press it in time, and the match will still be successful, because SuperPILOT will Accept whatever had been typed by the student before time ran out.

You can use timed responses to encourage quick answers, or to make progress through a lesson automatic, even if no interactive responses are typed. Use the TIM function to test whether or not the student responded to the last Accept instruction within the set response time. If the last Accept instruction was terminated by exceeding the time set by pr:t , and not by the student, the TIM function will return the

value  $\emptyset$ . See the discussion of the PROblem instruction, earlier in this chapter, for details about setting the response-timing control option.

If the student completed the last response within the set maximum time, or if the pr:t timing option was not selected, the TIM function returns the number of seconds the student required to complete the last response. You can use this information in your lessons to score the student on speed of response, to judge the difficulty of certain questions, or to discover which questions might be worded more clearly. The Advanced Programming chapter describes the TIM function in greater detail.

## Response Editing

The way a response is edited by the Accept instruction depends, in part, on the control options you set up to handle student input. These options are set by the most recently executed PROblem instruction that contains any control options in its object field. If no option is given, the student's response is edited according to the following rules:

1. All spaces before the first non-space character are removed.
2. Any multiple spaces (between characters or after the last character) are compressed into single spaces. If a pr:s option was selected, all spaces are removed.
3. If a pr:l or pr:u option was selected, all letters are converted to lowercase letters, or to uppercase letters, respectively. These options do not affect nonalphabetic characters.

In addition, you can specify no response editing at all, by using the eXact modifier, discussed later on. See the section Using Compute for String Editing, in the Computation Instructions chapter, for further response editing possibilities, which can augment those discussed here. The Compute instruction can capitalize the first letter of a stored string, delete or replace all examples of a character, or convert all letters to capitals (like pr:u ).

When you try to Match words in a student's response, it is important to know as much as possible about the probable form of that response. Response editing removes some of the random variation from one student's response to another's. For example, converting letters to all capitals or to all small letters (item 3) means you don't have to worry about which way a student will type them.

Without any control options set by a PProblem instruction, an unmodified Accept instruction will store this response:

```
My      name is   MAGUILLI
CUDDLE, so there  !
```

as

```
My name is MAGUILLICUDDLE, so there !
```

The spaces before "My" have been removed entirely, the multiple spaces between words have been reduced to one space each, and all uppercase or lowercase letters have been left as typed.

If you had set the S (Spaces) option in a pr:s instruction, the same response would have been stored like this:

```
MynameisMAGUILLICUDDLE,sothere!
```

And if you had set both the S option and the U (Uppercase) option in a pr:su instruction, the same response would have been stored like this:

```
MYNAMEISMAGUILLICUDDLE,SOTHERE!
```

Finally, if you had used the eXact modifier with the Accept instruction in an ax: instruction, the same response would have been stored like this, regardless of any control options set by a PProblem instruction:

```
My      name is   MAGUILLICUDDLE,      so there  !
```

## Answer Counting

An internal Answer-Counter records how many times in a row a student has responded to the same question. This Answer-Count is stored in the system variable named %a .

Before a lesson's first Accept instruction is encountered, the Answer-Count system variable is set at 0. At the first Accept instruction, the Answer-Count is set to 1. Until a new Accept instruction is encountered, the Answer-Count is increased by 1 each time the old Accept instruction is executed again. The Answer-Count continues to grow until a different Accept instruction is executed, setting the Answer-Count back to 1.

The Answer-Count value stored in %a can be used in other instructions for counting the number of times the student has attempted to answer the same question. Counting attempted answers can be useful in scoring the student's effort and in evaluating the clarity of your lesson's questions.

A frequent use of the Answer-Count is to let the student go on to the next question after a specified number of unsuccessful attempts at answering the current question or problem. This is important, as

otherwise the student could remain stuck indefinitely at one question, perhaps one that is ambiguously worded. Instructions whose execution is conditional on the Answer-Count value can let your lesson respond differently to each attempted answer. For Answer-Counts of 1 through 99, your instructions can use the Answer-Count conditioner to test the value of %a automatically. For example:

```

pr:u                                (Convert all response letters
                                   to capitals.)

*start                               (Label for this section.)
t:What is the best feeling in      (Display this text.)
 : the world?

a:                                   (Accept student's response.)
m:love                             (Did student type love ?)
ty:Yes, I think so too, even      (If Yes, display this text.)
 : though I'm only a computer.

jy:next                             (If Yes, Jump to label next .)
t1:Hint:  it makes you warm and  (If wrong answer #1, display
 : cuddly.                        this text.)
t2:Hint:  it makes you feel      (If wrong answer #2, display
 : wanted and secure.            this text.)
t3:Well, to me, the best feeling  (If wrong answer #3, display
 : is to love and be loved.      this text.)
j3:next                             (If wrong answer #3, Jump to
                                   the section labelled next .)

t:Think about it, and then try    (After wrong answer #1 and #2,
 : again to guess what I'm      display this text.)
 : thinking of.

j:start                             (Jump back to label start .)

*next                               (Label for next section.)

```

See the Overview of the Language chapter for a discussion of the Answer-Count conditioner. See the discussion of System Variables, in the Advanced Programming chapter, for information on using %a in other instructions.

## Variables

After a response has been edited according to the response-editing options described in a previous section, it is automatically stored in the system variable %b, also called the Answer Buffer. The same string is then assigned to any string variables whose names appear correctly in the object field of the Accept instruction. Finally, the first identifiable number found in the response is assigned to any simple numeric variables whose names appear correctly in the object field.

Any number and any combination of string variables and simple numeric variables can be assigned values by an Accept instruction. The names of these variables can appear in any order, but they must be placed in the object field of the Accept instruction according to certain strict rules:

1. Before placing its name in the object field of an Accept instruction a string variable must be created, with space for storing strings of the expected size, by using it in a Dimension instruction. For example, to create the string variable `s$`, you could use this instruction:

```
d:s$(30)
```

(Create string variable `s$`,  
with enough reserved space to  
store up to 30 characters.)

2. When it appears in the object field of an Accept instruction:
  - a) a string-variable name must be immediately preceded by a dollar sign (for example, the variable `s$` would appear as `$$`);
  - b) a simple numeric-variable name must be immediately preceded by a pound sign (for example, the variable `n` would appear as `#n`).
  - c) multiple variable names may appear in any order, and they must be separated by spaces.

For example, when an Accept instruction uses the two variables created in the previous part, it might look like this:

```
a:#n $$ #q2
```

(Accept the student's response.  
Store the entire accepted,  
edited response in string  
variable `s$`; store the  
response's first number in  
simple numeric variable `n`;  
create simple numeric variable  
`q2` and store the response's  
first number in it, too.)

3. Substring variables, string pseudo-variables, and numeric variable array elements (all variables written with subscripts) are not allowed to appear in the object field of an Accept instruction. The response should first be stored in a string variable or simple numeric variable whose contents can then be stored in a subscripted variable in a Compute instruction. For example, to store a numeric response in element 13 of a previously dimensioned numeric variable array `j`, you could use these instructions:

```
a:#n
```

(Accept the student's response,  
and store its first number in  
simple numeric variable `n`.)

```
c:j(13)=n
```

(Make a copy of the number now  
stored in `n` and store it in  
element 13 of the numeric  
array `j`.)

If the object field of an Accept instruction contains any variable name that does not conform to these rules, an error message is given (if you are in Author Mode), the Error Flag is raised, and any remaining variables are not given new values.



If there has been no response, or the student pressed only the RETURN key, %b and any object string variables contain the null string (no characters). If no identifiable number is found in the response, any simple numeric variables named in the object retain the value they had before the Accept instruction. If no number is found in the response and there is a numeric variable in the object field, the internal Error Flag is raised. Other instructions can test the state of the Error Flag automatically, by using the Error conditioner (see the Overview of the Language chapter).

To find a number in the Answer Buffer, the Apple II scans the characters stored in %b, starting with the first character and looking for a numeric digit (0 through 9). If a numeric digit is found, the preceding character is checked to see if it is a decimal point or a minus sign. If it is a decimal point, the character before that is checked to see if it is a minus sign. Finally, sixteen characters beginning with the first numeric character (minus sign, decimal point, or digit) are scanned until the first one is found that could not be part of this number. The numeric characters collected to that point are then converted to a number, and that number is stored in all simple numeric variables whose names appear in the Accept instruction's object field.

If more than sixteen numeric characters occur in a row, any characters beyond the sixteenth numeric character are simply ignored. A capital or lowercase E is included only if the next character is a numeric digit or a + or - followed by a numeric digit. Only the first two digits after an E are used: any further digits are ignored. A number with many digits is rounded to six digits, and may be expressed in scientific, E-type notation.

Here are some sample Accepted responses, and the number that would be stored in a simple numeric variable appearing in the object field of the Accept instruction:

Response Accepted by <u>a:#n</u>	<u>Number Stored in Simple Numeric Variable n</u>
4 score and 7 years ago,	4
Apple is No.1 , 4 me.	.1
I have 5,864,327 dollars	5
Retread number 13	13
Mr. Jones is 44	44
five point seven	Unchanged (No number.)
123456	123456

1234567	1.23457E06	(All characters are evaluated; rounding to six places.)
-12345678901234567890123E-23	-.123457	(All characters are evaluated; only six digits printed.)
12e-1234567	1.2E-11	(First two digits after E are used.)
4e37	4.E37	
4e38	Unchanged	(Number too large.)

If the number exceeds the limits for real numbers, an error message is given (in Author Mode) and the Error Flag is raised (in either mode), and all object numeric variables keep the same number that was stored in them before the Accept instruction. The complete response is stored in any string variables that appear before the first numeric variable in the object. Any string or numeric variables that appear after the first unaccepted numeric variable remain unchanged. Numbers must be in the approximate ranges  $-3.4E38$  to  $-1.2E-38$ ,  $\emptyset$ , and  $1.2E-38$  to  $3.4E38$ .

## Modifiers

**AX:** (Accept Exact) The Accept instruction can be modified by the eXact modifier to accept the student's response exactly as it was typed, with no editing. AX: suppresses the normal removal of leading spaces, the compression of multiple spaces, and temporarily overrides the effect of any S, L, or U option set by a PProblem instruction. The unedited response is assigned to system variable %b and to any object string variables (see the previous section). All other control options, such as the Time option and the execution-time command options, remain in effect.

**AS:** (Accept Single) The Accept instruction can be modified by the Single modifier to accept a single keystroke as a suitable response. AS: immediately accepts the student's first keystroke and places the exact character typed in the Answer Buffer %b and in any object variables.

The student does not press the RETURN key after typing the single character response: it is accepted as soon as it is typed, and lesson execution resumes. If the RETURN key is pressed as the only response, a space character is accepted. The Escape execution-time command, if enabled, can be used with the AS: instruction.

The AS: instruction does not generate a new line; the cursor remains where the single keystroke left it. The next Type instruction's text, for instance, will begin right after the Accepted character on the screen.

AP: (Accept Point) The Accept instruction can be modified by the Point modifier to accept the x and y coordinates of a graphics screen point selected by the hand-controller knobs, at the moment when either of the hand-controller buttons is pushed. ( AP: may also be used with other input devices, such as touch screens.) AP: stores the coordinates in system variables %x and %y for use by other instructions. This modification of the Accept instruction is discussed more fully in the Special Effects Instructions chapter.

## Programming Notes

1. An Accept instruction may be used as an unlabelled, relative branch destination for Jump, Use, and End instructions. When one of those instructions contains the destination @a , program execution branches back to the last Accept instruction previously executed.
2. Any Accept instruction (but not an Accept Point instruction) may be used as an entry into Immediate Execution Mode when the lesson is run from Author Mode. If you type CTRL-I (type I while holding down the CTRL key), lesson execution waits while you use Immediate Mode to change lesson variables, try out different approaches, etc. The following CTRL-I takes you out of Immediate Mode and resumes execution of the lesson that was interrupted, beginning with the instruction after the Accept instruction you used to enter Immediate Mode.
3. You should make a practice of testing for the Error Flag after each Accept instruction that stores the student's answer in a numeric variable. This is the best way, if not the only way, to confirm that the student did type a number. If no number is typed in response to an instruction of the form a:#n , the value of n will remain unchanged (or remain 0 if this is the first time n has been used in the lesson), and the Error Flag will be raised. If you use an error-handling routine at this point, you can force the student to enter a number (like 14 instead of fourteen ), so that the value of the variable changes as intended by the student.

## Example

d:n\$(50);q\$(50)	(Reserve space in memory for up to fifty characters each in string variables n\$ and q\$ .)
t:Please type your name, and : then press the RETURN key.	(Display this text.)
a:\$n\$	(Accept response and store it in string variable n\$ .)
c:q\$="Now please type your age."	(Store this string in string variable q\$ .)

t:Hi, \$n\$ . \$q\$	(Display text, substituting the current values of n\$ and q\$ for \$n\$ and \$q\$ and the following spaces.)
a:#a	(Accept response; create simple numeric variable a and store the response's first number in it.)
je:error	(If the Error Flag is up, no number was typed, so Jump to section labelled error .)
c:r=3*a	(Multiply age stored in a by 3, and store result in simple numeric variable r . This is the right answer.)
*prob	(Label for this section.)
c:q\$="What is three times your age?"	(Store this string in string variable q\$ .)
t:\$q\$	(Display text, substituting the value of q\$ for \$q\$ and the following space.)
a:#b	(Accept response; create simple numeric variable b if this is its first use; store the response number in b .)
je:error	(If the Error Flag is up, no number was typed, so Jump to section labelled error .)
tl(b=r):Right the first try!	(If Answer-Count is 1 and if b=r , display this text.)
tc(%a>1):Yes, that's right.	(If b=r ( c is last-expression conditioner) and Answer-Count %a is greater than 1, display this text.)
j(b=r):next	(If b=r , Jump to section labelled next .)
t(%a<5):Nope, try again.	(Wrong answer: if Answer-Count is still less than 5, display this text.)
jc:prob	(If Answer-Count %a is less than 5 ( c is last-expression conditioner), Jump back to label prob for another try.)
t:No, 3 times #a is #r .	(5th bad answer: display text, substituting age a for #a and first following space, and right answer r for #r and first following space.)
j:next	(Student has had 5 wrong answers so Jump ahead to next problem.)

*error	(Label for error section.)
t1:Please type a number.	(If Answer-Count is 1, display this text.)
t2:I mean a number like 7 or 23.	(If Answer-Count is 2, display this text.)
t(%a>2):You must type a number.	(If Answer-Count %a is greater than 2, display this text.)
t:\$q\$	(Display the last prompt again, so student will always see it on the screen, no matter how often an error is made.)
j:@a	(Jump back to last Accept instruction executed.)
*next	(Label for next section.)

In the Accept instruction a:#a , where the student is typing an age, if the student were to respond

MY AGE IS 34 YEARS AND 7 MONTHS.

the Accept instruction would store the number 34 in numeric variable a . However, if the student responds

IN 2 DAYS I WILL BE 34 YEARS OLD

the number 2 will be stored in a .

## M: Match

m:any text or controller characters

The Match instruction looks for any copy of its object text in the most recently Accepted student response. Subsequent instructions can then use the success of this search, or its lack of success, as a condition for their execution, by means of the Yes and No conditioners. For example:

pr:u	(Convert all response letters to capitals.)
t:What color is an orange?	(Display this text.)
a:	(Accept student's response.)
m:ORANGE	(Did student type ORANGE ?)
ty:That's right, a ripe orange	(If Yes, display this text.)
: is usually orange-colored.	
tn:Well, I mean a nice, ripe orange.	(If No, display this text.)

You will generally use the Match instruction to compare a student's response with a response or set of responses that you have supplied. This requires you to predict fairly accurately your students' probable responses, both the correct ones and (often more important) the

incorrect ones. Much of your skill and ingenuity as a programmer and teacher will go into this work.

The Match instruction may be used with two modifiers and four special controller characters to produce almost unlimited sets of Match conditions. Simple match conditions are easier to write than complex ones; but simple conditions can allow misinterpretation of responses and thereby lead to confusion.

When you are typing a Match instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation colon) on the next line. You may also break the instruction wherever you wish, by pressing the RETURN key. If you then wish to continue the Match instruction, you must begin the new line with a continuation colon. Be sure you do not introduce any unwanted spaces when continuing a Match instruction.

Even though the Editor may break your instruction in the middle of a word, the instruction will be executed correctly. When the instruction is executed, any breaks and continuation colons are ignored. The object text is treated as if it were one long unbroken string.

Regardless of the number of lines the instruction occupies, only the first 250 characters of the instruction are actually used at execution time. The continuation colons are not counted in this total. Any portion of the instruction beyond the 250th character is simply ignored.

The Match instruction scans the most recently Accepted student response, looking for any sequence of characters that matches the text in the Match instruction's object field. In the simplest case, the Match object text is a single word or number that the instruction tries to match, character for character, with a word or number in the last Accepted response.

## Controller Characters

To allow more complex Match conditions, any of four special controller characters may be included in the object text. Any number and combination of these special characters may be used, and in conjunction with any of the modifiers. Each affects the Match in a different way:

- % A percent sign is the symbol for a required beginning or end of Match text item. For a successful Match, the student's response must not have a visible character in the position corresponding to the % sign in the Match text item. Without the % sign, the Match text NO will successfully match the responses I DON'T KNOW , NOW? , I WILL NOT! , SNOW JOB , and NONSENSE . The Match text %NO will still successfully match three of those responses ( NOW? , I WILL NOT! , and NONSENSE ). But the Match text %NO% will successfully match only a response in which NO has either a space or the beginning of the response before it, and a space or the end of the response after it.

- \* An asterisk is the wildcard symbol: it will successfully Match any character the student types, and is therefore useful for Matching response words that have small predictable spelling errors. For example, the Match text B\*C\*CLE will successfully Match these responses: BYCICLE , BICYCLE , and CRAB COCLETTE . For a successful Match, the student's response must contain a character corresponding to each asterisk in the Match text item, but any character at all will Match. Any number of \* symbols at the end of a Match text item are ignored.
  
- ! An exclamation point is the or symbol. If an ! appears between two Match text items, a successful Match will occur if either item appears anywhere in the student's response. The Match instruction m:APPLE!PEAR!PLUM!ORANGE will successfully match the responses CRABAPPLE , SPEARMINT , PLUM CRAZY , ORANGES ARE TERRIBLE , and MY FAVORITE FRUITS ARE APPLES AND BANANAS .
  
- & An ampersand is the and symbol. If an & appears between two Match text items, a successful Match will occur only if both items appear in the response, and IN THE SAME ORDER as they appeared in the Match instruction. The Match instruction M:WAY&DOWN&RIVER will match the responses WAY DOWN UPON THE SWANEE RIVER , and SWAYING THE LANDOWNER'S DRIVER , but not I WENT DOWN THE RIVER A WAYS .

To search for more than one Match item in a response, when you don't know the order in which the student will type those items, you must use a series of Match instructions. You can easily make the execution of each Match in the series conditional upon the success of the previous Match instructions by using the Y or N conditioners. Match instructions can also use the Jump modifier, discussed later in this section, to jump quickly through a set of Match instructions looking for the one that applies to the response.

## Editing of Match Text and Responses

Before the Match instruction's object text is actually compared to the student's response, all leading spaces are removed from each Match text item. All the remaining Match text, including any spaces between words, is used exactly as you typed it.

Remember that the Accepted response has also been edited (unless the eXact modifier was used with Accept to suppress all input editing). In general, the Accept instruction removes all leading spaces from the response, and compresses all multiple spaces into single spaces. Any input control options selected by a PProblem instruction may cause further response editing. In particular, be aware of the U option (converts all letters in the response to capitals), the L option (converts all letters to lowercase), and the S option (removes all spaces from the response).

The way you type your Match instruction object text must correspond to the way responses are edited, according to the various defaults and PProblem instruction options. See the discussions of the Accept

instruction and the PProblem instruction in this chapter. The effect of these options on your Match instructions is discussed further in the Programming Notes section below.

## Modifiers

Any of the special controller characters may be used in conjunction with any of these modifiers in the same Match instruction. Also, any sensible combination of modifiers may be used on any instruction.

**MS:** (Match Spell) You can use Spell to modify the Match instruction to allow a successful Match when up to one character of the student's response differs from the corresponding character in the Match item. If multiple Match items are specified, each corresponding response is allowed one wrong character. This modification lets you successfully Match the student's response, even though the response contains a minor misspelling. For example, the instruction `ms:FIBBER` will successfully match the responses `THREE FIBBERS`, `I LIKE FILBERTS`, `A GIBBERING IDIOT`, `K9FIB4ER12`, and `HIFI BERRIES`. You can augment the action of this modifier, which also allows one character to be wrong anywhere in the response word, by using the `*` controller character, which allows another wrong character at a specific place in the response word.

**MJ:** (Match Jump) The Match instruction can be modified by Jump, so that if the Match fails, the program will branch automatically to the next Match instruction. The instruction `mj:` is the same as the instruction `m:` followed by the instruction `jn:@m`. Using the `mj:` instruction can simplify a series of consecutive Match instructions dealing with a single response; execution will simply "drop through" the unsuccessful Match instructions until it stops at the one that succeeds in Matching the response. For example:

<code>pr:u</code>	(Convert all response letters to capitals.)
<code>t:Tell me your favorite color, and</code> <code>: I'll tell you what I think of.</code>	(Display this text.)
<code>a:</code>	(Accept student's response.)
<code>mj:RED</code>	(Did student type RED? If No, Jump to next Match.)
<code>t:Roses, cherry pies, sunburn.</code>	(Yes, so display this text.)
<code>j:part2</code>	(Jump to section part2.)
<code>mj:BLUE</code>	(Did student type BLUE? If No, Jump to next Match.)
<code>t:Blueberry pie, distant mountains.</code>	(Yes, so display this text.)
<code>j:part2</code>	(Jump to section part2.)
<code>mj:GREEN</code>	(Did student type GREEN? If No, Jump to next Match.)
<code>t:Green fields, spinach, waves.</code>	(Yes, so display this text.)
<code>j:part2</code>	(Jump to section part2.)
<code>...</code>	...
<code>...</code>	and so on, through the colors
<code>...</code>	...
<code>m:VIOLET</code>	(Did student type VIOLET?)
<code>ty:Twilight, deep water, bruises.</code>	(If Yes, display this text.)



tn:Sorry, I don't know that color. (If No, none of the Matches was successful, so display this text.)

\*part2 (Label for section part2 .)

The example at the end of this section demonstrates the use of the MJ: instruction in more detail.

## Programming Notes

A Match instruction may be used as an unlabelled relative branch destination for Jump, Use, and End instructions, as well as by the Goto execution-time command (if enabled). When one of those instructions contains the destination @m , program execution jumps to the next Match instruction.

PR: A letter in the Match instruction's object text will not successfully Match a corresponding letter in the student's Accepted response unless both letters are uppercase (capitals) or both are lowercase. Unless you know that all student responses will be typed entirely in uppercase letters or entirely in lowercase letters, you will often wish to convert all response letters to one case or the other. To do this, use the PROblem instruction with the U control option (which converts all response letters to uppercase) or with the L option (which converts all response letters to lowercase).

## Example

```
r:***** (Comments to author.)
: * SOLAR SYSTEM LESSON: *
: * VISITING THE PLANETS *
: * REVISION 4: 5/29/82 *
: *****

pr:u (Convert all response letters
      to capitals.)

t:How many known planets are there (Display this text.)
: in our solar system?

a: (Accept student's response.)
m:%9!%NINE% (Did student type 9 or
              NINE ?)

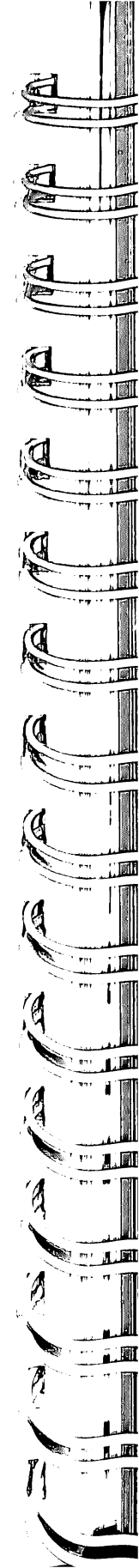
tn:No, try again. (If No, display this text.)
jn:@a (If No, go back to last Accept.)
t:Good. Now, try to name the (Right answer: display these
: nine planets in order, starting three lines of text.)
: with the one closest to the sun.

a: (Accept student's response.)
ms:MERCURY&VENUS&EARTH&MARS&JUPI (Did student type all nine
: TER&SATURN&URANUS&NEPTUNE&PLUTO words in this order, allowing
                                  for a wrong letter in each
                                  word?)

ty:Very good! (If Yes, display this text.)
jy:visit (If Yes, Jump to label visit .)
```

t(%a<3):No, try it again.	(Wrong answer: if Answer-Count is less than 3, display text and Jump back to last Accept.)
jc:@a	(Third wrong answer: display this text, and go on.)
t:No, the planets are Mercury, : Venus, Earth, Mars, Jupiter : Saturn, Uranus, Neptune, and Pluto.	
*visit	(Label for this section.)
t:Which one of the planets : would you like to visit?	(Display this text.)
a:	(Accept student's response.)
mj:NO	(Did student type NO ? If not, Jump to next Match.)
t:Come on, it's vacation time.	(Yes, so display this text, and Jump back to visit .)
j:visit	(Did student type MERCURY , allowing misspelling of letters 2 and 5? If not, Jump to the next Match.)
mj:M*RC*RY	(Yes, so display these lines of text...
t:Well, Mercury is a warm place : to visit, if you are on the : side facing the sun. You see... ...	...and Jump to label next .)
j:next	(Did student type VENUS , allowing misspelling of letters 2 and 4? If not, Jump to next Match.)
mj:V*N*S	(Yes, so display this text...
t:If you like sunny days, Venus : will be a disappointment. It is : always cloudy on Venus, although... ...	...and Jump to label next .)
j:next	(And so on, trying to Match EARTH, MARS, JUPITER, SATURN, URANUS, and NEPTUNE.)
...	
...	
...	
...	
...	
m:PL*T*	(Did student type PLUTO , allowing misspelling of letters 3 and 5?)
tn:That's not one of the planets. : Try again.	(If No, none of the Matches was successful, so display this text.)
jn:visit	(If No, Jump back to visit .)
t:Pluto is a very cold place : to visit, probably the coldest : you could go in our solar system. : Here's why.... ...	(Yes, so display this text... ...and go on to next .)
*next	(Label for next section.)

Note that the instruction mj:NO will match any of the following responses successfully: NO PLANET , NONE OF THEM , NOT MERCURY , I DON'T KNOW , and TO JUPITER, RIGHT NOW! . You could make this Match more selective by using the beginning-or-end-of-word controller character % in the instruction mj:%NO . The instruction mj:M\*RC\*RY will match any of these responses successfully: THE PLANET MERCURY , FIRST PLANET:MARCERY , and I'LL HAVE RUM,RC&RYE .



## Chapter 4

# Control Instructions

72	J: Jump
72	Unlabelled Destinations
72	Programming Notes
73	Example
74	U: Use
75	Unlabelled Destinations
76	Programming Notes
77	E: End
78	Unlabelled Destinations
79	Programming Notes
80	Example
83	L: Link
84	Modifiers
86	Programming Notes
87	Example
88	XI: Execute Indirect
89	Example
91	W: Wait
92	Programming Notes
92	Example

# J: Jump

---

`j:label` or unlabelled destination

The Jump instruction changes the sequence of execution in a lesson by branching to the label or to the unlabelled instruction specified in the object field. A label used as a destination is typed without any preceding asterisk. For example, to Jump to the label `review`, this instruction could be used:

`j:review` (Jumps to label `review`.)

If the specified label is not found in your lesson, an error message is given (in Author Mode) and the Error Flag is raised (in both Author Mode and Lesson Mode). Execution then continues at the next instruction.

You should be careful not to use the same label more than once in your lesson (remember, if the first six characters of two labels are the same, SuperPILOT will consider them to be identical). If SuperPILOT encounters a label that is identical to an earlier one, the earlier one will become invisible to SuperPILOT: all future Jumps that specify that label name will proceed to the most recent occurrence of the label that SuperPILOT has encountered. For further insights on how SuperPILOT keeps track of label destinations, see the discussion of the Wipe-labels option of the PProblem instruction in the previous chapter.

## Unlabelled Destinations

The Jump instruction can use the Accept, Match, and PProblem instructions as destinations for a Jump, even though these destinations are not preceded by any explicit label:

`j:@a` (Jumps back to the last Accept instruction executed.)  
`j:@m` (Jumps ahead to the next Match instruction.)  
`j:@p` (Jumps ahead to the next PProblem instruction.)

These Jump instructions could use explicit labels as destinations, instead of `@a`, `@m`, and `@p`. However, there is a tradeoff to consider. For greater lesson speed, more efficient use of labels, and increased modularity of your programs, use the unlabelled destinations. For greater clarity and readability, however, use explicit labels.

## Programming Notes

Like all Apple SuperPILOT instructions, the Jump instruction can be combined with conditioners and expressions in many ways, giving you a large degree of control over the lesson. For example:

jy:@p	(Jumps to next PROblem if the last Match was successful.)
jn:start	(Jumps to label start if the last Match was not successful.)
j3:review	(Jumps to label review if the Answer Count is 3.)
je:errorhandler	(Jumps to label errorhandler if the Error Flag is raised.)
j(n>13):@m	(Jumps to the next Match if the assertion that "the number now stored in n is greater than 13" is a True assertion.)
jc:@a	(Jumps to the last Accept if the last execution-modifying expression (in this case, the expression n>13 ) was True.)

See the Overview of the Language chapter for more details about using conditioners, combinations of conditioners, and expressions. See the Execution-Time Commands chapter for information about the Goto command, which a student can use to cause a Jump while a lesson is running.

## Example

pr:u	(Mark new section; convert all response letters to capitals.)
*start	(Label for this section.)
t:Guess my number, 1 to 99.	(Display this text.)
a:#n	(Accept response; store number in numeric variable n .)
je:errorh	(If Error Flag is up, no number was typed, so Jump to the section labelled errorhandler . Writing only six characters of a twelve-character label is not very clear, but legal: only the first six characters are significant.)
j(n<1!n>99):errorhandler	(If number typed is less than 1 or greater than 99, Jump to the label errorhandler .)
m:57	(Did student type 57 ?)
ty:You got it!	(If Yes, display this text.)
jy:@p	(If Yes, Jump ahead to the next PROblem instruction.)
t(n>57):No, that's too large.	(If n is greater than 57, display this text.)
t(n<57):No, that's too small.	(If n is less than 57, display this text.)
t(%a<20):Try again.	(If Answer Count %a is less than 20, display this text.)

jc:start	(If Answer Count %a is less than 20, Jump back to start .)
t:Too bad. My number was 57. j:@p	(20th guess, so display text, and Jump to next PProblem.)
*errorhandler	(Label for error routine.)
t:Please type a number, : from 1 through 99.	(Display this text.)
j:@a	(Jump back to last Accept.)
pr:	(Mark next section; continue to convert response letters to capitals.)

Note: The Match instruction m:57 was included to illustrate use of the Jump instruction with the Yes conditioner. You would not normally test a student's numeric response with a Match instruction. M:57 will successfully Match responses such as 57 , 2.3571 , or 57.98 . A better way to test this response would be to use these lines:

t(n=57):You got it!	(If response stored in n is 57, display this text.)
jc:@p	(If response stored in n is 57, Jump ahead to next pr: )

## U: Use

---

u:label or unlabelled destination

The Use instruction, like Jump, changes the sequence of execution in a lesson by branching to the label or to the unlabelled instruction specified in the object field. Unlike Jump, however, the Use instruction "remembers" where the branching operation originated, and can return to that spot later, when an End instruction is encountered.

Ordinarily, your lessons employ the Use instruction to execute a program subsection or subroutine. A subroutine is any program section that is executed after a Use instruction, up to the corresponding End instruction that terminates that section. Understanding the use of subroutines requires knowledge of two instructions: the Use instruction being discussed here, and the End instruction discussed in the next section of this chapter.

Usually, a subroutine starts with an identifying label and terminates with an End instruction. For example, a subroutine to calculate scores might look like this:



*score	(Label for subroutine score .)
c:p=5-%a	(Compute number of points p by subtracting Answer Count %a from 5.)
t:You received #p points on : this problem.	(Display this text, substituting value of variable p for #p and one space.)
e:	(End this subroutine.)

Each time you want to call this subroutine in your lesson, just insert the Use instruction with your subroutine's label in the object field. Your lesson would execute the subroutine labelled score , in the example above, each time it encountered this instruction:

u:score	(Jump to subroutine score and return after next e: .)
---------	---

Note: A label used as a destination is typed without any preceding asterisk.

When the u: instruction is executed, program execution branches to the subroutine with the specified label. At the subroutine's End, program execution returns to the instruction following the u: that called the subroutine.

If the same labelled program section is encountered without being called by a Use instruction, it is not considered a subroutine, and its End instruction will have a very different effect (see the discussion of the End instruction).

If the specified label is not found in your lesson, an error message is given (in Author Mode) and the Error Flag is raised (in both Author Mode and Lesson Mode); then lesson execution continues at the instruction following the Use instruction. However, SuperPILOT still "remembers" where the bad u: instruction is, and the next End instruction may cause lesson execution to return to the instruction following the bad u: instruction.

You should be careful not to use the same label more than once in your lesson (remember, if the first six characters of two labels are the same, SuperPILOT will consider them to be identical). If SuperPILOT encounters a label that is identical to an earlier one, the earlier one will become invisible to SuperPILOT: all future Use instructions that specify that label name will proceed to the most recent occurrence of the label that SuperPILOT has encountered. For further insights on how SuperPILOT keeps track of label destinations, see the discussion of the Wipe-labels option of the PROblem instruction in the previous chapter.

## Unlabelled Destinations

While the Use instruction normally branches to a labelled subroutine, it can also use the Accept, Match, and PROblem instructions as destinations, even though these destinations are not preceded by any

explicit label. The form of such an instruction is the same as for the Jump instruction, discussed in the previous section: u:@a , u:@m , u:@p .

In general, however, explicitly labelled subroutines are more appropriate, as they can then be freely Used from any part of the lesson. Using an unlabelled destination in the object of a Use instruction can cause you to lose control over the sequence of instruction execution. Use is effective for calling specific subroutines that begin with a label and end with an End instruction; for other program branching, such as to unlabelled destinations, Jump is generally a much safer choice.

## Programming Notes

Using subroutines makes your programs simpler, more efficient, and easier to understand, correct, or modify. Once you have written a useful feature as a subroutine, that feature can be used throughout your lesson and is easily transferred to other lessons, saving you much valuable programming time.

The two instructions Use and End together define a subroutine. Read the next section in this chapter for more details about subroutines and the End instruction. See the Execution-Time Commands chapter for information about the Escape command, which you can use to cause a u:sysx while a lesson is running.

Note: You can Use a subroutine from Immediate Mode. If you enter Immediate Mode from an Accept instruction in a running lesson, you can type u:destination , substituting the name of a subroutine in the lesson for the word destination . The subroutine will then be executed, and you will be returned to Immediate Mode when the last e: instruction is executed.

For example, suppose you have written a lesson to evaluate student career interests. It begins with a number of questions for the student to answer, then a subroutine called evaluate uses the collected data to display a career profile to the student. To test this subroutine, you would normally have to run the program many times from start to finish, each time giving slightly different answers to the questions. But a faster method of testing different outcomes would be to enter Immediate Mode at an Accept instruction point, use Compute instructions to enter sample data, and then type

```
u:evaluate
```

The subroutine will be run in its entirety without taking you out of Immediate Mode. When it ends, you can use another Compute instruction to change the sample data a little (or a lot), then call the subroutine again.

A comprehensive example, making use of both the Use instruction and the End instruction, follows the section on the End instruction.

## E: End

---

e:[label or unlabelled destination]

The End instruction terminates the most recently called active subroutine if any subroutine is being executed. If no subroutine (reached by the Use instruction) is currently being executed, an End instruction ALWAYS terminates the lesson (or Immediate Execution Mode), no matter what appears in the End instruction's object field.

e: (Terminate the lesson--or Immediate Execution Mode-- if there is no currently active subroutine.)

If any subroutine (reached by the Use instruction) is active, an e: instruction terminates the active subroutine that was most recently called. Usually, a subroutine starts with an identifying label, and terminates with an End instruction. For example, a subroutine to handle an error caused by the student typing a non-number response might look like this:

```
*error (Label for subroutine error .)
t(1):You must type a number. (Lower Error Flag by evaluating
                               the expression (1) , then
                               display this text.)
e: (End subroutine by branching to
   instruction after u:error
   that called this subroutine.)
```

If the e: has no object, lesson execution branches to the instruction following the last u: instruction. In the same circumstance, an e: with an object also terminates the subroutine, but branches instead to the specified label or unlabelled destination.

```
e:prob3 (End subroutine and branch to
         the label prob3 .)
```

(See the discussion of unlabelled destinations, below, for a way of handling error subroutines that improves on the above example.)

Once a Use instruction calls a subroutine, that subroutine is considered to be active until a corresponding End instruction is encountered. If subroutines are nested (one subroutine calls another subroutine, etc.), any End instruction terminates the most recently called subroutine. The earlier subroutine remains active until there has been exactly one End for each Use instruction.

You can nest subroutines up to ten levels deep. If, at any time during execution, the program has encountered eleven Use instructions unmatched by End instructions, it prints an error message (in Author

Mode) and raises the Error Flag (in both Author Mode and Lesson Mode). Lesson execution then resumes at the instruction following the illegal Use instruction.

If the label specified by an End instruction is not found, the End instruction is treated as an e: instruction with no object field. An error message is given (in Author Mode), the Error Flag is raised (in Author Mode and Lesson Mode), and lesson execution continues at the instruction following the last Use instruction.

## Unlabelled Destinations

The End instruction is usually used with no object, or with a specific label as the object destination. However, it can also use the Accept, Match, and PProblem instructions as destinations, even though these destinations are not preceded by any explicit label:

- |      |  |
|------|--|
| e:@a | (Terminate the lesson, or terminate the most recently called subroutine by branching back to the last a: instruction.)                                   |
| e:@m | (Terminate the lesson, or terminate the most recently called subroutine by branching to first m: instruction after the u: that called that subroutine.)  |
| e:@p | (Terminate the lesson, or terminate the most recently called subroutine by branching to first pr: instruction after the u: that called that subroutine.) |

Using the last Accept instruction as an unlabelled destination is a common practice, especially with error subroutines. The advantage of this format is that the student is allowed to respond to the same Accept instruction again. For example, consider the following subroutine:

*error	(Label for this subroutine.)
t:You must use the number keys to : type a number, like 12 or 173.	(Display this text.)
t:	(Display a blank line.)
t:\$q\$	(Display the contents of string variable q\$ .)
e:@a	(Terminate this subroutine and branch back to the last a: instruction executed.)

This format allows the student to make a mistake, get a helpful comment from the error subroutine, and then answer the same question again, at the same Accept point in the lesson. Notice that the contents of a string variable, q\$ , are displayed by a Type instruction just before

the subroutine is terminated. This variable could be used to store the question being asked at each point in the lesson where this subroutine might be called. This technique avoids the situation where repeated failures by the student could cause the original question to be gradually scrolled off the screen.

The above End instructions could use explicit labels as destinations, instead of @a , @m , and @p . Both methods have their advantages. For greater program modularity and more efficient use of labels, use these unlabelled destinations. For greater clarity and program readability, however, use explicit labels.

## Programming Notes

When an End instruction is encountered at any point in your program, one of two things may occur:

1. If the number of End's executed (including this one) exceeds the number of Use's executed, the lesson is terminated.
2. If the number of Use's executed equals or exceeds the number of End's executed (including this one), the most recently called active subroutine is terminated, and lesson execution continues.

A subroutine is any program section that is executed after a Use instruction, up to the corresponding End instruction that returns execution of the lesson back to the point of the Use instruction. If this same program section is encountered without being called by a Use instruction, it is not considered a subroutine, and its End instruction may terminate some other active subroutine or end the entire lesson.

It is therefore wise to place your subroutines at the end of the lesson and to have a definite end point before your lesson gets to the subroutines. Otherwise, program execution may accidentally "fall into" the subroutines, with unexpected results.

If you want to end your lesson at a certain point where a subroutine may not have been Ended, you might wish to use this ending:

```
*end                (Label for end section.)
e:end              (Ends all active
                  subroutines by branching
                  to label end again
                  and again until no active
                  subroutines are left, at
                  which point the lesson
                  terminates.)
```

## Example

This example is for both Use and End Instructions

```
r:***** (Remarks to author.)
: * ADDITION LESSON--LEVEL III *
: *   LAST REVISION--4/8/82   *
: *
: *   F(IND "REMARK" TO ADD   *
: *   NEW PROBLEM SECTIONS.  *
: *
: *   THE SUBROUTINES IN THIS *
: *   LESSON ARE: ANSWER, TEST, *
: *   SCORE, AND ERROR.      *
: *****

pr:u (Convert all response letters
      to capitals.)
d:a$(50) (Reserve space in memory for
          up to 50 characters in
          string variable a$ .)
t:To end this lesson, type "STOP" (Display this text,
t:in place of any response.      (Display this text,
t: (Display this blank line,
t:What is your name?            (Display this text.)
a:$a$ (Accept student's response;
      store it in string variable
      a$ .)

pr: (Mark start of this section,
     without changing options.)
c:x=23.5 (Store value in variable x .)
c:y=57.8 (Store value in variable y .)
u:test (Branch to subroutine test ,
       and return after next e: .)

pr: (Mark start of this section,
     without changing options.)
u:score (Branch to subroutine score ,
       and return after next e: .)
c:x=12.97 (Store value in variable x .)
c:y=3.1415 (Store value in variable y .)
u:test (Branch to subroutine test ,
       and return after next e: .)

pr: (Mark start of this section,
     without changing options.)
u:score (Branch to subroutine score ,
       and return after next e: .)
c:x=327.9 (Store value in variable x .)
c:y=48.88 (Store value in variable y .)
u:test (Branch to subroutine test ,
       and return after next e: .)
```

pr:

(Mark start of this section,  
without changing options.)

u:score

(Branch to subroutine score ,  
and return after next e: .)

r:\*\*\*\*\*

(Remarks to author.)

: \* PLACE NEW PROBLEM SECTIONS \*

: \* JUST BEFORE THIS REMARK. \*

: \*\*\*\*\*

\*end

(Label for final section.)

e:end

(Since program did not branch  
here from a u: , this e:  
ends the lesson. The label  
just gives extra insurance  
that the program will end  
even if some subroutine is  
still active.)

\*test

(Label for subroutine test .)

c:z=x+y

(Compute sum of x and y ,  
and store result in z .)

t:OK, \$a\$ , what is #x + #y ?

(Display text, substituting  
student's name for \$a\$  
and one space, and x and  
y values for #x and #y  
and following spaces.)

u:answer

(Branch to subroutine answer ,  
and return after next e: .)

ue:error

(If the Error Flag is raised,  
branch to subroutine error ,  
and return after next e: .)

t(n=z):That's right.

(If student's answer n  
equals correct sum z ,  
display text.)

ec:@p

(If same condition n=z is  
true, end subroutine by  
branching to first pr:  
after u:test that called  
this subroutine.)

t4:No, the answer is #z .

(Wrong answer: if the Answer  
Count is 4, display this  
text, substituting value of  
z for #z and one space.)

e4:@p

(If Answer Count is 4, end  
subroutine by branching to  
first pr: instruction  
after u:test that called  
this subroutine.)

t:No, try again.

(Display this text.)

j:test

(Jump back to label test .)

\*answer

(Label for subroutine.)

a:#n	(Accept student response; store in simple numeric variable n.)
m:STOP!HALT!QUIT!END	(Did student type STOP or HALT or QUIT or END?)
ey:end	(If last Match was successful, end this subroutine by branching to label end.)
je:error	(If Error Flag is up, no number was typed, so branch to subroutine error.)
e:	(End subroutine by branching to the instruction after u:answer that called this subroutine.)
*error	
t:You must type a number.	(Label for subroutine.) (Display this text.)
u:answer	(Branch to subroutine answer, and return after next e.)
je:error	(If Error Flag raised again, Jump back to label error.)
e:	(End subroutine by branching to the instruction after u:error that called this subroutine.)
*score	
c:p=5-%a	(Label for subroutine.) (Compute number of points p by subtracting Answer Count %a from 5.)
c:s3=s3+p	(Add p points to total score s3.)
t:You received #p points on : this problem. Your total : score so far is #s3 .	(Display text, substituting value of variable p for #p and one space and value of variable s3 for #s3 and one space.)
e:	(End subroutine by branching to the instruction after u:score that called this subroutine.)

Note: When the Type instruction asking for a response occurs in the main portion of the program and the Accept instruction occurs in a subroutine, there may be a slight delay before the Apple II is ready to start receiving the student's response. Normally, this delay will not cause a problem, but if it does, you can increase the length of the t: instruction, so the student will still be reading it while the Apple II is finding the subroutine:

```
t:OK, $a$, here's another problem:
: Please type the answer to #x + #y .
```



Or you can reduce the impact of this delay by placing the contents of that Type instruction in a variable before calling the subroutine and then including the Type statement itself within the subroutine:

```
d:t$(50)                (Reserve memory space for up
                        to 50 characters in string
                        variable t$ .)
c:t$="How much is #x + #y ?" (Store this text in t$ .)
u:answer                (Branch to subroutine answer ,
                        and return after next e: .)
...
*answer                (Label for subroutine.)
t:$t$                  (Display contents of t$ .)
a:#n                   (Accept student response;
                        store response in numeric
                        variable n .)
...
```

## L: Link

---

l:lesson name[label]

The Link instruction terminates the current lesson and starts running the lesson specified in the object field. All variables created and used in the lesson containing the Link instruction are maintained and are still available in the new lesson, unless the Erase ( x ) modifier is used with the Link instruction. Program execution starts at the beginning of the new lesson if no label follows the lesson name. For example,

```
l:part-two
```

would end the current lesson and start executing the lesson named part-two from the beginning of that lesson. If you wanted the new lesson to start running from the point labelled test , the first lesson would use this instruction:

```
l:part-two,test
```

The lesson containing the Link instruction and the lesson specified in the object field of that Link instruction must be stored on the same Lesson diskette in the normal one-disk-drive Lesson Mode system. For use in systems having more than one disk drive, see the appendix Using More Disk Drives.

If the specified lesson file is not found on the lesson diskette, an error message is given (in Author Mode), the Error Flag is raised (in both Author Mode and Lesson Mode), and lesson execution continues with the next instruction.

## Modifiers

**LX:** The Erase ( x ) modifier can be used with the Link instruction to erase the current lesson's variables and release all storage space reserved for those variables before starting the new lesson. If you just break an existing too-large lesson into two parts, you will probably want to use the l: instruction without the x modifier, so that the numbers and strings stored in the first portion's variables will still be available to the instructions in the second portion. But if you want to develop and then Link together truly independent lessons, each Dimensioning its own variables and not using information stored by any other lesson, you should use the lx: instruction. If you try to Link independent lessons using the unmodified l: instruction, so much of the available variable space may be reserved by the first lesson that the second lesson may not be allowed to Dimension or create all its variables.

After an lx: instruction, all variables used by the previous lesson are "forgotten", and space reserved for those variables is released for use by the new lesson. If the old lesson was using a special text-character set (see the tx: instruction), the standard ASCII character set is reloaded for use by the new lesson.

**LP:** You can Link your SuperPILOT lesson to a Pascal program by using the Pascal modifier ( P ). The LP: instruction can be a very powerful tool, allowing you to make use of new or previously written Pascal programs. Because the LP: instruction is bidirectional, you can also return to the SuperPILOT system when the Pascal program ends, if you wish. There are actually three branching options available to you, as follows:

1. Link to a Pascal program; do not return to SuperPILOT when the Pascal program ends. To exercise this option, you need only include an instruction of the following form in your lesson:

```
lp:demo
```

where demo represents the Pascal program named demo.code . If this program is on a diskette in the system at the time the lp:demo instruction is encountered, SuperPILOT will run it. No modifications to the demo.code program are necessary.

2. Link to a Pascal program; restart SuperPILOT when the Pascal program ends. You can exercise this option by typing your instruction in the same form as above. You must also make two additions to your Pascal program, however, to enable it to return to SuperPILOT. First, you must specify uses chainstuff at the beginning of the Pascal program. Second, you must include the following command right after the Begin statement of the outer block of the program:

```
setchain(`system.startup.`)
```

At the conclusion of the Pascal demo.code program, SuperPILOT restarts itself.

3. Link to a Pascal program; return to a specified SuperPILOT lesson when the Pascal program ends. Within this option, you have two further options: you can specify the return lesson in advance, or you can allow the student to select the return lesson. In either case, the Pascal program must include the same two modifications described above: `uses chainstuff` and `setchain('system.startup.')` .

To choose the return lesson yourself, the `LP:` instruction in the original lesson should take the following form:

```
lp:demo;stars
```

where `demo` represents the Pascal program named `demo.code` , and `stars` is the name of the SuperPILOT lesson you want to return to when `demo.code` has ended. This arrangement will put the student into the `stars` lesson as soon as the `demo.code` program ends.

If you want to allow the student the option of selecting the return lesson, the `LP:` instruction does not need to name a return lesson. Instead, insert the following additional command anywhere in your Pascal program:

```
setcval(choice)
```

where `choice` is the Pascal string-variable name where the student's selection will be stored. Your Pascal program could contain a menu of SuperPILOT lessons, for example, and the student's selection could be stored in `choice` (or any other string-variable name you specify). Then, when the `setcval` command is encountered, your Pascal program will know to return to the chosen SuperPILOT lesson at the program's termination. The contents of the parentheses after `setcval` need not be a string-variable name, but may be a string literal instead, giving the exact name of the return SuperPILOT lesson, enclosed in single quotes:

```
setcval('stars')
```

Any return lesson named in the original `LP:` instruction will be overridden by a return lesson named in a `setcval` command.

Finally, if you specify only an asterisk as the string literal, as in

```
setcval('*')
```

you will be returned to Immediate Mode at the end of the Pascal program.

Note: You must not type the `.code` suffix when specifying a Pascal program in an `LP:` instruction. The `.code` is supplied automatically by SuperPILOT.

## Programming Notes

You can use the Link instruction to join various program portions into a smoothly-running continuous lesson. It is necessary to break a lesson into separate lesson sections whenever the lesson becomes too large to be handled in the Lesson Text Editor. The break can come at any point in the original lesson--just end the first section with a Link instruction that starts the next section. Since the variables from the first section are still available in the Linked section, you should not re-Dimension any string variables or numeric variable arrays.

When you break a lesson in this manner, you must remember that Jump and Use instructions in one portion cannot transfer program execution to a label in another section as they did before. You can use the Link instructions in place of the Jump instructions, however, specifying the name of the destination lesson as well as the destination label:

```
l:lesson,label
```

An instruction with this form will cause a successful Link from one lesson to a specific label in a second lesson, as long as the second lesson is on a diskette in the system at the time of execution. So this method is comparable to using a Jump instruction within a single lesson.

Note: The Use instruction cannot specify a lesson name. When you break a lesson into smaller lessons that call the same subroutine, you must be sure a copy of that subroutine is placed in each lesson.

If you create logically distinct lesson sections, these sections can be combined in different ways and joined to other lessons by using the Link instruction. By building up a library of useful lesson sections, you can eventually save yourself much programming time. Independent lesson sections, each having its own variables, can be Linked using the lx: instruction.

One use for the lx: instruction is in a hello lesson. Each time a Lesson diskette is booted in the primary drive (i.e., in Lesson Mode), SuperPILOT looks first for a lesson named hello . If a lesson is found with that name, it is run immediately--the student does not see the diskette's menu and is not permitted to select a lesson. A hello lesson that consists of this one line:

```
lx:drumhanger
```

```
(Start running the lesson  
named drumhanger ,  
forgetting any variables  
now in use.)
```

will automatically run the lesson drumhanger .

## Example

pr:u (Convert all response letters to capitals.)  
(Display this text.)

t:This lesson is all about United States Presidents. We've had 40 of them so far; some of them good, some of them not so good.  
t:You may select any one you want to learn more about. Just press the RETURN key, and I'll show you a list of our Presidents.  
a: (Accept student's response.)

\*choice (Label for this section.)  
(Display this list.)

t:GEORGE WASHINGTON JOHN ADAMS  
t:THOMAS JEFFERSON JAMES MADISON  
. . . .

t:RICHARD NIXON GERALD FORD  
t:JIMMY CARTER RONALD REAGAN  
t:  
th:Which one would you like?  
a: (Display this blank line.)  
(Display this text, leaving text cursor on same line.)  
(Accept student's response.)

msj:WASHINGTON (If student typed WASHINGTON correctly or with only one spelling error, continue to next instruction. If not, Jump to next Match.)

t:I cannot tell a lie: George Washington is one of my favorites.  
t:Let's see how much you already know about him.  
l:washington (Display this text.)  
(Display this text.)  
(Terminate this lesson, and start running lesson named washington from its start.)

. . . . (More Presidents.)

ms:REAGAN (Did student type REAGAN correctly or with only one spelling error? )

ty:All right, pardner, we'll take a gander at Ronald Reagan.  
ty:Pass the jelly beans, please. (If last Match was successful, display this text.)  
(If last Match was successful, display this text.)

ly:reagan (If last Match was successful, terminate this lesson, and start running lesson named reagan from its start.)

*error	(Label for this section.)
t:Sorry, you must type one of the	(Since no previous Match was
: Presidents' names, EXACTLY as it	successful, display this
: appears on the list.	text.)
t:Press RETURN, and we'll try again.	(Display this text.)
a:	(Accept student's response.)
j:choice	(Jump back to label choice .)

The lessons named washington , reagan , etc., could each end with a similar section for choosing the next President to be studied, or they could Link back to this main lesson to handle the new selection. (Note: if there were a lesson for each of the Presidents, you would need more than 40 lessons on the Lesson diskette, while only 24 can be listed on the menu of the Lesson Text Editor. If your student system has more than one disk drive, you could store half the President lessons on a different Lesson diskette, and as long as both diskettes are in drives during the running of the main lesson, the Links will be successful. If there is only one drive in the student system, you would want to create two main lessons--Early Presidents and Recent Presidents, for example--and then keep all the lessons that can be Linked on the same diskette.)

## **XI: eXecute Indirect**

---

xi:string-variable name

The eXecute Indirect instruction executes the contents of the string variable whose name appears in the object as if those contents were a SuperPILOT instruction. An instruction such as a lengthy Graphics or Sound instruction, once you have stored it in a string variable, can be executed easily, anywhere in your lesson, by using an instruction of the form xi:a\$, where a\$ is the name of the string variable where the instruction is stored. You can also use this instruction, in combination with Accept, Compute, or File Input to create and execute new instructions during the course of a lesson. For example, to execute the contents of string variable a\$ as a SuperPILOT instruction, you could use this eXecute Indirect instruction:

xi:a\$	(Execute the contents of
	string variable a\$ as a
	SuperPILOT instruction.)

Before being used in an xi: instruction, the string variable must appear in a Dimension instruction, and then be assigned a string that is a single legal SuperPILOT instruction. The instruction stored in the string variable cannot begin with a label. The stored instruction may have all the usual modifiers, conditioners, and expressions, in addition to any conditioners and/or expressions that determine the execution of the xi: instruction. You can use either the Compute instruction or the Accept instruction to assign new contents to a string variable.

For example, this program segment:

d:a\$(42)	(Reserve space to store up to 42 characters in string variable a\$.)
c:a\$="t:What is your name?"	(Store the string t:What is your name? in string variable a\$.)
xi:a\$	(Execute contents of variable a\$ as an instruction.)

would cause the following to be printed on the screen:

What is your name?

Note: The xi: instruction is one of the few instructions that lowers the Error Flag. This occurs just before executing the contents of the string variable.

## Example

t:While this little program is	(Display this text.)
: running, you can type any four-	
: line SuperPILOT program and	
: execute it as many times	
: as you wish.	
t:	(Display this blank line.)
d:a\$(42);b\$(42);c\$(42);d\$(42);x\$(1)	(Reserve space in memory for up to 42 characters in each of these string variables: a\$, b\$, c\$, and d\$; and one character in x\$.)
*start	(Label for this section.)
t:Type up to four instructions,	(Display this text.)
: or CTRL-C.	
t:	(Display this blank line.)
t:Press just the RETURN key when	(Display this text.)
: you're done.	
ax:\$a\$	(Accept exact response; store in variable a\$.)
j(len(a\$)=0):exec-3	(If only RETURN pressed, the LENGTH of the string in a\$ equals 0, making this the last instruction, so Jump to the label exec-3.)
ax:\$b\$	(Accept second instruction.)
j(len(b\$)=0):exec-2	(If last, Jump to exec-2.)
ax:\$c\$	(Accept third instruction.)
j(len(c\$)=0):exec-1	(If last, Jump to exec-1.)
ax:\$d\$	(Accept fourth instruction.)
j:exec	(Jump to label exec.)

*exec-3 c:b\$=""	(Set b\$ to no instruction.)
*exec-2 c:c\$=""	(Set c\$ to no instruction.)
*exec-1 c:d\$=""	(Set d\$ to no instruction.)
*exec	(Label for execution section.)
xi:a\$	(Execute contents of variable a\$ as an instruction.)
xi:b\$	(Execute contents of variable b\$ as an instruction.)
xi:c\$	(Execute contents of variable c\$ as an instruction.)
xi:d\$	(Execute contents of variable d\$ as an instruction.)
pr:t2	(Set response time to two seconds.)
t:Press the X key within two : seconds to stop the program : from repeating. as:\$x\$	(Display this text.)  (Accept single response; store it in string variable x\$ .)
pr:t0	(Set unlimited response time.)
j(x\$="x"!x\$="X"):end	(If student typed lowercase or capital x , Jump to end .)
j:exec	(If x was not pressed, Jump to label exec .)
*end	(Label for this section.)
t:Do you want to enter a new : four line program? (Y or N) as:\$x\$	(Display this text.)  (Accept single response; store it in string variable x\$ .)
j(x\$="y"!x\$="Y"):start	(If student typed lowercase or capital y , Jump to start .)
e(x\$="n"!x\$="N"):	(If student typed lowercase or capital n , end lesson.)
t:Please type either Y or N. j:end	(Display this text.) (Jump to label end .)

The four instructions are stored in the string variables a\$ , b\$ , c\$ , and d\$ . If only the RETURN key is pressed, the remaining instructions are skipped. String variables that correspond to no instruction are set to the null string, "" . Then the four stored instructions are executed by four xi: instructions. It would be easy to extend this program to accomodate several more instructions.



## W: Wait

w: number of seconds or simple numeric-variable name

The Wait instruction temporarily stops the lesson for the specified number of seconds, or until a key on the keyboard is pressed, whichever comes first. The length of this wait can be from 0 seconds through 32767 seconds.

You can use the w: instruction to introduce a pause with a set maximum time, during which the student can read some text on the screen, or examine a graphic display, or solve a problem, or think about a new idea. When the specified time is up, the lesson automatically goes on to the next instruction. Or, if the student wants to proceed before the allotted time has passed, pressing any key causes the lesson to continue immediately. For example, if you wanted the student to look at a picture on the screen for up to thirty seconds, you could use either of these equivalent sets of instructions:

t:When you are ready to go	(Display this text.)
: on, press the spacebar.	
w:30	(Wait up to thirty seconds, and then go on.)

or

t:Press the spacebar to continue.	(Display this text.)
c:s7=30	(Assign value 30 to simple numeric variable s7 .)
w:s7	(Wait up to the number of seconds specified by the contents of variable s7 , and then go on.)

If you type an actual number in the object field of a Wait instruction, that number should be an integer from 0 through 32767. If you type a number containing a decimal point or an E, only the part of the number before the decimal or E is used. A number beginning with a decimal point, E, or + sign causes an error. A number less than 0 or greater than 32767 causes no wait.

If a simple numeric-variable name appears in the Wait instruction's object field, the number stored in that variable can be any real number from 0 through 32767. Only the integer portion of the stored number (discarding any portion to the right of the decimal point) is actually used.

The Wait interval can be ended by a key-press that occurs BEFORE the Wait interval is completed. Moreover, if a key is pressed before the Wait interval is started, that "unused" key-press will end the Wait as soon as it begins. If more than one key is pressed before a Wait instruction, however, the Apple II "forgets" those extra key-presses at the end of the Wait interval. This prevents pretyped keys from

causing a whole series of Wait intervals to be skipped; only the first Wait interval would be ended by keys typed in advance.

## Programming Notes

Do not confuse the Wait instruction with the Time option in the PProblem instruction. A `w:30` instruction introduces a pause of up to thirty seconds anywhere in the lesson. A `pr:t30` instruction sets a maximum time of thirty seconds for the student to respond to all future Accept instructions. However, the Wait instruction takes fewer instructions and does not leave the cursor on the screen.

## Example

<code>t:Picture yourself on a</code>	(Display this text.)
<code>: tropical island, surrounded</code>	
<code>: by bananas and papayas,</code>	
<code>: cute little monkeys, and</code>	
<code>: beautifully colored parrots.</code>	
<code>sx:bubble</code>	(Play tropical sounds stored in file named bubble .)
<code>t:Just close your eyes and</code>	(Display this text.)
<code>: imagine you are there...</code>	
<code>w:15</code>	(Wait up to fifteen seconds.)
<code>t:Peering down into the calm</code>	(Display this text.)
<code>: lagoon at your feet, you see</code>	
<code>: something that looks like this:</code>	
<code>gx:lagoon</code>	(Show graphics picture stored in file named lagoon .)
<code>t:Count the number of eyes and</code>	(Display this text.)
<code>: tentacles on this animal.</code>	
<code>w:10</code>	(Wait up to ten seconds.)
<code>...</code>	
<code>...</code>	(and so on...)

## Chapter 5

# Computation Instructions

94	D: Dimension
94	String Variables
95	Numeric Variable Arrays
96	Programming Notes
97	Example
98	C: Compute
99	Using Compute for Assignment
100	Constants
100	Variables
101	Expressions
102	Functions
102	Operators
103	Precedence
103	Using Compute for String-Editing
105	Example
107	Example

# D: Dimension

---

```
d:string-variable name(maximum number of characters to be stored)[;...]  
or  
d:numeric-variable array name(largest subscript to be used)[;...]  
or  
d:numeric array name(largest subscript1,largest subscript2)[;...]
```

The Dimension instruction reserves space in Apple's memory for the specified string variable or numeric-variable array. String variables and numeric-variable arrays must first be Dimensioned before they can be used anywhere else in an Apple SuperPILOT lesson. It is not necessary to Dimension simple (non-array) numeric variables.

The Dimension instruction tells the Apple how much space to save for the variable and stores an initial value in the variable. Immediately after Dimensioning, string variables contain no characters at all (the null string), and elements of a numeric-variable array contain values of 0. You then change these values by using Compute, Accept, or File Input instructions.

## String Variables

For a string variable, designated by a \$ following the variable name, the specified size gives the maximum number of characters that can be stored in that variable. The size limit for a single string is 255 characters. For example, if you want your lesson to store up to fifty characters in a string variable named s\$, you could use this line at the beginning of the lesson:

```
d:s$(50)
```

(Reserves space in Apple's memory for storing a string up to fifty characters long in string variable s\$.)

SuperPILOT reserves a total of 1600 character locations for the sole purpose of Dimensioning your string variables. You can use these locations to Dimension all of your string variables in a program.

If you attempt to Dimension more than 1600 character spaces in a lesson, an error message is displayed in Author Mode, and the Error Flag is raised in both Author Mode and Lesson Mode. Any string variables that you attempt to Dimension in excess of the 1600 character limit will be unavailable for storing strings in your lesson.

Note: There are two important cautions about string-variable Dimensions:

1. Do not Dimension strings bigger than necessary. A string variable always reserves the full number of character locations given in the Dimension instruction for that variable. If you store a string of less than the maximum Dimensioned size in the variable, not all of the reserved locations are used, but

the unused locations are still reserved and unavailable for storing characters of any other string variable.

2. Do not Dimension the same string more than once. Once a set of character-storing locations is reserved for a specified string variable by a Dimension instruction, these locations remain permanently reserved for the rest of the lesson. If another Dimension instruction re-Dimensions the same string variable, the first set of locations reserved for that variable remain reserved, but are made completely unavailable, and a new additional set of locations is reserved.

## Numeric Variable Arrays

Numeric variable arrays are like simple numeric variables, except that a "family" of variables is created, using the same name but followed by a distinguishing number in parentheses. The number that identifies each of the variables, or elements, in a numeric array is called the subscript of that element. For example, the first element of numeric-variable array  $p$  would be specified as  $p(\emptyset)$ , the second element would be  $p(1)$ , the third would be  $p(2)$ , and so on. This numeric variable array might be pictured like this:

$p \quad | \quad (\emptyset) \quad | \quad (1) \quad | \quad (2) \quad | \quad (3) \quad | \quad (4) \quad | \quad (5) \quad |$

Numeric arrays may be one-dimensional (requiring one subscript to specify each element) or two-dimensional (requiring two subscripts to specify each element). If  $q$  is a two-dimensional numeric variable array, its first element would be  $q(\emptyset, \emptyset)$ ; following elements could be  $q(\emptyset, 1)$ ,  $q(\emptyset, 2)$ , and so on. There could also be more elements, such as  $q(1, \emptyset)$ ,  $q(1, 1)$ ,  $q(1, 2)$ , etc., and  $q(2, \emptyset)$ ,  $q(2, 1)$ ,  $q(2, 2)$ , etc. This two-dimensional numeric variable array might be pictured like this:

$q \quad | \quad (\emptyset, \emptyset) \quad | \quad (\emptyset, 1) \quad | \quad (\emptyset, 2) \quad | \quad (\emptyset, 3) \quad |$   
 $\quad | \quad (1, \emptyset) \quad | \quad (1, 1) \quad | \quad (1, 2) \quad | \quad (1, 3) \quad |$   
 $\quad | \quad (2, \emptyset) \quad | \quad (2, 1) \quad | \quad (2, 2) \quad | \quad (2, 3) \quad |$

When you Dimension a numeric variable array, the specified size gives the largest array-element subscript allowed for each dimension of the array. The subscript of the first element in a numeric array is always  $(\emptyset)$  or  $(\emptyset, \emptyset)$ . For example:

$d:n(3\emptyset)$

(Reserves space in Apple's memory for storing up to 31 real numbers in numeric variable array elements  $n(\emptyset)$  through  $n(3\emptyset)$ . This is a one-dimensional array, so only one subscript is needed to specify any of its elements.)

d:v(12,4)

(Reserves space in Apple's memory for storing up to  $13*5=65$  real numbers in numeric variable array elements v(0,0) through v(12,4). This is a two-dimensional array, so two subscripts are needed to specify any of its elements.)

SuperPILOT reserves a total of 2000 locations in memory for the sole purpose of Dimensioning your numeric variable arrays. Either dimension of a two-dimensional array can be given a largest subscript value as high as 198, so long as the other dimension does not cause the array to exceed 2000 elements. For example, a two-dimensional array whose first dimension has a largest subscript value of 19 cannot be given a largest subscript value greater than 9 for the second dimension. Remember that the first subscript number in either dimension is 0, not 1. If you were to dimension such an array, SuperPILOT will reserve 2000 locations for it:  $(19+1)*(9+1)=2000$ .

Note: There are two important cautions about numeric variable array Dimensions:

1. Do not Dimension arrays bigger than necessary. A numeric variable array always occupies the full amount of locations given in the Dimension instruction for that array. If you store fewer array elements than the maximum Dimensioned size of the array, not all of the reserved locations are used, but the unused locations are still reserved and unavailable for storing anything else.
2. Do not Dimension the same array more than once. Once a set of locations has been reserved for a specified numeric variable array by a Dimension instruction, those locations remain permanently reserved for the rest of the lesson. If another Dimension instruction re-Dimensions the same array, the first set of locations reserved for that array remain reserved, but are made completely unavailable, and a new additional set of locations is reserved.

## Programming Notes

It is a good practice to place all D: instructions together at the beginning of each lesson so that they are executed one time only. If a Dimension instruction occurs in a program portion that is looped through several times, additional number-storage locations are used up each time the variable is re-Dimensioned, just as if you were Dimensioning new variables. Since there are only 2000 locations available for Dimensioning numbers and 16000 locations for Dimensioning strings, the repeated Dimension instruction might soon cause the message

R-error (Too many arrays or strings)

You can Dimension more than one variable in the same instruction, by separating the variable names by semicolons. For example, the instruction

```
d:x$(40);y(9);z$(20)
```

will Dimension three variables: string variables x\$ and z\$, and numeric variable array y. Note: No spaces are allowed in the instruction between the semicolon and the following variable name.

## Example

```
d:a$(20);b(20)
```

(Reserve space in memory for storing up to 20 characters in string variable a\$, and 21 real numbers in numeric variable array elements b(0) through b(20) .)

```
t:What is your name?
```

(Display this text.)

```
a:$a$
```

(Accept student's response, and store in string variable a\$ .)

```
c:m=1
```

(Store the value 1 in simple numeric variable m. This sets the initial value of m .)

```
t:Okay, $a$, this will give you  
: some practice adding numbers.
```

(Display text, substituting student's name for \$a\$ and one space following.)

```
*test
```

(Label for this section.)

```
t:What is #m + #m ?
```

(Display text, substituting the number currently stored in variable m for each #m and one space following.)

```
a:#n
```

(Accept student's response, and store the first number in it in simple numeric variable n .)

```
c(n=m+m):b(m)=1
```

(If n=m+m, the student's answer was right, so store value 1 in numeric array element b(m) .)

```
c(n<>m+m):b(m)=0
```

(If n is not equal to m+m, the answer was wrong, so store the value 0 in element b(m) .)

```
c:m=m+1
```

(Increment value of m by 1.)

```
j(m<21):test
```

(If m is less than 21, Jump back to the label test .)

```
*score
```

(Label for section of lesson that uses the scores stored in the elements of numeric-variable array b .)



Note: More advanced programmers would probably replace the two lines

```
c(n=m+m):b(m)=1
c(n<>m+m):b(m)=Ø
```

with the single equivalent instruction

```
c:b(m)=(n=m+m)
```

(Evaluate the assertion  $n=m+m$  .  
If it is true, store the  
result 1 in numeric variable  
array element  $b(m)$  . If  
false, store the result  $\emptyset$  in  
element  $b(m)$  .)

## C: Compute

---

```
c:name of variable=expression[;...]
```

or

```
c:/name of string variable editing option [editing option]
```

The Compute instruction can be used for three purposes:

1. to store a fixed or computed number in a simple numeric variable or in an element of a numeric-variable array,
2. to store a fixed or computed string of characters in a string variable or substring variable,
3. to edit the contents of a string variable.

Examples:

```
c:n=2+(3/x)
```

(Calculate the current value  
of  $2+(3/x)$  , and store it in  
simple numeric variable  $n$  .)

```
c:s$="Apple"
```

(Store the characters Apple  
in string variable  $s\$\$  .)

```
c:/s$ /px
```

(Edit the contents of string  
variable  $s\$\$  , replacing each  
character "p" with an "x" ;  
then store the new string  
back in string variable  $s\$\$  .)

The most common use of the Compute instruction is to do calculations and to store numbers and strings in their appropriate variables (as shown in the first two examples above). Most of this chapter discusses that kind of use. The final topic in this chapter is the use of the Compute instruction for editing a string (third example, above).



This chapter does not discuss constants, variables, functions, operators, and expressions in great depth. The discussion that follows lets you use some of Compute's features without a detailed understanding of these items. Later, when you are more familiar with Apple SuperPILOT, you may wish to read the Advanced Programming chapter to learn more about the calculating powers available in the Compute instruction and in conditioning expressions.

## Using Compute for Assignment

Only three instructions can store a number or a string in a variable: the Accept instruction, the Compute instruction, and the File Input instruction. Storing a number or a string in a variable is also called assigning the number or string to the variable. When Compute is used to store a number or a string in a variable, the instruction has four parts:

1. The C instruction name, together with any conditioners and instruction-modifying expression, always comes first, followed by the colon. Spaces following the colon are permitted.
2. The next item is the name of the variable where you wish to store a number or a string.
3. Next comes the assignment or "store-this-value" indicator, which is an equals sign (=). Spaces are permitted on either side of the equals sign. Note: only the first equals sign in a Compute instruction is interpreted as an assignment indicator, all subsequent equals signs are treated as relational operators in an expression. See the Advanced Programming chapter for a discussion of this distinction.
4. The last part is the value (number, string, variable name, or complete expression) to be stored in the variable whose name appears on the left of the assignment indicator. Spaces are permitted within the expression.

The last three parts of the instruction (everything after the colon) may be repeated with different variable names and values, as long as you separate each assignment from the next one with a semicolon. For example,

```
c:r$="Right";w$="Wrong";x=17
```

(Store the string Right in string variable r\$, store the string Wrong in string variable w\$, and store the number 17 in simple numeric variable x.)

The expression portion of a Compute instruction, to the right of the assignment indicator (=), may include constants, variables, and expressions. Each is discussed below.

## Constants

These are just numbers and strings.

<u>Item Name</u>	<u>Item</u>	<u>Examples</u>
Numeric constant	Number	512.37 , 7.1238E+12
String constant	String	"APPLE" , "All 4 1"

Examples in Compute instructions:

c:n=512.37	(Store the number 512.37 in simple numeric variable n .)
c:s\$="APPLE SEEDS"	(Store the string APPLE SEEDS in string variable s\$ .)
c:h1=60;w1=30	(Store the number 60 in simple numeric variable h1 , and store the number 30 in simple numeric variable w1 .)

## Variables

A variable is a location in the Apple II's memory where you can store a number or a string of characters. You don't have to know where each item has been stored: you just refer to it by the name of that variable. Once a number or a string is stored in a named variable, you can use the variable's name in place of the stored number or string.

<u>Item Name</u>	<u>Stored Item</u>	<u>Example Variable Names</u>
Simple Numeric Variable	Number	x , p4 , 13 , q
Element of Numeric Variable Array		
One-dimensional	Number	e(0) , r9(2) , w(23)
Two-dimensional	Number	n(0,4) , g1(13,2)
String Variable	String	s\$ , a5\$ , t\$
Substring Variable		
Single-character	Character	s\$(1) , a5\$(23) , t\$(5)
Multi-character	Characters	s\$(1,7) , a5\$(7,12)
System Variable		
Answer Count	Number	%a (SuperPILOT stores
Answer Buffer	String	%b numbers or strings
X-Coordinate	Number	%x in each of these
Y-Coordinate	Number	%y system variables
C-Coordinate	Number	%c without being told
R-Coordinate	Number	%r to do so by an
Spin Angle	Number	%s instruction.)
I/O Annunciator	Number	%o (You store numbers
External Control	Number	%v to control external
		%w devices.)

## Examples in Compute Instructions:

c:p5=n

(Make a copy of the number now stored in simple numeric variable n , and store it in simple numeric variable p5 .)

c:t\$=s\$

(Make a copy of the string now stored in string variable s\$ , and store the copy in string variable t\$ .)

c:x=a;a=b;b=x

(Make a copy of the number in a and store it in x ; make a copy of the number in b and store it in a ; make a copy of the number in x (which was originally in a ) and store it in b . Note: This procedure can be used to "switch" the values of two variables, in this case a and b .)

A string variable is created and given its maximum size by using it in a Dimension instruction. Only then can Compute or Accept instructions store strings in the variable. If you refer to a string variable before you store any string in it, it contains no characters at all.

A numeric-variable array is created and given its maximum subscripts by a Dimension instruction. Only then can Compute or Accept instructions store numbers in the elements of that array. If you refer to an element of a numeric array before you store any number in it, that element is given a value of  $\emptyset$ .

Simple numeric variables are created simply by using them in Compute or Accept instructions. It is not necessary to Dimension a simple numeric variable; just start using it by placing it in one of these two instructions, or in an expression (as a conditioner for any other instruction, or within the object text of a Type instruction). It will be given a value of  $\emptyset$  at the place of its first occurrence (unless, of course, you assign it a different value).

For more information about the different kinds of variables, see the Advanced Programming chapter.

## Expressions

An expression may be a constant, a variable, or a combination of constants and variables with items that indicate some calculation or other operation to be done on the constants and variables. If the expression includes one or more operations, they will be either functions or operators.

## Functions

A function normally takes the form of a three-letter abbreviation for the name of the function, followed by a parenthetical expression. The name of the function determines the kind of calculation or other operation to be performed on the contents of the parentheses, which can be a constant, a variable, or another expression. A complete discussion of functions can be found in the Advanced Programming chapter.

<u>Item Name</u>	<u>Item</u>	<u>Examples</u>
Function	Calculated Number or String	pd1(x) , ins(x,t\$,p\$), flo(x)

Examples in Compute instructions:

c:x=pd1(0)	(Store a number from 0 through 255, indicating the position of game control #0, in simple numeric variable x .)
c:a=flo(a\$)	(Find the first number in string variable a\$ , and store it in simple numeric variable a .)
c:x=ins(1,b\$,"thousand")	(Beginning at first character position, search through b\$ for the string thousand ; if found, store the number of its first character position in x .)

## Operators

Operators are used with constants, variables, and functions to indicate all the usual calculator operations.

<u>Operator Symbol</u>	<u>Operation</u>	<u>Example Expressions</u>
+	Addition	99+12.45 , 3.4+pd1(t) , e+1
-	Subtraction	16-43.2 , 5-sin(x) , g-h7
*	Multiplication	983*12 , y*3.14 , b*m
/	Division	2.3/.056 , w6/u , p/sin(13)

Example in a Compute instruction:

c:v=d/t	(Divide the number stored in d by the number stored in t and store the result in v .)
---------	---

There are also quite a few operators not offered on many calculators. These operators let you do exponentiation (multiplying a number by itself a specified number of times), comparisons between two numbers or

strings, concatenation (joining together) of two strings, and the logic operations And, Or, and Not. These operators are all discussed in the Advanced Programming chapter.

## Precedence

In expressions with several operators, it may not be clear which operation will take place first. For example, the expression  $12+4/2$  might be evaluated as  $12+4$  divided by  $2 = 16/2 = 8$ ; but the correct evaluation is  $12$  plus  $4/2 = 12+2 = 14$ . The Advanced Programming chapter discusses the special rules for precedence among operators, but there is a way to avoid any possible confusion: you can use parentheses to indicate how you want such an expression to be evaluated. The rule is simple: the sub-expressions in the innermost set of parentheses are evaluated first, then those results are used in evaluating the surrounding expressions.

Example in a Compute instruction:

```
c:d=(17-13)*(5/(2+.5))
```

In this expression, the innermost set of parentheses surrounds the sub-expression  $2+.5$ , so the first operation is to add  $.5$  to the number  $2$ . That leaves the expression as  $(17-13)*(5/2.5)$ . Next, the sub-expressions  $17-13$  and  $5/2.5$  must be evaluated (the order does not make any difference), leaving the expression  $4*2$ . Finally, the result,  $8$ , is stored in simple numeric variable  $d$ . Of course, any of the numbers in such an expression could be replaced by a variable name, by a function, or by yet another sub-expression in parentheses.

## Using Compute for String Editing

In addition to its normal task of evaluating an expression and assigning the result to a variable, the Compute instruction also has a slightly different use: it can do certain quick easy editing changes to the contents of a specified string variable. The normal Compute instruction follows this form:

```
c:name of variable=expression[;...]
```

To do string editing, you must use a somewhat different form in typing the Compute instruction's object field:

```
c:/string-variable name {editing option}
```

Note: The string variable's name is preceded by a slash (/), and that there is no assignment indicator (=) following the name. Any number of editing options may follow the variable name, and the list of options may be colon-continued onto subsequent instruction lines. For the sake of clarity, you may find it helpful to separate the options with spaces.

The string editing form of the Compute instruction differs from the assignment form in one other important way. When using Compute for

assignment, you can assign values to as many variables as you like, separating the assignments with semicolons. When using Compute for string editing, however, you may edit only one string per instruction. You may specify several edit options to be performed on that string, but you must begin a new Compute instruction to edit a different string. Also, if you include a string edit in the same Compute instruction with one or more assignments, the string edit must be the last command in the instruction. Note: Be careful to adhere to these rules, because failure to do so may give strange and undesired values to your strings, and you will probably not be warned of the problem with error messages.

These are the editing options you can use:

<u>Form of Option</u>	<u>Name of Option</u>	<u>Effect on String</u>
U	Uppercase	Converts all letters in the string to capital (uppercase) letters.
L	Lowercase	Converts all letters in the string to small (lowercase) letters.
C	Capitalize	If the first character in the string is a letter this option converts it to a capital letter.
/xy	Replace	Replaces every character x with character y ( x and y are any single characters).
/x/	Delete	Removes every character x from the string ( x is any single character).

For example, to delete every "R" from the string "RAPPLER" stored in string variable w\$, you could use this instruction:

```
c:/w$ /R/
```

After this instruction, string variable w\$ would contain the string "APPLE".

The Replace and Delete options operate on only a single character in the string. The string character (and the replacement character, if any) may be any printing character. You cannot Replace or Delete non-printing characters. You must type the character itself: a variable name cannot be used instead of a Replace or Delete character.

Note: You can delete or replace the character "/" in the existing string, but you cannot use "/" as a new, replacement character.

When you use more than one editing option in a single Compute instruction, the options are executed one at a time, starting with the leftmost option. For example, in these lines

d:b\$(30)	(Reserve space in memory to store up to 30 characters in string variable b\$ .)
c:b\$="omallallampopa"	(Assign string "omallallampopa" to string variable b\$ .)
c:/b\$ /o/ c /ai /ls /m/	(Get the string in b\$ , remove every "o", then capitalize the resulting string's first character, then replace each "a" with "i", then replace each "l" with "s", and last remove each "m" and store edited string back in b\$ .)
t:\$b\$ is an old, old man.	(Display text, substituting the edited string for \$b\$ .)

the string-editing Compute instruction first removes all "o"s from the string in string variable b\$ , making the new string "mallallamppa". Then it capitalizes the first character in the string, which is now an "m", making "Mallallamppa". Next it replaces every "a" with "i", and every "l" with "s", resulting in "Mississimppi". Last, it deletes every "m" from the string, and stores "Mississippi" back into string variable b\$ . Note that /m/ did not remove "M", which is considered a different character.

## Example

pr:l	(Convert all response letters to small letters.)
d:n\$(20);b\$(40)	(Reserve space in memory to store up to 20 characters in string variable n\$ and up to 40 characters in string variable b\$ .)
th:Please type your first name:	(Display this text, leaving the cursor waiting at end.)
a:\$n\$	(Accept student's response; store in string variable n\$ .)
c:/n\$ c	(Capitalize the first letter of the name stored in n\$ .)
t:Well, \$n\$ , type the number of : dollars you would like to have.	(Display text, substituting student's name for \$n\$ and following space.)
a:\$b\$	(Accept student's response; store in string variable b\$ .)
*number	(Label for section that deciphers numbers.)
c:/b\$ /,/	(Remove all commas from the string in b\$ .)

<code>c:r=flo(b\$)</code>	(Find the first number in <code>b\$</code> and store it in simple numeric variable <code>r</code> .)
<code>c(ins(1,b\$,"thousand ")):r=r*1000</code>	(If the word <code>thousand</code> is in <code>b\$</code> , multiply <code>r</code> by <code>1000</code> .)
<code>c(ins(1,b\$,"million ")):r=r*1e6</code>	(If word <code>million</code> is in <code>b\$</code> , multiply <code>r</code> by <code>1,000,000</code> .)
<code>t:You would like #r dollars?</code>	(Display text, substituting student's number for <code>#r</code> and following space.)
<code>a:\$b\$ #p</code>	(Accept student's response; store in string variable <code>b\$</code> and in simple numeric variable <code>p</code> .)
<code>je:match</code>	(If Error Flag is up, response contained no number, so Jump to section labelled <code>match</code> .)
<code>j:number</code>	(Response contained a number, so Jump back to <code>number</code> .)
<code>*match</code>	(Label for this section.)
<code>m:no!not!more!less!stupid</code>	(Did student type one of these words?)
<code>jn:nextsection</code>	(If No, Jump to label <code>nextsection</code> .)
<code>t:Sorry. Try typing the number : of dollars you would like, : again, using numbers only. a:\$b\$</code>	(Yes, so display these lines of text.)
<code>j:number</code>	(Accept student's response; store in string variable <code>b\$</code> .)
<code>*nextsection</code>	(Jump back to label <code>number</code> .)
	(Label for next section.)

In this example, we used Compute's string-editing feature twice. When the student types a name, the `pr:l` instruction at the beginning makes sure it is converted to all small letters. Then the Compute instruction `c:/n$ c` capitalizes the first letter of the name. Thus, every name comes out with the first letter a capital and the rest small letters, no matter how the name was typed originally.

In processing the student's typed number, we used another Compute instruction to remove all commas. Removing the commas is necessary if you want the FLO function to convert a string such as `$3,000,000` to the correct number. With those commas in the string, FLO would return the number 3, not 3E6. Of course, you will still have to explain to the student how to read a response that has been converted to scientific E-type notation.



# Example

```
r:*****
: * VANISHING VOWELS GUESSING *
: * GAME:  LAST REVISED 3/6/82 *
: *
: * OBJECTIVE:  SPELLING AND *
: * VOCABULARY IMPROVEMENT *
: *****

pr:l
d:w$(15);g$(15);x$(15);s(1,2);c$(4)

c:s(0,0)=0;s(1,0)=0
c:s(0,1)=1;s(1,1)=1
c:s(0,2)=30;s(1,2)=30
t:This is the game of
t:
t:          VANISHING VOWELS
t:
th:I will think of a word and make
: all its vowels vanish. Then I
: will show it to you. You will
: see the carat mark ( ^ ) where
: each vowel used to be.
t: You will then have to guess
: what the word is, type it on the
: keyboard, and then press the
: RETURN key. But there is a 30
: second time limit, so hurry!
t:
t:If you are ready to begin,
t:type 0 to get a hard word;
t:type 1 to get an easy word.
as:#d

je:error
j(d>1!d<0):error

*start
c(d=0):c$="hard"
c(d=1):c$="easy"
```

(Remarks to author.)

(Convert all response letters to lowercase.)

(Reserve space in memory for up to 15 characters each in string variables w\$, g\$, and x\$ ; up to 4 characters in string variable c\$, and six elements of numeric array s , from (0,0) to (1,2) .

(Set initial values for the elements of s .)

(Display these lines of text and blank lines.)

(Display these lines of text, leaving the cursor at the end so that the next instruction begins on the same line.)

(Display these lines of text and blank lines.)

(Accept the student's single keystroke response; store it in numeric variable d .)

(If the Error Flag is raised, jump to error section.)

(If student did not type 0 or 1 , Jump to error section.)

(Label for this section.)

(Depending on current value of d , assign either the string hard or the string easy to string variable c\$ .)

```

u:word
pr:lt30
t:
t:
t:Here`s $c$ word number #(s(d,1))
t:
t:          $x$
t:
t:What do you think it is?
t:
a:$g$
t:
t(g$=w$):That`s right!
tc:You guessed it in
: #(tim(1)) seconds.
cc:s(d,0)=s(d,0)+1
c(g$=w$&tim(1)<s(d,2)):s(d,2)=tim(1)
j(g$=w$):score
th:Sorry,
th(tim(1)=0): your time is up and
th: you did not guess the word
t: $w$ .
w:4
*score
t:
t:Your score on $c$ words is
: #(s(d,0)) right and
: #(s(d,1)-s(d,0)) wrong.
t:
t(s(d,0)>0):Your fastest
: correct answer for the $c$
: words is #(s(d,2)) seconds.
t:

```

```

(Branch to subroutine word ,
returning here when next
e: instruction is executed.)
(Convert all response letters
to lowercase; set a response
time limit of 30 seconds.)
(Display these blank lines and
line of text, substituting
current values for c$ , d ,
s(d,1) , and x$ , including
leading dollar and pound
signs and following spaces.)
(Accept student`s response;
store it in string variable
g$ .)
(Display this blank line.)
(If student typed correct word,
display text, substituting
elapsed answer time for
#(tim(1)) and next space.)
(If student typed correct word,
add 1 to value of s(d,0) .)
(If student typed correct word
and elapsed response time is
less than previous best time,
store new time in s(d,2) .)
(If student typed correct word,
Jump to score .)
(Display these lines of text as
one paragraph, including the
second line only if time had
run out; substitute current
value of w$ for $w$ and
following space.)
(Pause four seconds.)
(Label for this section.)
(Display these blank lines and
lines of text, substituting
current values for c$ , d ,
and s elements, including
leading dollar and pound
signs and following spaces.)
(If at least one correct answer
has been given for this group
of words, display this text,
substituting current values
for c$ , d , and s(d,2) ,
including leading dollar and
pound signs and next spaces.)
(Display this blank line.)

```

c:s(d,1)=s(d,1)+1	(Add 1 to the current value of s(d,1) .)
pr:tØ	(Set unlimited response time.)
t:Press Ø for a hard word.	(Display these lines of text.)
t:Press 1 for an easy word.	
as:#d	
je:error	(Accept single student response and store it in d .)
j(d>1!d<Ø):error	(If the Error Flag is raised, jump to error section.)
	(If student did not type Ø or 1 , Jump to error section.)
j:start	(Jump to label start .)
*word	(Label for this section.)
c:n=s(d,1)	(Assign value of s(d,1) to simple numeric variable n .)
j(d=1):easy	(If d=1 , Jump to easy .)
*hard	(Label for this section.)
c(n=1):w\$="artichoke"	(Assign one of these strings to w\$ , depending on the current value of n .)
c(n=2):w\$="nervous"	
c(n=3):w\$="pedigree"	
c(n=4):w\$="rambunctious"	
. . .	(Additional words added here.)
j:edit	(Jump to edit section.)
*easy	(Label for easy subroutine.)
c(n=1):w\$="grease"	(Assign one of these strings to w\$ , depending on the current value of n .)
c(n=2):w\$="baseball"	
c(n=3):w\$="towel"	
c(n=4):w\$="heart"	
. . .	(Additional words added here.)
*edit	(Label for edit section.)
c:x\$=w\$	(Assign the contents of string variable w\$ to string variable x\$ .)
c:/x\$ /a^ /e^ /i^ /o^ /u^	(Edit x\$ , replacing each of its vowels with a carat.)
e:	(Terminate this subroutine and return to the instruction after one that called it.)
*error	(Label for this subroutine.)
t1:Please type either Ø or 1.	(If answer count is 1, display this text.)
t(%a>1):You must type Ø or 1.	(If answer count is greater than 1, display this text.)
j:@a	(Jump back to last Accept.)

This example uses a numeric array to keep track of the score. There are three values to record for responses to the hard words and the same three

values to record for responses to the easy words. So the first subscript in numeric array  $s$  is either  $\emptyset$  or 1, depending on whether the student has requested a hard or an easy word. And the second subscript is either  $\emptyset$ , 1, or 2, depending on whether the element stores the number of right answers, the number of trials, or the lowest elapsed answer time. For example, the number of correct guesses of easy words the student has made is recorded in element  $s(1,\emptyset)$ , and the fastest answer time the student has achieved on a hard word is recorded in element  $s(\emptyset,2)$ . The simple numeric variable  $d$  is used frequently in place of the first subscript. This allows an instruction to work equally well regardless of the level of difficulty chosen by the student.



## Chapter 6

# Special Effects Instructions

- 114 G: Graphics
  - 114 Graphics Commands
  - 118 Writing Graphics Instructions
  - 120 Repeat-Factors
  - 121 SuperPILOT Graphics Screen
  - 122 Apple II High-Resolution Graphics Screen
  - 122 Text Screen
  - 124 Text and Graphics Cursors
    - 124 Full Screen (No Viewport)
    - 125 With a Viewport
  - 126 Graphics Colors
  - 127 Modifier
  - 127 Example
  
- 129 GX: Execute Graphics File
  - 129 Normal Graphics Files
  - 129 Quick-Draw Graphics Files
  - 130 Example
  
- 131 TS: Type Specify
  - 131 Type Specify Commands
  - 135 Repeat-Factors
  - 136 Example
  - 137 Text Animation
    - 140 Trailing
    - 142 Example
  - 143 Text Colors
    - 145 Programming Notes
    - 145 Output Devices
    - 146 Example
  - 147 Example



151 TX: Execute Character Set File  
154 Example

155 S: Sound  
157 Modifier  
157 Example

158 SX: Execute Sound File  
159 Example

160 AP: Accept Point  
161 Response Timing  
162 Programming Notes  
163 Example  
165 Using Other Pointing Devices

165 V: Audio/Visual Device Control

## G: Graphics

---

`g:command[;...]`

The Graphics instruction, with its associated commands, produces graphic images on the screen. For example, you could Erase the Screen in blue (color 6) using this instruction:

```
g:es6
```

You could then set the drawing Color to orange (color 5); Move to the position `x=23,y=100` ; and Draw a line from there to the position `x=200,y=12` :

```
g:c5;m23,100;d200,12
```

In the summary of the graphics commands below you will find frequent reference to the text and graphics cursors. It is important to note that these are not the same thing. Their positions and movements are determined by different sets of rules and their uses are likewise very different. You will find a complete description of these cursors and their behavior later in this chapter in the section Text and Graphics Cursors.

### Graphics Commands

This section summarizes all the commands that can be used with the Graphics instruction. Two of these commands (the first two listed below) duplicate commands available with the TS: instruction. This overlap is needed to maintain compatibility with both Common PILOT and Apple PILOT. Details about the text and graphics screens, coordinates, and color are discussed in the following sections.

The commands that affect the text mode are listed below.

<u>Command</u>	<u>Form</u>	<u>Description</u>
Text Viewport	<code>g:vl,r,t,b</code>	Sets left, right, top, and bottom character positions for the text viewport. Affects text displayed by T: instructions only, not text displayed by g:tabc instructions. The default viewport is the full screen. After you set a smaller viewport, the text cursor and the graphics cursor are maintained independently.

The instruction `g:v` is equivalent to `g:v0,39,0,23` : it resets the text viewport to occupy the full screen. The text cursor again follows the graphics cursor after this command.

If you set a viewport in a lesson where scrolling may occur, be careful when setting the top and bottom boundaries of the viewport. Depending on the set line spacing (`ts:l`) and the set text size (`ts:s`), the number of lines in the





graphics cursor and text cursor are separated when a `g:x` command is reached, they will be joined at the graphics cursor position before the command is executed. (To place an ASCII character at the text cursor position, use `ts:x`.)

Printing characters are displayed starting at the current graphics cursor position, although the `g:t` command usually is used for this purpose. The first character sent to the screen following ASCII 27 or 16 is used as part of the screen-control command, and not displayed. The first two characters sent following ASCII 30 are used as part of the screen-control command, and not displayed. If a `g:x` command does not contain enough character codes to complete the screen command specified by a `g:x27`, `g:x16`, or `g:x30`, the instruction is handled as if the missing character codes were 32 (space). For further details, see the ASCII Character Codes appendix.

The balance of the `G:` commands, which follow, affect only the Graphics mode. Graphics screens, coordinates, and color are discussed in the following sections.

**Erase Screen in Graphics Color**      `g:esn`      Erases the entire screen in graphics color `n` (0 through 10). Does not change the Plot and Draw color. Moves the graphics cursor to the origin and the text cursor to the top left corner of the text viewport.

<u>n</u>	<u>Color</u>	<u>n</u>	<u>Color</u>	<u>n</u>	<u>Color</u>
0	Black1	4	Black2	8	Reverse
1	Green	5	Orange	9	Single-dot On
2	Violet	6	Blue	10	Single-dot Off
3	White1	7	White2		

Note the differences between this instruction and `g:es` (no number specified), discussed above.

**Color**      `g:cn`      Sets the drawing color to `n` (0 through 10) for future Draw and Plot commands. This color is not changed by a screen-erasing command.

**Offset or Origin**      `g:ox,y`      Defines the current offset position as the absolute point `x,y` and moves the graphics cursor there without plotting. Default is 0,0. Screen extends from `x=0` (left) to `x=559` (right), and from `y=0` (bottom) to `y=510` (top).

The following five absolute coordinate positioning commands may each also be used as a move to relative (Turtlegraphics) coordinates (see below). While you will probably prefer to use the simpler Turtlegraphics commands for most of your graphics programming, the absolute positioning has been retained by SuperPILOT to insure compatibility with Apple PILOT. Graphics coordinates may be real numbers, but are rounded to the nearest integer after plotting.

Move	g:mx,y	Moves the graphics cursor to the point x,y relative to offset, without plotting.
Plot	g:px,y	Plots a point at x,y relative to the offset, and leaves the graphics cursor there. Uses the color set by g:c .
Quit	g:qx,y	Plots a point at x,y relative to the offset, and leaves the graphics cursor there. Uses the color Single-dot Off (10).
Draw	g:dx,y	Draws a line from the current graphics cursor position to the point x,y relative to the offset, and leaves graphics cursor there. Uses the color set by g:c .
Redraw	g:rx,y	Draws a line from the current graphics cursor position to the point x,y relative to the offset, and leaves the graphics cursor there. Uses the color Single-dot Off (10).

Turtlegraphics is a graphing system devised by Seymour Papert and his co-workers at the Massachusetts Institute of Technology. To make graphics easier for children who might have difficulty understanding Cartesian coordinates, Papert, et. al., invented the idea of a "turtle" who could walk a given distance and turn through a specified angle while dragging a pen along.

The next two commands are specific to Turtlegraphics. Either one will set the direction in which the turtle is to move. The other five Turtlegraphics commands are those same five commands just listed; their Turtlegraphics counterparts are differentiated by there being only one variable, d (relative distance), rather than the two Cartesian coordinates, x and y .

Absolute Angle	g:an	Specifies the angle, in degrees, along which the turtle will move, placing the angle in system variable %s . The angle may be specified in the range -359 degrees to 359 degrees. If you do not specify an angle, the initial angle will be 0 degrees, or straight up. 90 degrees is pointing right. Erasing the graphics screen (g:esn) will always return the cursor to the origin (0,0), with an angle of 0 degrees. The number you specify for n will be rounded to the nearest integer.
----------------	------	--

Spin	g:sn	Spins the angle by n number of degrees from its current direction, and also places the resulting angle in system variable %s . Positive numbers will spin the direction clockwise; negative numbers, counterclockwise. The number you specify for n will be rounded to the nearest integer.
------	------	---

Move	g:md	Moves the graphics cursor along the current angle ( %s ) the distance d from the current graphics
------	------	---

cursor position ( %x,%y ), without plotting. The graphics coordinates will be rounded to the nearest integers.

- Plot            g:pd            Plots a point along the current angle ( %s ) at distance d from the current graphics cursor position ( %x,%y ) and leaves the graphics cursor there. Uses the color set by g:c .
- Quit            g:qd            Plots a point along the current angle ( %s ) at distance d from the current graphics cursor position ( %x,%y ) and leaves the graphics cursor there. Uses the color Single-dot Off (10).
- Draw            g:dd            Draws a line from the current graphics cursor position ( %x,%y ) along the current angle ( %s ) at distance d from the current graphics cursor position, and leaves the graphics cursor there. Uses the color set by g:c .
- Redraw          g:rd            Draws a line from the current graphics cursor position ( %x,%y ) along the current angle ( %s ) at distance d from the current graphics cursor position, and leaves the graphics cursor there. Uses the color Single-dot Off (10).

## Writing Graphics Instructions

The object field of each G: instruction may consist of a single graphics command or several commands separated by semicolons. Spaces within a command are allowed, but there must be no space between a semicolon and the following command letter.

All numbers may be integers, simple numeric variables, or arithmetic expressions. Coordinates must be separated by a comma.

The values of n (color) must be from 0 through 10. Values of n outside this range will cause an error message in Author Mode and raise the Error Flag in both Author Mode and Lesson Mode. Note: Graphics colors 8-10 are different from text colors 8-10 used for text foreground and background (see the TS: instruction discussion later in this chapter).

The values of coordinates x and y may be any real number. However, the screen display is limited to those absolute x positions (offset x plus or minus relative x ) from 0 (left edge) through 559 (right edge) and absolute y positions (offset y plus or minus relative y ) from 0 (bottom edge) through 511 (top edge). X or y coordinates outside this display range will not be visible, but are still maintained in %x and %y , within the range 32767 through -32767. Thus, you can draw a line off the screen, spin, and draw a second line back onto the screen.

If the value specified for %x or %y is outside the allowable range, SuperPILOT changes the value so that it is within the range. For a value outside the range, you can determine the value that will be

stored in the system variable by using the following procedures. If the specified coordinate is greater than 32767, subtract 65536 as many times as it takes to bring the value back into the allowable range. If the specified coordinate is less than -32767, add 65536 as many times as it takes to bring the value back into the allowable range. So, for example, suppose you give the following Move command:

`g:m50000,-100000`

(Move the graphics cursor  
50000 x coordinates in a  
positive direction and  
100000 y coordinates in a  
negative direction.)

Assuming that both coordinates start at 0, their new values as stored in the system variables are -15536 for %x and 31072 for %y :

$\%x = 50000 - 65536 = -15536$  (within the range)

$\%y = -100000 + 65536 = -34464$  (not within the range)  
 $= -34464 + 65536 = 31072$  (within the range)

Imaginative use of the Graphics instructions G: and GX: can make a dramatic difference in the effectiveness of your lessons. Even a very simple drawing or other visual effect on the screen gives the student new interest in the lesson and relief from reading endless text.

When you are developing a new drawing or other visual effect, it may be easier to use several G: instructions with just one or a few object commands each, rather than one G: instruction with a long list of commands. Changing the sequence of commands is much easier when these commands are not buried in a long command list. The separate commands take up more space in the text file and execute slightly more slowly. If lesson text space is at a premium or you are fighting for every ounce of speed in execution, you can combine the separate instructions into a long list of commands in a single instruction. Do this only when you are satisfied with the complete drawing.

If you type a long Graphics instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters and continues it (beginning with a continuation colon) on the next line. You may also break the instruction anywhere you wish, by pressing the RETURN key. If you then wish to continue the Graphics instruction, you must begin the new line with a continuation colon or a new G: instruction.

Even though the Editor breaks your instruction in the middle of a command when it reaches the thirty-ninth character, the instruction will be executed correctly. When the instruction is executed, any breaks and continuation colons are ignored, whether they were introduced by you or by the Editor. The object text is treated as if it were one long unbroken string. This means that if you break a graphics instruction and then colon-continue the list of commands on the next line, you must not leave out any separating semicolons or introduce any illegal space between a semicolon and the following command letter.

Regardless of the number of lines a single G: instruction occupies, only the first 250 characters of the instruction are actually used at execution time. The continuation colons are not counted in this total. If the instruction is longer than 250 characters, any characters after the 250th character are simply ignored.

(The following discussion includes examples that you may want to try in Immediate Mode. To do so, you should first type in the following instruction:

```
g:v0,39,0,3;es0
```

This will create a separate, 4-line text area at the top of the screen. You may type g:es0 after each trial to clear the entire screen to black.)

## Repeat-Factors

The relative graphics instructions also allow repeat-factors. These repeat-factors allow you to specify how many times you want SuperPILOT to carry out the same sequence of commands within an instruction. For example, to draw a square 100 units on a side, you could command:

```
g:d100;s90
g:d100;s90
g:d100;s90
g:d100;s90
```

However, the square may be more easily achieved by simply commanding:

```
g:*4(d100;s90)
```

The asterisk must always precede the repeat-factor, and a parenthetical expression must always follow it, the contents of which will be executed the number of times specified. The number you specify must be an integer in the range of 0 through 32,767. If you do not specify a repeat factor, the default of 1 is used. Variables and expressions are allowed. For example, the following command will draw a polygon that has a number of sides equal to x :

```
g:*x(d100;s(360/x))
```

Commands containing repeat-factors may be combined with other graphics commands, using the normal semicolons for command separation. They may also be nested, one inside another, as in this example that will draw a circle of orange triangles:

```
g:o280,240;c5;*18(*3(s120;d150);s20)
```

Translated into English, this says:

o280,240	Originate the drawing slightly below the middle of the graphics screen.
c5	Select Color 5, orange.

*18	Repeat the next expression 18 times.
*3	Repeat the next expression 3 times.
s120;d150	Spin 1/3 of the way around (120 degrees), then draw 150 units distance. (Set direction for each leg of the triangle and then draw it.)
s20	Spin slightly in preparation for the next of the 18 triangles. (18 triangles with 20 degree separation will bring us full circle: $18*20=360$ .)

## SuperPILOT Graphics Screen

You can think of the student's screen as divided into a grid of vertical columns and horizontal rows. Two different grids are used for the same screen, one by the ordinary Type instruction T: for placing text characters, and the other by the Graphics instruction G: for drawing dots.

The G: instruction can draw points and lines on the screen in positions determined by a grid of 560 columns by 512 rows. You can specify any particular screen position by its column and row. The columns are numbered from 0 (the leftmost position) through 559 (the rightmost position). The rows are numbered from 0 (at the bottom of the screen) through 511 (the top of the screen).

When you plot a point on the student's screen, your Graphics commands specify that point's location by giving its horizontal position, or x-coordinate, and its vertical position, or y-coordinate, relative to some absolute screen position. The reference screen position is called the offset. By default, the offset position is at column 0, row 0. With this offset, the relative x,y coordinates of a point are the same as its actual column,row screen position. Thus this instruction:

```
g:p20,35
```

would plot a point on the screen at column 20, row 35. But you can use the Offset command to change the offset to any screen position. For instance, if you used the Offset command to change the offset to column 10, row 17, then the same Plot command:

```
g:o10,17
g:p20,35
```

would plot a point on the screen at column 30 ( $10+20$ ), row 52 ( $17+35$ ). This allows you to draw an image on the screen in one place, and then draw it again in a new place using the same instructions, just by changing the offset position.

If a simple numeric variable appears in the place of a coordinate, the contents of that variable need not be an integer. Since the screen itself is only 560 columns wide by 512 columns high, you can draw lines that extend far beyond the actual boundaries of the screen. Such a line just appears at the first actual screen position on its path and disappears at the last actual screen position on its path.

## Apple II High-Resolution Graphics Screen

This description of SuperPILOT graphics coordinates is further complicated by the fact that Apple SuperPILOT uses the Apple II high-resolution graphics screen for all displays. This screen can actually display dots at only 280 different horizontal positions and at only 192 vertical positions.

Normally you do not have to think about this, as Apple SuperPILOT automatically converts SuperPILOT coordinates into the correct screen-dot locations. However, it is the 280 by 192 screen-dot locations that determine the ultimate resolution of your drawings. If you Plot a SuperPILOT graphics point at  $x=400, y=300$  and another at  $x=401, y=300$  the two points will be displayed as the same actual screen-dot.

An Apple II Hi-Res screen-dot occupies an area two x-coordinate values wide ( $280 * 2 = 560$ ) by two-and-two-thirds y-coordinate values high ( $192 * 2 - 2/3 = 512$ ). This information is useful when you want to know which graphics locations are occupied by text characters, or how much to increase the x-coordinate to skip over three screen dots.

Text characters occupy an area seven Apple II Hi-Res screen-dots wide by eight screen-dots high, or  $7 * 2 = 14$  x-coordinate values wide by  $8 * (8/3) = 21 - 1/3$  y-coordinate values high. The Computation Instructions chapter and the section on the AP: instruction in this chapter both show examples that use this text character information in order to draw graphics that remain outside a set text window. Experimenting in the Graphics Editor, while watching the graphics and text positions displayed by the Graphics Editor's What-information command, may also help you to understand this material. Again, text-character positioning is normally handled for you automatically by Apple SuperPILOT.

Text characters displayed by the Graphics T or X commands always appear in the usual white foreground on a small, black background. Text characters displayed by the Type instruction T: usually appear the same way, but you can change to one of two other screen modes for other kinds of Type character displays. See the TS: section later in this chapter for a description of screen modes.

Color adds still another complication. The colors violet and blue can appear only at odd horizontal screen-dot positions, while green and orange appear only at even horizontal screen-dot positions. Lines drawn in white often appear to be more than one screen-dot wide. (Other colors, depending on the color monitor you are using, may appear more than one screen-dot wide.) Other horizontal screen-dots may also be affected when you plot colors near them. See the appendix Apple II Colors for more information.

## Text Screen

The usual T: instruction can display characters on the screen in positions determined by a grid of 40 columns by 24 rows. The columns are numbered from 0 (the leftmost column) through 39 (the rightmost



column). The rows are numbered from 0 (the topmost row) through 23 (the bottom row).

You need to know these text-character positions if you change the window for text display, using the Graphics (or Type Specify) instruction's Viewport command, discussed later in this section. By default, the text window is the entire 24-row by 40-column screen. It is often convenient to change this. For example, you might wish to draw graphics in the upper half of the screen and display text in a window or viewport that occupies the lower half of the screen.

Once you have set a text Viewport, all T: instructions will display text only at the character positions that fall within the specified window. For the purposes of the T: instruction, it is as if the normal text screen had shrunk to the size of the new viewport. If a T: instruction tries to display a line of text that extends beyond the width of the viewport, the line is broken when it reaches the right edge of the viewport and is continued on the next line. Lines are broken at the space after the last word that fits in the viewport on the current line. If a word is longer than the viewport line, the word is broken to fit on two lines. Reformatting your Type instruction's displayed text to fit in any size viewport occurs automatically. If the bottom line of the viewport already contains text, the next Type instruction causes the text (and graphics, if any) in the viewport to scroll up, making room at the bottom for the new line, while the old top line in the viewport disappears.

This Type instruction:

```
t:Chameleons are red,  
 : chameleons are blue;
```

normally causes this display:

```
Chameleons are red, chameleons are blue;
```

But if you first set the viewport to only ten characters wide:

```
g:v0,9,0,23  
t:Chameleons are red,  
 : chameleons are blue;
```

the display looks like this:

```
Chameleons  
are red,  
chameleons  
are blue;
```

Note: Do not confuse the T: instruction with the T command in the G: instruction. An instruction such as g:tccc also displays characters on the screen, using the same character set, but those characters can begin at any graphics position and are not limited to the normal text-character positions or to the viewport.

## Text and Graphics Cursors

You are already familiar with the text cursor: it is a pointer, not always visible on the screen, which determines where the next character displayed by a T: instruction will appear on the screen. Each time another character is displayed by the T: instruction, the cursor moves to where the next character will appear.

There is also a graphics cursor, with a similar function: it is an invisible pointer that determines where the next graphics point will be plotted. This point is usually on the screen, but its position is not limited to the screen's boundaries. The graphics cursor can be moved only by graphics commands. Most graphics commands start their action at the current graphics cursor position and leave the cursor at the location where that action ends. For example, the instruction

```
g:d20,35
```

draws a line that starts at the current graphics cursor position and ends at the position  $x=20,y=35$  (relative to the Offset position). After this instruction, the graphics cursor position is the same as the position of the line's last point.

The positions of the graphics and text cursors are always stored in the appropriate system variables. System variable %x contains the horizontal position of the graphics cursor, while %y contains the graphics cursor's vertical position. When the graphics cursor is within the screen boundaries, the value of %x will be from 0 (left edge of screen) to 559 (right edge), and the value of %y will be from 0 (bottom of screen) to 511 (top of screen).

The %c and %r system variables contain the coordinate values of the text cursor. The horizontal coordinate, or column, is stored in %c, and is always in the range of 0 (leftmost column) to 39 (rightmost column). The vertical coordinate, or row, is stored in %r, and is always in the range of 0 (top row) to 23 (bottom row).

There are a number of ways you can make use of the graphics cursor system variables in your lessons. Some of them, including the use of %x and %y with the Accept Point instruction, are shown in examples later in this manual.

The %c and %r system variables may also be put to valuable purpose. Any time you move the text cursor relative to its current position, the column and row system variables will help you do so. Also, if the text and graphics cursors become separated because you are using a viewport, you can use %c and %r when you want to move the graphics cursor to the text cursor position.

## Full Screen (No Viewport)

When no viewport has been set, so the entire screen is available for text, the text cursor follows any movement of the graphics cursor as long as the graphics cursor does not go beyond the boundaries of the screen. If a G: instruction draws a line to row 53, column 97 of the

graphics screen, both cursors end up at row 53, column 97 (the text cursor is placed on the normal text-character position that includes the position of the graphics cursor). This lets you use the `g:t` instruction to type labels at appropriate places in your graphics image.

If the graphics cursor is placed at a point that is beyond the boundaries of the screen, the text cursor assumes the screen position whose coordinates are each closest to the corresponding coordinate of the graphics cursor. Thus, if you Draw a line to the point `300,5000` the graphics cursor ends up at `300,5000`, but the text cursor is placed at the text character position that includes the graphics screen position `300,511`.

Note 1: the text cursor is NOT moved to the graphics cursor position by Graphics instructions that do not move the graphics cursor. This means that you can display some text on the screen, using the `T:` instruction (which moves the text cursor), and then issue a `g:x8,8,8,8` instruction, for instance, to backspace the text cursor. This Graphics instruction does NOT move the text cursor to the graphics cursor position (wherever that might be on the screen); it just moves the text cursor four characters back.

Note 2: the graphics cursor does NOT follow the text cursor. After a `T:` instruction displays a line of text on the screen, the graphics cursor remains just where it was before the `T:` instruction.

## With a Viewport

When you use the Graphics (or Type Specify) instruction's Viewport command to set a window for text display, the behavior of the text cursor changes. First of all, the text cursor is now limited to those text character positions that lie within the viewport. Further, when a `G:` instruction moves the graphics cursor to a new location, the text cursor now remains exactly where it was.

Once you have used the Viewport command to set a viewport that does not include the entire screen, the text cursor and the graphics cursor are maintained completely independently. `T:` instructions move the text cursor (within the viewport) and `G:` instructions move the graphics cursor (anywhere on the screen, including the text viewport).

This lets you maintain a scrolling text dialog in the text viewport, while concurrently drawing and changing a graphics image in any other part of the screen. One advantage of this is that your graphics image (if it is outside the text viewport) does not scroll up and off the screen along with the text. Since the normal `T:` instruction now displays text only in the text viewport, you must use the `g:t` command to place text labels in your graphics image. Remember, though, that Graphics commands do not respect the text viewport, so a command like `g:es0` or `ts:0` erases the entire screen, including the text viewport. The only exception to this is the instruction `g:es` or `ts:es` (with no color number after the `ES` command), which erases only the text viewport.

Either of these equivalent instructions

```
g:v0,39,0,23
```

or

```
g:v
```

reestablishes the full-screen text viewport, after which moving the graphics cursor once again drags the text cursor with it.

## Graphics Colors

Graphics colors are selected by number in the graphics instructions `g:c` and `g:esn`, where `n` is a numeric digit from 0 through 10, or a simple numeric variable whose value is any number from 0 through 10. If your lesson specifies a graphics color number outside of this range, an error message will be displayed in Author Mode, and the Error Flag will be raised in both Author Mode and Lesson Mode. The colors corresponding to the integers 0-10 are as follows:

<u>n</u>	<u>Color</u>	<u>n</u>	<u>Color</u>	<u>n</u>	<u>Color</u>
0	Black1	4	Black2	8	Reverse
1	Green	5	Orange	9	Single-dot On
2	Violet	6	Blue	10	Single-dot Off
3	White1	7	White2		

The grouping of these colors is significant. There are many strange interactions between the method color monitors use to produce their colors and the way the Apple II stores so much color information into relatively little memory. You may notice various odd effects when you draw any non-horizontal line across a colored background: black and white lines may appear as dashed or incomplete, colored lines may appear with black and white zebra stripes, or a jagged colored "shadow" may appear around a line. To minimize these normal phenomena, use colors Black1 and White1 with Green and Violet, but use colors Black2 and White2 with Orange and Blue. For a detailed account of how the Apple II produces color and why these effects occur see Appendix F.

If you Plot a point using colors 0, 3, 4, or 7, the display of that point occupies two horizontal Apple II Hi-Res screen-dots to let the color show. Non-horizontal lines that are Drawn in colors 0 through 7 often appear wider than one screen dot. The colors Reverse (8), Single-dot On (9), and Single-dot Off (10) can be used to draw finer lines that are only one screen-dot wide. For black-and-white monitors, you may wish to use mostly colors 0, 8, 9, and 10 for your drawings.

Reverse (8) is not actually a color. A line drawn using Reverse takes its color from whatever the line passes over: it changes White1 to Black1 and Black1 to White1, White2 to Black2 and Black2 to White2, Single-dot On to Single-dot Off and Single-dot Off to Single-dot On, Green to Violet and Violet to Green, Orange to Blue and Blue to Orange. Reverse is a rather magical "color." It lets you draw a line, say, across a complex background, so that every part of the line shows up. When the Erase Screen command uses the color Reverse, it switches the

entire foreground and background coloring of a picture without disturbing the picture itself.

The drawing color is set by default to Whitel (number 3), but if you change it with a `g:cn` instruction, it remains set to your color after that. The color used for erasing the screen in a `g:esn` command is independent of the color used for drawing. An instruction such as `g:es` (with no color number after the `g:es` command) does not erase using a default color of black. Rather, it erases the text viewport portion of the screen in the set text background color (0-20). Text colors are explained in the Type Specify section later in this chapter.

When Graphics commands place text characters on the screen, they use the default character set. Regardless of what text-screen mode the Type instruction is using, or what plotting or background color you are currently using for graphics, the `g:t` and `g:x` commands always display their characters in the usual white, with each character on its own small, black background.

Note: Graphics colors 8-10 are not the same as text colors 8-10. Text colors are discussed later in this chapter.

## Modifier

The eXecute modifier may be used with the Graphics instruction to place entire stored graphics images on the screen. For more details see the section on the `GX:` instruction later in this chapter.

## Example

<code>r:THIS DRAWS A BOX:</code>	(Note to the author.)
<code>g:es4</code>	(Erase Screen in color Black2.)
<code>g:c6</code>	(Choose drawing color Blue.)
<code>g:m30,10</code>	(Move to the point <code>x=30,y=10</code> .)
<code>g:d30,110</code>	(Draw blue line from <code>x=30,y=10</code> to the point <code>x=30,y=110</code> .)
<code>g:d80,110</code>	(Draw blue line from <code>x=30,y=110</code> to the point <code>x=80,y=110</code> .)
<code>g:d80,10</code>	(Draw blue line from <code>x=80,y=110</code> to the point <code>x=80,y=10</code> .)
<code>g:d30,10</code>	(Draw blue line from <code>x=80,y=10</code> to the point <code>x=30,y=10</code> .)
<code>r:THIS DRAWS TWO LARGE FOURS</code>	(Note to the Author.)
<code>d:a4\$(50)</code>	(Reserve space in memory to store up to fifty characters in string variable <code>a4\$</code> .)
<code>c:a4\$="g:a90;m50;a0;d100;s-140;d70;</code>	(Store the quoted string, a Graphics instruction which draws a figure four, in variable <code>a4\$</code> .)
<code>:s-130;d50"</code>	
<code>g:ol0,10</code>	(Set the reference Offset to the point <code>x=10,y=10</code> .)

g:c5 xi:a4\$	(Choose drawing color Orange.) (Execute the contents of a4\$ as a SuperPILOT instruction. This draws an orange figure four with its base at x=60,y=10 .)
t:FOUR	(Display this text starting at the graphics cursor position, next to the last figure four.)
g:o200,350	(Set the reference Offset to the point x=200,y=350 .)
g:c7 xi:a4\$	(Choose drawing color White2.) (Execute the contents of a4\$ as a SuperPILOT instruction. This draws a white figure four with a base at x=250,y=350 .)
g:v30,36,5,15	(Establish a text window from text column 30 to column 36, and from row 5 to row 15.)
T:'Twas brillig, and the slithy :toves did gyre and gimble in the :wabe: all mimsy were the boro- :goves, and the mome raths :outgrabe!	(Display this text in the text window.)
g:tfour	(Display the word "four," starting at the graphics cursor position, which has remained next to the last figure four.)
g:es8	(Reverse all the colors on the screen.)
g:es8	(Reverse again, back to the original colors.)

**Note:** The seven G: instructions that draw the box, at the beginning of this example, could also be written as a single G: instruction with a long list of commands. It would look like this, in the Lesson Text Editor:

```
g:es4;c6;m30,10;d30,11;d80,11;d80,10;
:d30,10
```

In this example, the G: instruction to draw a figure "4" was stored in a string variable and then executed by an XI: instruction. This figure-drawing G: instruction could also have been written as a subroutine and then executed by a U: instruction.

Before the Viewport is set, the text cursor follows the graphics cursor, and the T: instruction displays text starting where the last graphics command ended. After setting the Viewport, the T: instruction displays its text within the Viewport, independent of any graphics moves. Note how the text in the Viewport scrolls without affecting the graphics on the rest of the screen. Any graphic element that happens to be within the Viewport, however, will be scrolled away right along with the text.

## **GX: eXecute Graphics File**

---

gx:file name of a graphics image stored on the lesson diskette

The eXecute Graphics instruction reads a graphics file stored on the lesson diskette, and redrawing it on the screen. Unlike the G: instruction, no graphics commands appear in the object field of the GX: instruction. Instead, the file name of the stored graphics image is the object. For example, to use the stored graphics image whose file name is picture, your lesson would use this instruction:

gx:picture

(Find the graphics image whose file name is picture, stored on the lesson diskette, and redraw it step-by-step on the screen.)

The graphics image file is created and named while you are in the Graphics Editor. If you are using a normal one-disk-drive Lesson Mode system, all image files used by GX: instructions must be on the same diskette as the lesson that includes the GX: instructions. For using a system with additional disk drives, see the appendix Using More Disk Drives.

If a graphics file with the specified name is not found on the lesson diskette, you are given no message, even if you are running a lesson in Author mode, and the Error Flag is not raised. Lesson execution simply goes on to the next instruction. This lets you write and test lessons that include GX: instructions before you develop the stored graphics images named in the GX: instructions. Later, when you provide the correctly named image files, your lesson will automatically incorporate those images.

### **Normal Graphics Files**

A GX: instruction that specifies a normal file name (such as picture) recreates the stored image step-by-step, just as it was created in the Graphics Editor. The previous contents of the screen are not removed by this instruction. The stored graphics image will be drawn over any text or graphics already on the screen unless a g:es (erase screen) instruction precedes the GX: instruction.

### **Quick-Draw Graphics Files**

It is also possible, when you create a graphic in the Graphics Editor, to give your graphics image a file name with an exclamation point anywhere in the name. For example, you might give the name sunset! to a drawing of the sunset. Such a file is called a quick-draw file, and the Graphics Editor saves the color of every screen-dot for that image, in addition to the sequence of steps that created the image. A GX: instruction such as

gx:sunset!

(Find the graphics image whose file name is sunset! , stored on the lesson diskette, and quickly display that complete image on the screen all at once.)

specifies the name of a quick-draw graphics file, and quickly replaces the entire contents of the student's screen with the stored image. The stored quick-draw image is not redrawn step-by-step, but is placed on the screen all at once, erasing anything that was on the screen before the instruction. Note: However, it takes much more space on the lesson diskette to store a quick-draw file than to store an ordinary graphics file.

Unlike a normal step-by-step file, the quick-draw image cannot be restricted to certain portions of the screen. A quick-draw file always replaces everything on the student's screen, including any text, even in a viewport.

If the dot-by-dot quick-draw image file is not found on the diskette when a quick-draw file name is specified, SuperPILOT looks for the step-by-step command file with the same file name. Lesson execution goes on to the next instruction without affecting the screen only if neither portion of your quick-draw file can be found. (Note: If you have access to an Apple Pascal system, you can look at the directory listing of your lesson diskette graphics files in the Pascal Filer. A dot-image file has the suffix .FOTO after its file name, while a command-sequence file has the suffix .GRAF . The quick-draw file sunset! appears as two files: SUNSET!.GRAF and SUNSET!.FOTO .)

See the section of the Apple II SuperPILOT Editors Manual dealing with the Graphics Editor for further details on creating and using the graphics images.

## Example

pr:u	(Convert response letters to capitals.)
g:es5;m0,0	(Erase the Screen to the background color Orange, and move the text cursor to the lower left corner, where the graphics cursor already is.)
t:           This is an Apple.	(Display this text at the bottom of the screen.)
gx:apple	(Redraw the stored graphics image named apple , step-by-step on the screen.)
w:5	(Wait five seconds, or until student presses any key.)
gx:globe!	(Quickly replace current screen contents with the quick-draw graphics image globe! .)



g:m0,0;t	What is this a map of?	(Move graphics and text cursors to lower left corner of the screen, and display text.)
a:		(Accept student's response.)
m:	WORLD!EARTH!PLANET!GLOBE	(Did student type one of these match words?)
...		...
...		...

## TS: Type Specify

ts:command[;...]

When you modify the Type instruction with the Specify modifier, you can specify how and where the text will appear. Apple SuperPILOT gives you the power to pick foreground and background colors, single or multiple spacing between lines, single- or double-sized characters, animation sequences, etc.

### Type Specify Commands

There are 17 different commands that can be given with the TS: instruction, each of which is issued by a single letter (and in some cases a variable name or a number or numbers) immediately following the colon. The following table gives the name of each command, its proper form, and a description of its operation. The descriptions of the first nine commands listed (through Quiet) are complete, and the example immediately following the table illustrates their use in a SuperPILOT lesson. However, to better understand the use of the remaining eight commands, you will want to read the Text Animation and Text Colors discussions at the end of this section. Specifically, Text Animation covers the use of the Animate, Walk, and Delay commands; and Text Colors covers the use of the Background, Foreground, Erase Text Screen, Inverse, and Normal commands.

<u>Command</u>	<u>Form</u>	<u>Description</u>
Viewport	ts:vl,r,t,b	Sets the boundaries of the text window (viewport). The instruction ts:vl0,30,5,20 would set the text window horizontally from position 10 on the left to position 30 on the right, and vertically from position 5 at the top to position 20 at the bottom. The default viewport is the full screen, with coordinates of 0,39,0,23, which can be called by the command ts:v . Expressions (enclosed in parentheses) and variable names may be substituted for any of the boundary values. See the Text Instructions chapter for details on how the text viewport operates.
Goto XY	ts:gx,y	Makes the text cursor go to the specified x and y coordinates relative to the text viewport set by a ts:v (or g:v ) command. The text screen extends from x=0 (left) to x=39 (right) and from y=0

(top) to y=23 (bottom). With a full-screen viewport, the command `ts:g32,7` moves the text cursor to the character position in the 33rd column from the left and the 8th row from the top (remember that the first column and the first row are numbered 0). If either or both of the x or y coordinates you specify are beyond a boundary of the set text viewport, the cursor moves to the boundary. Expressions (enclosed in parentheses) and variable names may be substituted for either coordinate value.

Size of Type      `ts:sn`      Sets size of text characters: 1=regular-size, 2=double-size characters. Double-size characters are often more dramatic, more readable, and display color better than single-size characters. When using double-size characters, you should not set the vertical dimension of the viewport to an odd number of rows. A setting of `ts:v0,39,0,10`, for example, will produce a text viewport of 11 rows (0-10), and when double-size characters are scrolled up, only half of the top line will be visible.

Thickness of Type      `ts:tn`      Selects normal or boldface type. (1=normal type, 2=bold face type.) Boldface is very effective for highlighting parts of text. Many authors use boldface for all printing of normal-size white characters on black background, because it reduces color fringes when the text is viewed on a color monitor.

Modes of Type      `ts:mn`      There are three basic modes in which you can type:  
1=normal: each character Typed erases the previous contents of that character position. This is the default mode.  
2=overstrike: each new character is Typed on top of the old contents of that character position, partially erasing it. Best results are achieved when using a black text background.  
3=x-or: eXclusive-OR is a special mode that lets you display a new character over an existing character or graphic, and then restore that prior character or graphic by displaying the new character again, in the same location.

Which mode you choose will depend upon your needs at the time. Normal creates the most consistently clear characters. X-OR lets you temporarily overwrite complex backgrounds that would normally take a major effort to restore. Overstrike is seldom used, but can be used to achieve some very clever effects, such as characters that slowly become readable as you Type partial versions, one on top of the other. Further details on these three modes is given in the Screen Modes section of Appendix A.

Transmit ASCII      `ts:xn`      Sends to the screen at the text cursor position the character whose ASCII number is n. For example, since ASCII character #8 is a backspace, the instruction `ts:x8,8,8` --or the equivalent expression `ts:*3(x8)` --moves the text

cursor on the screen back three spaces. The 128 ASCII characters are listed and described in Appendix A. (To send an ASCII character to the graphics cursor position, use the `g:x` command.)

Line Spacing      `ts:ln`      This is equivalent to typewriter spacing: single-spacing is every line used, double-spacing gives one blank line between each typed line, etc. You can specify up to 15 lines of space between lines of text, by using the maximum value, `ts:15`. If you specify an amount greater than 15, the last set line spacing is left unchanged. Expressions (enclosed in parentheses) and variable names may be substituted for `n`.

The blank lines are always one normal character height high, so that when using double-size characters with double-spacing, there will be a single-size line between each double-size line of text. To get double-sized blank lines between double-size lines of text you must select triple-spacing. (One line of double-size text plus two lines of single-size blanks). This scheme may seem strange on paper, but, in practice, you will find you rarely want more than one blank line between double-size character lines.

Note: The `ts:l` command specifically tells SuperPILOT to place the indicated number of blank lines between each of the lines of text displayed AFTER the `ts:l` command. It will NOT place the indicated number of spaces between the text that precedes the command and the text that follows it. So, for example, if the current line spacing is set to 1, and you issue a `ts:15` command, the next text line will be displayed with only a single line space, and then future text will be displayed after five line spaces.

Print            `ts:p`            Send all text output simultaneously to the screen and to a printer when connected to the Apple II at the printer slot 1. Any printer will work with your SuperPILOT lessons, as long as it uses a standard serial or parallel interface. If your SuperPILOT lesson encounters a `ts:p` instruction and a printer is not attached to the system, lesson execution will proceed to the next command in that instruction or to the next instruction, without raising the Error Flag. If a printer is attached but not turned on, lesson execution will wait at a `ts:p` instruction until the printer is turned on.

Characters printed on paper will be unaffected by Type Specify settings for Viewport, Goto, Size, Thickness, Mode, Background, Foreground, or Inverse, and the characters will always appear in the printer's standard style regardless of the character set you have specified in your lesson. Any graphics in your lesson will be ignored by the printer. If your printer is an Apple Silentye, however, you can print

on paper the exact contents of the screen by issuing these instructions:

```
ts:p  
t:#{chr(17))
```

If there is a graphic image on the screen when SuperPILOT executes the `t:#{chr(17))` instruction, it will be printed on paper in shades of black and gray corresponding to the intensities of the colors used in the image. Any text also appearing on the screen at that time will also be printed by the `Silentype:` the effect of the `t:#{chr(17))` instruction is to reproduce each screen-dot in a corresponding location on the paper.

Note: If you wish to print part of a lesson on paper, but do not want that text to appear on the screen, you can do so by setting both the Background and Foreground colors to black (`Ø`) or to any other combination that causes only a solid background to appear (see the Text Colors Chart later in this chapter). Because the printer ignores any text color settings used to display text, the text will be printed even though it will not appear on the screen.

- Quiet      `ts:q`            Turn off output to the printer.
- Animate    `ts:an$`           Types the contents of `n$` very rapidly on the screen. This method does no formatting. It is used mainly for typing special characters (bees, frogs, little people) that you have created using the character set editor or to quickly display text such as prompt messages. The animation occurs when you repetitively Type slightly different versions of the same character or group of characters over and over again. See the Text Animation section later in this chapter for details on the use of the Animate command.
- Walk        `ts:wl,r,u,d`       Moves ("walks") the cursor one space in the specified direction, either left, right, up, or down. For example, the instruction `ts:wuur` could be used to move a knight on a chessboard two spaces up and one space right. (The commas separating multiple moves are optional.) This command is used mainly for animating strings that use a specially designed character set. See the Text Animation section later in this chapter.
- Delay       `ts:dn`            Causes a pause in lesson execution for a length of time specified by `n`. A value of 1 for `n` is equal to a delay of 1/60th of a second. See the Text Animation section for details on using the Delay command with Animate and Walk to create special effects.

Background Text Color `ts:bn` Sets the color of the background used to display text on the screen. The 21 available colors are given in the section Text Colors later in this chapter. The text colors 0-7 are the same as the graphics colors 0-7. The default background color is 0 (black).

Foreground Text Color `ts:fn` Sets the color of the foreground used to display text on the screen. The available colors are given in the Text Colors discussion later in this section. The text colors 0-7 are the same as the graphics colors 0-7. The default background color is 3 (white).

Inverse `ts:i` Switches background and foreground colors. The specified background color will be used for the foreground, and the specified foreground color will be used for the background. The default is black printing on a white background.

Normal `ts:n` Cancels a prior `ts:i` instruction, so that the background and foreground colors are returned to their originally set values. Default is to white characters on a black background.

Erase Screen in Graphics Color `ts:esn` Causes the entire screen to be erased in the graphics color specified by `n`. For example, to erase the entire screen in Green, you would use the command `ts:es1`. To reverse each screen color present, you would use `ts:es8`. If you have previously set a viewport, it will be ignored: the entire screen is always erased regardless of the viewport boundaries.

However, if you specify no value for `n` in your `ts:es` command, only the text viewport will be erased, using the text background color previously set by a `ts:b` command.

This is really a Graphics command, usually called by the identical command `g:es`, but it works the same way as a Type Specify command.

Note: The Erase and Viewport instructions above can be used with a `g:` in place of the `ts:`. For example, the instruction `g:esn` works exactly the same as `ts:esn`. The `g:` format for these two instructions was retained from Apple PILOT in order to insure that programs written in that earlier version would run equally well under SuperPILOT.

## Repeat-Factors

The Type Specify instruction also allows repeat-factors, which operate in the same way as repeat-factors with the graphics commands. With the `TS:` instruction, repeat-factors are of the greatest value when used in Animation sequences.

For example, if you stored an animation sequence in string variable a\$, you could give the following command to execute the two commands within the parentheses 30 times:

```
ts:*30(wr;aa$;)
```

The asterisk must always precede the repeat-factor, and a parenthetical expression must always follow it, the contents of which are executed the number of times specified. The number you specify must be an integer in the range of 0 through 32,767. If you do not specify a repeat-factor, the default of 1 is used. Variables and expressions are allowed. For example, if your lesson stores the student's name in string variable n\$ and the student's age in numeric variable y, the following command prints the student's name once for each year:

```
ts:*y(an$;wd)
```

For examples of repeat-factors in use, and for an explanation of how the animation commands work, see the Text Animation section later in this chapter.

## Example

The following example illustrates the use of several text formatting commands of the TS: instruction. It is structured so the subroutine score will be called at the end of each section labeled question1, question2, etc. Each of these question sections Types a question, Accepts an answer, looks for a Match, and Types a response before calling the score subroutine. Examples that include text color and text animation are found later in this chapter.

pr:u	(Convert all response letters to capitals.)
d:a\$(4)	(Reserve space in memory for storing up to 4 characters in string variable a\$.)
c:s=0	(Store an initial value of 0 in simple numeric variable s.)
u:score	(Branch to subroutine called score, returning here when next e: is executed.)
*question1	(Labels for questions.)
...	
*question2	
...	
*question20	
*score	(Label for subroutine score.)

<pre>ts:v0,39,18,23;es;sl;t1</pre>	<pre>(Set a viewport to occupy the last six lines of the screen; erase the new viewport; and display future text in normal size and thickness.)</pre>
<pre>cy:s=s+1</pre>	<pre>(If last Match was successful, add 1 to the value of s.)</pre>
<pre>t:Correct: #s      Incorrect: #(q-s)</pre>	<pre>(Display text, substituting the current values both for #s and the next space, and for #(q-s) and the next space.)</pre>
<pre>t:Press RETURN for the next : question.</pre>	<pre>(Display this text.)</pre>
<pre>t:Type STOP to end the lesson.</pre>	<pre>(Display this text.)</pre>
<pre>a:\$a\$</pre>	<pre>(Accept student's response and store in string variable a\$.)</pre>
<pre>e(q=20!a\$="STOP"):end</pre>	<pre>(If there are no more questions (here, assume 20 questions), or if student typed STOP, terminate the subroutine and jump to label called end.)</pre>
<pre>ts:v;es;s2;t2</pre>	<pre>(Reset the viewport to the full screen; erase the entire screen; display future text double size and boldface.)</pre>
<pre>c:q=q+1</pre>	<pre>(Add 1 to the current value of numeric variable q.)</pre>
<pre>t:Question ##q :</pre>	<pre>(Display text, substituting the current value of q for #q and following space.)</pre>
<pre>e:</pre>	<pre>(Terminate subroutine; return to the instruction after the one that called subroutine.)</pre>
<pre>*end</pre>	<pre>(Label for section end.)</pre>

This example uses a viewport at the bottom of the screen to display the student's score after each question and to give the prompt lines. When a new question is requested by the student, this information disappears, to free the entire screen for the question and to avoid confusing the student with prompts that do not apply at that moment. The questions and answers are displayed in double-size boldface type, which further sets them apart from the less important score and prompt information. All of the switchin to new viewports and type styles is done in the score subroutine, so that new question and answer blocks may be added without repeating format instructions.

## Text Animation

To better understand the discussion below, type the following instructions into the Lesson Text Editor and store them on your LESSON diskette under the lesson name WALKING .

```
d:r1$(15);r2$(15);r3$(15)
c:r1$=" a / bc/ de";r2$=" fg/ hi/ jk";r3$=" lm/ nop/ qr"
tx:maxwell
ts:g10,10;*30(ar1$;d15;ar2$;d15;ar3$;d15;wr)
```

Be sure to include the spaces as indicated in the Compute instruction. When this lesson is Run, a figure known as Maxwell will walk across your screen.

SuperPILOT creates the effect of animation in much the same way that a cartoonist does--by presenting complete, slightly different images in rapid succession. The images you use, like the image of Maxwell in the above example, are not produced in the Graphics Editor or even with Graphics commands. They are actually groups of characters that have been specially designed in the Character Set Editor. By using the TX: instruction in your lesson, you instruct the Apple II to substitute your new character designs for the letters and other keyboard characters in your Animate commands.

You are not required to use special character sets with your animation effects, however: you may use text animation to move words and sentences around on the screen, flash warning messages, and so forth. But the most dramatic use of the animation commands is with specially designed figures created in the Character Set Editor. If you have not already familiarized yourself with the use and capabilities of this Editor, you may want to do so before reading this discussion. See The Character Set Editor chapter of the Apple II SuperPILOT Editors Manual.

The commands you use with the TS: instruction to achieve an animated effect are Animate, Walk, and Delay. The Animate command displays an image, the Walk command moves the text cursor to a new position for the next image to appear, and the Delay command causes a pause between one image and the next, so you can control how fast your animation will "move." We'll consider each of these three commands separately, then put them together to show their interaction.

The Animate command takes the form ts:a , followed by either a literal string or the name of a string variable. If you use a string variable, it must be Dimensioned and assigned a value in a Compute, Accept, or File Input instruction, just as any other string variable, or else its name in the Animate command will be displayed as a literal string.

When the Animate command is executed, the string or string-variable contents are displayed on the screen according to the following rules:

1. The characters are displayed beginning at the current text cursor position, and the cursor will return to that same position at the end of that Animate command.



2. A slash character (/) in the string or string variable is not displayed on the screen; instead, it causes the cursor to move to the next line, directly below the cursor's starting position, before displaying future characters. For example,

```
ts:aone/two/three
```

will produce the following on the screen:

```
one
two
three
```

Note: If multiple line spacing has been specified by a `ts:l` command, the slash will cause that multiple spacing to occur.

3. When the object of the command is a literal string, any spaces at the beginning of the string are not displayed. Leading spaces in the contents of a string variable are displayed, however. Internal spaces and following spaces are displayed when they occur in either a literal string or the contents of a string variable.

This rule is important to note for at least two reasons. First, you frequently need leading spaces in the strings you animate, to avoid the phenomenon of "trailing." See the Trailing discussion below. Second, if the string to be animated enters the lesson through an Accept instruction, leading spaces are automatically "edited out," unless you modify the instruction with the eXact modifier, as in `ax:$a$`.

4. Any characters that fall outside the right edge of the screen or text viewport are not displayed. Likewise, any characters that fall below the bottom edge of the full screen are not displayed. However, if the bottom edge of the viewport is set to a value less than 23, the animated image will scroll up, if necessary, to display the last line of the animation.
5. A semicolon cannot be used as part of a literal string in the object of an Animate command, because it signals the end of the command.

The Walk command is used between Animate commands to move the cursor to a new location, thus creating the effect of movement. The form of the Walk command is `ts:w`, followed by one or more of four letters: `r`, `l`, `u`, or `d`, for right, left, up, or down movements. Each of these letters moves the cursor one character position in the specified direction. Multiples of these four letters may be given to move the cursor. For example:

```
ts:aa$;wlld;aa$
```

causes the contents of string variable `a$` to be displayed twice: first at the current cursor position, and second at a position two character spaces to the left and one character space down.

Finally, you use the Delay command to slow down the animation process. This command takes the form `ts:d`, followed by a numeric value to designate the length of the delay. A value of `60`, as in `ts:d60` will cause a delay of one second. The numeric value may be an expression, enclosed in parentheses, or it may be a simple numeric variable. Negative values are allowed, but they result in no delay. A delay of `10` seconds may be achieved with a `ts:d6000` command, but values greater than `6000` cause delays of unexpected length.

## Trailing

A problem frequently encountered by authors using animation commands for the first time is a phenomenon called trailing: when an image moves across the screen, it seems to leave part of itself everywhere it goes, in a kind of trail. Sometimes this effect may be desirable, but for animations like the walking Maxwell above, it will never do. To understand why this occurs, we need to remind ourselves that no motion is actually occurring on the screen, despite what our eyes tell us. The image we are animating is simply being displayed on the screen in one location and then another--it will remain in the first location unless it is erased or something else is printed in its place. For example, if we give the following command:

```
ts:g30,10;*10(aLet's Go/Dodgers!;w1) (Set the cursor to position
                                     30,10; then produce this
                                     animation sequence a total
                                     of ten times: display the
                                     indicated text in two lines
                                     and move one space left.)
```

the screen will appear like this:

```
Let's Goooooooooooo
Dodgers!!!!!!!!!!!!
```

Each time the string is displayed, each character in the previous display is replaced by a new character, except for the ones on the far right; they remain on the screen because other characters are not printed over them.

This problem is avoided by strategically placing blank spaces in your strings. These blank spaces are displayed with the rest of the string, "erasing" the contents of the character space where they appear. The result is that each image seems to disappear completely when a new one is displayed. For example, the above command could be changed by adding a space at the end of each line, as follows:

```
ts:g30,10;10(aLet's Go /Dodgers! ;w1)
```

No trail will be left when this command is executed.

However, adding a space at the end of the line solves the trailing problem only as long as the Walk command moves the cursor in the left direction and only one space at a time. If the string is redisplayed

to the right, above, or below the previous display, or if it is displayed two or more spaces to the left, a different trail will appear. The best protection against such an occurrence is the careful design of your strings before you place them in your animation commands. Graph paper can be very useful for this purpose. For example, let's say we want to animate the following message:

```
Look Ma...
I can fly!
```

Furthermore, we want to be able to "move" the message up, right, and left, but we will never move more than one space at a time in any direction. We could write the message out on a grid, like this:

	L	o	o	k		M	a	.	.	.	
	I		c	a	n		f	l	y	!	

Then we can indicate a border of blank spaces on each edge of the message opposite a direction we will want it to move. For example, to avoid a trail when moving up, we put a border of blanks on the bottom edge of the message. Here is what our grid will look like after placing an asterisk in each space that must be blank:

*	L	o	o	k		M	a	.	.	.	*
*	I		c	a	n		f	l	y	!	*
*	*	*	*	*	*	*	*	*	*	*	*

If we want the ability to display the message two spaces above its previous position, we would need another extra line of spaces at the bottom.

We can now translate this grid into a string to be used in an animation command. Simply begin at the top left corner and read to the right, remembering to place a slash mark at the end of each line. A Compute instruction that stores this string in a previously Dimensioned string variable might look like this:

```
c:f$=" Look Ma... / I can fly! / "
```

Note: No slash marks are needed at the very beginning of the string when it is stored in a string variable. However, if we were to include this string literally in an Animate command, we would need to begin the message with a slash, so that the leading space would not be dropped:

```
ts:a/ Look Ma... / I can fly! / ;wur
```

Note: The beginning slash also causes the cursor to drop down a line before displaying the message. If this approach is used, remember to begin the animation with the cursor a line above where you want the image to appear.

This example, like the others in this section, uses an English message as the string to be animated, because it is easier to describe on a printed page. However, the strings used could easily be replaced by combinations of characters that make no literal sense, but when used with a special character set produce images like Maxwell. Also, you will usually find it much more convenient to store your animation sequences in string variables before including them in Animate commands. String literals were used as examples here for clarity.

## Example

```
d:w1$(10);w2$(21);w3$(21)
```

(Reserve space in memory to store up to 10 characters in string variable w1\$ and up to 21 characters each in string variables w2\$ and w3\$ .)

```
c:w1$=" WARNING! "  
c:w2$=" POSSIBLE /EXPLOSION!"  
c:w3$=" REACTOR / TOO HOT! "
```

(Store these strings in string variables w1\$ , w2\$ , and w3\$ .)

```
*warnings
```

(Label for a subroutine to display warning messages.)

```
ts:gl5,10;m3;i
```

(Move the cursor to screen location 15,10 relative to the viewport boundaries; use Type Mode 3 (X-Or); reverse the foreground and background colors.)

```
ts(x>1000):*8(aw1$;wd;aw2$;d20;wu)
```

(If the current value of x is greater than 1000, perform this animation eight times: display the contents of w1\$ , move down one line and display the contents of w2\$; pause 1/3 second; move up one line.)

```
ts(h>4700):*8(aw1$;wd;aw3$;d20;wu)
```

(If the current value of h is greater than 4700, perform this animation eight times: display the contents of w1\$ , move down one line and display the contents of w3\$; pause 1/3 second; move up one line.)

ts:g0,0;ml;n

(Move the cursor to the top left corner of the viewport; set the Type Mode back to normal; set the foreground and background colors back to normal.)

e:

(Terminate the subroutine, returning to the instruction following the one that called the subroutine.)

A repeat-factor of eight is given for displaying the warning messages, but they will appear only four times; every other time the message is "displayed," the original screen contents are restored, because the X-Or Type Mode has been set ( ts:m3 ). The effect is that the warning seems to appear on top of the screen contents for an instant, then disappear again. Note: The warning message should be placed over a solid background; if it appears over text, graphics lines, or mixed screen contents of any kind, it may be difficult or impossible to read.

## Text Colors

With SuperPILOT you are not limited to displaying text on the screen in the familiar white-on-black fashion. You may, instead, use the ts:b and ts:f commands, discussed above, to specify text printing colors. When your lesson executes the instruction ts:bl4;fl , for example, the text-printing instructions that are executed thereafter will produce characters of foreground color 1 (Green) on a square of background color 14 (Violet/White). Below is a list of the color combinations available to you. Each of these 21 colors may be specified as the foreground color or the background color, or both.

n	Color	n	Color	n	Color
0	Black	8	White/Black	13	Green/White
1	Green	9	Green/Black	14	Violet/White
2	Violet	10	Violet/Black	15	Orange/White
3	White	11	Orange/Black	16	Blue/White
4	Black	12	Blue/Black	17	Green/Orange
5	Orange			18	Green/Blue
6	Blue			19	Violet/Orange
7	White			20	Violet/Blue

Table 6-1. Text Color Chart

For convenience, the first eight text colors (0-7) are numbered the same as the first eight graphics colors, and include all of the "solid" hues available. (Note: You do not have to worry about using 0 and 3 with some colors and 4 and 7 with others, as you must with the Black and White graphics colors. Unlike graphics colors, these text color pairs are identical.) The remaining 13 colors are each a mixture of two of the six solid colors: black, white, green, violet, orange, and blue.

The method that the Apple II uses to mix these six colors is a complex one beyond the scope of this discussion. (If you would like to explore the Apple II's color capabilities more deeply, see Appendix F.) But you can use colors effectively--and impressively--without understanding how the Apple II makes them, because SuperPILOT's text color scheme has been designed to eliminate virtually all of the color-mixing problems that might otherwise result.

Simply select a background color and a foreground color and include their corresponding numbers in your Type Specify instruction. For example, `ts:b9;f16` or `ts:f12;b20`. With only one exception, discussed below, the result is that each future character displayed on the screen will be the color you selected for the foreground, and the character space behind it will be the color you selected for the background.

The exception to this rule has to do with the four solid colors other than Black and White. These four--Green (1), Violet (2), Orange (5), and Blue (6)--are fussy about whom you pair them with. In order to keep them happy, you need to think of them in two groups (the shaded areas in the chart of colors above). The first two colors, Green and Violet, like each other fine, but they will refuse to appear in the same character space with any color that is partly Orange or partly Blue. Likewise, Orange and Blue will not let you pair them with any color that contains Green or Violet.

If you do specify an unfriendly combination, the foreground color will be changed so that all the colors are from the same group. When this happens, it causes no problems for the Apple II and has no ill effect on the running of your lesson. It means only that the foreground color you see is not the one expected. For example, if you specify a background of Green (1) and a foreground of Orange/White (15), the Orange part of the foreground is changed to Green. The result is Green/White characters, no different in appearance from color number 13.

Here's a summary of the exception, in a nutshell:


If your foreground or background choice is either Green (1) or Violet (2), your other choice must not contain either Orange or Blue, or the foreground color will change.

If your foreground or background choice is either Orange (5) or Blue (6), your other choice must not contain either Green or Violet, or the foreground color will change.

Exceptions to rules are usually a nuisance, but in this case the finicky nature of the solid colors has a hidden benefit. When one of them is paired with either color 18 or color 19, one of two new colors is created, not otherwise available to you. The new colors are Green/Violet and Orange/Blue. Here's how you get them:

To create Green/Violet, pair either 1 or 2 with either 18 or 19.

To create Orange/Blue, pair either 5 or 6 with either 18 or 19.



Of course, you can avoid even having to think about these exceptions, if you wish, by never using colors 1, 2, 5, or 6. If you do not use one of these four solid colors for either your background or foreground color, both the foreground and background colors always appear as their names suggest. In fact, if you avoid these four colors entirely, you still have over 400 possible color combinations to select from, maybe all the options you care to have. The choice, as always, is yours.

## Programming Notes

1. It is a good idea, whenever possible, to use the Erase Screen command, `ts:es`, before displaying text in a new background color. This will avoid the "ragged-right" effect caused by text that does not fill the entire 39-character line. Set your background color (and a viewport, if desired), erase the screen, and then display your text.
2. Text colors generally look better when you specify double-size (`ts:s2`). Some color pairs produce a slight halo effect around the characters, which may make them more difficult to read in single-size.
3. If you specify the same color for both the foreground and the background, no characters will be visible, as you might expect. This effect can be put to good use to "hide" some or all of your text under certain lesson conditions and disclose it under other conditions. For example, a "fill-in-the-blanks" lesson could be designed with the missing words always in place--by changing the text foreground color, you control whether words or blanks appear on the student's screen. Note: The four shaded colors in the Text Color Chart cause this same "blank space" effect in certain pairs. These pairs are 1-5, 1-17, 2-6, 2-20, 5-17, and 6-20.

## Output Devices

The foreground and background colors you specify in your lessons have no effect on the appearance of text displayed by a printer. Printer output is insensitive to all Type Specify settings for text color, Size, Thickness, and Mode, as well as to different character sets.

There will be important differences in text appearance depending on the screen being used, however. Color television sets vary, and the colors they produce may not be exactly what you expect. To get the best results, adjust the tint control to produce violet (2) as well as possible.

On a black-and-white television, shades of gray will appear in place of the colors, depending upon the colors' densities. No matter what color combinations you choose, the readability of the text on a black-and-white television will be at least as good as if the set were a color set. If the screen is a black-and-white computer monitor, however, most color

combinations are unreadable. If you know that your lesson may be displayed on a black and white monitor, you can take one of these precautions:

1. Use a color combination that is readable on a black-and-white monitor. Background colors of Black (0 or 4) or White (3 or 7) are the safest choices. Use caution with any other selection.
2. Include a Type instruction at the beginning of the lesson, asking the student to identify the monitor in use. Subsequent instructions could then change the color settings depending upon the answer the student gives.

## Example

pr:u	(Convert all response letters to capitals.)
d:p\$(80)	(Reserve space in memory for up to 80 characters in string variable p\$ .)
*question1	(Label for first section.)
ts:f3;b6;es;s2;t2	(Set foreground color white and background color blue; erase the screen in blue; display future characters in double-size, boldface.)
t:Who was the only President to : serve two nonconsecutive terms? ts:b19	(Display this text.)
a:\$p\$	(Set background color violet/orange.)
ms:CLEVELAND	(Accept student's response, and store it in string variable p\$ .)
ts:es;i	(Did student type this text correctly or with only one spelling error?)
ty:That's right!	(Erase screen violet/orange; display future characters violet/orange on white background.)
tn:No, that's not right.	(If Match was successful, display this text.)
ts:n	(If Match was unsuccessful, display this text.)
t:Grover Cleveland served as : President from 1885-1889 and : from 1893-1897.	(Return to white character on violet/orange display.)
u:score	(Display this text.)
	(Branch to subroutine score to compute new score.)



\*score

(Label for score subroutine.)

The brief example above illustrates how text colors can be used to vary the appearance of the screen. Questions are displayed in a different color from answers, and the information part of the answer is a different color from the "right or wrong" part--thus the colors serve as visual cues in learning reinforcement. The following example places this question-and-answer block in the context of a complete lesson to illustrate how a wide variety of TS: commands can work together in creating a special effect.

## Example

```
r:*****
: * UNITED STATES PRESIDENTS *
: * GRADE LEVEL 6 *
: * REINFORCEMENT FOR STUDENTS *
: * WHO HAVE COMPLETED UNIT ON *
: * PRESIDENTS *
: *****
(Notes to author.)

pr:u
(Convert all response letters
to capitals.)

d:p$(40);a$(4)
(Reserve space in memory for
up to 40 characters in
string variable p$ and up
to 4 characters in string
variable a$ .)

c:s=0
(Set an initial value of 0 for
numeric variable s .)

r:THIS SECTION DIMENSIONS AND
: COMPUTES STRINGS TO BE USED
: IN THE MAXWELL ANIMATION.
(Comments to the author.)

d:m1$(15);m2$(15);m3$(15);m4$(15)
(Reserve space in memory for
up to 15 characters each in
string variables m1$ ,
m2$ , m3$ , and m4$ .)

d:f1$(11);f2$(11);f3$(11)
(Reserve space in memory for
up to 11 characters each in
string variables f1$ ,
f2$ , and f3$ .)

c:m1$=" /DAG/EBH/FC "
c:m2$=" /DAG/EBR/FC "
c:m3$=" /DAU/EB /FC "
c:m4$=" /ZAG/MBH/FC "
(Store these strings in the
string variables that were
just Dimensioned.)

c:f1$="# /# !%/ $ "
c:f2$="# !%/# / $ "
c:f3$="!%/# /# / $ "

r:DISPLAY BEGINNING SCORE INFO
(Comment to author.)
```

u:score	(Branch to the subroutine called score , returning here when the next e: is executed.)
*question1 t:Who was the only President to : serve two nonconsecutive terms? a:\$p\$	(Label for first question.) (Display this text.)  (Accept student's response, storing it in string variable p\$ .)
ms:CLEVELAND	(Did student type this text correctly or with only one spelling error?)
u:response	(Branch to subroutine called response , returning here when next e: instruction is executed.)
t:Grover Cleveland served as : President from 1885-1889 and : from 1893-1897. u:score	(Display this text.)  (Branch to subroutine called score , returning here when next e: is executed.)
*question2 t:Which President was never elected : to either the Presidency or the : Vice-Presidency? a:\$p\$	(Label for second question.) (Display this text.)  (Accept student's response, storing it in string variable p\$ .)
m:FORD u:response	(Did student type this text?) (Branch to subroutine called response, returning here when next e: instruction is executed.)
t:Gerald Ford became Vice-President : and then President when Spiro : Agnew and then Richard Nixon : resigned. u:score	(Display this text.)  (Branch to subroutine called score , returning here when next e: is executed.)
. . .	(Further questions follow the same format.)
r:PLACE NEW QUESTIONS JUST ABOVE : THIS REMARK. REMEMBER TO CHANGE : HIGHEST-QUESTION NUMBER BELOW IN : THE SCORE SUBROUTINE.	(Comments to author.)
*score	(Label for subroutine score .)

ts:v0,39,18,23;b6;es;s1;t1	(Set viewport to occupy last six lines; set background color to blue; erase the new viewport in blue; display future characters in single-size normal thickness text.)
cy:s=s+1	(If last match was successful, add 1 to current value of s .)
t:Correct: #s      Incorrect: #(q-s)	(Display text, substituting current values of s and q-s for #s and #(q-s) and following spaces.)
t:Press RETURN for the next question.	
t:Type STOP to end the lesson.	
a:\$a\$	(Accept student's response, storing it in string variable a\$ .)
e(q=20!a\$="STOP"):end	(If this is the last question (here, 20 questions), or if student typed STOP , then terminate subroutine and branch to label end .)
ts:v;b19;es;s2;t2	(Set viewport to full screen; set background to violet/orange and erase viewport; display future characters in double-size, boldface.)
c:q=q+1	(Add 1 to the value of q .)
t:Question ##q :	(Display text, substituting the value of q for #q and the following space.)
e:	(Terminate subroutine and branch back to instruction after one that called it.)
*response	(Label for subroutine called response .)
ts:es;i	(Erase screen in current background color; switch background and foreground colors for future text.)
tn:Sorry, that's wrong.	(If last Match unsuccessful, display this text.)
tsn:n	(If last Match unsuccessful, reset normal use of the foreground and background colors.)
en:	(If last Match unsuccessful, terminate subroutine and branch to the instruction after one that called it.)
ts:g6,2	(Move the cursor to screen position 6,2.)
t:That's right!	(Display this text.)

ts:n;s1;t1;g0,0;f3;b0

(Reset normal use of the background and foreground colors; use single-size normal thickness text; move cursor to screen position 0,0; set foreground white and background black.)

tx:maxwell

(Use MAXWELL character set for future text.)

r:MAXWELL RAISES FLAG AND SALUTES

(Comment to author.)

ts:am1\$;wrrr;af1\$;d60;w111

(Display this animation, in which Maxwell raises a flag and then salutes.)

ts:am3\$;d30

ts:am2\$;wrrr;af2\$;d30;w111

ts:am3\$;d30

ts:am2\$;wrrr;af3\$;d30;w111

ts:am4\$

ts:g0,5

(Move the cursor to screen position 0,5.)

r:PLAY "HAIL TO THE CHIEF"

(Comment to author.)

s:25,90;27,30;0,30;29,30;30,90;29,60;

(Play these notes, "Hail to the Chief.")

:27,30;25,30;0,30;27,30;25,30;0,30;

:22,30;20,90;18,90

ts:s2;t2;b19

(Display future characters double-size boldface; set background color to violet/orange.)

tx:

(Use standard character set for future text.)

e:

(Terminate subroutine and branch to the instruction after one that called it.)

\*end

(Label for subroutine end.)

r:COMPUTE PERCENTAGE CORRECT

(Comment to author.)

c:x=fix(100\*s/q)

(Multiply the current value of s by 100, divide by the current value of q, drop off any decimal places and store result in x.)

ts:v0,39,20,23;es

(Set viewport to last four screen lines; erase it in current background color.)

t:You answered #x % of the questions  
: correctly.

(Display text, substituting value of x for #x and the following space.)

t(x>=90):An excellent score!

(If x is at least 90, display this text.)

t(x<90&x>=70):A good score.

(If x is less than 90, but at least 70, display text.)

t(x<=60):I think you could use some  
: review.

(If x is 60 or less, display this text.)

w:5

(Wait for 5 seconds.)

e:

(Terminate subroutine and  
branch back to instruction  
after one that called it.)

The preceding example is a complete lesson that gives a quiz on U.S. Presidents. It uses most of the TS: commands, adding type variety and color ("red, white, and blue," of course) to an otherwise simple question and answer format. Maxwell even raises the flag and salutes after a right answer, making use of the MAXWELL character set on the LESSON diskette that comes with the SuperPILOT system. Only two questions are included here, but additional questions could be added easily, using only seven instructions each.

## **TX: eXecute Character Set File**

---

tx:[file name of a character set stored on lesson diskette]

The eXecute character set instruction replaces the usual set of text character display images by a set of images stored under the specified file name on the lesson diskette. Following this instruction, any text character typed by the student or displayed by the program causes one of the images from the new character set, rather than the usual image, to appear on the screen. This instruction lets your lessons display alphabets other than the standard alphabet. To return to the standard character set, use a TX: instruction with no object.

Unlike the usual T: instruction, the TX: instruction's object field does not contain text to be displayed on the student's screen. Instead, the file name of the stored character set is the object. For example, to begin using the stored character set whose file name is GREEK , your lesson would use this instruction:

tx:greek

(Replace the old set of character images with the character set whose file name is GREEK , stored on the lesson diskette, and use the new images for future character displays.)

The character set file is created and named while you are in the Character Set Editor. If you are using the normal, one-disk-drive Lesson Mode system, all character set files used by TX: instructions must be on the same diskette as the lesson that includes the TX: instructions. For using systems with additional disk drives, see the appendix Using More Disk Drives.

If no character set file with the specified name is found on the lesson diskette, no message is given, even if you are running a lesson in Author mode, and the Error Flag is not raised. The current character set is not changed in such a case, and execution of the lesson continues with the next instruction. This lets you write and test lessons that include TX: instructions, before you develop the stored character sets named in the TX: instructions. Later when you provide the correctly named

character set files, your lesson will automatically incorporate these character sets.

When you finish using the new character set, and want to return to the standard character set, use this instruction:

```
tx:                (Resume using the standard character set
                   for future character displays.)
```

Three other situations automatically return your lessons to using the standard character set:

1. Every time Apple SuperPILOT is started or restarted, the standard character set is called.
2. If a special character set is in use when an LX: instruction is executed, the standard character set is recalled. The un-modified L: instruction does NOT change the character set being used.
3. When a lesson is running in Author Mode and a special character set is being used, the standard character set is recalled in order to display any error message. The lesson automatically returns to the special character set following the error message.

Suppose the GREEK character set is designed so that a drawing of a Greek "alpha" corresponds to the usual drawing for "A". After your lesson executes the instruction tx:greek , an alpha will appear on the screen every time the student presses the A key and every time the T: or G:T instructions specify an A as the character to display.

The images stored in the character set file may be any drawings at all; they are not necessarily restricted to drawings of alphabetic characters. For instance, you might design several images to look like a person in various stages of walking. A sequence of g:t or ts:a instructions, using the characters associated with your images, could then create the appearance of a person walking across the screen. See the discussion of Text Animation in the Type Specify section earlier in this chapter for details on how to create animated effects.

The new character set changes only the drawings associated with screen displays of characters, and only when the student runs the lesson. Internally, the Apple II continues to use the standard characters, just as the keys themselves do not change (unless you put labels on them.) Thus, when you are typing instructions in the Lesson Text Editor you will always use the standard (ASCII) letters, even if you know that they will later appear in some other form when they are displayed on the student's screen.

You may specify as many different character sets as you wish in your lesson, and SuperPILOT will use them when a TX: instruction tells it to. However, only two character sets can reside in the Apple II's memory at one time. The standard character set on the Author diskette is one of these, and the first time a TX: instruction calls a different character set, it is retrieved from the diskette and stored in the

other memory space. If these are the only two character sets you use in your lesson, SuperPILOT will not have to return to the diskette to get them when you change from one to the other.

However, if you use more than one character set in addition to the standard one, there will be a delay each time one of the "extra" character sets is retrieved from the diskette and "swapped" for the other one. You will need to keep this in mind, especially when you are developing animations or other effects that must be executed promptly. Try to arrange your lesson so that the student has something to do (like read some text on the screen) while SuperPILOT is busy loading the new character set.

If you have an Apple Pascal system at your disposal, you may change the standard character set on your Author diskette to any other character set you wish. In the Pascal Filer, select the T(ransfer option, place the Author diskette in one drive and the diskette containing the desired character set in the other drive, then enter this command:

```
LESSON:NEWSET.FONT,AUTHOR:SYSTEM.CHARSET
```

where NEWSET is the name of the desired character set and LESSON is the name of the diskette containing NEWSET . You will then be prompted as follows:

```
Remove old AUTHOR:SYSTEM.CHARSET ?
```

Answer Y and the transfer process will begin. When the transfer is completed, NEWSET will be the standard character set on your Author diskette. Thereafter, any lesson diskette initialized by the utility program on the Author diskette will also use NEWSET as the default character set.

If you want to change back to the Author diskette's original character set, you can do so. This character set is on the Lesson diskette that came with your SuperPILOT system, under the name ASCII. Type this:

```
LESSON:ASCII.FONT,AUTHOR:SYSTEM.CHARSET
```

to return to the original default character set.

The same process can be used to change the default character set on a lesson diskette without having to make any changes to the Author diskette. Simply substitute the appropriate volume names in the above example.

See the portion of the Apple II SuperPILOT Editors Manual dealing with the Character Set Editor for further details on creating and using stored character sets.

## Example

pr:u	(Convert all response letters to uppercase.)
tx:jibberish	(Replace old set of character images with character set file named JIBBERISH , and use the new images for future character displays.)
*start	(Label for this section.)
t: Skveid endbr8foø dks ksie : *kej8jism vhhfke 2hdn3 j sjak?	(Display JIBBERISH image corresponding to each of these characters.)
a:	(Accept student's response, displaying the JIBBERISH image corresponding to each key the student presses.)
m:KRIEDF!BUROPDS!ERSD!T*3BF	(Did the actual keys pressed by the student include any of these four sequences?)
ty:Qwebghe!	(If Yes, display JIBBERISH images corresponding to these characters.)
jy:next	(If Yes, Jump to label next .)
t:Gewnt. Towo andkei.	(No, so display JIBBERISH images corresponding to these characters.)
j:start	(Jump back to label start .)
*next	(Label for this section.)
tx:	(Resume using the standard character set for future character displays.)
t:Now we will try the same thing : again, this time in English.	(Display the standard images corresponding to these characters.)
...	...
...	...

The file named JIBBERISH was created earlier, in the Character Set Editor, and stored on the lesson diskette. Each standard symbol typed in the Lesson Text Editor has a corresponding drawing or character image in the JIBBERISH file. After the instruction tx:jibberish , any character displayed by a T: instruction or typed by the student will appear on the screen, not in its usual form, but as the JIBBERISH image which corresponds to that character.

To write the lesson, the author must have a table telling which standard character corresponds to each JIBBERISH image. To Type or Match a given series of JIBBERISH images, the author then uses the corresponding series of standard characters in a T: or M: instruction, knowing that the appropriate transliteration will take place when the lesson is running.



## S: Sound

S:pitch,duration[;pitch,duration;pitch,duration...;pitch,duration]

The Sound instruction plays a musical note or notes over the Apple II's speaker. You can use this sound to get the student's attention, to signal a wrong answer (or a right one), or to dramatize a visual effect. You can even use the Sound instruction to play simple tunes. For longer, more complicated musical pieces or sound effects, see the SX: instruction.

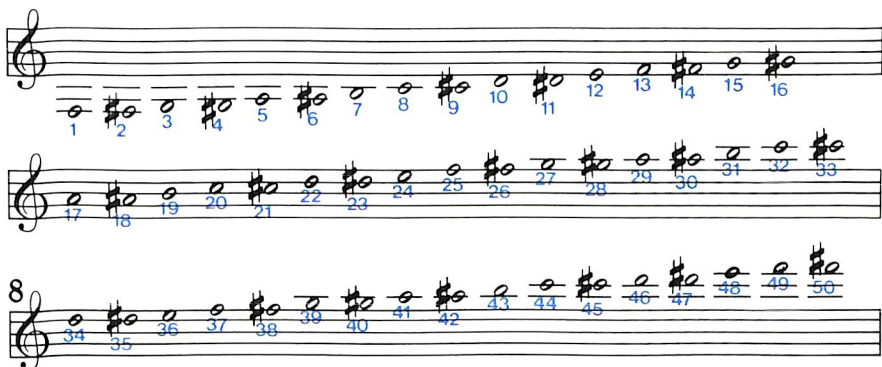
To play a note, the Sound instruction's object field must specify how high or low a note to play (that is, its pitch), and then how long to play that note (its duration). For example, to play a middle C that lasts about half a second, you could use this instruction:

s:8,100

(Play a note of pitch 8 and duration 100 over the Apple's built-in speaker.)

The note's pitch should be an integer from 0 through 50. Pitch 0 makes no sound at all; it is used to specify a rest or silence. Pitches 1 through 50 yield a rising tempered (approximately) chromatic scale, in which each pitch number gives a note that is one half-step higher than the previous pitch number. A note of pitch 8 is approximately middle C. There are 12 half-steps in an octave; so pitch 20 is a C an octave above middle C, pitch 32 is a still higher C, and pitch 44 is the highest C you can play.

To translate a printed musical sequence into a Sound instruction, you can refer to the following chart, using the number below each note for the pitch value:



[ DIAGRAM OF CHROMATIC SCALE, WITH 50 NUMBERED PITCHES ]

For musical pieces of any length, however, you will find it more convenient to use the Sound Effects Editor to write your program and then call it with a `SX:` instruction. See the next section of this chapter and also the Sound Effects Editor chapter of the Apple II SuperPILOT Editors Manual for instructions.

A note's duration must be an integer from  $\emptyset$  (the shortest note) through 255 (the longest). A duration of  $\emptyset$  specifies a note that is so short it sounds almost like a "click," especially with lower pitch numbers. A note of duration  $2\emptyset\emptyset$  lasts about a second.

Actually, any integer from  $\emptyset$  through 32767 may be used for pitch and duration. Pitch numbers higher than 5 $\emptyset$  are converted to pitch 5 $\emptyset$ ; duration numbers higher than 255 are changed to duration 255. The name of a simple numeric variable may appear in place of any pitch or duration number. The number stored in this variable need not be an integer, but it must be within the range limits imposed on pitch and duration. A non-integer variable value is not rounded, it is truncated: any portion of the number to the right of the decimal point is simply discarded.

Several notes may be specified in a single `S:` instruction by listing their pitch,duration pairs in sequence, separated by semicolons. For example, this instruction plays a C scale:

```
s:8,1 $\emptyset\emptyset$ ; 1 $\emptyset$ ,1 $\emptyset\emptyset$ ; 12,1 $\emptyset\emptyset$ ; 13,1 $\emptyset\emptyset$ ;           (Play eight notes over  
:15,1 $\emptyset\emptyset$ ; 17,1 $\emptyset\emptyset$ ; 19,1 $\emptyset\emptyset$ ; 2 $\emptyset$ ,1 $\emptyset\emptyset$            the Apple II's speaker.)
```

The spaces are not necessary; they just make the instruction easier to read.

When you are first developing a sound effect, it is usually better to use several `S:` instructions with just one or a few object notes each, rather than one `S:` instruction with a long list of notes. Changing individual notes in the sequence is much easier when these notes are not buried in a long list of notes.

Later, when you are satisfied with your tune, you can always combine the separate instructions into one instruction with a long note-list. Combining separate `S:` instructions into one may make the program a little more compact and easier to read in the Lesson Text Editor, but it has little effect on the sounds produced. At most, you will notice a very slight decrease in the "space" between successive notes after combining them into one instruction.

If you do type a long Sound instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation colon) on the next line. You may also break the instruction anywhere you wish, by pressing the RETURN key. If you then wish to continue the Sound instruction, you must begin the new line with a continuation colon or a new `S:` instruction.

Even though the Editor may break your instruction in the middle of a note specification when it reaches the thirty-ninth character, the

instruction will be executed correctly. When the instruction is executed, any breaks and continuation colons are ignored, whether they were introduced by you or by the Editor. The object text is treated as if it were one long unbroken string. This means that if you break a Sound instruction, and then colon-continue the list of notes on the next line, you must not leave out any separating semicolons or introduce any illegal spaces in the middle of a number.

Regardless of the number of lines a single S: instruction occupies, only the first 250 characters of the instruction are actually used at execution time. The continuation colons are not counted in this total. If the instruction is longer than 250 characters, any characters after the 250th character are simply ignored.

## Modifier

The Sound instruction can be used with the eXecute modifier to play musical compositions and sound effects created in the Sound Effects Editor. For example, your lesson could play the sound effect stored in the diskette file named DRUMROLL by using this instruction:

```
sx:drumroll
```

(Find the sound effect whose file name is DRUMROLL, stored on the lessons diskette, and play it over the Apple II's built-in speaker.)

For more information, see the next section in this chapter, on the SX: instruction.

## Example

```
pr:u
```

(Convert all response letters to capital letters.)

```
c:n=1
```

(Set value of variable n to 1.)

```
t:Wait until you hear a "beep";
```

(Display these lines of text.)

```
: then quickly type the names of
```

```
: five famous women.
```

```
*names
```

(Label for this section.)

```
w:2
```

(Wait two seconds or until any key is pressed.)

```
s:30,10
```

(Play a short "beep" on the Apple's speaker.)

```
a:
```

(Accept student's response.)

```
s:20,20
```

(Play a lower, longer "beep.")

```
c:t=tim(0)
```

(Store the response time in simple numeric variable t.)

```
t:It took you #t seconds
```

(Display these lines of text, substituting the response time t for #t.)

```
: to name them.
```

```
j(n=2):next
```

(If n=2, Jump to the label next.)

t:Do you think you could name	(N=1 , so display this text.)
: five famous men in less time?	
t:Give it a try, when	(Display this text.)
: you hear the "beep."	
c:n=2	(Set value of variable n to 2.)
j:names	(Jump back to label names .)
*next	(Label for this section.)
t:Was it easier to think of	(Display these lines
: famous women or famous men?	of text.)
a:	(Accept student's response.)
t:I see. Think about the reasons	(Display this text.)
: for this, while you listen to	
: the following tune.	
s:Ø,255	(Short silence.)
s:8,5Ø; 11,5Ø; 8,1Ø5	(Plays tune of "Five foot
s:8,5Ø; 12,5Ø; 8,1Ø5	two, eyes of blue..." .)
s:8,5Ø; 13,5Ø; 8,5Ø; 13,5Ø	
s:8,25; 13,75; 8,47; Ø,3	
s:13,5Ø; 16,5Ø; 13,5Ø; 16,25; 13,75	
s:11,1Ø5; 13,11Ø; 16,22Ø	
t:Do you think this describes	(Display this text.)
: a famous woman?	
...	...
...	...

## **SX: eXecute Sound File**

**sx:**file name of a sound effect stored on the lesson diskette

The eXecute Sound instruction reads a sound effect file stored on the lesson diskette and plays the stored sound effect over the Apple II's built-in speaker. Unlike the S: instruction, no pitch or duration specifications for notes appear in the object field of the SX: instruction. Instead, the file name of the stored sound effect is the object. For example, to use the stored sound effect whose file name is DRUMROLL , your lesson uses this instruction:

<b>sx:drumroll</b>	(Find the sound effect whose file name is DRUMROLL , stored on the lesson diskette, and play it over the Apple II's built-in speaker.)
--------------------	--

The sound effect file is created and named while you are in the Sound Effects Editor. These stored sound effects may contain more complex tunes, effects that are not tunes, and effects that you may want to use in more than one lesson. If you are using a normal, one-disk-drive Lesson Mode system, all sound effect files used by SX: instructions must be on the same diskette as the lesson that includes the SX: instructions. For using a system with additional drives, see the appendix Using More Disk Drives.

If no sound effect file with the specified file name is found on the lesson diskette, no message is given, even if you are running a lesson in Author mode, and the Error Flag is not set. Lesson execution simply goes on to the next instruction. This lets you write and test lessons that include SX: instructions, before you develop the stored sounds named in the SX: instructions. Later, when you provide the correctly named sound files, your lesson will automatically incorporate those sounds.

See the portion of the Apple II SuperPILOT Editors Manual dealing with the Sound Effects Editor for further details on creating and using stored sound effects.

## Example

sx:loony	(Find the sound effect whose file name is LOONY, stored on the lesson diskette, and play it over the Apple II's built-in speaker.)
sx:tunes	(Play the sound effect whose file name is TUNES.)
d:a\$(4)	(Reserve space for storing up to four characters in string variable a\$.)
c:a\$="sx:"	(Store the string sx: in string variable a\$.)
d:s\$(15);t\$(10)	(Reserve space for storing up to fifteen characters in string variable s\$ and up to ten characters in string variable t\$.)
pr:su	(Remove all spaces from typed responses, and convert all response letters to capitals.)
*grammy	(Label for tunes section.)
t:Here are the tunes now stored : in this "library":	(Display this text.)
t: What a Fool Believes	(Display these lines of text.)
t: Isn't She Lovely?	
t: Roll Over Beethoven	
t: Chopin's Prelude 12	
t: L'homme Armee by Josquin	
t:	(Display this blank line.)
t:Type the name of the piece : you want to hear:	(Display this text.)
a:\$t\$	(Accept student's response, remove all spaces; store first ten remaining characters in string variable t\$.)

<pre>c:s\$=a\$!!t\$ xi:s\$ t:Would you like to hear another? a: m:YES!OK!BET!SURE!RIGHT!FINE jy:grammy *next</pre>	<pre>(Join the name of the tune stored in t\$ to the sx: instruction name stored in a\$ . Store resulting string, such as sx:ISN'TSHELOVELY? , in string variable s\$ .) (Execute the string stored in string variable s\$ as an Apple SuperPILOT command. This plays the selected tune if it is available.) (Display this text.) (Accept student's response.) (Did student type one of these words?) (If Yes, Jump to grammy .) (Label for next section.)</pre>
--	--

This lesson portion plays the selected tunes, if they are available and if you gave them ten-character names (you can limit the names to any shorter length, if you want, by Dimensioning t\$ smaller).

Note: In fact the Sound Effects Editor accepts file names of ten characters or less. So, although it is possible to offer the student the full name of a musical composition, such as "L'homme Armee by Josquin," on the display screen, in the Sound Effects Editor only the first ten letters of the name of this piece are accepted.

## AP: Accept Point

---

ap:

The Accept Point instruction waits for the student to indicate a point on the screen, using the game controls or any input pointing device that the system has been modified to accept. It then stores the indicated screen location for use by later instructions. If any variable names appear in the object field of this instruction, they are ignored.

When the AP: instruction is executed, a pair of intersecting "cross-hairs" appears on the screen. (However, you may modify SuperPILOT so that the cross-hairs do not appear; see the section Using Other Pointing Devices at the end of this chapter.) By using the game controls, the student can move the intersection of the cross-hairs to any point on the screen. Game control 0 sets the horizontal position, and game control 1 sets the vertical position. When the cross-hairs are centered on the desired location, the student presses the button on either game control (or presses any key except SHIFT, CTRL, REPT, or RESET) to save the graphics coordinates of the indicated screen position. The horizontal coordinate (a number from 0 through 559) is stored in system variable %x , the vertical coordinate (a number from

Ø through 511) is stored in system variable %y , and the graphics cursor is moved to the Accepted location.

These system variables can be used in other instructions to let the student point to various words or graphic items on the screen or to let students do their own graphics. For example, you could Accept a Point and then Draw a line from that point to the previous point, over and over, using instructions such as these:

g:esØ;c9	(Clear the screen to Black1; set drawing color to Single-dot On.)
c:hl=Ø;vl=Ø	(Set the initial values for simple numeric variables hl and vl to Ø.)
ap:	(Display cross-hairs; Accept the coordinates of the cross-hairs location when student presses game-control button, or any key.)
c:h=%x;v=%y	(Store x-coordinate %x in simple numeric variable h and y-coordinate %y in simple numeric variable v .)
g:dhl,vl	(Draw a line from the last ap: location to the point x=hl, y=vl .)
c:hl=h;vl=v	(Assign new screen positions in h and v to variables hl and vl .)
j:@a	(Jump back to last Accept; in this case, the ap: .)

System variables are discussed in the Computation Instructions chapter.

## Response Timing

If the last Accept instruction was an AP: instruction, the TIM(Ø) function returns the amount of time the AP: instruction waited for the student's response to Accept the coordinates of the cross-hairs' location. For example, you can display the student's response time with these instructions:

ap:	(Accept cross-hairs location when student presses a game-control button, or any key.)
c:t2=tim(Ø)	(Store the student's response time in simple numeric variable t2 .)
t:You took #t2 seconds : to choose your point.	(Display this text, substituting the response time t2 for the variable #t2 and one space.)

The TIM function is described in the Advanced Programming chapter.

Normally, the Accept Point instruction waits indefinitely for the student to adjust the cross-hairs. Program execution continues after the student presses either game-control button or a key to Accept the coordinates of the indicated screen location. However, if the Tn response-timing option is set by the last PR: instruction, the AP: instruction waits for a maximum of n seconds (where n is an integer from 0 through 32767). If the student does not press a game-control button after n seconds, program execution continues just as if the button had been pressed. In this case, the coordinates Accepted indicate where the cross-hairs are placed when the time ran out, and TIM(0) returns a value of 0, not n.

For example, to set a maximum response time of ten seconds (at the same time terminating any other PROblem control options), your lesson can use this instruction:

```
pr:tl0
```

(Set maximum response time for future Accept and Accept Point instructions to ten seconds.)

See the discussion of the PROblem instruction in the chapter Text and Response Instructions for details about setting the response-timing control option.

## Programming Notes

1. If you want to find out which key the student pressed in response to an AP: instruction, you can use the KEY(0) function. Just include the following instruction (or a similar one) immediately after the ap: instruction in your lesson:

```
c:n=key(0)
```

(Store the ASCII number of the last key that was pressed in simple numeric variable n.)

The 0 in key(0) is only a "place-holder" and is not used by SuperPILOT in any way.

2. An AP: instruction cannot be used as an entry point into Immediate Mode, as an A: instruction can.



## Example

```
r:*****
: ** "CATCH THE DOT" PROGRAM **
: **      FINAL VERSION      **
: **                               **
: **  USES THE GAME PADDLES  **
: *****

c:r=5
pr:utr

g:v0,39,0,1;es0;c8

*start
c:h2=rnd(560)

c:v2=rnd(465)

g:ph2,v2

t:You have five seconds to catch
: the dot.
ap:

c:t=tim(0)

s(t=0):1,255

tc:Sorry, time's up.

jc:replay

t(abs(h2-%x)>2!abs(v2-%y)>3):Missed!

jc:replay

s:30,7;35,7;39,7;42,25;0,7;39,7;42,150
```

(Remarks to author.)

(Store number 5 in simple numeric variable r .)

(Convert response letters to capitals; set maximum response time to r , in this case five seconds.)

(Allow text only in top two lines; Erase screen to black; set plotting color to Reverse.)

(Label for this section.)

(Store a randomly-selected horizontal coordinate in numeric variable h2 .)

(Store a randomly-selected vertical coordinate in numeric variable v2 .)

(Plot a dot at screen point x=h2,y=v2 .)

(Display this text.)

(Accept the coordinates of the screen point under the cross-hairs.)

(Store the response time in numeric variable t .)

(If time had expired, play this low note.)

(If same condition is true, display this text.)

(If same condition is true, Jump to label replay .)

(If the Accepted point misses the Plotted point by more than one screen-dot, display this text.)

(If same condition is true, Jump to label replay .)

(The Accepted coordinates were both within 2 of the Plotted coordinates, so play tune.)

t:Good! You got it in #t seconds.	(Display text, substituting response time t for #t and one space .)
*replay	(Label for this section.)
th:Want to try again?	(Display this text, leaving cursor on the same line.)
a:	(Accept student's response.)
m:%NO!%N%!NEVER!FORBID!KIDDING	(Did student type any of these negative answers?)
ey:	(If Match is successful, end this lesson.)
g:ph2,v2	(Erase dot by plotting point at x=h2,y=v2 in color Reverse for the second time.)
j:start	(Jump back to start .)

This example is a game of "Catch the Dot." It shows a white dot on the screen at a random location, then displays the Accept Point cross-hairs. The student has a maximum of five seconds to center the cross-hairs on the dot, using the game controls. The game continues when the student presses either game-control button, or when five seconds is up, whichever comes first.

The coordinates of the Accepted point are then compared with the coordinates of the dot. The student has succeeded in catching the dot if the Accepted x-coordinate is within plus or minus 2 of the dot's x-coordinate, and the Accepted y-coordinate is within plus or minus 3 of the dot's y-coordinate. Since a screen dot occupies an area two x-coordinate values wide by two-and-two-thirds y-coordinate values high, this test lets the student catch the dot if the Accepted point is within one screen dot of the "target" dot, in any direction.

Note: Plotting in the color Reverse first reverses a black background dot to white. Plotting the same point again reverses the white dot back to black, effectively erasing the dot.

The randomly-selected y-coordinate for the "target" dot has been limited to the range from 0 through 464. This ensures that the dot will not be placed in the text viewport. The viewport allows two lines of text at the top of the screen. A text character is seven screen-dots wide by eight screen-dots high. Each screen-dot occupies an area that is two x-coordinate values wide by two-and-two-thirds y-coordinate values high. Therefore, the two rows of viewport text occupy an area 16 screen-dots high (2\*8), or about 43 y-coordinate values high (16\*2.67). We subtract 43 from 511, the screen's top y-coordinate, to calculate the highest allowable y-coordinate of 468, and then subtract another four from that to make sure the "target" dot never actually touches the bottom of the text.



## Using Other Pointing Devices

Unless you have modified your Author diskette, the AP: instruction will Accept input only from the game controls and the keyboard, and will always display the cross-hairs. However, you may wish to install other pointing devices to your system for the purpose of student input, such as a touch screen or a graphics tablet. With a modification to the Author diskette, SuperPILOT can be customized so the cross-hairs do not appear, or input from other devices is accepted from the AP: instruction. The details for making the necessary conversion are explained in a document available through your Apple dealer, SuperPILOT Technical Support Package.

Manufacturers of pointing devices, or others interested in developing the software to allow a pointing device to be used with SuperPILOT, can use the SuperPILOT Technical Support Package to obtain information on modifying the system to interface with videotapes, videodisks, and similar devices. See the next section on the V: instruction.

## V: Audio/Visual Device Control

---

v:command[;...]

The V: instruction can be placed in your SuperPILOT lessons to control almost any peripheral device, like videotapes and videodisks. For example, if you were to write a geography lesson, you might allow the student to select a certain country from a menu, and then have a videotape of that country's landscape begin. The commands in the object text of the V: instruction direct the videotape machine to begin at a certain location or to perform other functions. Because the V: instruction can be customized to a wide variety of machines, its use in your programs can greatly expand the educational power of your SuperPILOT system.

In order to accommodate the great diversity of A/V equipment that can be used with SuperPILOT, the form of the V: instruction has been kept flexible. The actual commands that you include in the object text are defined by you, depending on the kind of equipment you wish to use. Two system variables are available for your use: %v and %w. They differ only in the numeric ranges they allow. The %v system variable allows signed numeric values in the range -32767 to 32767; the %w system variable allows unsigned numeric values in the range 0 to 65535.

In order to use the V: instruction with specialized A/V equipment, you need to modify your SuperPILOT system. If the necessary interface for the equipment you want to use has already been developed elsewhere, your Apple dealer can assist you in obtaining it. If this is not available, or if you have Apple Pascal programming experience and prefer to make the modifications yourself, your dealer can help you obtain the SuperPILOT Technical Support Package. This package includes the necessary software and documentation to convert SuperPILOT to accept the V: instruction, and it also contains instructions for customizing the AP:

instruction (see the discussion earlier in this chapter) to utilize touch screens, graphics tablets, or other input pointing devices.

The V: instruction always accesses the peripheral device connected to the Apple II through slot #2. Therefore, without making any modifications to your SuperPILOT system you can use the V: instruction to send literal text to a device connected through slot #2. A printer in that slot, for example, would print the contents of the V: instruction's object text.

## Chapter 7

# File Handling Instructions

- 168 K: Keeping Student Records
- 170     Modifier
- 170     Creating a Recordkeeping File
- 171     Creating a Separate Record Diskette
- 172     Example
  
- 174 General File Information: FOX: , FIX: , FO: , FI:
- 174     Files
- 175     Records
- 175     Opening a File
- 176     Closing a File
  
- 176 FOX: Create and Open New File
- 177     The Largest-Record Number
- 177         Deleting a Data File
- 178     The New File
- 180     Example
  
- 181 FIX: Open Existing File
- 182     The Largest-Record Number
- 182     Example
  
- 184 FO: Output to Open File
- 185     The Record Number
- 185     Example
  
- 188 FI: Input from Open File
- 189     The Record Number
- 189     The String Variable
- 190     Example

# K: Keeping Student Records

---

k:any text or spaces or variable names

The Keep instruction is a new feature of SuperPILOT; it has not been available in earlier versions of the PILOT language. It gives you the capability to keep records on the performance of students using your lessons. When you have the recordkeeping option in use, Keep instructions placed at strategic places in a lesson will send data back to a special file. You can later access this file by using SuperPILOT LOG, a statistical package available from your Apple dealer, so that you can see how each student progresses, how students compare with each other, what parts of your lesson may be ineffective, and other useful information.

You will probably find that the Keep instruction is the only one you need for most of your file-handling needs. The FOX: , FIX: , FO: , and FI: instructions, described later in this chapter, have been retained by SuperPILOT primarily so that Apple PILOT lessons written with these instructions will continue to run under SuperPILOT. The Keep instruction for the most part makes these four instructions unnecessary for new lesson development, except in cases where you need to retrieve data from a file for use in a lesson (use the FI: instruction for this purpose).

In a lesson program, the Keep instruction is very similar to the Type instruction: you can add the same modifiers, conditioners and expressions between the K and the colon that are allowed with the Type instruction, and the object field may contain the same combination of text and variables that a text instruction does. The difference is that the object field of the Keep instruction is not displayed on the student's screen, but is stored instead in the special recordkeeping file you have set up, called system.log .

For example, you might include the following lines at the beginning of a lesson:

pr:u	(Convert all response letters to capital letters.)
d:n\$(30)	(Reserve space in memory to store up to 30 characters in string variable n\$ .)
*studentinfo	(Label for this section.)
t:What is your name?	(Display this text.)
a:\$n\$	(Accept student response and store it in string variable n\$ .)

k:Name: \$n\$

(Store this text in the special recordkeeping file system. log , substituting the contents of string variable n\$ for \$n\$ and the one following space.)

The system.log file automatically keeps track of the name of the lesson being run, any Links that are executed to other lessons, and the normal End of the lesson. So with only the addition of the lines above, you can keep track of which students have completed a lesson. With the addition of some additional Keep instructions, as below, you can gather further information on the student's progress and the operation of the lesson:

c:p=1;s=0

(Create two simple numeric variables, p and s to hold the number of the problem set and the student's score, respectively.)

...

(Early part of an arithmetic lesson.)

...

t:Your score for the first ten  
: problems is #s .

(Display this text, replacing #s and the one following space with the current value of s , computed earlier in the lesson.)

u:record

(Branch to the subroutine named record , returning here after encountering the next e: .)

...

(Continue the lesson.)

...

\*record

(Label for the subroutine to transfer information to the recordkeeping file.)

k:Score for problem set #p : #s

(Store this text in the special recordkeeping file, system.log , substituting the current values of p and s for #p and #s and the one space following each.)

e:

(End this subroutine and branch back to the instruction after the last u: instruction.)

Note: Recordkeeping is not supported by SuperPILOT in Author Mode. This means you can test a lesson containing Keep instructions without adding to a system.log file on the diskette (or causing an error message if no system.log file exists). If you want to test the recordkeeping function, you can do so by booting the Lesson diskette in Lesson Mode and running the lesson as a student would.

Because recordkeeping is not supported in Author Mode, you can complete and test your lesson before creating the system.log file. When it is



created after you finish editing your lesson and other files, it is stored as the last file on the diskette. This placement is best because system.log can then expand to occupy additional blocks on the diskette if necessary. If one or more files are stored on the diskette after system.log, its maximum length will be "frozen" at the number of blocks it currently has (four, when first created).

If the system.log file is full when the lesson diskette is first booted in Lesson Mode, a brief warning message appears on the screen, but lesson execution is halted and records are not kept. If the file is filled during the course of a lesson, however, no warning is given and future data sent to the file from the lesson is lost without raising the Error Flag. When it is possible that the records kept from a lesson will cause the system.log file to exceed its current size, be sure that it is the last file on the diskette, with enough free blocks after it to accommodate the student's records. (If you have a multiple-drive student system, you can use a special SYSLOG diskette, described later in this section, to avoid this problem.)

## Modifier

The Keep instruction may be modified by the letter S, for Save, which updates the system.log file immediately. Without the Save modifier, SuperPILOT keeps all the data accumulated from the lesson's K: instructions in the Apple II's memory buffer, but it is not written into the system.log file until it is necessary for SuperPILOT to do so (at the end of the lesson, when a Link is encountered, or when the Apple II's memory buffer is full).

When a KS: is executed, however, the object text of that instruction, as well as the object text of any Keep instruction executed since the last KS: instruction, is transferred to the system.log file. It is a good idea to use the Save modifier at strategic points in your lesson, so that the data you want to Keep is not lost if there is a power failure or if the student turns off the computer before the lesson is over. Updating the file may take a few seconds, however, so you may want to give the student something to do (like reading some text) while SuperPILOT is busy.

## Creating a Recordkeeping File

This section is limited to an explanation of the mechanics of creating a student record file and writing information into it. The information stored there may be read in the Editor in Author Mode, but you will need the optional Apple SuperPILOT Log package to perform the sorting and analysis that will make that information meaningful. Detailed instructions on how to make your program supply necessary relevant information is contained in the Apple SuperPILOT Log User's Guide.

An individual diskette can have one recordkeeping file. The name of this file is system.log.



To place a recordkeeping file on the diskette:

1. Boot the system in Author Mode.
2. Select "Lesson Text Editor"
3. Select "New"
4. When asked, "Create which lesson?", type `system.log` .
5. As long as the file does not already exist (in which case you are notified), the file is created and you are placed in the editor.
6. At this point you may enter some notes to yourself, beginning them with R: , as with any SuperPILOT remarks.
7. When done, exit the file by pressing Q for Quit, and then type Y and press RETURN when you see the prompt "Save this lesson as SYSTEM.LOG?"

As long as `system.log` resides on a diskette, data is put into it, as instructed by Keep instructions, every time any lesson on that diskette is run. If you do not want to keep records, you need to DELETE `system.log` , using the Lesson Text Editor. This will, of course, destroy any information already stored in it, so you may want to print the information before you delete it.

## Creating a Separate Record Diskette

If you have two or more disk drives on line in your student system, you may keep records on a diskette other than the one containing the lesson. Simply initialize a blank diskette and give it the volume name `SYSLOG` . (Details on how to initialize a diskette may be found in the Apple II SuperPILOT Editors Manual.) A `system.log` file is placed on the `SYSLOG` diskette automatically when it is initialized.

Once `SYSLOG` is set up, it automatically handles all of your programs' recordkeeping chores. You must be sure that it is in a disk drive when Lesson Mode is first started up, either by turning on the Apple II or by pressing RESET. SuperPILOT looks for the `SYSLOG` diskette only when a lesson diskette is booted; if `SYSLOG` is not found then, SuperPILOT is not able to access it unless the student reboots.

For testing students, it may be best to include the `system.log` file on the same diskette as the test lesson, rather than using a separate `SYSLOG` diskette for recordkeeping. Otherwise, clever students may be able to run test lessons without recordkeeping, simply by taking the `SYSLOG` diskette out of the drive when booting the system. There are a number of advantages to using a `SYSLOG` diskette, however, including increased recordkeeping space (an entire diskette) and centralized recordkeeping for lessons run from different diskettes by the same student.

## Example

<code>d:n\$(50)</code>	(Reserve space for string variable <code>n\$</code> to store up to fifty characters.)
<code>t:Please type your name, and : then press the RETURN key.</code>	(Display this text.)
<code>a:\$n\$</code>	(Accept response and store it in string variable <code>n\$</code> .)
<code>k:Name:     \$n\$</code>	(Store this text in the special recordkeeping file system. <code>log</code> , substituting the current contents of <code>n\$</code> for <code>\$n\$</code> and one space.)
<code>t:Now, \$n\$ , type your age.</code>	(Display text, substituting the name stored in string variable <code>n\$</code> for <code>\$n\$</code> and the first following space.)
<code>a:#a</code>	(Accept response; create simple numeric variable <code>a</code> and store the response's first number in it.)
<code>je:error</code>	(If the Error Flag is up, no number was typed, so Jump to section labelled <code>error</code> .)
<code>k:Age:     #a</code>	(Store this text in system. <code>log</code> , substituting the current value of <code>a</code> for <code>#a</code> and the following space.)
<code>c:r=3*a</code>	(Multiply age stored in <code>a</code> by 3, and store result in simple numeric variable <code>r</code> . This is the right answer.)
<code>*prob</code>	(Label for this section.)
<code>t:What is three times your age?</code>	(Display this text.)
<code>a:#b</code>	(Accept response; create simple numeric variable <code>b</code> if this is its first use; store the response number in <code>b</code> .)
<code>je:error</code>	(If the Error Flag is up, no number was typed, so Jump to section labelled <code>error</code> .)
<code>t1(b=r):Right the first try!</code>	(If answer count is 1 and if <code>b=r</code> , display this text.)
<code>tc(%a&gt;1):Yes, that's right.</code>	(If <code>b=r</code> ( <code>c</code> is last-expression conditioner) and answer count <code>%a</code> is greater than 1, display this text.)
<code>j(b=r):next</code>	(If <code>b=r</code> , Jump to section labelled <code>next</code> .)
<code>t(%a&lt;5):Nope, try again.</code>	(Wrong answer: if answer count is still less than 5, display this text.)

jc:prob	(If answer count %a is less than 5 ( c is last-expression conditioner), Jump back to label prob for another try.)
t:No, 3 times #a is #r .	(5th bad answer: display text, substituting age a for #a and first following space, and right answer r for #r and first following space.)
j:next	(Student has had 5 wrong answers so Jump ahead to next problem.)
*error	(Label for error section.)
t1:Please type a number.	(If answer count is 1, display this text.)
t2:I mean a number like 7 or 23.	(If answer count is 2, display this text.)
k2:\$n\$ failed to type a number : twice in a row.	(Store this text in system.log , substituting the student's name for \$n\$ .)
t(%a>2):You must type a number.	(If answer count %a is greater than 2, display this text.)
j:@a	(Jump back to last Accept instruction executed.)
*next	(Label for next section.)
ks:Wrong guesses: #(%a-1)	(Store this text in system.log , after subtracting 1 from the current answer count and substituting this value for #(%a-1) ; update the system.log file now.)

This example is identical to the one at the end of the section on the Accept instruction in the Response Instructions chapter, except that four Keep instructions have been added. These quick additions, however, make it possible for the author to monitor several aspects of program execution. The student's name and age are recorded for later evaluation, along with the number of wrong answers the student gave before solving the problem. The Keep instruction that appears in the error section will tell the author if the lesson instructions are unclear (or if maybe the student is "goofing off").

Note: The last instruction in this example calculates the number of wrong answers the student gave, but it is included at the beginning of the following section. This placement is necessary to avoid having to place duplicate Keep instructions at each point in the program where a branch to next might occur: regardless of how the student arrived at the next section, this one instruction will record the number of guesses. Another approach would be to create a subroutine to handle each of the items of information you want to store in system.log ,

then place U: instructions at appropriate locations in the program. Finally, note that this example does not store information in the file in a form that the optional Apple SuperPILOT Log can use. For information on how to write lessons that can make use of this package, see the Apple SuperPILOT Log User's Guide.

## General File Information:

### **FOX: , FIX: , FO: , FI:**

---

The information contained in this section will be useful to you if you plan to use the FOX: , FIX: , FO: , and FI: instructions in your lessons to handle file input and output. Usually, you will find that the Keep instruction handles your data file needs more simply and efficiently than these four instructions, which have been retained from Apple PILOT primarily for reasons of compatibility. Use these four instructions if you need to save program data in a file other than system.log or if you need to retrieve data from a file during the running of a program. Otherwise, you may find that Keep is the only file handling instruction you need.

### Files

Your lessons can easily store student responses, scores, or other information onto the lesson diskette, and then retrieve the same information later in the same lesson or in another lesson. Information is stored on the lesson diskette in a named "file," similar to the diskette files that store your lessons, graphics, character sets, and sound effects. It is a different type of file, however, known as a data file. You give each file a specific name, and refer to that file by its name from then on.

The files you name are accessed during the running of your lessons by four SuperPILOT instructions: FOX: , FIX: , FO: , and FI: . Each of these is discussed in its own section later in this chapter.

When you create a lesson, graphic, character set, or sound effect using one of the special Editors, that file's name is added to the list that you see each time you select the appropriate Editor. However, the names of files created by instructions within a lesson that is running never appear on any Editor list. You must remember what files your lessons have created, or refer to the instructions that create these files, in the text of the lesson itself.

Note: If you have access to an Apple Pascal system, you can use the Pascal File to list all the files on a SuperPILOT lesson diskette. Each SuperPILOT file appears in the Pascal File with a characteristic suffix added to the usual SuperPILOT file name. Lesson files end in .TEXT , graphics files end in .GRAF or .FOTO , character-set files end in .FONT , sound-effects files end in .SONG , and files created by instructions in a running lesson end in .DATA . Pascal programmers should note that the format for a data file is:

```
Datafile:file of string[255];
```



## Records

An individual item of information is saved in a file sub-unit called a record. Each record occupies half of a block on the lesson diskette and can hold one string only, up to 255 characters long. A record may hold a word, a sentence, the string representation of a number, a SuperPILOT instruction: anything that can be stored in a string variable. A number held in a numeric variable cannot be put in a file record directly. The number in the numeric variable must first be transferred to a string using the STR(x) function, after which that string can be saved in a file record.

Each record in a file has a record number. The file's first record is number 0, the second is record number 1, and so on. You can store any item into any of the file's records in any order, and some records may remain empty. However, you must remember the record number of each item your lesson saves. When you retrieve that information later, your instructions will have to ask for each item by its record number.

## Opening a File

Before your lesson can store any item in a record, or retrieve the contents of any record, you must first tell Apple SuperPILOT which file you are talking about. This process of announcing the file's name and preparing the file for storing or retrieving information is called opening the file.

SuperPILOT has two different ways to open standard data files. The FOX: instruction creates and opens a new file on the lesson diskette; the FIX: instruction reopens an existing file. Each time you open a file, your file-opening instruction specifies the maximum record number that your storage or retrieval instructions can use. You must remember this maximum number, and use it consistently every time you open the file.

Once you have opened a data file, all your subsequent storage ( FO: ) or retrieval ( FI: ) instructions are assumed to refer to records within that file. Only one data file can be open at any time.

The recordkeeping file system.log may be open simultaneously with a data file, but this practice should be avoided (unless you are using a separate SYSLOG diskette for recordkeeping). When system.log is the last file on the diskette, it can expand in size (that is, in the number of blocks it occupies) as necessary to accept new input from the lesson's Keep instructions. However, as soon as a data file is opened in a lesson, that file is moved to the end of the diskette, and the system.log file will no longer be able to expand to new blocks if necessary. You should therefore use Keep instructions in the same lesson with data file instructions only if you are certain that system.log will not exceed its current size (four blocks, or about 10,000 characters, when first created). Of course, if a separate SYSLOG diskette is being used for recordkeeping, this limitation does not apply.

Note: Common PILOT uses the FO: instruction to store information in an open file and to open a file when no file is currently open. In

Apple PILOT and SuperPILOT, you must use the special file-opening instruction FOX: or FIX: to open a file.

## Closing a File

Announcing that you are finished using the currently open file is called closing the file. In general, you must open a file yourself, but Apple SuperPILOT automatically closes the file for you. For example, any instruction that opens a new file automatically closes any currently open file first, except for the recordkeeping file, system.log .

The following instructions and commands close any open file:

FOX: , FIX: , LX: , CTRL-C in response to A: ,

final E: , end of lesson without E:

Pressing the RESET key (or turning the Apple II's power off) closes any open file, but in a potentially deleterious fashion: if the Apple II is in the process of writing a record onto the disk, the information in that single record will be lost. Therefore, it is prudent not to turn off the computer or to press RESET while the red light on the disk drive is "on."

Closing a file just means you are temporarily through storing or retrieving information in that file. The file and its contents remain on the lesson diskette, but that information is not available until the file is reopened. See the FOX: instruction for information on actually deleting a lesson-created file from a diskette.

## FOX: Create and Open New File

---

fox:largest-record number,name of new file

or

fox:∅,name of file to be deleted

Syntactically, this instruction is a modified FO: instruction, but its use is quite different. The FOX: instruction first closes any file that is already open, and then creates and opens a new file on the lesson diskette, using the file name specified. Subsequent storage and retrieval instructions ( FO: and FI: ) will refer to records within this file as long as it remains open; the record numbers specified by those instructions must not exceed the largest-record number given in the FOX: instruction.

For example, if you wanted your program to create and open a new file named comments , for storing and retrieving up to 21 records numbered ∅ through 2∅, you might use this instruction:

fox:20,comments

(Create the new file comments ;  
and open it for storing and  
retrieving information in  
records numbered 0 through  
20.)

You can also use a string variable name in place of a literal file name. The file will be created under the name currently stored in the string variable. This gives you the added flexibility of being able to name a file with input taken from an Accept or File Input instruction, as well as with a name assigned in a Compute instruction. For example:

d:n\$(10)

(Reserve space in memory for  
storing up to 10 characters  
in string variable n\$ .)

t:Hello, what's your first name?

(Display this text.)

a:\$n\$

(Accept student response and  
store it in string variable  
n\$ .)

fox:20,n\$

(Create a new file with the  
name currently in string  
variable n\$ , and open it  
for storing and retrieving  
information in records 0  
through 20.)

After these instructions, you could store a student's response in record number 12 of the file comments , as follows:

d:c\$(80)

(Reserve space in memory for  
storing up to 80 characters  
in string variable c\$ .)

c:c\$=%b

(Assign the contents of the  
Answer Buffer %b to  
string variable c\$ .)

fo:12,c\$

(Store the contents of string  
variable c\$ into record 12  
of the currently open file.)

and retrieve the stored response later using this instruction:

fi:12,c\$

(Retrieve the contents of  
record 12 in the currently  
open file, and store in  
string variable c\$ .)

If you wish to reopen an existing file without creating any new file, see the FIX: instruction, discussed later in this chapter.

## The Largest-Record Number

When you use the FOX: instruction to create a new file, the largest-record number should be an integer no less than 1 and no greater than 548, or a simple numeric variable containing any number in the same

range. In practice, however, the maximum number of records is limited by the amount of space available on the lesson diskette when the FOX: instruction is executed. There are a minimum of two records in a file, numbered 0 and 1 .

## Deleting a Data File

If a FOX: instruction specifies a largest-record number of 0, this does not create a new one-record file. Instead, SuperPILOT looks for an existing lesson-created file with the specified file name. If such a file is found, the Apple DELETES that file from the diskette. This allows your lessons to remove from the diskette any files that will no longer be needed, before the lesson comes to an end. Also, any other lesson can use this instruction to delete a file after retrieving that file's stored information.

If the file specified for deletion is not found, no error message is given and the Error Flag is not raised. Lesson execution quietly goes on to the next instruction. This lets you attempt to delete a file BEFORE creating a new file with the same name (for instance), which you may want to do if you feel there may not be enough space on the diskette for the new file. For example:

fox:0,a\$	(Delete a file--if it exists--with the name in a\$ .)
fox:20,a\$	(Create a new file with the name in a\$ , and open it for storing and retrieving information in records numbered 0 through 20.)

The usual process followed when creating a file is the opposite: FOX: creates the new file and THEN automatically deletes any old file with the same name.

## The New File

A file name for a new file must not exceed ten characters, and it must not contain any colons. It does not matter whether you type the file name in capital letters, lowercase letters, or both. Internally, all file name letters are always converted to capital letters. The new file name (or name of a file to be deleted) may be a literal or it may be stored in a string variable. For example, the following sequence of instructions:

```
d:a$(10)
c:a$="lie&lay"
fox:15,a$
```

is the equivalent of the one instruction:

```
fox:15,lie&lay
```



The latter method is faster to type and does not use any string-variable memory space. The method of storing the file name in a string variable first does have at least one advantage, however: you can have the student name the file during program execution. For example:

d:i\$(10)	(Reserve space in memory for storing up to ten characters in string variable i\$ .)
t:What is your I.D. number?	(Display this text.)
a:\$i\$	(Accept student's response; store in string variable i\$ .)
fox:15,i\$	(Create a new file with the name in i\$ , and open it for storing and retrieving information in records numbered 0 through 10.)

When the FOX: instruction creates a new file, it reserves enough space on the lesson diskette to store the specified number of records (from record 0 through the specified largest-record number). It then fills the entire storage area of the file with null characters (ASCII code 0), erasing any old information that may have been stored there. From then on, the file always contains exactly the same number of records it contained when created, even if you actually store information in only one record.

Each diskette block can contain two records. Records 0 and 1 are in the first block of the file, records 2 and 3 are in the second block, and so on. However, note that the instruction fox:0,x\$ attempts to DELETE the file whose name is stored in x\$ , so that the smallest file you can actually create contains two records (record 0 and record 1).

If there is not enough room on the diskette for a file of the specified size, the file created occupies all of the largest unused area on the diskette. A line at the bottom of each Editor menu tells how many blocks are available at that time. But note that fox: instructions, encountered in any running of this lesson or other lessons on the same lesson diskette, may change the amount of diskette space available.

If a file with the specified file name already exists on the lesson diskette, the new file is first created in the remaining diskette space, and THEN the old file with that name is deleted. This leaves some unused blocks where the old file was, not necessarily at the end of the diskette. When the Editors create or change a file, they "compact" the diskette after storing the new file. This makes sure all the unused diskette blocks are in one place at the end of the diskette. The diskette is not compacted, however, after a fox: instruction creates a new file, so successive fox: instructions creating files with the same file name may leave "pockets" of unused blocks here and there on the diskette. The total number of unused blocks may therefore be much larger than the largest contiguous set of unused blocks that can store a file. This situation will be remedied the next time you perform an action that normally compacts the diskette.

In a normal one-disk-drive Lesson Mode system, your lesson creates and uses files on the same diskette the lesson is stored on. For using systems with additional disk drives, see the appendix Using More Disk Drives.

## Example

pr:e	(Turn on the Escape option.)
d:r\$(80)	(Reserve space in memory for storing up to 80 characters in string variable r\$ .)
fox:20,comments	(Create the new file comments ; and open it for storing and retrieving information in records numbered 0 through 20.)
c:n=0	(Store the number 0 in simple numeric variable n ; this variable will contain the record-number counter.)
j:start	(Jumps over sysx routine to label start .)
*sysx	(Label for Fescape routine; the lesson Uses this routine if any response begins with @ .)
c:r\$=%b	(Assign the contents of the Answer Buffer %b to string variable r\$ .)
fo:n,r\$	(Store the contents of string variable r\$ in record n of the currently open file.)
c:n=n+1	(Add one to the record-number count stored in variable n .)
t(n>20):No more room for comments.	(If n exceeds 20, display text.)
prc:	(If n exceeds 20, turn off the Escape option.)
e:@a	(End sysx routine; jump back to same Accept that initiated this Escape routine.)
*start	(Label for main body of lesson.)
...	...
...	...

You might use a lesson portion like this example at the beginning of a program that you are developing. It lets you store a comment at any response point, by typing an @ sign followed by the comment. It then returns you to the same response point to Accept your normal response and continue with the lesson. Note that this example creates a new file comments every time you run the lesson, deleting any previous comments file. You would need to write another lesson or lesson portion to retrieve the comments stored by this example. See the examples at the

end of the other sections in this chapter for more uses of these instructions.

## **FIX:      Open Existing File**

---

`fix:largest-record number,name of old file`

Syntactically, this instruction is a modified `FI:` instruction, but its use is quite different. The `FIX:` instruction first closes any file which is already open, and then uses the file name to open a file which already exists on the lesson diskette. The Error Flag is raised if the named file is not found on the lesson diskette. Subsequent storage and retrieval instructions (`FO:` and `FI:`) will refer to records within this file as long as it remains open; the record numbers specified by those instructions must not exceed the largest-record number given in the `FIX:` instruction.

For example, if you wanted your program to open an existing file named `comments`, for storing and retrieving up to 21 records numbered 0 through 20, you might use this instruction:

`fix:20,comments`

(Find the existing file `comments`; open it for storing and retrieving information in records numbered 0 through 20.)

After these instructions, you could store a student's response in record number 12 of the file `comments`, as follows:

`d:c$(80)`

(Reserve space in memory for storing up to 80 characters in string variable `c$`.)

`c:c$=%b`

(Assign the contents of the Answer Buffer `%b` to string variable `c$`.)

`fo:12,c$`

(Store the contents of string variable `c$` into record 12 of the currently open file.)

and retrieve the stored response later using this instruction:

`fi:12,c$`

(Retrieve the contents of record 12 in the currently open file, and store in string variable `c$`.)

If you wish to create a new file, deleting any old file by the same name, see the `FOX:` instruction, discussed earlier in this chapter.

## The Largest-Record Number

The largest-record number must be an integer from 0 through 32767, or a simple numeric variable containing any number in the same range. In practice, however, the maximum number of records is usually limited to the number of records available in the specified diskette file. The size of that file was set earlier, when the file was created by a FOX: instruction.

Each time the FIX: instruction reopens a file it will normally specify the same largest-record number the FOX: instruction used when creating the file. However, SuperPILOT does not force you to be consistent; you may use a different largest-record number each time you open the file, if you wish.

If you create a file specifying one largest-record number, and later reopen the file with FIX: , giving a larger one, your storage and retrieval instructions can then specify record numbers corresponding to records which are outside the original file. Any attempt to store information into records beyond the originally created file, and any attempt to retrieve information from such beyond-the-file records, causes an error message (in Author Mode) and raises the Error Flag (in either Author or Lesson Mode).

You may occasionally find it useful to open an existing file of unknown length, using FIX: with an arbitrarily high largest-record number. You can then retrieve information from successive records until the Error Flag is raised to indicate that you have gone beyond the end of the file. In general, however, it is best to create your files with the proper largest-record number in the first place and then stick to that number in future use of the file.

The file name for the existing file to be opened may be stored in a string variable. It does not matter whether you type the file name in capital letters, lowercase letters, or both. Internally, all file name letters are always converted to capital letters.

If no file with the specified name is found on the lesson diskette, the FIX: instruction raises the Error Flag. The E conditioner can be used on an instruction following the FIX: instruction to handle this situation.

On a normal one-disk-drive Lesson Mode system, your lesson's instructions always use files on the same diskette the lesson is stored on. For using systems with additional disk drives, see the appendix Using More Disk Drives.

### Example

```
d:f$(10);1$(30)
```

```
(Reserve space to store up to  
10 characters in string  
variable f$ and up to 30  
characters in string variable  
1$ .)
```

t:What is your first name?	(Display this text.)
a:\$f\$	(Accept student's response; store in string variable f\$ .)
fix:9,f\$	(Try to open an existing file whose name is stored in f\$ , for storing and retrieving information in records 0-9.)
je(1):new	(If Error Flag is up, there is no old file f\$ , so lower Error Flag by evaluating the expression (1) , and Jump to the label new .)
j:old	(Old file exists, so Jump to section labelled old .)
*new	(Label for new-student section.)
fox:9,f\$	(Create and open a new disk file with the name stored in f\$ , to store and retrieve information in records 0-9.)
t:Well, \$f\$ , I see this : is your first time here. : What is your last name?	(Display this text, substituting student's first name for \$f\$ and following space.)
a:\$l\$	(Accept student's response; store in string variable l\$ .)
fo:0,l\$	(Store contents of l\$ into record 0 of the open file.)
j:main	(Jump to label main .)
*old	(Label for old-student section.)
fi:0,l\$	(Retrieve student's last name from record 0 of the open file; store that information in string variable l\$ .)
t:Well, \$f\$ \$l\$ , I see this : is not your first time here.	(Display this text, substituting student's first name for \$f\$ and last name for \$l\$ .)
*main	(Label for main lesson.)

This example asks the student to type a first name, and then tries to open (using FIX: ) an existing file which has the same file name as the student's first name. If there is a file by that name already, the student has used this lesson before, and the lesson then retrieves the student's last name from record 0 of the file. If there is no old file with the student's first name, such a file is created (using FOX: ), and the new student's last name is Accepted and stored into record 0 of the file.

Note: Only the first ten characters of the student's first name will be placed in string variable fS . This prevents a longer first name from creating an illegally long file name. If you wanted to use longer names in messages to the student, you could Accept the student's first name into two different string variables, one (limited to 10 characters) for the file name, and another (Dimensioned to any length) for use in messages:

```
a:$f$ $m$
```

## **FO: Output to Open File**

fo:record number,string (or name of string variable containing string)

The FO: instruction stores a string or the contents of the specified string variable into the specified record of the diskette file that is currently open. Each record can store one string only, up to 255 characters in length. Before the FO: instruction can store an item of information into a file, that file must first be created and opened with a FOX: instruction, or reopened with a FIX: instruction.

For example, to store the words "This is a recording..." into record 13 of a file that you have already opened, you could use these instructions:

d:s\$(50)	(Reserve space to store up to 50 characters in string variable s\$ .)
c:s\$="This is a recording..."	(Store the quoted string in string variable s\$ .)
fo:13,s\$	(Store the contents of string variable s\$ in record 13 of the currently open file.)

or this single instruction:

fo:13,This is a recording...	(Store the words "This is a recording..." in record 13 of the currently open file.)
------------------------------	---

In a real lesson, however, you would also have to include instructions to open a file. Suppose you want to store your words into an existing file named mabel says :

fix:20,mabel says	(Open the existing file mabel says for storing and retrieving information in records 0-20.)
fo:13,This is a recording...	(Store the words "This is a recording" in record 13 of the currently open file.)

## The Record Number

The record number must be an integer from 0 through the largest-record number specified by the last file-opening instruction ( FIX: or FOX: ), or a simple numeric variable containing any number in the same range.

If you create a file specifying one largest-record number, and later open the file again with FIX: , giving a larger maximum, your storage and retrieval instructions can then specify record numbers corresponding to records that are outside the originally created file. Any FO: instruction that tries to store information into records that are beyond the file created by FOX: causes an error message (in Author Mode) and raises the Error Flag (in Author or Lesson Mode).

In general, it is best to create your files with the proper maximum record number in the first place, and then stick to that number in future use of the file.

Note: Common PILOT uses the FO: instruction to open a file if no file is currently open, as well as to store information in records of an open file. Both Apple PILOT and SuperPILOT require you to use the special file-opening instructions FOX: and FIX: to open a data file; the FO: instruction is used only for storing information in a record of an already-open file.

## Example

This rather long example can be used to create a new diskette file and store information in any of its records, or to change the information stored in any record of an existing file.

pr:u	(Convert all response letters to capital letters.)
d:f\$(10)	(Reserve space to store up to 10 characters in string variable f\$ .)
d:x\$(255)	(Reserve space to store up to 255 characters in string variable x\$ .)
*name	(Label for file-name section.)
t:Type the name of your file.	(Display this text.)
a:\$f\$	(Accept student's response; store first ten characters in string variable f\$ .)
fix:999,f\$	(Try to reopen existing file with the name stored in f\$ , for storing and retrieving records up to record 999 .)
je(1):number	(If Error Flag is up, no old file with that name was found, so lower Error Flag by evaluating the expression (1) and Jump to label number .)

t:A file with the name \$f\$ already : exists. Do you want to use that : file name anyway?	(Display this text, substituting the file name for \$f\$ .)
a:	(Accept student's response.)
m:YES!OK!RIGHT!SURE!COURSE	(Did the student type one of these words?)
jn:name	(If No, Jump back to name .)
t:Okay. Do you want to change : the records in that file, or do : you want to erase the old file : and start a new file named \$f\$ ?	(Yes, so display this text.)
a:	(Accept student's response.)
mj:CHANGE!ALTER	(Did the student type one of these words? If not, Jump to next Match instruction.)
j:record	(Yes, so Jump to record .)
m:ERASE!START!NEW	(Did the student type one of these words?)
jy:number	(If Yes, Jump to number .)
t:I don't understand. Do you : want to erase the file or : change its records?	(Display this text.)
j:@a	(Jump back to last Accept.)
*number	(Label for new-file section.)
t:How many records do you want : the new file to contain?	(Display this text.)
a:#n	(Accept student's response, and store response number in simple numeric variable n .)
te:Please type a number.	(If Error Flag is up, no number was typed, so display text.)
je:@a	(If Error Flag is up, Jump to last a: .)
c:t=n-1	(Subtract 1 from number of records n to obtain the largest-record number t .)
t(t<2!t>200):Choose a number : from 2 to 200 .	(If t is less than 2 or is more than 200, display text.)
jc:number	(If last expression was true, Jump back to number .)
t:Starting a new file named \$f\$ , :containing records 0 through #t .	(Display this text, substituting file name for \$f\$ and larg- est-record number for #t .)



fox:t,f\$	(Create and open a new diskette file with the name stored in f\$ , for storing and retrieving information in records 0 through t .)
te:Sorry, the diskette has no room :for your file. To restart the : system, press any key.	(If Error Flag up, diskette was full, so display text.)
we:l00	(If Error Flag up, wait l00 seconds or until keypress.)
ee:	(If Error Flag up, end this lesson here.)
fi:t,x\$	(Attempt to retrieve contents of last record, number t , storing contents in string variable x\$ .)
te(l):Warning: space restrictions :caused your new file to be created : with fewer than #n records.	(If Error Flag up, record t was not in the new file, so lower flag and display text.)
*record	(Label for this section.)
t:Type the number of the record :whose contents you wish to alter.	(Display this text.)
a:#r	(Accept student's response; store in simple numeric variable r .)
t(len(%b)=0):Closing file \$f\$ .	(If only the RETURN key was pressed, display this text.)
ec:	(If last expression was true, End this lesson here.)
te:Please type a number.	(If Error Flag is up, no number was typed, so display text.)
je:@a	(If Error Flag up, Jump back to last a: .)
fi:r,x\$	(Retrieve contents of record r; store in string variable x\$.)
te:Record #r is not in the file.	(If the Error Flag is up, display this text.)
je(l):record	(If Error Flag up, lower flag and Jump back to record .)
t:Old Record #r : \$X\$	(Display this text.)
th:New Record #r :	(Display this text, leaving cursor after the colon.)
ax:\$x\$	(Accept the student's exact response; store it in x\$ .)
j(len(%b)=0):record	(If only RETURN was pressed, leave record r unchanged, and Jump back to record .)
fo:r,x\$	(Store contents of x\$ into record r of the open file.)
j:record	(Jump back to label record to get the next record number.)

This example demonstrates most of the concepts you need for storing information in a diskette file. Pay special attention to the many error-handling instructions.

In the section labelled `name`, the lesson Accepts a file name from the student, and tries to open an old file with that name, using a largest-record number large enough to include any existing file. If it succeeds, it lets the student give another file name or choose between erasing the old file and changing records in the old file.

If no old file with the specified name is found, or if the student wants to erase the old file and start a new file with that name, the lesson then Accepts the number of records the new file is to contain. At the end of this section, labelled `number`, the lesson creates and opens a new file using the chosen name and the largest-record number calculated from the specified number of records. (Since all files start with record number  $\emptyset$ , the largest-record number is one less than the number of records in the file.)

In the section labelled `record`, the lesson Accepts a record number from the student, retrieves and displays the current contents of that record, and then Accepts and stores new information in that record. This section repeats until the student presses the RETURN key instead of specifying a record number.

The student-supplied name for the file is automatically limited to ten characters maximum length by assigning the name to a string variable that was Dimensioned to ten characters. The student does not need to know about this limitation, because any retrieval program will also limit the name to ten characters maximum. As long as the student types the same name each time, the file will be found.

See the end of the `FI:` instruction, next in this chapter, for an example of a lesson that can retrieve the information in the files created by this example.

## **FI: Input from Open File**

---

`fi:record number,name of string variable`

The `FI:` instruction retrieves the contents of the specified record from the diskette file that is currently open, and stores that information in the specified string variable. Before the `FI:` instruction can retrieve an item of information from a file, that file must first be created and opened with a `FOX:` instruction, or re-opened with a `FIX:` instruction.

For example, to retrieve the information that was earlier stored in record 13 of a file that you have already opened, you could use these instructions:

d:s\$(50)	(Reserve space to store up to 50 characters in string variable s\$.)
fi:13,s\$	(Retrieve contents of record 13 of currently open file, and store in string variable s\$.)

If the file whose information you want to retrieve is not already open, your lesson would have to include instructions for opening the correct file. Suppose the information you want is in record 13 of a file named mabel says :

d:s\$(50)	(Reserve space to store up to 50 characters in string variable s\$.)
fix:20,mabel says	(Find the existing file whose name is mabel says , and open it for storing and retrieving information in records 0-20.)
fi:13,s\$	(Retrieve contents of record 13 of currently open file, and store in string variable s\$.)

## The Record Number

The record number must be an integer from 0 through the largest-record number specified by the last file-opening instruction (FIX: or FOX: ), or a simple numeric variable containing any number in the same range.

If you create a file specifying one largest-record number, and later open the file again with FIX: , giving a larger number, your storage and retrieval instructions can then specify record numbers corresponding to records that are outside the originally created file. However, any FI: instruction that attempts to retrieve the contents of such beyond-the-file records causes an error message (in Author Mode) and raises the Error Flag (in both Author and Lesson Mode).

Sometimes it may be useful to retrieve successive records from a file of unknown length until the Error Flag indicates you have gone beyond the end of the file. In general, though, it is best to create your files with the proper largest-record number in the first place, and then stick to that number in future use of the file.

## The String Variable

When an item of information is retrieved, it must be stored in a string variable, and that string variable's name must appear in the FI: instruction.

Each record in a file contains a single string, up to 255 characters long. When your FI: instruction retrieves the contents of a record, the string stored in the specified record is then stored in the

specified string variable. The record string is shortened to the maximum Dimensioned length for the receiving string variable, if necessary.

## Example

This example lets you retrieve and display the contents of each record in an existing lesson-created file.

d:f\$(10);x\$(255)	(Reserve space to store up to 10 characters in string variable f\$ and up to 255 characters in string variable x\$ .)
*name	(Label for file-name section.)
t:Type the name of the file	(Display this text.)
:whose contents you want to see.	
a:\$f\$	(Accept student's response; assign first 10 characters to string variable f\$ .)
e(len(%b)=0):	(If only the RETURN key was pressed, End lesson here.)
fix:999,f\$	(Open an existing diskette file with the name stored in f\$ , for storing and retrieving records up to record 999.)
te:Sorry, I can't find that file.	(If Error Flag is up, no file with that name was found, so display this text.)
je(1):name	(If Error Flag up, lower flag by evaluating the expression (1) and Jump back to name .)
c:r=0	(Store initial number 0 in the record-count r .)
*display	(Label for display section.)
fi:r,x\$	(Retrieve contents of record number r of the file now open, and store the string retrieved in string variable x\$ .)
te:That's all of file \$f\$ .	(If Error Flag is up, record r was not in the file, so display this text.)
we:5	(If the Error Flag is up, wait 5 seconds.)
je(1):name	(If Error Flag up, lower flag and Jump back to name .)
t:Record #r : \$X\$	(Display text, substituting record number for #r and one space and record contents for \$x\$ and one space.)

c:r=r+1

(Increment record-counter r  
by 1 .)

j:display

(Jump back to display to  
retrieve next record.)

This example lesson demonstrates most of the concepts you need for retrieving information that has been stored in a lesson-created diskette file. In the name section, the lesson Accepts a file name from the student, and opens an existing file with that name, if the file is on the lesson diskette. The FIX: instruction uses a largest-record number large enough to include all the records of any file. If the student just presses the RETURN key, without typing a file name, that response ends the lesson.

Next, in the section labelled display , the lesson retrieves and displays each record in the open file, starting from record number 0 . When the Error Flag is raised, this indicates that the lesson has gone beyond the last record in the originally created file.

The student-supplied name for the file is automatically limited to ten characters maximum length by assigning the name to a string variable that was Dimensioned to ten characters. The student does not need to know about this limitation, because the storage program also limited the name to ten characters maximum. As long as the student types the same name each time, the file will be found.

See the end of the FO: instruction, earlier in this chapter, for an example of a lesson that can create or change the files whose contents are retrieved by this example.



## Chapter 8

# Execution-Time Commands

194 Goto Command  
195 Example

197 @ Escape Command  
198 Example  
198 Example

200 CTRL-C

200 CTRL-I

201 RESET

# Goto Command

---

The form of the typed response is:

goto label or unlabelled destination (any text may follow)

There are several commands available to the student when a lesson is running, in addition to the normal interplay of displayed questions and typed answers. Some of these commands, such as CTRL-C and RESET, can be used at any time, in any lesson. However, the student can use the Goto command only if the Goto option has been selected by a PProblem instruction. The instruction that allows the student to use the Goto command looks something like this (the instruction may select other options at the same time):

```
pr:g                                     (Let the student use the Goto
                                         execution-time command in
                                         typed responses to Accept
                                         instructions.)
```

For details about setting control options, see the PProblem instruction, in the Response Instructions chapter.

After the Goto control option is selected by a PProblem instruction such as pr:g, the student's typed responses to the Accept instruction are treated in the usual way unless the response begins with the letters goto (or GOTO). A response of the form

```
goto destination
```

is interpreted as a Goto command, which causes a Jump to the specified destination. The choices for the destination are:

goto label	Jumps to the specified label.
goto @a	Re-executes the current Accept instruction.
goto @m	Jumps ahead to the next Match instruction.
goto @p	Jumps ahead to the next PProblem instruction.

The command goto @a is not useful, since it merely re-executes the same Accept instruction that just Accepted the Goto command. The command goto @m Jumps to the next Match instruction, which then treats the Goto command itself as the Accepted response in which to look for the Match words. Any additional text may follow the Goto command, and will be used as part of that Accepted response.

The Accepted response, which includes the first characters goto and the destination, may be used in the part of the lesson to which execution Jumps. This Accepted response is stored in the Answer Buffer (system variable %b) but it is NOT stored in any string variables or numeric variables that may appear in the Accept instruction's object field. The values previously stored in the Accept instruction's object variables are not changed by the Accept instruction that Accepts the



Goto command. The Answer-Count (system variable %a ) is incremented in the usual way by this Accept instruction.

If a destination label is specified, but that label is not found in the lesson, program execution continues at the instruction following the Accept instruction. In this case, the Accepted response is still stored only in the Answer Buffer %b , and not in any of the Accept instruction's object variables. If an unlabelled destination is specified, but that destination is not found, an error message is displayed (in Author Mode), and the program resumes with the instruction following the Accept instruction where the invalid Goto command was given (in Author or Lesson Mode).

The Goto command can be a very useful tool for developing a lesson. You can use Goto commands to repeat a section under test again and again, trying different responses to determine what problems may arise. Conversely, you can skip over whole sections of the lesson that have known errors or that you have already tested.

If you select the Goto option for your own use as a testing aid, you can later eliminate Goto commands for student use by removing the G specifier from the PProblem instruction. A student's response beginning with goto will then have no special effect, and will be treated like any normal response.

However, the Goto command can also be a valuable tool for the advanced student. If the lesson tells the students what choices are available, the students can then direct their own progress through the lesson. For example, some students may wish to take a test immediately, to see if they already know the material in this lesson. Others may wish to review certain sections, or skip over sections.

## Example

pr:gl	(Allow use of the Goto command and convert all response letters to lowercase.)
*setl	(Label for this section.)
c:x=1	(Assign a value of 1 to simple numeric variable x .)
u:prompt	(Branch to subroutine prompt , returning here after next e: instruction is executed.)
t:Solve this problem:	(Display this text.)
th: 10 - 3 =	(Display this text, leaving the cursor on the same line.)
a:#a	(Accept student's answer, and store it in simple numeric variable a .)
...	(Further problems in this set.)

. . .	(Further problem sets, each one with higher numbered label.)
*set15	(Label for this section.)
c:x=15	(Assign a value of 15 to simple numeric variable x .)
u:prompt	(Branch to subroutine prompt , returning here after next e: instruction is executed.)
t:Solve this problem:	(Display this text.)
th: $-(10^2-57+16-(13+33))-7 =$	(Display this text, leaving the cursor on the same line.)
a:#a	(Accept student's response, and store it in simple numeric variable a .)
. . .	(Further problems in this set.)
*prompt	(Label for subroutine prompt .)
ts:v0,39,20,23;es	(Set text viewport to occupy the last four screen lines; erase this new viewport.)
t:This is problem set #x .	(Display this text.)
t(x<15):For harder problems, type : GOTO SET#(x+1) .	(If the current value of x is less than 15, display this text, substituting current value of x+1 for #(x+1) and the following space.)
t(x>1):For easier problems, type : GOTO SET#(x-1) .	(If the current value of x is greater than 1, display text, substituting current value of x-1 for #(x-1) and the following space.)
ts:v0,39,0,19;es	(Set viewport to occupy all but the last four screen lines; erase this viewport.)
e:	(Terminate this subroutine and return to the instruction after one that called it.)

If you use a subroutine similar to the one in the above example, the student will always have a reminder at the bottom of the screen that he or she can break out of a problem set and begin an easier or more difficult one. When you allow the Goto option in a lesson, you make it much easier to "class test" it, because you can use Keep instructions ( K: ) to store information about which sections students select most often.

## @ Escape Command

---

The form of the typed response is:

@[any text]

The student can use the Escape command only if the Escape option has been selected by a PProblem instruction. The instruction that allows the student to use the Escape command looks something like this (the instruction may select other options at the same time):

```
pr:e                                (Let the student use the
                                     Escape execution-time
                                     command in responses to
                                     a: instructions.)
```

For details about setting control options, see the PProblem instruction in the Response Instructions chapter.

After the Escape control option is selected by a PProblem instruction such as `pr:e`, the student's typed responses to the Accept instruction are treated in the usual way, unless the response begins with the character `@`. A response whose first character is an `@` is interpreted as an Escape command, which causes program execution to branch to an author-supplied subroutine labelled `sysx`. This branch to the `sysx` subroutine, initiated by the Escape command, is equivalent to the following Use-subroutine instruction:

```
u:sysx                              (Jump to the subroutine
                                     labelled sysx, and return
                                     to the instruction after
                                     this u: instruction when
                                     the subroutine ends with an
                                     e: instruction.)
```

For more information about using subroutines, read the discussions of the Use and End instructions in the Control Instructions chapter. Do not confuse the End instruction `E:` with the PProblem instruction's Escape option `pr:e`. The same letter means very different things.

If PProblem instruction selects the `E` option, the author must provide a subroutine beginning with the label `sysx`. Normally, the subroutine should End with an `E:` instruction. The `E:` returns to the instruction after the `A:` that Accepted the `@` (Escape) command, unless the `E:` specifies some other branch destination.

If the Escape command is issued, but no subroutine labelled `sysx` is found, program execution continues at the instruction following the Accept instruction that Accepted the Escape command. However, the next `E:` instruction may cause a "return" to the same instruction following this Accept instruction.

The Accepted response, which includes the first character `@`, may be used in subroutine `sysx`. This Accepted response is stored in the

Answer Buffer (system variable %b ) but it is NOT stored in any string variables or numeric variables that may appear in the Accept instruction's object field. The values previously stored in the Accept instruction's object variables are not changed by the Accept instruction that Accepts the Escape command. The Answer-Count (system variable %a ) is incremented in the usual way by this Accept instruction.

The subroutine sysx can be as simple as an end-the-program routine, providing you or the student an effective escape. To do that, just put the label sysx as the last line of the lesson. The sysx subroutine can also be relatively complex, letting you save and record comments, or pass data to a special function you have written.

## Example

This example shows a sysx subroutine that can be used to reveal the values of three program variables during program execution.

pr:e	(Activate the Escape option.)
. . .	(Body of the lesson.)
*sysx	(Label for Escape routine; program branches here if any response begins with @ .)
t:The current angle is #d degrees.	(Display these lines of text, substituting the actual values of the variables d , t , and s at the point in the lesson where the sysx routine was called.)
t:The elapsed time is #t seconds.	
t:The current speed is #s ft./sec.	
e:@a	(End this subroutine by branching back to the same Accept instruction that initiated this Escape.)

For small-scale testing, you may find it more convenient to use Immediate Execution Mode (see the Overview of the Language chapter). However, for repeated tests of a similar nature, such as in the example above, a sysx routine can be a significant time-saver.

## Example

This example shows a sysx routine that would act as a calculator if the student preceded an expression by the @ sign:

pr:eu	(Let student use the Escape option; convert response letters to capitals.)
d:x\$(50)	(Reserve space for string variable x\$ to store up to 50 characters.)

```

t:You can do a calculation at any      (Display all of this text.)
: response point. Just type an
: "@" followed by any mathematical
: expression (for example, @56/32.9).
: You will see the answer to your
: calculation, and then return to the
: same response point in the lesson.
: Type "@ end" to stop this lesson.
t:
t:If 12 people can write 16 lines of
: corn in 6 hours, how many lines
: can 19 people write in 8 hours?
a:                                     (Accept student's response.)
...

*sysx                                 (Label for Escape subroutine;
                                     program jumps here if any
                                     response begins with @ .)

m:END                                 (Did student type END ?)
jy:stop                               (If Yes, Jump to label stop .)
c:x$=%b                               (No, so store Answer Buffer %b
                                     in string variable x$ .)

c:a1=len(x$)-1                       (Set a1 to one less than the
                                     length of the string in
                                     x$ .)

c:x$="c:n4="!!x$(2,a1)               (Make a new string, consisting
                                     of "c:n4=" plus all but the
                                     first character of old x$ .
                                     Store new string back into
                                     string variable x$ .)

xi:x$                                 (Execute the contents of x$
                                     as a SuperPILOT
                                     instruction.)

te:That was not a correct            (If Error Flag is up, then
: mathematical expression.          display this text.)
ee(1):@a                             (If Error Flag is up, lower it
                                     by evaluating (1) ; then
                                     return to Accept instruction
                                     that started this Escape.)

t:The answer is #n4                  (No error, so display answer.)
e:@a                                  (End subroutine sysx by a
                                     return to Accept instruction
                                     that started this Escape.)

*stop e:stop                         (Ends this lesson.)

```

When the student types a response of, say,

```
@(12*19*8)/(16*6)
```

the program Escapes to the sysx subroutine, which removes the leading @ sign, makes up the new string "c:n4=(12\*19\*8)/(16\*6)", and stores that string in variable x\$. Then it executes the string stored in x\$

as a SuperPILOT Compute instruction, shows the student the result, and returns to the same Accept instruction from which this Escape was initiated.

Note: The PR: instruction converts response letters to capitals for the convenience of the Match instruction. It is not necessary to do this conversion for the Escape command to work correctly.

## **CTRL-C**

---

The form of the typed response is:

Type C while holding down the CTRL key

In Lesson or Author Mode if a CTRL-C is typed in response to any Accept instruction, the lesson currently running is immediately terminated, and the system restarts itself. To type CTRL-C, type a C while holding down the CTRL key. This execution-time command provides a handy escape from any lesson while it is running.

You do not need to set any option to enable this command, and you cannot set any option to disable this command. It is always available in any lesson during the response to an Accept instruction.

Note: The CTRL-C command does not terminate the lesson when given in response to an AP: instruction.

If you want the student to know about the CTRL-C command, your lesson can use an instruction such as this:

```
t:To stop this lesson at any           (Display this text on
: response, hold down the CTRL key,    the student's screen.)
: and type c while continuing
: to hold down the CTRL key.
```

Finally, if you type CTRL-C when one of the Editor Menus is on the screen, you are returned to the previous prompt.

## **CTRL-I**

---

The form of the typed response is:

Type I while holding down the CTRL key

When used in Author Mode in response to an A: instruction, CTRL-I will place you in Immediate Execution Mode. The "greater than" prompt appears on the screen (assuming you have not changed the character set) to tell you that any command you enter will be executed immediately. To leave Immediate Execution Mode, simply type CTRL-I again, and you

will be returned to the lesson at the instruction following the Accept instruction where you were before.

CTRL-I also allows you to enter Immediate Execution Mode even when a lesson is not running. In Author Mode when the Lesson Text Editor Menu is on the screen, you may select the Run option, and then, instead of typing in the name of a lesson to Run, just type CTRL-I and you are placed in Immediate Mode. CTRL-I again will not return you to the Lesson Text Editor Menu, however. Instead, type CTRL-C or e: .

Finally, you can allow your students to use CTRL-I for access into Immediate Execution Mode. In Lesson Mode this may only be done when the Lesson diskette is first booted and the title page appears on the screen. A prompt line appears as follows:

(Press CTRL-I for immediate mode)

Immediate Mode is not available from Lesson Mode when a lesson is running, so that students will not be able to intentionally or unintentionally change values of variables or otherwise interfere with lesson execution. If you do not want your students to have access to Immediate Mode even at the startup stage, you can bypass the diskette title page by including a lesson named hello on the diskette. A hello lesson always runs immediately upon booting a lesson diskette in Lesson Mode, so the diskette's title page is not displayed.

See the Overview of the Language chapter for more information on how to use Immediate Execution Mode.

## RESET

---

The response is:

Press the RESET key [while holding down the CTRL key]

When you are running any program in Apple SuperPILOT, you can press the Apple II's RESET key any time you want to stop the program. This causes the system to restart itself, just as it starts itself when you first turn it on.

The only time you should avoid pressing the RESET key is when the system is busy storing information on the diskette. If the disk drive's red "In Use" light is off, it is always safe to press the RESET key.

Sometimes a programming error may result in a loop that performs some instructions over and over and over. Pressing the RESET key may be the only way to break out of such a loop.

Note: Newer Apple II keyboards are equipped with a switch that lets you change the normal RESET command to CTRL-RESET (press the RESET key while holding down the CTRL key). For simplicity, the descriptions in this manual use the term RESET to apply to the act of resetting the Apple II, whether this is accomplished on your system as a CTRL key operation or not.



## Chapter 9

# Hints for Beginners

204 Using the Student's Name

206 Using Numbers

207 Counting Answers

209 Example

This section should prove most useful to those who have little experience with mathematics and computer programming. You will learn three simple procedures that make your lessons much more effective: using the student's name, using numbers, and counting answers. You can use these three procedures just as they are in your own lessons, or you can use what you learn here as a springboard for understanding many of the more complex operations offered in Apple SuperPILOT.

You need to understand one concept before you read any further: the variable. A variable is just a fancy term for a place in the Apple II where you can temporarily store a number or a word that you may want to use later. The reason such a storage place is called a variable is that you can change or vary what is stored there while the program is running. You give this variable a name, and refer to it by name in your lesson programs. If the item you plan to store in the variable is a number that you want to use in a mathematical way, the name you give is usually a one-letter name, like `t`. If, on the other hand, you plan to store letters, numbers, or other characters in a literal sequence exactly as you type them (called a string), the variable's name will end with a dollar-sign, as in `w$`. Variable names in SuperPILOT may have a number as well as a letter in them, such as `t5` and `w2$`, but no name can begin with a number and no name can be more than two characters long (not counting the dollar-sign).

Once the variable has a name, you can put things in it yourself, or you can let the student put things in it, as you will soon see. You will tell the Apple to "store this number in `t`," or "store this word in `w$`." To use what you have stored, you can put the variable's name (in this case, `t` or `w$`) in a program instruction, and Apple SuperPILOT will automatically substitute the stored number or word wherever that name appears.

Using a variable name instead of a number or word can be convenient in two ways. It saves you from typing the actual number or word each place you need it in the lesson, because you can type the one-letter variable name instead. And you can easily change that number or word throughout a lesson, just by changing what you put in the named variable.

## Using the Student's Name

---

This is an easy trick that will make your lessons seem much more personal. It lets any of your messages to the student incorporate the student's name. To do this, you will store the student's name in a string variable, which is the term for a variable that holds words or any literal sequence of characters. We will name this variable `w$`.

Before you can use a string variable, you must first use a Dimension instruction to tell SuperPILOT how big this variable should be. Let's assume that no first name will be longer than, say, 30 characters. This is the instruction that tells SuperPILOT you need a string variable named `w$`, big enough to store up to 30 characters:

d:w\$(30)

(Reserve a place called w\$  
for storing words up to  
30 characters long.)

The next step is to tell the student to type her or his first name:

t:Please type your first name.

(Display this text on the  
student's screen, except  
for the t: .)

And now you need to tell the Apple II to accept the student's typed name. Here is an instruction that not only accepts the name but also stores it in the place called w\$ :

a:\$w\$

(Accept the student's response  
and store the first 30  
characters in the string  
variable named w\$ .)

Notice that the Accept instruction requires you to put an extra dollar sign in front of the normal name of the string variable.

That's all there is to it. From now on, any message in your lesson can use the student's name, which is stored in w\$ . Here's how you would use the stored name in a Type instruction:

t:Well, \$w\$ , how old are you?

(Display the text following  
the t: on the student's  
screen, after replacing  
\$w\$ and one space with the  
word stored in the string  
variable named w\$ .)

Notice that the Type instruction also requires you to put an extra dollar sign in front of the name for the string variable. What's not so obvious, but equally important, is that you must put a space after the variable's name, too. In this case, the dollar sign before the variable's name and the space after it form a special signal to the Type instruction. They say, "Don't display the \$w\$ literally, but in its place display the student's name that is stored in w\$ ." Since the space after the variable's name is part of the special signal, that space is not displayed on the student's screen. When the example is printed on the student's screen, the comma will immediately follow the student's name.

In this example, you used the Accept instruction to store a word in your string variable w\$ . That lets the student type the word that is to be stored. But you can also put a word in that variable yourself, from within the lesson. Here's a Compute instruction that stores the word Apple in the string variable named w\$ :

```
c:w$="Apple"
```

(Store the word Apple in the string variable named w\$.)

Notice that the Compute instruction does not require you to put any extra dollar sign in front of the name of the string variable.

You can store new words in a string variable as often as you wish during a lesson. Each time you put a new word in that variable, the old word stored there is forgotten.

## Using Numbers

---

Often, you may want your lessons to use a number typed by the student. For instance, the previous instruction asked the student "How old are you?" To use the student's answer, you will store the number part of the answer in a simple numeric variable, which is the term for a number-storage place. We will name this variable `n`.

Since all numbers (as stored by SuperPILOT) use the same amount of memory space, you do not need to use a Dimension instruction before using a numeric variable. Instead, you can Accept the student's age, name a numeric variable, and store the student's age in the numeric variable, all in one instruction.

```
a:#n
```

(Accept the student's typed response, and store the response's first number in the numeric variable `n`.)

Notice that the Accept instruction requires you to put a pound-sign ( `#` ) in front of the name of the numeric variable. Another thing to notice is that this instruction only recognizes actual digits, such as in 12 or 99, not words that spell numbers, such as twelve or ninety-nine. What gets stored in the numeric variable named `n` is the first recognizable number typed by the student, ignoring any words or other non-number typing. If the student types

```
I am twelve years and 2 months and 26 days old.
```

the number `2` will be stored in the numeric variable named `n`.

Well, that does it. From now on, any message or calculation in your lesson can use the student's age, which is stored in `n`. Here's how you would use the stored age in a Type instruction:

t:Do you like being #n years old? (Display the text that follows the t: on the student's screen, after replacing #n and one following space with the number stored in the numeric variable n .)

Here again, the Type instruction requires that you put a pound sign ( # ) in front of the normal name for the numeric variable. But again, what's not so obvious is that you must also put an extra space after the variable's name. The pound sign before the name and the space after the name are the Type instruction's signal to use these characters as the name of a numeric variable, and to display the number stored there. Since the first space after the variable's name is part of the special signal, that space is not displayed on the student's screen. When the example is printed on the student's screen, there will be only a single space immediately following the student's age.

In this example, you used the Accept instruction to let the student store a number in your numeric variable. But you can also put a number in a numeric variable yourself from within the lesson. Here's a Compute instruction that stores the number 2237.84 in the numeric variable named n :

c:n=2237.84 (Store the number 2237.84 in the numeric variable named n .)

Notice that the Compute instruction does not require you to put any extra pound sign in front of the name of the numeric variable.

You can store new numbers in a numeric variable as often as you wish during a lesson. Each time you put a new number in that variable, the old number stored there is forgotten.

## Counting Answers

---

Suppose your lesson contained the following instructions:

*dozen	(Label for this section.)
t:How many eggs are there	(Display this text on the student's screen.)
: in a dozen?	(Accept the student's answer.)
a:	(Did the student's answer
m:12!twelve!TWELVE	"match" any of these three words: 12 , twelve , or TWELVE ?)
ty:That's right!	(If one of the Match words was found in the student's answer, display this text.)

jy:next

(If one of the Match words was found in the student's answer, Jump ahead to continue the lesson at the label next , for the next problem, ignoring any other instructions in between.)

    t:No, that's not the right  
    : answer. Try again.

(If the program gets to here, none of the Match words were in the student's answer, so display this text.)

    j:dozen

(Jump back to label dozen for another try.)

    \*next

(Label for the next section, marking a place that a Jump instruction can specify.)

If the student does not know how many eggs make a dozen, these instructions will present the question over and over and over, forcing the student to go on guessing indefinitely. Since the idea is to instruct, not to punish, you need a way to limit how many times a student tries to answer the same question.

You can do this very easily, using Apple SuperPILOT's Answer-Count conditioner on any instruction. The Answer-Count conditioner is a number from 1 through 9 that you put immediately after an instruction name. That instruction will then be executed only if the student has tried to answer the same question a number of times equal to the number you wrote after the instruction's name. For example, this instruction

    t3:No, there are twelve eggs  
    : in a dozen.

(Display the text after the t3: only if the latest Accept instruction has been executed 3 times in a row.)

will be skipped over unless the student has given an incorrect answer to the same question for the third time in a row.

So, to rescue the poor student after three unsuccessful guesses, we could add two extra instructions to the previous set:

    \*dozen  
    t:How many eggs are there  
    : in a dozen?  
    a:  
    m:12!twelve!TWELVE

(Label for this section.)

(Display this text on the student's screen.)

(Accept the student's answer.)

(Did the student's answer "match" any of these three words: 12 , twelve , or TWELVE ?)

ty:That's right!

(If one of the Match words was found in the student's answer, display this text.)

jy:next

(If one of the Match words was found in the student's answer, Jump ahead to continue the lesson at the label next , for the next problem, ignoring any other instructions in between.)

t3:No, there are twelve eggs  
: in a dozen.

(If program gets here, answer was wrong. If this is third answer, display this text.)

j3:next

(If third answer, Jump ahead to the label next .)

t:No, that's not the right  
: answer. Try again.

(If the program gets to here, none of the Match words were in the student's answer, so display this text.)

j:dozen

(Jump back to label dozen for another try.)

\*next

(Label for the next section, marking a place that a Jump instruction can specify.)

Note: A colon-continuation of an instruction, as shown in the two-line Type instructions for example, continues all the conditioners and modifiers, as well as the instruction name, in the second line. A continued Type instruction will not appear on the student's screen in the same two-line format. Instead, the displayed words will all go on the first line, if possible, only jumping to the next line if there are still some words that just won't fit on the first line.

## Example

---

Here's an example of a lesson portion that stores and uses the student's name, accepts and stores a number from the student, and uses both the name and the number in messages that show the student what two times that number is. It repeats this number-doubling game until the student types the word "end" or until nine numbers have been doubled, and then it goes on to the next section of the lesson. The section called next is not supplied in the example. It is there to show where the lesson branches to from the Jump instruction. If you try typing this example into the Lesson Text Editor and running it, it will just end when it goes to the section labelled next .

d:f\$(30)

(Create the string variable named f\$ , with space for storing up to 30 characters.)

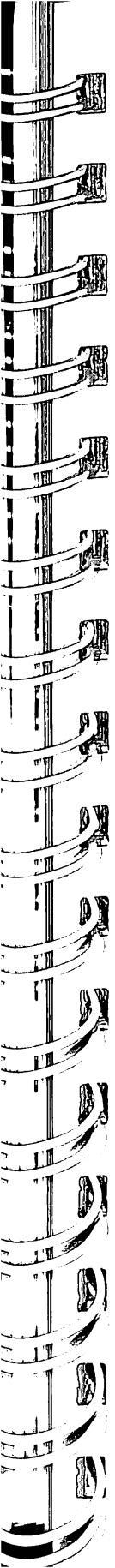
t:Type your first name, please.

(Display this text on the student's screen.)

a:\$f\$	(Accept student's typed name and store it in the string variable named f\$ .)
t:Very good, \$f\$ .	(Display this text, after substituting the student's name (stored in the string variable named f\$ ) for \$f\$ and one following space.)
*double	(Label for this section.)
t:Type a number and I will double : it, or type "end" to stop.	(Display these lines of text.)
a:#d	(Accept student's typed number and store it in the numeric variable named d .)
m:end!END!stop!STOP!quit!QUIT	(Did the student type any of these six words: end , END, stop , STOP , quit , QUIT ?)
jy:next	(If Yes, Jump to label next .)
c:a=2*d	(Make a copy of the number now stored in the variable named d , multiply it by 2, and store the result in the numeric variable named a .)
t:Okay, \$f\$ , twice #d is #a .	(Display this text, substituting the name stored in string variable f\$ for \$f\$ and one following space, and substituting the numbers in numeric variables d and a for #d and #a and one space after each.)
t9:I'm tired of that, \$f\$ . : Let's try something else.	(If this is the 9th answer in a row, display this text, using the name stored in f\$ in place of \$f\$ and one following space.)
j9:next	(If this is the 9th answer in a row, Jump to label next .)
j:double	(Otherwise, Jump back to the label double , for another number.)
*next	(Label for the next section.)

There are a number of limitations to this lesson portion, some of which could be improved by more advanced techniques. For instance, a Problem instruction could be used to convert all response letters to small letters, saving you from having to type every Match word in both capital letters and small letters. You could also use the Compute instruction's string-editing feature to make sure the student's name is always displayed with the first letter capitalized.





More important confusions might arise over the way SuperPILOT displays numbers. Numbers from .1 through 999999, from -.1 through -999999, and 0 are shown normally, after rounding to six digits, if necessary. Thus, a number like 234.56789 is displayed as 234.568. But if the student types larger or smaller numbers, they are converted to scientific E-notation. If the student typed 123456789, for example, the number would be displayed later as 1.23457E08, which means 1.23457 times 10 to the eighth power. Students can also use scientific notation in typing their own numbers, if they wish.

If the student types more than sixteen digits in response to an Accept instruction, any part of the number beyond the sixteenth digit is ignored. For a fancier way to convert a student's response to a number, avoiding this sixteen-digit limitation, see the FLO function, in the Advanced Programming chapter.

If the student types a word with no numbers, or just presses the RETURN key without typing anything, the number stored in the student's numeric variable (named `d`) is not changed from the number previously stored in it. In this case, the same number typed the previous time is used again. To avoid this occurrence, you can use the Error Conditioner to detect such a no-number response and execute instructions that deal with that situation.



## Chapter 10

# Advanced Programming

214	Constants
214	Numeric Constants
214	String Constants
215	Variables
215	Simple Numeric Variables
217	Numeric Variable Arrays
220	String Variables
222	Substring Variables
224	Pseudo-Variables
225	System Variables
225	%A The Answer-Count
226	%B The Answer Buffer
226	%X The Horizontal Coordinate
226	%Y The Vertical Coordinate
226	%C The Column Coordinate
226	%R The Row Coordinate
226	%S The Spin Angle
227	%O The Game Port Control
227	%V The V: Parameter (Signed)
227	%W The V: Parameter (Unsigned)
227	Using System Variables
229	Functions
231	Transcendental Functions
232	Arithmetic Functions
234	String Functions
239	Input Functions
242	Operators
242	Arithmetic Operators
243	Relational Operators
245	Logical Operators
246	String Operator
246	Expressions
246	Operator Precedence
248	Limitations on Expressions

The following material requires some knowledge of mathematics and its use in programs. For a less detailed discussion of this material, read the chapters Computation Instructions and Hints for Beginners.

## Constants

---

### Numeric Constants

A numeric constant is simply a number. All numbers in Apple SuperPILOT are real numbers: they may be expressed as integers, decimal numbers, or in scientific E-type notation (see the examples below).

Examples of numeric constants:

13947	(Integer notation.)
43.9237	(Decimal notation.)
7.76E+12	(Scientific notation: this is 7.76 times 10 to the twelfth power, or 7,760,000,000,000.)

Example of a numeric constant in use:

c:n=332.786	(Store the number 332.786 in simple numeric variable n.)
-------------	---

SuperPILOT's real numbers are limited to these ranges:  $-3.40282E38$  through  $-1.1755E-38$ , 0, and  $1.1755E-38$  through  $3.40282E38$ . Instructions such as Compute and Accept issue an error message (in Author Mode) and raise the Error Flag (in both Author Mode and Lesson Mode) if they encounter numbers outside these limits.

No more than six digits are displayed for a number in integer or decimal notation. No more than six digits are displayed for the mantissa (the portion preceding the Exponent) of a number expressed in scientific E-notation. When a number is rounded, it is rounded to six significant digits BEFORE it is stored or used. The number is rounded up if the seventh significant digit is 5 or more and down if seventh digit is 4 or less, ignoring any digits beyond the seventh for rounding purposes.

### String Constants

A string constant is any sequence of characters, enclosed in double quotation marks. Only the characters between the quotation marks are considered to be the string, and not the quotation marks themselves. Some characters cannot be typed into a string directly, but you can use the string functions and operators in an expression to fabricate a string that includes almost any characters.

Examples of string constants:

"Apple"	(Character string in quotes.)
"29.73201"	(Any characters, in quotes.)

Example of string constant in use:

c:s\$="FAT CHANCE!"	(Store the string FAT CHANCE! in string variable s\$ .)
---------------------	--

## Variables

---

Variables are just places in the Apple where you can store numbers or strings of characters for use later in a lesson. Numeric variables are used for storing numbers, and string variables are used for storing strings of characters. Each variable is named by you. Once you store a number or string in that variable, you can use the variable's name instead of that number or string almost anywhere in your lesson. This saves you a lot of typing, and also lets you change that number or string throughout a lesson by simply changing what is stored in that variable.

Four types of variables can be used with SuperPILOT instructions. These are simple numeric variables, numeric variable arrays, string variables, and system variables. Each type of variable is discussed in this section.

Every numeric variable array and string variable must first appear in a Dimension instruction to reserve space for it in memory, before it can be used in Compute or in any other instruction.

Your lesson may create a maximum of 50 different names for its variables. This includes the total of all simple numeric variables, numeric arrays, and string variables, but does not include the individual elements of a numeric variable array or the system variables. If you have 50 variables in your lesson and you attempt to dimension a new string variable or create a new numeric variable, an error message is given in Author Mode and the Error Flag is raised in both Author Mode and Lesson Mode.

### Simple Numeric Variables

A simple numeric variable is a named place for storing a single number. The name of the simple numeric variable must be a single letter, or a single letter followed by an integer from 0 through 9.

When used in the object field of Type or Accept instructions, the name must be immediately preceded by a pound sign ( # ) and immediately followed by a space or by the end of the instruction line. When the Type instruction is executed, the first space after a simple numeric variable name is not displayed: the stored number is displayed instead of the pound sign, variable name, and following space.

A simple numeric variable is normally created by using a Compute or an Accept instruction to store some number in that variable. For example:

```
c:n=45                                (Create the simple numeric
                                        variable n and store the
                                        number 45 in it.)
a:#p                                    (Accept student's response;
                                        create the simple numeric
                                        variable p and store the
                                        first response number in it.)
```

If a simple numeric variable is created by using it for the first time in an expression to the right of the assignment indicator ( = ) in a Compute instruction, or in an Accept instruction that finds no number in the student's response, or in an execution-conditioning expression in any instruction, that variable is created and assigned a value of  $\emptyset$ .

A simple numeric variable does not need to be Dimensioned, since it stores only one number. However, Dimensioning a numeric variable array (see next section) may influence your use of simple numeric variables. There are 200 number-storage places available for the use of all the simple numeric variables and all the elements of all the numeric variable arrays created during a lesson. Dimensioning a large numeric variable array may reserve much of the 200 number-storage places, limiting the number of simple numeric variables your lesson can create. It is possible in such a situation to cause a "Too many strings or arrays" error message on creating a simple numeric variable. See the chapter Control Instructions for important information on how this memory space may be safely used.

**WARNING!** The simple numeric variable n is the SAME VARIABLE as the numeric variable array element n( $\emptyset$ ) or n( $\emptyset$ , $\emptyset$ ). Do not try to use a simple numeric variable and a numeric variable array that share the same name.

Examples of simple numeric variables:

```
n                                       (Name is a single letter.)
s9                                      (Name is a single letter
                                        followed by a single digit.)
```

Example of simple numeric variables in use:

```
c:n=5                                  (Store the number 5 in simple
                                        numeric variable n.)
c:q3=n+1                                (Get the number stored in
                                        simple numeric variable n,
                                        add 1 to it, and store the
                                        sum in simple numeric
                                        variable q3.)
```

t:What is #q3 minus 1 ?

(Display text, substituting the number stored in simple numeric variable q3 for #q3 and one following space.)

a:#r

(Accept student's response, and store first number (if any) in simple numeric variable r .)

## Numeric Variable Arrays

A numeric variable array is really a set of simple numeric variables sharing the same name. Each of the individual variables (called elements) in the array is distinguished from the others by a subscript number which appears in parentheses after the variable's name. It is sometimes useful to use a numeric variable array in your lesson, because the lesson can choose among the different elements of the array just by changing a number.

Numeric variable arrays may be one-dimensional (requiring a single number subscript) or two-dimensional (requiring a subscript consisting of two numbers separated by a comma). A one-dimensional numeric variable array may be used to store a list of numbers, such as the score on each problem in a lesson. A two-dimensional numeric variable array may be used to store a table of numbers, such as the closing stock prices for a dozen different stocks on each day of the week.

The name of the numeric variable array must be a single letter, or a single letter followed by an integer from 0 through 9. The name of an individual element in an array is the variable array's name followed by a subscript specifying that element.

The subscript for an element of a one-dimensional array is a single number in parentheses. The subscript for an element of a two-dimensional array is two numbers separated by a comma, with one set of parentheses around the entire subscript. A number in a subscript may be replaced by a simple numeric variable name, but not by an expression or by an element of a numeric variable array. This feature allows you to choose an array element by simply changing the subscript number which is stored in a simple numeric variable.

Examples of numeric variable array elements:

w6(23)

(The 24th element of the one-dimensional numeric variable array w6. Remember the first element is w(0) .)

q(n)

(The number stored in simple numeric variable n is used as the subscript denoting an element of one-dimensional numeric variable array q .)



a(2,5)

(The 6th element in the third row of the two-dimensional variable array a .)

WARNING! The first element of a numeric variable array, specified by subscript ( $\emptyset$ ) or ( $\emptyset, \emptyset$ ), is the SAME VARIABLE as the simple numeric variable by the same name. Thus n and n( $\emptyset$ ) and n( $\emptyset, \emptyset$ ) are the same variable.

Before a variable array can be used by Compute or other instructions, it must first appear in a Dimension instruction to reserve number-storage places for its elements. When the Dimension instruction is encountered in your lesson, it creates all the elements of the numeric variable array and stores a value of  $\emptyset$  in each. The Dimension instruction specifies the largest-subscript number for each of the array's dimensions. The minimum subscript number for each array dimension is  $\emptyset$ . The largest-subscript number is limited by the amount of number-storage space available. There are 200 number-storage places to be used by all the simple numeric variables and all the elements of all the numeric variable arrays in a lesson. A numeric array whose largest-subscript is Dimensioned as (9,9) has 100 elements, which is half of all the numeric variable memory space you can Dimension. See the chapter Computation Instructions for important information on how this space may be used.

The results of these instructions:

d:a(6)	(Reserve space for storing up to seven numbers in one-dimensional numeric variable array a .)
c:a(1)=32.34	(Store 32.34 in a(1) .)
c:a(2)=557	(Store 557 in a(2) .)
c:a(5)=8.921	(Store 8.921 in a(5) .)

might be pictured like this:

a	( $\emptyset$ )	(1)	(2)	(3)	(4)	(5)	(6)
	$\emptyset$	32.34	557	$\emptyset$	$\emptyset$	8.921	$\emptyset$

The results of these instructions:

d:b(2,3)	(Reserve space for storing up to 12 numbers in two-dimensional numeric variable array b .)
c:b( $\emptyset$ ,2)=43	(Store 43 in b( $\emptyset$ ,2) .)
c:b(1, $\emptyset$ )=7.8	(Store 7.8 in b(1, $\emptyset$ ) .)
c:b(2,3)=9219	(Store 9219 in b(2,3) .)



might be pictured like this:

b	( $\emptyset$ , $\emptyset$ )	( $\emptyset$ ,1)	( $\emptyset$ ,2)	( $\emptyset$ ,3)
	$\emptyset$	$\emptyset$	43	$\emptyset$
	(1, $\emptyset$ )	(1,1)	(1,2)	(1,3)
	7.8	$\emptyset$	$\emptyset$	$\emptyset$
	(2, $\emptyset$ )	(2,1)	(2,2)	(2,3)
	$\emptyset$	$\emptyset$	$\emptyset$	9219

Numeric variable array elements cannot appear in the object field of an Accept instruction. Accept must first store a number in a simple numeric variable; then Compute can store that number in an element of the numeric variable array. A numeric variable array element may appear in a Type instruction, however, as long as it is completely enclosed within parentheses, with the required pound sign immediately preceding it.

Example of a numeric variable array in use:

d:p(3 $\emptyset$ )	(Reserve space for 31 elements of numeric variable array p . The elements go from p( $\emptyset$ ) through p(3 $\emptyset$ ) .)
...	(Early part of lesson.)
*prob5	(Label for test problem 5.)
th:23 + 13 =	(Display this text, leaving the cursor after the equals sign.)
a:#g	(Accept student's response, and store first number in simple numeric variable g .)
t(g<>36):No, try again.	(If student's answer now stored in g is not equal to 36, display this text.)
jc:prob5	(If answer g is not 36 (same condition), Jump back to the label prob5 .)
c:p(5)=%a	(Right answer, so store Answer-Count %a in the array element denoted by subscript 5.)
*prob6	(Label for test problem 6.)

When the lesson above is finished, each element of numeric variable array p contains the Answer-Count for the corresponding test problem. The Answer-Count tells how many attempts the student made before giving the correct answer.

To display the scores stored in numeric variable array `p` , you might use instructions such as these:

<code>*score</code>	(Label for scores section.)
<code>t:Score on problem #n was #(p(n)) .</code>	(Display text, substituting the number stored in <code>n</code> for <code>#n</code> , and the number in <code>p(n)</code> for <code>#(p(n))</code> and one space.)
<code>c:n=n+1</code>	(Add 1 to the number in <code>n</code> , and store it back in <code>n</code> .)
<code>j(n&lt;31):score</code>	(If the number in <code>n</code> is less than 31, Jump back to score .)

## String Variables

A string variable is a memory storage place where you can store a single string of characters for use in your lesson. It is often helpful to use a string variable's name in your lesson in place of the string that is stored in that variable. This lets you change that word or phrase throughout the lesson simply by changing the string that is stored in that named variable.

The name of the string variable must be a single letter, or a letter and a single integer from 0 through 9, and must be immediately followed by a dollar sign. You may use a numeric variable and a string variable with the same basic name in a lesson; the string variable is always distinguishable by the dollar sign after its name.

Examples of string variables:

<code>a\$</code>	(Name is a single letter, followed by a dollar sign.)
<code>s8\$</code>	(Name is a single letter followed by a single digit, followed by a dollar sign.)

When used in the object field of Type or Accept instructions, the string variable's name must be immediately preceded by a dollar sign and immediately followed by a space or by the end of the instruction line. Thus, the string-variable name `s$` appears as `$$` . When the Type instruction is executed, it replaces the preceding dollar sign, the string-variable name, and the first space after the name with the string of characters currently stored in that string variable. When the Accept instruction is used to store a student's response in a string variable, the response receives full Accept editing unless all editing has been suspended by Accept's `eXact` modifier ( `ax:$$` ).

Before a string variable can be used by Compute, Type, or any other instruction, it must first appear in a Dimension instruction to reserve space for its characters in memory. The Dimension instruction specifies the maximum number of characters that can be stored in the variable. This maximum must be a number from 1 through 255 characters. The

Dimension instruction creates the string variable, but it does not store any characters in the variable. Until you use an Accept or Compute instruction to store a string of characters in a string variable, it contains no characters at all.

There are a total of 16000 character-storage locations for all the characters stored by all the string variables in a lesson. When you Dimension a string, the specified maximum number of character locations are made permanently unavailable for use by any other string variable. See the chapter Computation Instructions for important information on how this memory space can be used.

A string is stored in a string variable from left to right (the first or leftmost character of the string is stored in the string variable's first position). A length-of-string indicator is stored with the string to tell SuperPILOT how long the stored string is. This length is reset each time a new string is stored in the variable. If the string being stored is longer than the maximum length Dimensioned for the string variable, the extra (rightmost) characters of the string are discarded.

The result of these instructions:

```
d:s$(8)                (Reserve space for storing
                        strings up to 8 characters
                        long in string variable s$ .)

c:s$="Apple"           (Store the string Apple in
                        string variable s$ .)
```

might be pictured like this:

```
s$  

|   |   |   |   |   |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|
| A | p | p | l | e |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|


      (Current length of s$ is 5 characters.)
```

Example of string variables in use:

```
d:a$(20);b$(50)        (Reserve space for storing
                        strings up to 20 characters
                        long in string variable a$ ,
                        and up to 50 characters long
                        in string variable b$.)

c:a$="apples"           (Store the string apples in
                        string variable a$ .)

c:a$=a$!!" and pears"  (Make a copy of the string in
                        string variable a$ ; join
                        the string and pears to it.
                        Store the resulting string
                        back into a$ , replacing
                        a$'s previous contents.)

t:What are your        (Display these lines of text.)
: favorite fruits?
```

a:\$b\$

(Accept student's response and store in string variable b\$ .)

t:Yes, \$b\$ are nice.

(Display text, replacing \$b\$ and the first following space with the string now in string variable b\$ .)

t:Have you ever tried \$a\$ ?

(Display text, substituting the contents of a\$ where \$a\$ and one space appear.)

In the example above, if the student replies that "mangos and papayas" are favorite fruits, the Apple II responds:

Yes, mangos and papayas are nice.

Have you ever tried apples and pears?

And if the student replies, "I think fruit is ghastly!" , the Apple II responds:

Yes, I think fruit is ghastly! are nice.

Have you ever tried apples and pears?

## Substring Variables

A string variable may sometimes be treated like a one-dimensional array of characters. When used this way, you can think of the string variable as a family of character variables, each storing one character.

The result of these instructions:

d:s\$(8)

(Reserve space for storing strings up to 8 characters long in string variable s\$ .)

c:s\$="Apple"

(Store the string Apple in string variable s\$ .)

might be pictured like this:

s\$ (1) (2) (3) (4) (5)

A	p	p	l	e					
---	---	---	---	---	--	--	--	--	--

(Current length of s\$ is five characters.)

The complete set of these character variables is the string variable itself, of course, and smaller sets can form substrings of the string variable. We often refer to these smaller parts as variables in their own right, as in the phrase substring variable, but it is important to note that the Apple does not look at them in this way. The string variable itself is the only one for which memory space is reserved and used, and if we remove or change the value of a string variable, its substrings will also be removed or changed. For convenience, character

variables and substring variables often both go under the name substring variables.

Individual character variables within the string variable use the string variable's name, immediately followed by a subscript consisting of a single number in parentheses. The subscript number corresponds to a character position in the stored string, a number from 1 (denoting the first character) through the number of characters currently stored in the string variable. Thus, the character variable storing the first character in string variable `s$` is `s$(1)` :

`s$(1)`      (1)

A
---

A substring variable uses the string variable's name, immediately followed by a subscript consisting of two numbers separated by a comma, with one set of parentheses around the whole two-number subscript. The first number gives the string position of the substring's first character variable, and the second subscript gives the number of character variables in the substring variable. The value of either subscript number can be from 1 through the number of characters currently stored in the main string variable, but the sum of the subscript numbers must also not exceed by more than 1 the number of characters currently stored. Thus, the substring variable storing three characters, starting with the second character in string variable `s$`, is `s$(2,3)`. This substring variable also includes character variables `s$(2)` through `s$(4)` :

`s$(2,3)`      (2)      (3)      (4)

p	p	l
---	---	---

Any subscript number can be replaced by a simple numeric variable, but not by a numeric variable array element or by an expression.

Examples of character variables and substring variables:

`r$(2)`

(Specifies the second character of the string stored in string variable `r$`.)

`x$(3,12)`

(Specifies a substring, twelve characters long, beginning with the third character stored in string variable `x$`.)

Example of substring variables in use:

`d:q$(20);z3$(20);s5$(20)`

(Reserve space for storing strings up to 20 characters long in each of three string variables: `q$`, `z3$`, and `s5$`.)

<pre>c:q\$="LABORATORY"</pre>	<pre>(Store the string LABORATORY in string variable q\$ .)</pre>
<pre>c:z3\$=q\$(1,5)</pre>	<pre>(Make a copy of the 5-character substring LABOR starting at character 1 of the string in q\$ , and store it in string variable z3\$ .)</pre>
<pre>c:s5\$=q\$(4,7)</pre>	<pre>(Make a copy of the 7-character substring ORATORY starting at character 4 of the string in q\$ , and store it in string variable s5\$ .)</pre>
<pre>t:This is a \$q\$ . t:Let's have more \$z3\$ t:and less \$s5\$ .</pre>	<pre>(Display this text, replacing \$q\$ , \$z3\$ , and \$s5\$ (and the one space after each) with the strings now in string variables q\$ , z3\$ , and s5\$ .)</pre>

The above example displays the following message on the screen:

```
This is a LABORATORY.
Let's have more LABOR
and less ORATORY.
```

## Pseudo-Variables

A pseudo-variable is a character variable or substring variable (see the previous section) that appears on the lefthand side of the assignment indicator ( = ) in a Compute statement. This allows you to change a single character or group of characters in a stored string, without affecting the other characters in that string.

If the new string of characters being stored in the substring variable is longer than the substring variable, the extra (rightmost) characters of the new string are discarded. If the new string is shorter than the substring variable, spaces are added to the right end of the new string until it is as long as the substring variable.

Note: You cannot CREATE strings or characters by the use of a pseudo-variable; you can only CHANGE characters that are already currently stored in the string of which the pseudo-variable is a portion. If a pseudo-variable includes any character positions that do not currently have characters stored in them, an error condition results.

For examples of pseudo-variables, see the previous section's examples of substring variables and character variables.

Example of pseudo-variables in use:

d:p0\$(20)	(Reserve space for storing strings up to 20 characters long in string variable p0\$ .)
c:p0\$="COMPUTER"	(Store the string COMPUTER in string variable P0\$ .)
c:p0\$(3,4)="workhorse"	(Replace 4-character substring starting at character 3 of string variable p0\$ with string constant workhorse , after discarding all but the 4 leftmost characters.)
t:\$p0\$	(Display the string now stored in string variable p0\$ .)

The above example when run, displays the following word:

COworkER

If the third instruction had been

c:p0\$(3,4)="rk"

the example would display this, instead:

COrk ER

## System Variables

In addition to all the variables that you can create yourself (see the previous sections), SuperPILOT maintains ten of its own variables for internal use and for use in your instructions. These ten variables are called system variables. System variables do not have the usual, author-created variable names, and they cannot be used in all of the same ways a normal variable can be used. For example, a system variable cannot appear in the object of an Accept instruction or in the object of a File Input instruction. Here are the ten system variables:

### %A The Answer-Count

The numeric value stored in the %a system variable is the Answer-Count: the number of successive times the most recent Accept instruction is executed without a different Accept instruction being executed. The Answer-Count records how many times in a row a student responds to the same question. Before your lesson executes its first Accept instruction, the value stored in %a is 0. Thereafter, the value the system stores in %a is always an integer that is 1 or greater. The value of %a for the first 99 responses to any Accept instruction can be automatically tested as a condition for executing other instructions by using the Answer-Count conditioner.

## **%B The Answer Buffer**

Before your lesson executes its first Accept instruction, there are no characters stored in the %b system variable. Thereafter, the string stored in %b by SuperPILOT is the student's most recent typed response, as received and edited by an Accept instruction. The next Match instruction searches the contents of the Answer Buffer for any occurrence of the author-supplied Match text. The Accept instruction stores up to 80 characters (before editing) in %b .

## **%X The Horizontal Coordinate**

When your lesson begins, the numeric value stored in the %x system variable is 0. Thereafter, the number stored in %x is the horizontal coordinate of the graphics cursor. The value of %x changes when a Graphics instruction or an Accept Point instruction moves the graphics cursor horizontally.

## **%Y The Vertical Coordinate**

When your lesson begins, the numeric value stored in the %y system variable is 0. Thereafter, the number stored in %y is the vertical coordinate of the graphics cursor. The value of %y changes when a Graphics instruction or an Accept Point instruction moves the graphics cursor vertically.

## **%C The Column Coordinate**

Before your lesson executes its first instruction, the numeric value stored in the %c system variable is 0. Thereafter, the number stored in the %c system variable is the vertical or column coordinate of the text screen where the text cursor is located, relative to the text viewport. The range of %c is from 0 (left edge of text screen) to 39 (right edge of text screen).

## **%R The Row Coordinate**

Before your lesson executes its first instruction, the numeric value stored in the %r system variable is 0. Thereafter, the number stored in the %r system variable is the horizontal or row coordinate of the text screen where the text cursor is located, relative to the text viewport. The range of %r is from 0 (top edge of text screen) to 23 (bottom edge of text screen).

## **%S The Spin Angle**

The value of %s is 0 until your lesson executes a G: instruction that changes the direction used for Draw and other graphics commands. Instructions in the form g:an or g:sn, where n represents the number of degrees in the specified Angle or Spin, reset the Spin Angle system variable %s. For a g:an instruction, the new value for %s is n. For a g:sn instruction, the new value for %s is n plus the previous value of %s.



## **%O The Game Port Control**

The %o system variable is used to access peripheral devices attached to the Apple II's game ports. When you use the %o system variable in a lesson instruction, it must be followed by a number, in parentheses, specifying the appropriate game port: %o(n), where n is an integer from 0-3. The instruction c:%o(n)=expr activates the peripheral device attached to game port n if expr<>0, and it deactivates it if expr=0.

## **%V The V: Parameter (Signed)**

The %v system variable is used by the V: instruction to communicate with a peripheral device such as a videotape or videodisk. Before you give it a value in a V: instruction, %v has a value of 0. It can be assigned a value in the range -32767 through 32767.

## **%W The V: Parameter (Unsigned)**

The %w system variable is identical in use and function to the %v system variable, except that %w cannot have a negative value. It can be assigned a value in the range 0 through 65535.

## **Using System Variables**

You must not use a system variable in the object field of an Accept instruction or a File Input instruction. A system variable cannot appear with subscripts (except for %o), so you must not specify substrings of the Answer Buffer %b. The value stored in a system variable can, of course, be assigned to an appropriate author-created variable, using a Compute instruction, and that assigned value can then be used in all the usual ways.

A system variable can be used in expressions, but the contents of a system variable are usually changed only by the SuperPILOT system. This means that you will rarely place a system variable on the lefthand side of an assignment operator (=) in a Compute instruction. Two noteworthy exceptions are the Answer-Count and Answer Buffer system variables.

It is possible to use Compute to store a string in system variable %b. Your lesson might edit a response, for instance, and then store the edited response back into %b. Or you might store a default answer in %b and then allow the student to use the default answer by simply pressing the right-arrow key until the answer is completely revealed (see the section on the Accept instruction in the chapter Response Instructions for details on this use of the right-arrow key).

It is also possible for Compute to store a number in system variable %a. An Escape (sysx) subroutine, for example, might reduce the Answer-Count by 1 so the Escape response itself would not be counted as an attempted answer. Or you can store a value in %a so that the Answer-Count conditioner can be used in place of an expression conditioner:

```

t:Type a number from 1-26, and I'll (Display this text.)
: tell you what letter of the
: alphabet that number represents.
a:#n (Accept student's response;
      store it in simple numeric
      variable n .)

c:%a=n (Store the value of n in
        the Answer-Count system
        variable %a .)

t1:A (If the Answer-Count is 1,
t2:B display an A ; if it's 2,
t3:C display a B ; etc.)
...

t26:Z

```

This example does not contain the error-handling instructions you will normally include when accepting a numeric response, but it does show how the Answer-Count system variable can be used for purposes other than counting how many times an Accept instruction is attempted. If instead of Answer-Count conditioners we use expression conditioners for each of the 26 Type instructions, as in `t(n=1):A`, execution time will be noticeably longer. So there is an advantage of speed if you use this method.

System variables cannot be used directly in a Type instruction to place the stored value into the displayed text. If you want a system variable's value to appear in a Type instruction, you must treat that value as an expression. For example:

```

t:Well, #(%b) , where are you from? (Display text, substituting
                                     the current contents of the
                                     Answer Buffer for #(%b)
                                     and the following space.)

```

or

```

t:That was guess number #(%a) . (Display text, substituting
                                  the current value of the
                                  Answer-Count for #(%a)
                                  and the following space.)

```

Example of system variable %a in use:

```

c:r=rnd(100) (Pick a random number from 0
              through 99, and store it in
              simple numeric variable r .)

*start (Label for this section.)
t:Guess my number, 0 through 99. (Display this text.)
a:#g (Accept student's response
      and store it in g .)

te:Type a number, please. (If Error Flag is up, no number
                           was typed, so display text.)

```

je:@a	(If Error Flag up, Jump back to last a: .)
j(g=r):right	(If guess g is equal to right answer r , Jump to right .)
t(g>r):No, that's too large.	(If guess g is greater than the answer r , show this text.)
t(g<r):No, that's too small.	(If guess g is less than right answer r , show this text.)
t(%a<20):Try again.	(If the number stored in the Answer-Count is less than 20, display this text.)
jc:start	(If Answer-Count less than 20, Jump back to label start .)
t:Too bad. My number was #r .	(This is 20th wrong answer, so give up and display this text, substituting right answer r for #r and one space.)
j:next	(Jump to next .)
*right	(Label for right section.)
c:s=21-%a	(Right answer, so Compute score, based on number of guesses. Store score in simple numeric variable s .)
t:You guessed it in #(%a) tries, : so you get #s points.	(Display text, substituting the current value of the Answer-Count for #(%a) and the next space, and substituting the current value of s for #s and the next space.)
*next	(Label for next section.)

## Functions

---

SuperPILOT will do a variety of special operations for you, each selected by typing a three-letter name. These special operations are called functions. SuperPILOT functions can calculate the cosine of a given angle, generate a random number, convert a number into a string, find a number within a string, note the setting of a game control, and perform many other useful tasks.

Each function needs certain items of information in order to do its operation for you. These items are called the function's arguments. From one to three arguments may follow the function's name, the entire set of arguments enclosed in parentheses. Multiple arguments are separated by commas.

For example, to calculate the square root of 1369, you could use the SQR function in a Compute instruction such as this:

c:x=sqr(1369)

(Calculate the square root of 1369; store it in simple numeric variable x.)

You can also use this function in place of a number in many other expressions and instructions. Here are some sample instructions:

c:x=(5\*12/sqr(1369))+16

(Evaluate expression on the right of the equals sign; store the result in x.)

g(x>sqr(1369)):dx,y

(If x is greater than the square root of 1369, Draw a line to x=X,y=Y.)

j(y<(5\*12/sqr(1369))+16):part2

(If y is less than result of remaining expression, Jump to label part2.)

t(sqr(x)<.75):Try a larger angle.

(If the square root of x is less than .75, display text.)

s(sqr(x)):30,100

(If the square root of x is non-zero, play this note.)

Functions have the highest priority among operations. For example,  $\text{sqr}(16)*2$  is executed as  $(\text{sqr}(16))*2$ . See the Expressions section for more details about precedence among operations.

Each function requires specific kinds of arguments. A numeric argument (shown in the following descriptions as x) may be a number, a simple numeric variable, an element of a numeric variable array, or an expression that yields a number. A string argument (shown in the following descriptions as x\$) may be a quoted string, a string variable, a substring variable, or an expression that yields a string. Expressions are discussed in the section following the descriptions of the functions.

When a function completes its operation, either a number or a string is the result. The function is said to return this number or string. For instance, the function  $\text{sin}(x)$  returns a number that is the sine of the angle x.

In the following discussion, a valid real number is any whole number, decimal number, or number expressed in scientific notation, as long as the value is in the approximate range from  $3.4\text{E}38$  to  $1.2\text{E}-38$ , or is zero, or is in the approximate range from  $-1.2\text{E}-38$  to  $-3.4\text{E}38$ . These are the approximate limits on numbers that can be Accepted into a numeric variable without causing an Arithmetic Error. Some functions or calculations may occasionally yield valid numbers slightly exceeding those limits.

Most specific limits are given as the numbers will be stored or used after rounding to six significant digits. For example, the Type instruction

```
t(x=1000000):That's correct.
```

displays its object text if  $x=99999.95$ , because the value of  $x$  is rounded to  $100000$  before the instruction is evaluated.

If the value of the argument given is outside the function's allowable range, an error message is displayed in Author Mode and the Error Flag is raised in both Author Mode and Lesson Mode. The rest of the instruction is not executed. If the value of the function was to be assigned to a variable, that variable remains unchanged. (This behavior is different from that of Apple PILOT; see Appendix E for details.)

## Transcendental Functions

There are two ways you can define an angle in a SuperPILOT instruction: by degrees or by radians. For most lessons, you will probably use degrees, as in `g:an` or `g:sn` instructions, where  $n$  is the number of degrees. However, in situations that call for radian measurement, you can use the following three transcendental functions to define your angles. For conversion purposes, note that an angle of  $\pi$  radians is equal to an angle of  $180$  degrees. If you want to convert an angle of  $d$  degrees into its equivalent radian measure  $r$ , use this conversion:

$$r=d*(3.14159/180) \quad \text{or} \quad r=d*(.017453)$$

These are only a few of the standard trigonometric functions. Consult any book on trigonometry for the ways to calculate other quantities using these functions. For example, you can find the tangent of an angle by dividing the angle's sine by its cosine.

**SIN(X)** Returns a number in the range from  $-1$  through  $+1$  that is the sine of angle  $x$ . The single numeric argument  $x$  is the size of an angle, expressed in radians. The value of  $x$  can be any valid real number in the range  $-102942$  through  $102942$ . Example:

```
c:s=sin(0.317) (Compute the sine of an angle
                of 0.317 radians, and store
                in numeric variable s.)
```

**COS(X)** Returns a number in the range from  $-1$  through  $+1$  that is the cosine of the angle  $x$ . The single numeric argument  $x$  is the size of an angle, expressed in radians. The value of  $x$  can be any valid real number in the range  $-102940$  through  $102940$ . Example:

```
t(cos(a)=.249):Good shot! (If cosine of a radians is
                           .249, display this text.)
```

ATN(X) Returns a number in the range  $-.785398$  through  $1.5708$  that is the radian measure of the angle whose tangent is  $x$ . This number is also called the arctangent of  $x$ . The single numeric argument  $x$  is the tangent of an angle. The value of  $x$  can be any valid real number.

A tangent does not uniquely determine a corresponding angle (in general, any two angles that differ by  $\pi$  radians have the same tangent). The  $\text{atn}(x)$  function usually returns the smallest positive angle having a tangent of  $x$ . However, if  $x$  is in the range from  $0$  through  $-1$ ,  $\text{atn}(x)$  returns the smallest negative angle whose tangent is  $x$ .  $X$  must be in the range  $-5.32511\text{E}18$  through  $5.32511\text{E}18$ . Example:

`c:a=atn(t)` (Compute the angle, in radians, whose tangent is  $t$ ; store in numeric variable  $a$ .)

## Arithmetic Functions

SGN(X) Returns  $-1$  if  $x$  is less than zero, returns  $0$  if  $x$  is zero, or returns  $1$  if  $x$  is greater than zero. The value of  $x$  can be any valid real number. Example:

`c:b=a*sgn(r)` (If  $r$  is greater than  $0$  then  $b=a$ ; if  $r$  equals  $0$  then  $b=0$ ; if  $r$  is less than  $0$  then  $b=-a$ .)

ABS(X) Returns a positive number that is the absolute value of  $x$ . If  $x$  is zero or positive, the absolute value of  $x$  is  $x$ . If  $x$  is negative, the absolute value of  $x$  is  $-x$ . The value of  $x$  can be any valid real number. Example:

`u(abs(a)>360):error` (If  $a$  is greater than  $360$  or less than  $-360$ , Use the subroutine called `error`.)

FIX(X) Returns an integer in the range  $32767$  through  $-32767$ , found by truncating  $x$ . Truncation is performed by discarding any part of  $x$  to the right of the decimal point, without rounding. Thus,  $\text{fix}(5.933)$  returns the value  $5$ ,  $\text{fix}(-3.957)$  returns the value  $-3$ . The value of  $x$  can be any valid real number in the range  $-32767.9$  through  $32767.9$ . Remember that the value of  $x$  is rounded to six significant digits BEFORE being used. Example:

`c:i=fix(n)` (Truncate the value of  $n$ ; store it in variable  $i$ .)

INT(X)

Returns an integer in the range from 32767 through -32767 that is the largest integer less than  $x$  or equal to  $x$ . Thus,  $\text{int}(7.946)$  returns the value 7, and  $\text{int}(-4.12)$  returns the value -5. The value of  $x$  can be any valid real number in the range -32766.9 through 32767.9.  
Example:

`c:a=int(1)`

(Find the largest integer less than or equal to 1; store in numeric variable a.)

RND(X)

Returns a random number in a range determined by the value of  $x$ . The value of  $x$  can be any valid real number less than 32768.0. For any  $x$  less than 1,  $\text{rnd}(x)$  returns a random real number in the range 0 through 0.999969. For any  $x$  equal to or greater than 1,  $\text{rnd}(x)$  returns an integer from 0 through  $\text{FIX}(X)-1$ .  
Note: Therefore when  $x$  is less than 2 and greater than or equal to 1,  $\text{rnd}(x)$  always returns a value of 0.  
Example:

`c:x=rnd(6)+1`

(Pick a random integer from 0 through 5, add 1 to it; store it in variable x.)

SQR(X)

Returns the square root of  $x$  (the number that, when multiplied by itself, yields  $x$ ). The value of  $x$  can be 0 or a valid real number greater than 0. This limits the values returned by  $\text{sqr}(x)$  from 0 through about  $2E19$ .  
Example (finds the length of the hypotenuse  $h$  for a right triangle with sides of lengths  $a$  and  $b$ ):

`c:h=sqr((a*a)+(b*b))`

(Compute  $a*a$  and  $b*b$ , add them together, take the square root of that sum; store it in variable h.)

LOG(X)

Returns a number between about 39 and -39, which is the base 10 logarithm of  $x$  (the power to which 10 must be raised to yield  $x$ ). The value of  $x$  can be any valid real number greater than 0. Example:

`c:i=log(v/r)`

(Divide the value of  $v$  by the value of  $r$ , compute logarithm of the quotient; store in variable i.)

LN(X) Returns a number between about 89 and -88, which is the natural logarithm of  $x$  (the power to which  $e$  must be raised to yield  $x$ ).  $e$  is approximately 2.71828. The value of  $x$  can be any valid real number greater than 0. Example:

c:f=2.3+ln(14.29) (Compute the natural log of 14.29; add 2.3; store sum in numeric variable f.)

EXP(X) Returns a number in the range 0 to about 6.8E38, which is  $e$  raised to the power  $x$ .  $e$  is approximately 2.71828. The value of  $x$  can be any valid real number in the range -88.0295 through 89.4159. Example:

c:n=exp(y)\*5 (Raise  $e$  to the power  $y$ ; multiply by 5; store in numeric variable n.)

## String Functions

ASC(X\$) Returns a number in the range from 0 through 255, which is the decimal ASCII code for the first character of  $x$ \$. Returns 0 if the length of  $x$ \$ is 0. Characters usually typed from the keyboard have decimal ASCII code numbers ranging from 32 through 122.

See the appendix ASCII Character Codes for the code number that corresponds to each character. Example:

j(asc(%b)=78):nope (If the decimal ASCII code of the first character in the Answer Buffer %b is 78, the student has typed a capital N, so Jump to the section labelled nope.)

CHR(X) Returns the ASCII character whose decimal ASCII code is  $\text{fix}(x)$ . See the appendix ASCII Character Codes for the character that corresponds to each code number. The value of  $x$  can be any valid real number in the range -32767.9 through 32767.9, but all of the available characters can be obtained using values of  $x$  in the range 0 through 159. For values of  $x$  in the range 160 through 255,  $\text{chr}(x)$  returns characters identical to those that are returned by  $\text{chr}(x-128)$ , except that they are stored with the higher ASCII code. For example,  $\text{chr}(168)$  returns the character whose ASCII code is  $168-128=40$ .

If  $x$  is greater than 255, the remainder left after dividing  $\text{fix}(x)$  by 256 is used in place of  $x$ , so  $\text{chr}(256)$  returns the character whose ASCII code is 0,  $\text{chr}(257)$  returns the character whose ASCII code is 1, and



so on. If  $x$  is less than 0, the remainder left after dividing  $\text{fix}(x)$  by  $-256$  is subtracted from 256, and that number is used in place of  $x$ , so  $\text{chr}(-1)$  returns the character whose ASCII code is 255,  $\text{chr}(-2)$  returns the character whose ASCII code is 254, and so on.

The characters corresponding to ASCII codes less than 32, when sent to the screen, do not appear as visible characters. Instead, many of them act as screen-handling instructions to clear the screen, erase a line, do a line feed or carriage return, and so forth. Example:

<code>d:s\$(10)</code>	(Reserve space to store up to 10 characters in string variable <code>s\$</code> .)
<code>c:s\$=chr(8)</code>	(Store the character whose ASCII code is 8 (a backspace character) in string variable <code>s\$</code> .)
<code>c:s\$=s\$!!s\$!!s\$!!s\$!!s\$</code>	(Concatenate five <code>s\$</code> strings and store back in string variable <code>s\$</code> . <code>s\$</code> now contains five backspace characters.)
<code>t:The U.S.A. consists</code>	(Display text.)
<code>th:of states.\$s\$ \$s\$</code>	(Display text, then backspace five times for each <code>\$s\$</code> and leave cursor there, in the blank area between the words "of" and "states.")
<code>a:#n</code>	(Accept student's response; assign number to simple numeric variable <code>n</code> .)
<code>t(n=50):Very good.</code>	(If <code>n</code> is 50, display text.)

#### FLO(X\$)

Returns any valid real number that is the value of the first number found in `x$`. Starting at the beginning of the string, `flo(x$)` reads each character until it finds the first numeric digit. It then examines the preceding character to see if it is a decimal point or minus sign. If a decimal point precedes the numeric digit, the character before that is examined to see if it is a minus sign. `FLO(x$)` then forms a number using all the characters from its current position up to the first subsequent character that cannot be interpreted as part of that number. Thus, `flo("My 3rd child is 14 years old.")` returns the value 3.

If no numeric digit is found in `x$`, `flo(x$)` causes an error message in Author Mode and the Error Flag is raised in both Author Mode and Lesson Mode.

Periods and hyphens in the string may be interpreted as parts of a number if they occur in the character just

before the number. Flo("This is no.4 on the list.") returns the number .4. Flo("too many--12, in fact--are missing") returns the number -12. A comma is not recognized as part of a number, so that flo("I've got \$23,244,056 in the bank.") returns the value 23. The Compute instruction's string-editing option can be used to remove commas from a string.

Since the string is scanned left to right, collecting and examining digits along the way, any number of leading zeros cause no trouble. However, if the number collected at any point along the string exceeds the limits for real numbers, the instruction is terminated, with an error message in Author Mode and with the Error Flag raised in both Author Mode and Lesson Mode, even though a later exponent, say, would bring the number back to within limits. This restricts you to about 38 or 39 digits preceding the decimal point, starting at the first non-zero digit, and restricts you to about 37 or 38 zeros before reaching the first non-zero digit, starting at the decimal point. Thus a 1 followed by 39 zeros, and a decimal point followed by 39 zeros, both cause an error regardless of what comes next.

At most two minus signs can appear in a number, one as the first character, and another after an E as the sign of an exponent. Flo(x\$) tolerates any number of decimal points in a number, ignoring all except the rightmost decimal point. Flo("1..2.3.4") returns the value 123.4. A decimal point after the E in an exponent ends the number. Only the first two digits following an E are used in the exponent; any subsequent digits are ignored. Thus, flo("1e2345") returns the value 1E23. Either capital E or lowercase e is interpreted as a numeric exponent character. Example:

```
c:n=flo(s$!!"e12")      (Concatenate the string "e12"
                          to the end of the string in
                          string variable s$ ; find
                          first number in the new
                          string; store the number in
                          numeric variable n .)
```

```
t(flo(%b)<27):Too small. (If the first number in the
                          Answer Buffer %b is less
                          than 27, display text.)
```

STR(X) Returns a string, from 1-12 characters in length, which represents the number x as it would be displayed on the screen. The value of x can be any valid real number. The value of x is always changed to standard SuperPILOT format before str(x) converts the number into a string, even if x is an actual number (not a numeric variable). Thus, str(123456789) returns the string 1.23457E08.

You can use the STR function any time you want to place a number directly into a string. In particular, a number must always be converted to a string before you can store it in a diskette file other than system.log . Example:

```
c:s3$=str(%a)           (Convert the Answer-Count %a
                        to a string, and store in
                        string variable s3$ .)
fo:5,s3$                (Store the contents of s3$
                        in record 5 of the file
                        that is currently open.)
```

Conversely, in order to do any arithmetic operation on a number in a string, you must first convert that part of the string to a number, using the FLO function. Example:

```
fi:5,r4$                (Retrieve the contents of
                        record 5 from the diskette
                        file that is currently open;
                        store in r4$ .)
c:n=flo(r4$)*12         (Find first number in r4$ ,
                        multiply by 12; store in
                        simple numeric variable n .)
c:t6$="Score:!!str(n)  (Convert number n to a
                        string, concatenate to
                        "Score:", and store in
                        string variable t6$ .)
fo:5,t6$                (Store the contents of t6$
                        in record 5 of the file
                        that is currently open.)
```

The two Compute instructions above could be combined:

```
c:t6$="Score:!!str(flo(r4$)*12)
```

INS(N,T\$,P\$) Starting at character number fix(n), looks in target string t\$ for the first occurrence of a sequence of characters that matches pattern string p\$ . Returns the value 0 if no such occurrence is found, or returns an integer from 1 through len(t\$), indicating the position of the character within t\$ where the sequence begins.

The integer returned by the INS function refers to character positions within t\$ numbered from the first character in the string, which is position number 1, and NOT from position n . The value of n merely indicates the character position within t\$ from which to begin the search for p\$ . If n is 1, the search begins at the first character of t\$ . If n is 14, the search begins at the 14th character, ignoring the first thirteen characters of t\$ . In either case, if the INS function finds the pattern string p\$ starting at the twenty-third character of t\$ , it returns the value 23. The search always proceeds toward the right through t\$ .

The value of `n` can be any valid real number less than 32768.0 . If `n` is less than 1, the value 1 is used instead. If `n` specifies a starting position beyond the last character position in `t$` , the INS function returns the value 0 . `N` may be a number, an expression, a simple numeric variable, or an element of a numeric variable array.

Either pattern string `p$` or target string `t$` may be an expression, a quoted string, string variable, or string variable substring. If pattern string `p$` is longer than the target string `t$` , the INS function returns the value 0 .

You can use the INS function to determine the existence of a given word within any string. Since SuperPILOT treats the value 0 as False, and any non-zero value as True, the number returned by INS can be used as a simple true/false expression modifying other commands. Example:

```
t(ins(1,%b,"TIGER")):Right. (If the word TIGER is
                             found in Answer Buffer
                             %b , the INS function
                             returns a non-zero number,
                             so display the text.)
```

That use of INS is similar to an unmodified Match instruction, but INS gives you extra information about where the pattern is found in the response. The number returned by INS can also be used to change a certain word found in the target string. Example:

```
d:s$(35);r$(5)           (Reserve space in memory to
                          store up to 35 characters
                          in string variable s$ and
                          up to 5 characters in
                          string variable r$ .)

c:s$="A hungry wolf in   (Store in s$ .)
:sheep's clothing."
c:r$="wolf"              (Store in r$ .)
c:p=ins(1,s$,r$)        (Find character position where
                          pattern string wolf starts
                          in target string s$ ; store
                          that position in p .)

c:s$(p,4)="lamb"        (Replace four characters of
                          s$ starting at position p
                          with the four characters
                          lamb .)

t:$s$                   (Display the string s$ .)
```

The INS function will return the value 10 , the position in `s$` where the first occurrence of the sequence of characters `wolf` started. We then use that number 10 as the starting position of a substring in `s$` to be

replaced by the string `lamb` . The new word had to be the same length as the word it was replacing, to avoid changing any other characters in `s$` .

LEN(X\$)

Returns an integer that is the number of characters currently stored in `x$` . This number may be as small as  $\emptyset$ , for the null string `""` , and may be as large as the maximum size specified for `x$` when space was reserved for that string in a Dimension instruction (cannot exceed 255). The LEN function helps you deal with strings or string-parts of unknown lengths, or whose lengths change from one running of a lesson to the next. Examples:

`t(len(%b)>7):`Try a shorter (If the number of characters in Answer Buffer `%b` is greater than 7, display text.)  
: word.

`c:d5=len(t$)-4` (Subtract 4 from the number of characters now in string variable `t$` ; store the result in variable `d5` .)

`t(ins(d5,t$,"Zaarb")):`Good. (If the word `Zaarb` is found in the last 5 characters of string `t$` , display text.)

## Input Functions

PDL(X)

Returns an integer from  $\emptyset$  through 255 that represents the position of game control number `x` . The value of `x` should be a number from  $\emptyset-3$ , but it can be any valid real number in the range 32767 to -32767. The value of `x` is rounded (not truncated) to the nearest integer value. If `x` is greater than 3, the value 3 is used instead. If `x` is less than  $\emptyset$ , the value  $\emptyset$  is used instead. If no game control number `x` is connected to your Apple II, `pd1(x)` always returns the maximum value 255.

The game controls are sometimes called "paddles" (hence the name of the PDL function) because they are used to control the position of game paddles in certain computer games. Any  $\emptyset-15\emptyset\text{K}$  ohm variable resistance can be used as a "paddle," if properly connected to the Apple II. See the Apple II Reference Manual for electrical details. If your Apple II is equipped with the usual single pair of game controls, those controls are numbered  $\emptyset$  and 1.

BTN(X)

Returns the number 1 if the button on game control number `x` is being pressed, or returns  $\emptyset$  if that button is not being pressed. The value of `x` should be a number from  $\emptyset-2$ , but it can be any valid real number in the range 32767 through -32767. The value of `x` is rounded (not truncated) to the nearest integer value. If `x` is greater than 2, the value 2 is used instead. If `x` is

less than  $\emptyset$ , the value  $\emptyset$  is used instead. If no game control number  $x$  is connected to your Apple II, `btn(x)` always returns the "pressed" value 1.

Note: Of the four controls, only the first three have corresponding buttons. Of course, button number 1 does not actually have to be mounted on game control number 1. Each of the numbered buttons is simply any switch properly connected to the Apple II. See the Apple II Reference Manual for electrical details. If your Apple II is equipped with the usual single pair of game controls, these controls are numbered  $\emptyset$  and 1.

You can use the `BTN` function to let the student indicate any simple two-state response. For instance, the student might press the game-control button to indicate when the game-control "paddle" or knob is at the desired position. Example:

```
*wait j(btn(1)= $\emptyset$ ):wait      (If button on control #1 is
                               not being pressed, Jump
                               back to the label wait .)
```

#### KEY(X)

Returns the ASCII code of any keyboard key pressed since the last `Accept` or `Wait` instruction ended; returns  $\emptyset$  if no key is pressed. This function responds to any key except `REPT`, `CTRL`, `SHIFT`, or `RESET`. The value of  $x$  can be any valid real number. Within this range, it does not matter what value of  $x$  you specify; this number is not used by the function in any way.

Once a key is pressed, `key(x)` will continue to return the ASCII number of the last key pressed until an `Accept`, `Accept Point`, or `Wait` instruction resets the keyboard. After an `Accept` instruction is executed, the value of `key(x)` is always reset to  $\emptyset$ , whether or not a key is pressed within the maximum time allowed by a `pr:t` command. `Key(x)` also is reset to  $\emptyset$  if a `Wait` is executed without being interrupted by a keypress, if a button is pressed in response to an `Accept Point`, or if no response is given to an `Accept Point` within the time allowed by a `pr:t` command. However, if either a `Wait` or an `Accept Point` is terminated by a keypress, the ASCII code of that key will be returned by `key(x)`. Example:

```
t:To go on, press any key. (Display text.)
```

```
*halt j(key(1)= $\emptyset$ ):halt    (If no key is pressed after
                               the last Accept or Wait,
                               Jump to label halt .)
```

Note: The example above waits for a keypress only if no key is pressed after the execution of the last `Accept` or `Wait` instruction.

Note: You can use the key(x) function in Immediate Mode, but you have to be quick. After typing t: #(key(1)), for example, press the RETURN key and then immediately type the key whose ASCII number you want to know. If you complete the keypress before SuperPILOT finishes evaluating the instruction, the ASCII number of the key you pressed is displayed on the next line; if you are too slow,  $\emptyset$  is displayed.

TIM(X)

Returns a number that is the time, in seconds, that elapsed while the last Accept or Accept Point instruction waited for the student to complete a response. The value of x can be any valid real number. It does not matter what value of x you specify; this number is not used by the function in any way.

After an Accept instruction, tim(x) returns the number of seconds that elapsed before the student pressed RETURN to terminate the response. The minimum time returned is  $\emptyset.125$  seconds. If the response is terminated by waiting longer than the maximum response time set by a pr:tn instruction, tim(x) returns the value  $\emptyset$ . Example:

```
pr:t5u          (Set maximum response time to
                 5 seconds, convert response
                 letters to capital letters.)
t:What color is an orange? (Display text.)
a:              (Accept student's response.)
c:n=tim(1)      (Store response time in simple
                 numeric variable n.)
t(n= $\emptyset$ ):Your time is up. (If student did not complete
                               response in 5 seconds,
                               display this text.)
m:ORANGE        (Did student type ORANGE ?)
ty:You got the right answer (If Yes, display this text.)
tn:You got the wrong answer (If No, display this text.)
t(n):in only #n seconds     (If n is not  $\emptyset$ , display
                               text, substituting the
                               response time n for #n
                               and one space.)
```

After an Accept Point instruction, tim(x) returns the number of seconds that elapsed before the student pressed either of the two game-control buttons to terminate the response. The minimum time returned is  $\emptyset.04$  seconds. If the response is terminated by waiting longer than the maximum response time set by a pr:tn instruction, tim(x) returns the value  $\emptyset$ .

# Operators

Many calculations, comparisons, and other arithmetic and string operations involving constants, variables, and functions are indicated in your lessons by operators. An operator is indicated by either one or two special symbols. Three operators (unary minus sign, unary plus sign, and logical NOT) affect only the element that follows the operator. All the other operators are placed between the two elements that are used in carrying out that operation.

Each operator requires an element or two elements of a certain type. A numeric element (indicated in these discussions as  $x$  or  $y$ ) may be a numeric constant, a simple numeric variable, an element of a numeric variable array, or a function or expression that yields a numeric result. A string element (indicated in these discussions as  $x\$$  or  $y\$$ ) may be a string constant, a string variable, a substring variable, or a function or expression that yields a string result. Expressions are discussed in the section following this one.

## Arithmetic Operators

The arithmetic operators handle all the usual "calculator" operations: addition, subtraction, multiplication, division, and exponentiation. Two additional operators indicate the sign of an element.

These operators must be used with numeric elements. The value of an element can be any valid real number, and the result of the operation must also be a valid real number. This means that the elements and the operation results should be in the range from about  $-1.18E-38$  through about  $-3.4E38$ , or be  $\emptyset$ , or be in the range from about  $1.18E-38$  through about  $3.4E38$ .

Operator Symbol	Name of Operation	Form of Operation	Example in Compute Instr.	(Amount Assigned to Variable p5)
+	unary plus sign	+x	c:p5=+13.3	(Positive 13.3)
-	unary minus sign	-x	c:p5=-14	(Negative 14)
+	addition	x+y	c:p5=t1+7	( t1 plus 7 )
-	subtraction	x-y	c:p5=19-q	(19 minus q )
*	multiplication	x*y	c:p5=i*r	( i times r )
/	division	x/y	c:p5=v/6	( v div. by 6 )
**	exponentiation	x**y	c:p5=b**8	( b raised to the power 8 )

The unary minus sign multiplies the element after the sign by  $-1$ . The unary plus sign is rarely used, since it does not change the element following it in any way.

An exponentiation operation (  $x**y$  ) that yields a value greater than about  $6.8E38$  or less than about  $-6.8E38$  causes an error. Also, the intermediate result  $x*\ln(y)$  must not exceed the range for valid real numbers. If  $y$  is zero, the result is one, except that if  $x$  is



zero the result is always zero. If  $x$  is negative, then  $y$  must be an integer in the range  $-32767$  through  $32767$ . When  $x$  is negative, the result is positive for an even  $y$ , and negative for an odd  $y$ .

Note: There is currently an error in the exponentiation routine that yields no result at all for a certain very small range of values that normally yields a result near  $1.18E-38$  or near  $-1.18E-38$ .

## Relational Operators

Relational operators let you compare two elements to see if they are the same, or to see which element would appear first in an ordered list. You can compare two numeric elements, or you can compare two string elements, but you must not compare two elements of different types.

A relational operator is used to make an "assertion," like " $x=y$ ". The operation then compares  $x$  and  $y$  to determine whether or not the assertion was true. If it was true, the operation yields the number 1 as the result. If the assertion was false, the number 0 is the result.

Two numeric elements may be compared, and the operation is fairly obvious. The numeric value of each element is determined, and those two values are compared. Thus the assertions  $9=9$  and  $9>3$  (9 is greater than 3) are both true, so the result in both cases would be 1.

Two string elements are compared in the following way. Each element is evaluated and reduced to a single resulting string. Then the ASCII code number for the first character in one string is compared to the ASCII code number for the other string's first character. This is a normal numeric comparison, just like comparing two numeric elements. If the first characters have different code numbers, the comparison ends. If those numbers are the same, the second character in each string is compared in the same way. If the two strings have different lengths, the shorter string has spaces (ASCII code 32) added to the end of it for the purposes of comparison. Thus, the assertions "Apple" = "Apple" and "Apple">"Apple"!!CHR(29) are both true, and the result is 1 in both cases. See the appendix ASCII Character Codes for the code number corresponding to each character.

<u>Operator Symbol</u>	<u>Name of Comparison</u>	<u>Form of Comparison</u>	<u>Example in Compute Instr.</u>	<u>(Amount assigned to Variable p5)</u>
=	is equal to	x=y x\$=y\$	c:p5=5.2=5.2 c:p5="ab"="ac"	( 1 , True) ( 0 , False)
<>	is not equal to	x<>y x\$<>y\$	c:p5=n<>4 c:p5="a"<>"e"	( 1 , unless n=4 ) ( 1 , unless e\$(1) is letter a )
<	is less than	x<y x\$<y\$	c:p5=5.2<5.1 c:p5="Z"<"a"	( 0 , False) ( 1 , True)
>	is greater than	x>y x\$>y\$	c:p5=5.2>5.1 c:p5="99">"aa"	( 1 , True) ( 0 , False)
<=	is less than or equal to	x<=y x\$<=y\$	c:p5=5<=5 c:p5="\$"<="&"	( 1 , True) ( 1 , True)
>=	is greater than or equal to	x>=y x\$>=y\$	c:p5=17>=17.01 c:p5=s\$>="A"	( 0 , False) ( 1 if ASC(s\$(1) is 65 or more)

Note: In a Compute instruction such as c:q=5 , the equals sign is NOT the relational "is equal to" operator but the assignment or "is assigned the following value" indicator. Almost every Compute instruction consists of the instruction name, followed by a colon, followed by a variable's name, followed by the assignment indicator ( = ), followed by the numeric or string element to be evaluated and assigned to or stored in the named variable.

The element to be evaluated may be an assertion containing a relational operator, like the "is equal to" ( = ) operator. Thus, in the Compute instruction c:q=x=y , the first = is the assignment indicator, while the second = is the relational operator in the assertion "the number stored in simple numeric variable x is equal to the number stored in simple numeric variable y ." If that assertion is true, the number 1 is assigned to simple numeric variable q ; if false, the number 0 is assigned to q .

A further confusion may arise when an assertion appears, not in a Compute instruction, but alone as a condition for executing an instruction. In the instruction c:q=5 , the = sign is the assignment indicator, causing the number 5 to be stored in numeric variable q . In the instruction t(q=5):Right! the = sign is the relational "is equal to" operator. If the number stored in numeric variable q is equal to 5, the assertion "q=5" is true, so the Type instruction is executed. If not, the assertion is false, so the Type instruction is skipped.

## Logical Operators

Logical operators are like relational operators in that they deal with truth or falsehood. Only numeric elements can be used with the logical operators, and only the truth or falsehood of each element is considered. In evaluating whether an element is true or false, only the value  $\emptyset$  is considered to be "false," and any number other than zero (whether positive or negative) is considered to be "true." When indicating the result of a logical operation, SuperPILOT uses the number  $\emptyset$  to indicate "false," and the number 1 to indicate "true."

<u>Operator Symbol</u>	<u>Name of Operator</u>	<u>Form of Operation</u>	<u>Example in Compute Instr.</u>	<u>(Amount assigned to Variable p5 )</u>
$\wedge$ NOT	NOT	$\wedge x$ notx	c:p5= $\wedge$ 5.23 c:p5=not $\emptyset$	( $\emptyset$ , False) ( 1 , True)
& AND	AND	x&y x and y	c:p5= $\emptyset$ &t2 c:p5=3and45	( $\emptyset$ , False) ( 1 , True)
! OR	OR	x!y x or y	c:p5=a!12 c:p5= $\emptyset$ or(1-1)	( 1 , True) ( $\emptyset$ , False)

The logical NOT operator (written either as  $\wedge$  or as not ) gives a result of 1 (true) if the following element is false (its value is  $\emptyset$ ), or gives a result of  $\emptyset$  (false) if the following element is true (its value is non-zero). In other words, the result is whatever the element after the operator was not.

The logical AND operator (written either as & or as and ) is used in this assertion: "the element before the operator is true (non-zero) AND the element after the operator is also true (non-zero)." If this assertion is true, the result is the number 1. If the assertion is false, the result is  $\emptyset$ . Thus, if the value of either element is  $\emptyset$ , the assertion is always false and its result is a  $\emptyset$ . The assertion is only true (yielding the result 1 ) if both elements have non-zero values. The assertion  $m\&\emptyset$  is false (result is  $\emptyset$ ) regardless of m's value. The assertion  $r\&t$  is true (result is 1) only if both r and t have non-zero values.

The logical OR operator (written either as ! or as or ) is used in this assertion: "either the element before the operator is true (non-zero) OR the element after the operator is true (non-zero), or both are true." If this assertion is true, the result is the number 1. If the assertion is false, the result is  $\emptyset$ . Thus, if the value of either element is non-zero, the assertion is always true and its result is a 1. The assertion is only false (yielding the result  $\emptyset$ ) if both elements have the value  $\emptyset$ . The assertion  $-5!m$  is true (result is 1) regardless of m's value. The assertion  $r!t$  is false (result is  $\emptyset$ ) only if both r and t have the value  $\emptyset$ .

## String Operator

The string operator lets you join one string element to another string element. This operation is known as concatenation.

Operator Symbol	Name of Operation	Form of Operation	Example in Compute Instr.	(String assigned to Variable s\$ )
!!	concatenation	x\$!!y\$	c:s\$="Ap"!!"ple"	( Apple )

The string that appears after the concatenation operator is added to the end of the string that appears before the operator. Example:

```
d:a$(20);b$(20);c$(20)      (Reserve space to store up to
                              20 characters each in string
                              variables a$ , b$ , and
                              c$ .)
c:a$="ippi";b$="iss"         (Store string "ippi" in a$ ;
                              store string "iss" in b$ .)
c:c$="M"!!b$!!b$!!a$        (Concatenate the indicated
                              strings, end to end.)
t:Mangroves mutter in the $c$ mud. (Display text, substituting
                              the combined string for
                              $c$ and one space.)
```

This example displays the message:

```
Mangroves mutter in the Mississippi mud.
```

## Expressions

Expressions are combinations of constants, variables, functions, and operators that perform arithmetic calculations or string manipulations, or even both. Of course, you have been seeing expressions all through this manual: every function argument is an expression, everything assigned to a variable is an expression, and every parenthesized condition for an instruction's execution is an expression. Frequently, an expression will appear in a SuperPILOT instruction enclosed in parentheses that are immediately preceded by a pound sign ( # ); this is the form used in the object text of a Type instruction, for example, when the expression is to be replaced by its value. An operator and its related elements form an expression that, when evaluated, may be used as an element in a larger expression.

## Operator Precedence

In expressions like  $3+4*2$  , you don't know whether the result is 14 or 11 unless you know the order in which the addition and multiplication operations are done. This order among operators in an expression is called precedence. The precedence of each operation in an expression can be determined in three ways:

1. Before anything is evaluated, each user variable and system variable is replaced by the value assigned to it. Each function is also replaced by the value that the function returns (this may require first evaluating any expression that appears as an argument to the function).
2. All operations within a set of parentheses (like this) are carried out before proceeding to the rest of an expression. Thus, the result of  $(3+4)*2$  is 14, while the result of  $3+(4*2)$  is 11. If you want to make the order of execution absolutely clear in an expression, use parentheses.

If parenthesized subexpressions are nested within each other, the innermost subexpression is evaluated first. Then the result of that subexpression is used as an element in evaluating the surrounding expression, and so on. Thus, the expression  $7*(6/(5-2))$  becomes  $7*(6/3)$  after the first step, then becomes  $7*2$  after the next step, and the final result is 14.

3. When the order of execution in an expression or sub-expression is not set by parentheses, the operations are carried out in the following sequence:

<u>Execution Order</u>	<u>Operator Symbols</u>	<u>Name of Operation</u>
First	^ NOT	logical NOT
	-	unary minus sign
	+	unary plus sign
Second	**	exponentiation
Third	*	multiplication
	/	division
Fourth	+	addition
	-	subtraction
	!!	concatenation
Fifth	=	: equal
	<>	: not equal
	<	relational : less than
	>	comparisons: greater than
	<=	: less than or equal
	>=	: greater than or equal
Sixth	& AND	logical AND
	! OR	logical OR

When operations from different precedence levels appear in the same expression, with no parentheses to alter the precedence, the operation with the highest precedence is executed first. This is the operation

that appears closest to the level marked "First" in the preceding table. Then the operation with the next highest precedence is executed.

In the expression  $3-7\&3^{**}3^2>7$  first the exponentiation  $3^{**}3$  is carried out. The resulting expression is  $3-7\&27^2>7$ . Then the multiplication  $27^2$  is done, making the expression  $3-7\&54>7$ . Next the subtraction  $3-7$  is carried out, leaving  $-4\&54>7$ . Then the comparison  $54>7$  is executed, and the expression is  $-4\&1$ . Finally, the logical AND gets its chance: the resulting, completely evaluated expression is 1.

You can see that several of the six levels of precedence contain more than one operator. Operations of equal precedence are simply executed starting with the leftmost operation first, and proceeding toward the right.

In the expression  $3*4+6/6*2/4-8$ , first all the multiplications and divisions are carried out, working from left to right.  $3*4$  yields 12. Then  $6/6$  yields 1, which result is multiplied by 2 to give 2, and that result is divided by 4 to give .5. This leaves the expression  $12+.5-8$ . Finally, all the additions and subtractions are executed, and the resultant value is 4.5.

## Limitations on Expressions

In general, an operation in an expression is carried out as soon as it becomes clear the operation does not depend on the outcome of another operation. In the expression  $3+4*5-6/7^{**}8+1$ , the first operation (addition) must wait to see if its second element is followed by an operation of higher precedence. Then, the second operation (multiplication) must also wait, to see if its second element is followed by an operation of still higher precedence. We now have two operations waiting to be executed. The third operation (subtraction) is not of higher precedence than either of the waiting operations, so both of the waiting operations can be carried out immediately.

In SuperPILOT, there can be a maximum of nine operations waiting to be executed. "Operations" that count in this context are operators, parenthesized subexpressions, and functions, if their execution must await the outcome of another operation. For example, the expression  $1+2*3-4/5+3*4-6/7+5*6-8/9+4*3^{**}2+1$  causes no problems, because there are never more than a couple of operations waiting. But the expression  $1+(1+(1+(1+(1+2*3))))$  will not be executed, because the multiplication operation is the tenth waiting operation. (The waiting operations are +, (, +, (, +, (, +, (, +, and \*.) In an expression such as  $1+\sin(x+2*3)$ , the function name and the left parenthesis together add two waiting operations.

Note: If the rules of expression syntax are not followed, the expression may be evaluated anyway, with unpredictable results. However, if the evaluation of an expression cannot be completed, an error message appears in Author Mode and the Error Flag is raised in both Author Mode and Lesson Mode.

String expressions are handled by SuperPILOT in a fixed amount of the Apple II's memory. In addition to the 16000 memory locations that can be used for Dimensioning string variables, there are 2800 locations that are reserved solely for SuperPILOT's use. SuperPILOT uses these 2800 locations for several purposes: to assign a string to a string variable or to perform string editing and concatenation, for example. This use is only temporary: all 2800 locations are freed for reuse when the execution of an instruction is completed.

If more than 2800 memory locations are needed by SuperPILOT to perform the string-handling directions of a single instruction, it uses some of the other 16000 locations temporarily, if they are not already reserved. So, if you have Dimensioned strings totaling all or nearly all of the 16000 allowable locations, it is possible (though not probable) that a complex string operation in your lesson could cause an "out of memory" error.

SuperPILOT also has 35 memory locations beyond the 2000 available to you for Dimensioning numeric variable arrays. These 35 locations are used for handling the simple numeric variables that you are not required to Dimension. If your lesson includes more than 35 simple numeric variables, SuperPILOT uses locations from the 2000, if any are free. As with string expressions, therefore, it is possible for the available memory to be used up if many simple numeric variables appear in the same lesson with very large numeric variable arrays.





## Appendix A

# ASCII Character Codes

252 Screen Command Characters

255 Screen Modes

258 Printing Characters

Several operations in SuperPILOT use the number codes by which the Apple II recognizes characters. They follow a standard coding scheme for characters called the ASCII character codes. For example, the ASC function in SuperPILOT returns the standard ASCII decimal code (the code can also be expressed in binary, hexadecimal, etc.) for the first character stored in a specified string variable. Similarly, the CHR function returns the character that corresponds to a specified ASCII decimal code number.

## Screen Command Characters

The "characters" corresponding to the ASCII decimal code numbers from 0 through 31 are not printing characters that are displayed on your screen, but commands to the display device. These tell the Apple II to "clear the screen," or "set a text viewport," and so forth. The command is carried out when a string containing the command character is sent to the screen for display by the Type instruction. You can also send one of these commands to the screen by using the Graphics instruction's X command, g:x , followed by the appropriate code number.

Most of the commands affect only the text cursor, and operate within the text viewport that is currently set. Since the text cursor itself is not visible, you usually do not see the effect of a command until the Type instruction or a response typed on the keyboard displays a printing character on the screen at the new cursor position.

The text cursor is also moved by any displayed Type text or keyboard response text, and by any Graphics command that moves the graphics cursor when the text viewport is set to occupy the full screen.

**Note:** However, the text cursor does NOT join the graphics cursor after Graphics instructions that contain no graphics moves. The only exception to this is the g:x30,x,y instruction, which moves the text cursor to location x,y immediately.

The text displayed by the Graphics instruction's T command is not influenced by these screen commands, even though they affect the display of all other text.

<u>Decimal Code</u>	<u>Standard Name</u>	<u>Command Name</u>	<u>Effect on the Screen</u>
0	NUL		(No action.)
1	SOH		(No action.)
2	STX	Single size	Switch to single-size text.
3	ETX	Cursor On	Allow the system cursor to be displayed after having been turned off by ASCII 6 (see below).

Decimal Code	Standard Name	Command Name	Effect on the Screen
4	EOT	Double size	Switch to double-size text.
5	ENQ	Return + line feed	Move the text cursor to the left edge of the text viewport and one line down.
6	ACK	Cursor Off	Turn off the operating system's cursor (used in error messages, for example). Has no effect on SuperPILOT's graphics or text cursors.
7	BEL	"Bell"	Make a "Beep" on the Apple II's built-in speaker.
8	BS	Backspace	Move the text cursor one character position to the left.
9	HT		(No action.)
10	LF	Line feed	Move the text cursor one line down without moving to the right or left.
11	VT	Clear to end of screen	Clear the text line to the right of the text cursor, and clear all of the text viewport below that line.
12	FF	Clear screen	Clear the entire text viewport and move the text cursor to the top left corner of the text viewport.
13	CR	Return	Move the text cursor to the left edge of the text viewport and one line down.
14	SO	Normal video	Set normal print (white characters on black background) for Type text and keyboard response text.
15	SI		(No action.)
16	DLE	Indent	Use the ASCII code of first character that follows, minus 32, as the number of spaces to insert starting at the current text cursor position.
17	DC1		(No action.)
18	DC2	Inverse video	Set inverse print (black characters on white background) for Type text and keyboard response text.
19	DC3		(No action.)

<u>Decimal Code</u>	<u>Standard Name</u>	<u>Command Name</u>	<u>Effect on the Screen</u>
20	DC4	Release viewport	Set the text viewport to occupy the full screen. Releases the viewport set by CHR(22) and CHR(23). To release the Viewport set by g:vl,r,t,b use g:v , not CHR(20).
21	NAK		(No action.)
22	SYN	Set top left of viewport	Set the top left corner of the text viewport to the text cursor. Does not affect the bottom or right edges of the text viewport. Unlike g:vl,r,t,b CHR(22) does not tell the outer SuperPILOT system about the set viewport. Clear this viewport using CHR(20) before returning to the main lesson.
23	ETB	Set bottom right of viewport	Set the bottom right corner of the text viewport to the text cursor. Does not affect the top or left edges of the text viewport. Unlike g:vl,r,t,b CHR(23) does not tell the outer SuperPILOT system about the set viewport. Clear this viewport using CHR(20) before returning to the main lesson.
24	CAN		(No action.)
25	EM	Home	Move the text cursor to the top left corner of the text viewport.
26	SUB		(No action.)
27	ESC	Screen mode	If the first character that follows is 1, 2, or 3, set the screen mode to that number for displaying Type text or keyboard response text: Mode 1: Normal display. Mode 2: Overprint. Mode 3: Exclusive Or. Ignore (do not print) the first character that follows if it is not 1, 2, or 3. To use these screen modes, however, you may use the ts:m command (see the chapter Special Effects). More details on screen modes are given in the next section.
28	FS	Forward space	Move the text cursor one character position to the right.

<u>Decimal Code</u>	<u>Standard Name</u>	<u>Command Name</u>	<u>Effect on the Screen</u>
29	GS	Clear to end of line	Clear the text line to the right of the current text cursor position.
30	RS	Goto XY	Use the ASCII code of first character that follows, minus 32, as horizontal character position relative to left edge of the text viewport. Use ASCII code of second character that follows, minus 32, as line number relative to the top edge of text viewport. Move text cursor to that position or the edge of the viewport closest to that position. The top line is 0, and the first character position on a line is 0. After a Graphics instruction like g:x30,32,38 the text cursor is moved immediately, not waiting for the end of the Graphics instruction.
31	US	Reverse line feed	Move the text cursor one line up without moving to the right or left.

## Screen Modes

Screen Mode 1 is the default mode normally used by the system. In addition, Mode 1 is the only mode used for text displayed by the Graphics instruction's T or X commands or by the Type Specify instruction's X command. Mode 1 is also used to display characters typed by students in response to an Accept instruction. To change the screen mode used for displaying the Type instruction's object text and response text, you can use either a Type Specify instruction or a Graphics instruction, as shown below:

<u>To set Screen Mode</u>	<u>Use this Type Specify instruction</u>	<u>Or use this Graphics instruction</u>
1	ts:m1	g:x27,49
2	ts:m2	g:x27,50
3	ts:m3	g:x27,51

<u>Mode</u>	<u>Effect on the Screen</u>
1	Normal display: each character, with its own small background, completely replaces whatever was on the screen in that place. This is the mode that is always used by the Graphics instruction's T command.

Mode            Effect on the Screen

- 2            Overprint display: shows each character's "on" dots in addition to any screen-dots that were already "on" in that place. In normal video, the dots of the character itself are "on," while the background dots are "off." In reverse video, the opposite is true.

Because of Apple II's method of producing color, this mode does not work for printing characters on colored backgrounds. A white character on any background other than black is practically invisible. However, you can use this mode for underlining a word, say, by Typing the word, then back-spacing, and then Typing the underline character in Mode 2. You can also create many other interesting "combination" characters by overprinting two or more characters in the same place.

This mode is usually bad for displaying keyboard response text, since it moves a white square cursor (all dots "on") in front of the response characters. In normal Mode 1, the old cursor is replaced by the typed response character, but in Mode 2 the typed character is just printed "on top of" the old all-white cursor.

The main use of Mode 2 is for displaying text in a part of the screen crossed by a few thin graphics lines. In Mode 1 the background of each character may erase gaps in the lines. In Mode 2 the lines will cross right through the characters without problems.

- 3            Exclusive OR display: shows each character's "on" dots in addition to any screen-dots that were already "on" in that place, except turns "off" any screen-dot specified as "on" by BOTH the former screen image and the character.

This mode is useful for printing text across a background consisting of large areas of both black and white. The characters will show up as white when printing over a black background, and as black when printing over a white background. If the character is then reprinted at the same location, the previous background is restored. This mode is not usually effective for displaying keyboard response text.

It is also possible to change the text line spacing, foreground color, and background color with ASCII 27. To change the line spacing, use the ASCII code that is equal to 32 plus the number of line spaces desired. For example, triple spacing would be specified by `g:x27,35`. The maximum line spacing is 15 (ASCII 47). To change the foreground text color, use the ASCII code that is equal to 64 plus the number of the foreground color desired. For example, the color Violet (text color 2) would be specified by `g:x27,66`. To change the background text color, use the ASCII code that is equal to 96 plus the number of

the background color desired. For example, the color Violet (text color 2) would be specified by `g:x27,98` .

The equivalent Type Specify commands can be substituted for the Graphics commands. For example, `ts:x27,98` has the same effect as `g:x27,98` . SuperPILOT's numbered text colors are given in the Text Colors section of the Special Effects Instructions chapter.

Note: These screen controls should never be used from SuperPILOT, since they control the screen without SuperPILOT's knowledge of what is happening and can cause the lesson to behave strangely. With SuperPILOT, use the appropriate Type Specify commands instead: `ts:l` for line spacing, `ts:f` for foreground color, and `ts:b` for background color.

# Printing Characters

---

The characters corresponding to the ASCII decimal code numbers from 32 through 127 are the standard printing characters that can be displayed on the screen. By knowing the ASCII code numbers for these characters, you can use the CHR function to put them in strings that let you display all the characters that are not directly typeable on the Apple II keyboard.

<u>Decimal</u> <u>Code</u>	<u>Standard</u> <u>Name</u>	<u>Decimal</u> <u>Code</u>	<u>Standard</u> <u>Name</u>	<u>Decimal</u> <u>Code</u>	<u>Standard</u> <u>Name</u>
32	space	64	@	96	
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	del

**Note:** SuperPILOT displays the del character, ASCII code 127, as an inverse space, appearing as a white square on the screen (this is your text cursor).

The ASCII code numbers 128-159 are available for you to design in SuperPILOT's Character Set Editor. Numbers 128-255 otherwise correspond to screen commands and printing characters that repeat the set from 1-127.



## Appendix B

# Using More Disk Drives

- 260 Diskette Names
- 261 Disk Drive Names
- 262 Lesson Mode
  - 263 Specifying a Diskette Name
  - 264 Specifying a Disk Drive Name
- 265 Author Mode

SuperPILOT is generally intended for use with two different Apple II computer system configurations: an Author Mode system with two disk drives, complemented by several Lesson Mode systems each using one disk drive. If your systems are set up with these standard configurations, you do not need to know the information in this appendix. In a standard system SuperPILOT handles all access to the disk drives automatically.

However, it is possible to use SuperPILOT effectively on systems with more than the usual number of disk drives. In particular, Lesson Mode can accommodate up to six disk drives, and can use files from diskettes in all the drives while running a lesson. The information in this appendix is for people using such nonstandard multiple-drive systems in Lesson Mode.

## Diskette Names

---

Each time you initialize a new diskette for storing your lesson files, SuperPILOT asks you to name your diskette. You can type any name up to seven characters long. Depending on the name you select, your diskette will be initialized as one of three types: a Lesson diskette, Resource diskette, or Recordkeeping diskette.

A Recordkeeping diskette is one entirely dedicated to storing program data from Keep instructions in your lessons. Any time you give the name SYSLOG to a diskette, it will be initialized as a Recordkeeping diskette. See the Keep discussion in the File Handling Instructions chapter for details on how this diskette is used.

A Resource diskette contains none of the system files that allow a diskette to be booted in Lesson Mode. It is dedicated entirely to storing the files you create in the SuperPILOT Editors. Since it cannot be booted, it is useless in a single-drive Lesson Mode system, but it is ideal as a secondary-drive diskette in Lesson Mode, because it has room for almost twice as many files as a Lesson diskette. You create a Resource diskette whenever you include a plus sign ( + ) in the name you give it during initialization: for example, TWO+TWO, OCTO+, etc.

Any diskette you initialize that does not have the name SYSLOG and does not have a plus sign in its name will be a standard Lesson diskette.

**WARNING!** Whenever you have more than one disk drive in Lesson Mode, it can be very dangerous to put two diskettes with the same name into a lesson system at the same time. SuperPILOT will be quite confused about which diskette is which. It may retrieve information from the wrong diskette or, worse yet, it may store information on the wrong diskette, possibly even scrambling the directory that tells where all the files are stored.

To avoid this problem, you may wish to develop the habit of giving each new Lesson or Resource diskette a new and different name. Be sure to write the diskette's name on the label when you initialize it (write very gently, with a felt-tip pen).

If you intend to write lessons that use diskettes from more than one disk drive in Lesson Mode, you **MUST** give your Lesson and Resource diskettes different names. If you are not sure they have different names, initialize (and name) a new diskette; then transfer the necessary files from one of the unknown diskettes onto the new one.

**Note:** If you have access to an Apple Pascal system, you can use the Pascal Filer to see the name of your lesson diskettes and even to change a lesson diskette's name.

In Author Mode there is much less risk of error due to diskettes with the same name. When you update the backup copy of a diskette, for example, it has the same name as your original and both diskettes will need to be in the system at the same time if you are to copy the revised original onto the backup. This causes no confusion for SuperPILOT because the original is always placed in the primary drive and the backup is always placed in the secondary drive. The system will transfer files only from drive 1 to drive 2, ignoring the names of the diskettes.

## Disk Drive Names

---

Each disk drive has a numeric "name." This name is determined by two things:

1. The number of the slot inside the Apple II where the drive's controller card is installed (these slots are numbered from 0 through 7, but SuperPILOT disk controller cards can be in slots 4, 5, or 6).
2. The name of the controller card's connector to which the drive's ribbon cable is attached (the upper connector is labelled "DRIVE 1," the lower connector is labelled "DRIVE 2").

Normally, you don't need to know these names, but for this discussion it will prove helpful. Here are the names for all of the six disk drives you can connect to your Apple II:

<u>Name of Drive</u>	<u>Controller Card</u>	<u>Description of the Disk Drive</u>
#4	Slot 6, Drive 1	This is the first disk drive. It is sometimes called "drive 1" or "the primary drive." Each Lesson Mode system must have at least this one drive. In Author Mode the Author diskette goes in this disk drive.
#5	Slot 6, Drive 2	This is the second disk drive. Sometimes it is called "drive 2" or "the secondary drive." Every Author Mode system must have a lesson diskette in this drive and an Author diskette in the drive named #4 .
#9	Slot 4, Drive 1	Additional disk drive.
#10	Slot 4, Drive 2	Additional disk drive.
#11	Slot 5, Drive 1	Additional disk drive.
#12	Slot 5, Drive 2	Additional disk drive.

The names may seem arbitrary, but they are correct. If you plan to use a multiple-disk-drive Lesson Mode system, you may wish to label your disk drives with their correct drive "names."

## **Lesson Mode**

---

Lessons that are running in Lesson Mode can use files that are stored on any diskette in any available disk drive. In general, the safest practice is to put all the diskettes needed by the lessons into their disk drives BEFORE you turn on the system. That way, when the system starts up it will discover where each diskette is located in the system. Thereafter, it is best to leave all the diskettes in their original drives. SuperPILOT will look in all the drives to find a diskette whose name was not found during the startup process, but it assumes the known diskettes will stay put.

In a standard one-drive Lesson Mode system, you need to tell the system only the file name of the lesson, graphic, character-set, or sound effect that is wanted. SuperPILOT automatically supplies the name of the only disk drive, #4 .

In a multiple-drive Lesson Mode system, SuperPILOT continues to supply the name of disk drive #4 for any file that you specify by file name alone. In such a system, however, you now have the option of using a different diskette or a different drive. You can do this in

either of two ways: by giving SuperPILOT the name of the diskette that holds the file you want, or by giving SuperPILOT the name of the disk drive that holds that diskette.

## Specifying a Diskette Name

This is the preferred method, because it allows your lessons to work correctly no matter which drives the diskettes are put in. Basically, any instruction that normally specifies a file name alone can also have the diskette name appended to the front of that file name, with a colon between the two names. SuperPILOT will then find the diskette with the given diskette name, in any drive, and retrieve the desired file from that diskette.

For example, suppose you store a sound effects file named `tunel` on a lesson diskette named `MYDISK`. On a one-drive system, you would put that diskette in the disk drive and run a lesson that uses your sound effect in an instruction such as

```
sx:tunel
```

(Play the sound effect stored in the file named `tunel`, on the diskette now in the disk drive named `#4`.)

A multiple-drive system can do the same thing, if that diskette is in the drive named `#4`. Suppose, however, that you want your lesson (which might be on the same diskette or on a diskette in a different drive) to use that sound effect even if the diskette containing the sound effect file is NOT in drive `#4`. Just add the diskette's name and the following colon to the file name in your `SX:` instruction:

```
sx:mydisk:tunel
```

(Play the sound effect stored in the file named `tunel`, on the `MYDISK` diskette, in any available disk drive.)

In a one-drive system, this instruction still works fine, even though the extra information is not necessary. In a multi-drive system, SuperPILOT first finds the diskette named `MYDISK`, and then retrieves the file `tunel` from that diskette.

The same principle can be used with the instructions `GX:`, `TX:`, `L:`, `LX:`, `FOX:`, and `FIX:`. In fact, when the title page asks you to type the name of the lesson you want to run, you can specify a lesson on any available diskette. Just type the name of the diskette (be sure that diskette is in one of the drives), followed by a colon, followed by the name of the lesson.

Here is an example of how your lesson might read a file named `tales`, previously stored on a diskette named `snail`:

<pre>d:r\$(255);q\$(18)</pre>	<pre>(Reserve space to store up to 255 characters in string variable r\$ and up to 18 characters in string variable q\$ .)</pre>
<pre>c:q\$="snail:tales"</pre>	<pre>(Store the string snail:tales in string variable q\$ .)</pre>
<pre>fix:30,q\$</pre>	<pre>(Find the file named tales , on the diskette named SNAIL , and open it for retrieving 31 records.)</pre>
<pre>fi:12,r\$</pre>	<pre>(Retrieve the contents of record number 12, in the currently open file, and store those contents in string variable r\$ .)</pre>
<pre>t:Tale #12: \$r\$</pre>	<pre>(Display text, substituting the tale stored in r\$ for \$r\$ and one space.)</pre>


The possibilities of a multiple-drive Lesson Mode system are endless. You can, for instance, have all your graphics files on a special graphics diskette, all your sound effects on a special sound effects diskette, and use those diskettes as a "library" of effects that any lesson can draw on. Or you can write an immense lesson, that runs for hours and hours, Linking from files on one diskette into files on another diskette, and so on.

## Specifying a Disk Drive Name

If you wish, you can use the name of the disk drive in place of the diskette's name. For example, if you want to run the sound effects file `tunel` , on the diskette now placed in the disk drive named `#9` , you could use this instruction:

<pre>sx:#9:tunel</pre>	<pre>(Play the sound effect stored in the file named tunel , on the diskette now in the disk drive named #9 .)</pre>
------------------------	--

SuperPILOT will look on whatever diskette is in drive `#9` , and retrieve the file named `tunel` from that diskette. This method works fine for specifying the name of the lesson you want to run next in the `hello` program, but is less satisfactory for instructions within a lesson. Every time you run this lesson, you will have to make sure a diskette containing the file `tunel` goes in drive `#9` . If you put the diskette in drive `#10` , SuperPILOT won't find it. On the other hand, you can put ANY diskette with a file named `tunel` in drive `#9` and your lesson will work just fine.



WARNING! Specifying a diskette by the name of the drive it is in does NOT let you use two diskettes having the same name. SuperPILOT quickly converts drive names into the name of the diskette found there, so it will still be hopelessly confused by finding two diskettes with the same diskette name.

## Author Mode

---

Author Mode uses only two disk drives, #4 and #5, even if more drives are connected to the system. The Author diskette must be in drive #4, and a lesson diskette must be in drive #5. No matter what name you give a file that you are creating in one of the Editors, that file will always be stored on the lesson diskette in drive #5. In fact, the Editors do not allow you to type a colon or pound sign when you specify the name of the file you are creating.





## Appendix C

# Error Messages

268 Language Error Messages

272 System Execution Errors

275 System Failure Errors

# Language Error Messages

---

When you run a lesson from Author Mode, SuperPILOT gives you various messages to help you locate instructions that are incorrectly written or improperly used, or that have been given numbers that are too big or too small, or in which an expected item is missing. Any time one of these messages is given, SuperPILOT's Error Flag is also raised.

An appropriate error message appears on the screen when SuperPILOT has trouble executing an instruction, temporarily stopping the lesson. Directly beneath the message, you are shown the instruction that contains the error, with the white square cursor located on or just before the part of the instruction where the error was detected. For example, if you try to divide by zero in a Compute expression, you may get a message like this:

```
A-error (Arithmetic error)
c:n=5/0
```

with the cursor placed on the offending zero. The lesson remains stopped until you press the RETURN key or the space bar. This lets you write yourself a note about the error, in order to correct the problem when you are next in the Lesson Text Editor.

Only one screen line is used to display the instruction that contains the error, so if the instruction is longer than 40 characters, the error itself may not be displayed. If only the first 40 characters of a line containing an error are displayed, and if the cursor appears on the 40th character, you can assume that the error occurs later in the instruction.

SuperPILOT almost always lets your lesson continue after an error is detected. When an error is found, the rest of the instruction is not executed and SuperPILOT moves on to the next instruction. This may have strange effects on your lesson, but you can finish the lesson in some fashion. As you do this, jot down the error message and the offending instruction line. Circle the character where the cursor is: either this character or the one following is the first character that was found to be unacceptable. When the lesson is over, you can use this information to make appropriate changes to your lesson in the Lesson Text Editor.

Note: These error messages are not displayed when you run a lesson in Lesson Mode. Instead, the Error Flag is raised and your lesson's error-handling instructions can detect this problem by using the Error Conditioner.

The SuperPILOT Language Error Messages and their explanations follow.

Error Message Displayed

Explanation

A-error (Arithmetic error)

In an expression, you tried to do something illegal, such as divide by 0, or use a number too big or too small. Most numbers must be in the ranges -3.4E38 to -1.2E-38, 0, and 1.2E-38 to 3.4E38. The arguments to some functions must be within smaller limits.

B-error (Value out of bounds)

A subscript number is not in the range specified by Dimension for that variable; or you used a subscripted variable that has not been Dimensioned; or an object-field command number such as a Graphics color is too large or is negative.

C-error (Invalid compute instruction)

A bad variable name, bad edit option, missing equal sign, or other syntax error has occurred in the non-expression part of a Compute instruction. Errors in the expression part of a Compute instruction are usually reported as E-errors or S-errors.

D-error (Diskette read error)

Your diskette is damaged (contains bad blocks of information); or a file was read into the Apple incorrectly for some reason, so the end of the current file was reached before the end of the program.

E-error (Invalid expression)

You have written an expression that does not make sense, such as an odd number of parentheses or a wrong character.

## Error Message Displayed

## Explanation

F-error (File input/output error)	In FOX: , FIX: , FO: , or FI: , you named a bad string variable; or in FOX: or FIX: the string variable contains a bad file name; or there was no unused space on the lesson diskette to create a new file; or FO: or FI: used a record number larger than specified by FOX: or FIX: or beyond the file created by FOX: ; or you tried to use FO: or FI: when no file was open.
G-error (Invalid graphic instruction)	Something is wrong in the list of commands for a Graphics instruction, such as non-integer number or extra space.
I-error (Invalid function argument)	The argument to a function such as SIN or PDL is too big or too small, or otherwise incorrect. Such errors usually reported as A-errors, S-errors, E-errors, or B-errors.
J-error (Invalid jump destination)	A branching instruction such as Jump, Use, or End, or a Goto command, tried to jump to a label that cannot be found. The offending destination is displayed after this message.
K-error (Error on System.Log file)	Unable to complete a read or write to the file system.log . Part of the file may have been lost and may not be recovered.
L-error (Invalid link destination)	A Link instruction specified a lesson that cannot be found.
M-error (Invalid modifier/condition)	Where your instruction's list of modifiers and conditioners should be, a character appears that is neither modifier nor conditioner.
O-error (Invalid instruction code)	The name of your instruction is not recognized by SuperPILOT.

Error Message Displayed

Explanation

P-error (SuperPILOT system error)

SuperPILOT has found an error but is unable to detect the source. Results of the most recently executed instruction may be invalid.

R-error (Too many arrays or strings)

You have tried to use up more storage places for variables than are available. You can store up to 235 numbers in simple numeric variables and elements of numeric arrays combined. You can store up to 1880 characters in reserved string variable locations and unreserved locations needed to store a string, combined.

S-error (Invalid syntax or format)

Something is wrong with the way you have typed your instruction, or an expression is too complex to be evaluated.

T-error (Invalid expression type)

You tried to store a number in a string variable, or a string in a numeric variable, or you have mixed strings and numbers illegally in an expression.

U-error (Too many subroutines)

You have nested Use subroutine invocations more than 10 levels deep. A maximum of 10 Use instructions can be waiting for their corresponding End instructions at any time.

V-error (Invalid variable name)

The object field of a Compute instruction did not start with a correct variable name, or starts with a string variable name you did not Dimension; or an Accept instruction used a bad variable name or one not preceded by # or \$, or the name of a string variable you did not Dimension.

## Error Message Displayed

## Explanation

W-error (Warning: System.Log is full)	There is no more contiguous space available for system.log on the diskette, even though there may be unoccupied blocks. Logging is turned off. Data has likely been lost from the last lesson that tried to write to system.log .
X-error (Invalid execute instruction)	There is something wrong with the string variable named in your XI: instruction.
Z-error (External device error #x)	Error in interfacing a device with V: or AP: , where x is the error code returned by the device interface procedure, as specified by the device's manufacturer. For procedures originally on the Author diskette, x will be the same as the error numbers listed later in this appendix.

As you can see, the problems reported by these messages overlap somewhat, and a given message does not always pinpoint a particular problem. Also, the point at which SuperPILOT discovers the problem may not be the true cause of a problem that may have arisen earlier in the instruction or even many instructions earlier.

The error messages are a useful aid, but you will often need to use some ingenuity to track down what is really causing the reported error. For example, if an error message points out a string constant as causing a T-error, it may not be a problem with the string at all. Instead, you may have forgotten to Dimension a string variable whose contents were to be concatenated to the constant.

Finally, some problems may not cause any error message at all. Instructions, modifier-and-conditioner lists, expressions, and objects that do not follow the rules of SuperPILOT syntax may be executed anyway with unpredictable results. Only by knowing what should happen in your lessons, and investigating carefully when anything unexpected occurs, will you be able to track down these problems.

## **System Execution Errors**

---

When the SuperPILOT system runs into difficulties such as a damaged diskette, a missing file needed to run the next part of the system, or a disk drive door left open, several different things may happen:

1. If you were running a lesson at the time, execution of the lesson usually just goes on to the next instruction. If that is impossible (because you are at the end of the current lesson, Linking to a new portion, say), the current portion of the lesson may be restarted.
2. If the system does not know how to recover, it may restart itself, returning to the Main Menu (in Author Mode) or to the lesson diskette's hello program (in Lesson Mode).
3. Other problems will cause the system to report a

SuperPILOT execution error #xxx

where the number corresponds to one of the following kinds of problems:

<u>Error Number</u>	<u>Description of the Problem</u>
0	System error of undefined nature.
2	The system specified an incorrect disk drive number.
3	The system attempted an illegal operation, such as reading from the printer.
4	(Not used on the Apple II.)
5	A disk drive is no longer available, after an operation was successfully started using that drive.
6	A diskette file is no longer in the diskette directory, after successfully starting an operation using that file.
7	An illegal file name was used (for example, a file name more than ten characters long.)
8	There was insufficient space on the specified diskette to store another file.
9	The specified diskette was not in any of the disk drives, or the specified device was not available.
10	The specified file was not on the specified diskette.
11	An attempt was made to create a diskette file when a file of that name already exists.
12	An attempt was made to open an already-open file.
13	An attempt was made to access a closed file.

Error  
Number

Description of the Problem

- 14 There was an error in reading a number (for example, the system expected a number but received a character.)
- 15 Characters were arriving at the Apple II faster than it could handle them.
- 16 The specified diskette is write-protected.

(Numbers 17 through 63 are not used.)

- 64 An attempt to store or retrieve information from a diskette file failed, because the diskette is damaged or the information was stored incorrectly.

(Numbers 65 through 99 are not used.)

- 100 A system error of unknown nature has occurred.
- 101 A number used by the system was too large or too small, or a string was too long.
- 102 A diskette file was read into the Apple II correctly, but the information in the file was incorrect.
- 103 Part of the system program was not available when it was needed, or a program portion was executed incorrectly.
- 104 The programs and data in the system have attempted to use up more of Apple II's memory than was available.
- 105 The system created an integer that was too large.
- 106 The system tried to divide something by zero.
- 107 Invalid memory reference (not used on the Apple II).
- 108 The "break" key was pressed (not available on the Apple II).
- 109 An error occurred in trying to read another part of the system's program into the Apple II from diskette.
- 110 An error occurred when your program tried to store or retrieve information in a diskette file. Usually reported as one of the errors numbered 2 through 64.
- 111 The system tried to use an instruction that was not available.



<u>Error Number</u>	<u>Description of the Problem</u>
112	The system tried to use a number that was too large or too small, or had the wrong format.
113	The system tried to store a string into a string variable that was too small for it.

After an error that is especially confusing to SuperPILOT, you may have to restart the system yourself by turning the Apple II's power off and then on again. All other errors cause the system to restart itself, usually after you press the Apple's space bar or the RESET key to continue.

Note: Most of these errors occur when the SuperPILOT system is doing its work, especially when you are using the special Editors. These errors rarely occur when you are running a lesson in Author Mode, or when a student is running a lesson in Lesson Mode, because SuperPILOT is almost always able to handle any problems that arise when a lesson is running.

Execution errors usually result from mistakes in handling the diskettes and disk drives: taking a diskette out of its drive at the wrong time, putting the wrong diskette in a drive, putting a diskette in its drive upside down or sideways, leaving the disk drive's door open, and so forth. In such cases you can just check your diskettes and disk drives, correct the problem, and restart the system.

## **System Failure Errors**

---

The other type of error message you may see is a System Failure error message. This type of error is not a failure of the SuperPILOT system itself, but has to do with the outer program structure in which SuperPILOT operates. The message displayed will resemble the following:

SYSTEM FAILURE NUMBER XX. PLEASE REFR  
TO PRODUCT MANUAL FOR EXPLANATION.

The X's will be replaced by the number of the error, and the word "REFER" will be abbreviated differently depending on the length of the error message. For more details on these error messages, see the appendix Error Messages of the Apple II SuperPILOT Editors Manual.

<u>Error Number</u>	<u>Description of the Problem</u>
01	Unable to load specified program
02	Specified program file not available
03	Specified program file is not code file
04	Unable to read block zero of specified file
05	Specified code file is unlinked
06	Conflict between user and intrinsic segments
07	UNASSIGNED ERROR CODE
08	Required intrinsics not available
09	System internal inconsistency
10	UNASSIGNED ERROR CODE
11	UNASSIGNED ERROR CODE
12	Original disk not in boot drive

## Appendix D

# SuperPILOT Language Summary

278	Expressions
278	Numbers
278	Strings
278	Variables
279	Functions
279	Transcendental Functions
279	Arithmetic Functions
280	String Functions
280	Input Functions
280	Operators
281	Order of Evaluating Operations
282	The Parts of an Instruction
282	Label
282	Instruction Name
283	Modifier
283	Conditioners
284	Colon
284	Object
284	Miscellaneous
284	The SuperPILOT Instruction Set
284	Text Instructions
284	Response Instructions
285	Control Instructions
285	Computation Instructions
286	Special-Effects Instructions
288	File-Handling Instructions

# Expressions

---

## Numbers

Numbers in SuperPILOT are all real numbers, which can be expressed as integers, decimal numbers, or in scientific E-type notation:

324	(Integer notation)
45.9716	(Decimal notation)
2.34851E-5	(E-type notation. This example is 2.34851 times 10 to the minus fifth power, or .0000234851.)

SuperPILOT rounds the main number to six significant digits if it contains more than six. A number is rounded up if the seventh digit is 5 or more, and down if it is 4 or less; this rounding takes place BEFORE the number is used or stored. Most numeric quantities can be any number in the ranges from -3.4E38 to -1.2E-38, 0, and 1.2E-38 to 3.4E38. Some of the SuperPILOT functions require argument numbers to stay in a smaller range, often from -32767 to 32767. Most numbers that appear as command parameters in the object field of instructions must stay in the range from 0 through 32767.

## Strings

Strings are any characters, enclosed in double quotes:

```
"Apple Computer, Inc."  
"Today's number is 345.99887, up 3/8ths."
```

## Variables

Many instructions may use variables in the object field. A simple numeric variable is a number storage place that holds one number. You give it a name such as `f` or `q3`. Created by using `Compute` or `Accept` to store a number: `c:f=5` or `a:#q`. In a `Type` or `Accept` instruction, name must be preceded by `#` and followed by a space or by the end of an instruction: `t:#f` or `a:#q3`.

A numeric variable array is a set of number-storage places with one name. Subscripts (one number or two, in parentheses) identify the elements in the array. You give the array a name like `m` or `p8`, and elements have names such as `m(3,4)` or `p8(13)`. Created by using `Dimension` to reserve enough space for the intended elements: `d:m(5,10)` or `d:p8(20)`.

A string variable is a word-storage place that holds a string of characters. You give it a name like `r$` or `u5$`. Created by using `Dimension` to reserve enough space for the intended strings (1 to 255 characters): `d:r$(30)`. In a `Type` or `Accept` instruction, name must be preceded by `$` and followed by a space or by end of instruction: `t:$r$` or `a:$u5$`.

You can also use subscripts after a string variable name to identify one character or you can use a substring of contiguous characters to make substring variables with names such as r\$(23) or u\$(7,22) .

A system variable is a numeric variable or string variable created and used by SuperPILOT. You sometimes store things in these variables, but more often you just use what SuperPILOT puts in them. There are ten system variables, with the following names and contents:

%A	Answer-Count	Number of times in a row the last Accept instruction has been executed.
%B	Answer Buffer	The last Accepted typed response.
%X	X-Coordinate	Graphics screen coordinates of the last point accepted by an AP: instruction.
%Y	Y-Coordinate	Graphics screen coordinates of the last point accepted by an AP: instruction.
%C	Col. Coordinate	Text screen coordinates at the current text cursor position.
%R	Row Coordinate	Text screen coordinates at the current text cursor position.
%S	Spin Angle	The current graphics angle, in degrees.
%O	Game Port	Allows access to peripherals attached to the game port.
%V	V: Parameter	Signed value for A/V device control.
%W	V: Parameter	Unsigned value for A/V device control.

There are 200 number-storage places for all simple numeric variables and numeric variable array elements. There are 1600 character-storage places for all the characters in all your string variables.

## Functions

A function is said to return the result of its calculation or operation. The information in parentheses after the function's name is called the function's argument or arguments. A string argument (shown as x\$ ) may be a quoted string, a string variable, a substring variable or an expression whose result is a string. A numeric argument (shown as x ) may be a number, a simple numeric variable, an element of a numeric variable array, or an expression whose result is a number. Except as noted in the descriptions, the value of any numeric argument can be any valid real number. If the value of the argument given is outside the function's allowable range and you try to assign the function's returned value to a numeric variable, that variable will remain unchanged.

## Transcendental Functions

SIN(X)	Returns sine of angle x , where x is in radians. X must be in the range -102942 through 102942.
COS(X)	Returns cosine of angle x , where x is in radians. X must be in the range -102940 through 102940.
ATN(X)	Returns radian measure of an angle whose tangent is x . X must be in the range -5.3E18 through 5.3E18.

## Arithmetic Functions

SGN(X)	Returns -1 if x is negative, 1 if x is positive, or 0 if x equals 0.
--------	--

**ABS(X)** Returns the absolute value of  $x$  .  
**FIX(X)** Returns integer found by truncating  $x$  (truncation discards any part of  $x$  to right of decimal, without rounding).  
 $X$  must be in range from  $-32767.9$  through  $32767.9$ .  
**INT(X)** Returns largest integer less than or equal to  $x$  .  $X$  must be in range from  $-32766.9$  through  $32767.9$ .  
**RND(X)** For  $x$  less than 1, returns random real number in the range from  $\emptyset$  through  $.999969$ . For  $x$  from 1 to  $32767.9$ , returns random integer from  $\emptyset$  through  $\text{fix}(x)-1$  .  
**SQR(X)** Returns the square root of  $x$  .  $X$  must be greater than  $\emptyset$ .  
**LOG(X)** Returns base 10 logarithm of  $x$  .  $X$  must be greater than  $\emptyset$ .  
**LN(X)** Returns natural logarithm of  $x$  .  $X$  must be greater than  $\emptyset$ .  
**EXP(X)** Returns  $e$  to the  $x$  power.  $X$  must be in the range  $-88.\emptyset295$  through  $89.4159$ .  $e$  is approximately  $2.71828$ .

## String Functions

**ASC(X\$)** Returns the decimal ASCII code for the first character in  $x\$$  . Returns  $\emptyset$  if  $x\$$  is the null string.  
**CHR(X)** Returns the character whose decimal ASCII code is  $\text{fix}(x)$  .  
 $X$  is normally in the range from  $\emptyset$  through 127.  
**FLO(X\$)** Returns the value of the first number found in string  $x\$$  .  
**STR(X)** Returns the number in  $x$  expressed as a string of characters.  
**INS(N,T\$,P\$)** Starting at character position  $\text{fix}(n)$  , searches through target string  $t\$$  for the first occurrence of pattern string  $p\$$  . Returns  $\emptyset$  if no occurrence found, or returns the position in  $t\$$  of the matching sequence's first character.  
**LEN(X\$)** Returns the number of characters currently stored in  $x\$$  .

## Input Functions

**PDL(X)** Returns an integer from  $\emptyset$  through 255 indicating the position of the knob on game control ("paddle") number  $x$  . The first two game controls are usually numbered  $\emptyset$  and 1.  
**BTN(X)** Returns 1 if the button on game control number  $x$  is being held down, or returns  $\emptyset$  if that button is not being pressed.  
**KEY(X)** Returns 1 if any keyboard key has been pressed since the last Accept or Wait instruction ended; returns  $\emptyset$  if no key has been pressed. The value of  $x$  is not used.  
**TIM(X)** Returns the number of seconds the student took to respond to the last Accept or Accept Point instruction. If Accept was ended by `pr:tn` timeout, `tim(x)` returns  $\emptyset$ . The value of  $x$  is not used.

## Operators

Most operators are placed between the two elements that are used in carrying out that operation. A numeric element (shown as  $x$  or  $y$  ) may be a number, a simple numeric variable, an element of a numeric variable array, or an expression whose result is a number. Except as noted in the descriptions, the value of any numeric element can be any valid real number. A string argument (shown as  $x\$$  or  $y\$$  ) may be a

quoted string, a string variable, a substring variable or an expression whose result is a string.

The standard operators are listed in the precedence table, which follows. This section discusses only items that might not be obvious.

- X \*\* Y**      The result of this operation is  $x$  raised to the power  $y$ . A result that would exceed the range  $-6.8E38$  to  $6.8E38$  causes an error. If  $y$  is  $\emptyset$ , result is 1, except if  $x$  is  $\emptyset$  the result is always  $\emptyset$ . If  $x$  is negative,  $y$  must be an integer in the range  $-32767$  to  $32767$ , and result is positive for even  $y$ , negative for odd  $y$ .
- X = Y**  
**X > Y**  
etc.      Relational comparisons form assertions, such as "x is equal to y." When  $x$  and  $y$  are evaluated, if the assertion is true, result is 1; if the assertion is false, result is  $\emptyset$ .
- X\$ = Y\$**  
**X\$ > Y\$**  
etc.      String assertions compare ASCII codes of the first character in each string. If the two numbers are different, assertion is evaluated for those numbers as true (result 1) or false (result  $\emptyset$ ). If the two first numbers are equal, the second character in each string is compared the same way, and so on. The two strings are made equal in length by adding spaces (ASCII 32) to the shorter one. Two strings are considered equal only if all the character-pairs compared are equal.
- NOT X**  
**X AND Y**  
**X OR Y**      The result of `not x` (or `^x`) is 1 if  $x$  is  $\emptyset$ , or the result is  $\emptyset$  if  $x$  is non-zero. `X and y` (or `x&y`) is an assertion that "x is non-zero AND y is non-zero." `X or y` (or `x!y`) is an assertion that "x is non-zero OR y is non-zero, or both are non-zero." If the assertion is true, the result is 1; if the assertion is false, the result is  $\emptyset$ .
- X\$ !! Y\$**      The concatenation operator joins string `y$` to the end of string `x$`, making one string out of two.

## Order of Evaluating Operations

Here is the order in which the various operations are carried out in an expression:

1. All variables are replaced by the values stored in them.
2. Functions are evaluated (this may require evaluating an expression that appears as an argument).
3. Operations within the innermost set of parentheses are evaluated before proceeding to the surrounding expression.
4. When the order of the operations in any part of the expression is not set by parentheses, the remaining operations are carried out in the following sequence:



	NOT	Logical NOT	not x	
	^	Logical NOT	^x	
First:	-	Unary Minus Sign	-x	
	+	Unary Plus Sign	+x	
Second:	**	Exponentiation	x**y	
Third:	*	Multiplication	x*y	
	/	Division	x/y	
	+	Addition	x+y	
Fourth:	-	Subtraction	x-y	
	!!	Concatenation	x\$!\$y\$	
	=	: Equal	x=y	x\$=y\$
	<>	: Not Equal	x<>y	x\$<>y\$
	<	Relational : Less Than	x<y	x\$<y\$
Fifth:	>	Comparisons : Greater Than	x>y	x\$>y\$
	<=	: Less Than or Equal	x<=y	x\$<=y\$
	>=	: Greater Than or Equal	x>=y	x\$>=y\$
	AND	Logical AND	x and y	
Sixth:	&	Logical AND	x&y	
	OR	Logical OR	x or y	
	!	Logical OR	x!y	

Operations of equal precedence are executed from left to right. There can be up to ten operations "on hold," awaiting the outcome of another operation before they can be evaluated. Open parentheses are counted as a waiting operation.

## The Parts of an Instruction

The parts of an SuperPILOT instruction should appear in the following order [items in square brackets are optional]:

```
[*label] instruction name[modifiers&conditioners][ (expression) ]:[object]
```

### Label

Any name up to six characters long identifying an instruction or lesson section. Where the identifying label is placed in the lesson, it is preceded by an asterisk ( \* ) and separated by a RETURN or space from the following instruction name. A branching instruction can then use the label name (without the \* ) in its object field as the destination of a jump.

### Instruction Name

Required except when continuing instruction from previous line. The one-, two-, or three-letter instruction names, and the identifying word we can associate with them, are:



R Remark	J Jump	D Dimension	G Graphics
T Type	U Use	C Compute	GX eXecute Graphics
	E End		AP Accept Point
PR PRoblem	L Link	K Keep	TS Text Specify
A Accept	XI eXecute	FOX Open New File	TX eXecute Char-Set
M Match	Indirect	FIX Open Old File	S Sound
	W Wait	FO File Output	SX eXecute Sound
		FI File Input	V External Device

## Modifier

One-letter element, in list of modifiers and conditioners that follows the instruction name and precedes the expression (if any) and the colon. Each modifier has a specific effect on one instruction. Used with other instructions, that modifier has a different effect or is ignored.

TH: (Hang modifies Type)	No carriage return after a Typed line.
AX: (eXact modifies Accept)	Accept exact response; no editing.
AS: (Single modifies Accept)	Accept a single-character response.
AP: (Point modifies Accept)	Accept x,y point from game controls.
MS: (Spell modifies Match)	Allow one-letter misspelling on Match.
MJ: (Jump modifies Match)	Jump to next Match if this Match is unsuccessful.
LX: (Erase modifies Link)	Start a new lesson without preserving the old lesson's variables.
LP: (Pascal modifies Link)	Run the Pascal program named in the object text of the instruction.
KS: (Save modifies Keep)	Write data accumulated from Keep instructions onto system.log file.

## Conditioners

One-letter or one-digit element, in list of conditioners and modifiers that follows the instruction name, or an expression. Each conditioner establishes a test for use with any instruction, which is skipped unless each named condition is met. Up to four one-letter or one-digit conditioners may be used effectively, in any order, one from each type:

Y (Yes conditioner)	Execute if last Match was successful.
N (No conditioner)	Execute if last Match was unsuccessful.
1 to 99 (Answer-Count conditioner)	Execute if the conditioner number equals the current Answer-Count.
E (Error conditioner)	Execute if Error Flag has been raised.
(expr) (Expression Cond.)	Execute if expression value is non-zero.
C (Last-expression conditioner)	Execute if last evaluated expression conditioner was True (non-zero).

One arithmetic or one assertion expression, in parentheses, may be used with any instruction, and must immediately precede colon. Instruction is skipped unless the evaluated assertion is true, or the evaluated expression is non-zero.

Arithmetic Expression:	(t-13)	Execute if result is non-zero.
Assertion Expression:	(t>13)	Execute if assertion is true.

## Colon

Every instruction line must contain a colon after the instruction name, modifier and conditioner list (if any), and expression (if any). Colon as the first character on a line continues the previous instruction up to 250 characters total.

## Object

Depends on the instruction. Preceding spaces are allowed. Type instruction displays those spaces; other instructions do not.

## Miscellaneous

Every instruction, including the last instruction in a lesson, must end with a RETURN character. Instructions may be typed in any mixture of upper- and lowercase letters, with these exceptions: the object text of both Type and Match will be used exactly as typed. Blank lines are allowed between different instructions, and are ignored. Only the first 250 characters of an instruction (not counting continuation colons) are used when the instruction is executed.

# The SuperPILOT Instruction Set

---

## Text Instructions

Remark	r:THIS IS A COMMENT.	Comments to author; not executed.
Type	t:What is an apple?	Displays object text on the screen.
	t:#n is right, \$\$\$.	Text may include variable names.

## Response Instructions

Problem	pr:egt20	Starts new section and sets options (if any option is set, any options not mentioned are turned off):
Escape	pr:e	If any response begins with @ , Uses subroutine labelled sysx .
Goto	pr:g	If a response begins with goto , uses remainder as object for Jump.

Lowercase	pr:l	Converts responses to lowercase.
Uppercase	pr:u	Converts responses to capitals.
Spaces	pr:s	Removes all spaces from responses.
Time	pr:tx	Sets a maximum response time of x seconds (t $\emptyset$ =default, unlimited).
Wipe-labels	pr:w	"Forgets" all previous labels, from this point forward.
	pr:	Starts new section without changing any previously selected options.
Accept	a: a:#n \$1\$	Accepts student's response. May assign response to variable(s).
Match	m:COMPUTER m:apple!banana	Looks for object text in student's last Accepted response.

## Control Instructions

Jump	j:next j:@m	Branches to label, next pr: (@p), next m: (@m), or last a: (@a).
Use	u:score u:@p	Branches usually to labelled subroutine that ends with e: . "Remembers" place to return to.
End	e: e:@a	Returns to instruction following u: that called subroutine, or returns and then branches like j: . If not called by u: , ends lesson.
Link	l:lesson2 l:lesson2,review	Starts new lesson. May start at a specified label.
eXecute Indirect	xi:q2\$	Executes contents of string variable as an SuperPILOT instruction.
Wait	w:5	Pauses specified number of seconds or until any key is pressed.

## Computation Instructions

Dimension	d:q2\$(4 $\emptyset$ ) d:r(4,5)	Sets maximum string size for string variable or maximum subscript for numeric variable array.
Compute	c:q2\$="APPLE" c:m=417.95 c:h3=cos(flo(a\$)-2)  c:/q2\$ /@/ c	Evaluates an arithmetic or string expression, and stores it in the variable to left of equals sign.  Edits contents of specified string variable, with these options:

LowerCase	c:/s\$ l	Converts all letters to lowercase.
UpperCase	c:/s\$ u	Converts all letters to capitals.
Capitalize	c:/s\$ c	Capitalizes first character, if it is a letter.
Delete	c:/s\$ /c/	Removes every character c .
Replace	c:/s\$ /xy	Replaces every character x with the character y .

## Special-Effects Instructions

Graphics	g:c6;p23,40;d98,13	Executes the graphics commands that appear in the object field:
Viewport	g:vl,r,t,b	Sets left, right, top, and bottom boundaries for text viewport. Default is full screen. Columns = 0 to 39 (left to right); rows = 0 to 23 (top to bottom).
Erase Screen	g:v g:es g:esx	Resets full screen text viewport. Erases text viewport to the set text background color. Erases screen to graphics color x (0-10). 0=Black1    4=Black2    8=Reverse 1=Green    5=Orange    9=1 Dot On 2=Violet    6=Blue    10=1 Dot Off 3=Whitel    7=White2
Color	g:cx	Sets drawing color x (0 to 10).
Offset	g:ox,y	Sets absolute x,y as reference point for relative locations; moves pen there. Graphics x=0 to 560 (left to right), y=0 to 511 (bottom to top).
Angle	g:ax	Sets angle for relative movement G: commands. X in degrees.
Spin	g:sx	Adds x degrees to current angle for relative movement commands.
Move	g:mx,y g:mx	Moves pen to relative x,y point. Moves pen a distance x in the set direction.
Plot	g:px,y g:px	Plots point at relative x,y . Plots point at distance x in the current set direction.
Quit	g:qx,y g:qx	Puts black point at relative x,y . Puts black point at distance x in the current set direction.
Draw	g:dx,y g:dx	Draws line to relative x,y . Draws line of length x in the current set direction.
Redraw	g:rx,y g:rx	Draws black line to relative x,y . Draws black line of length x in current set direction.

Repeat-factor	g:*x(...)	Executes commands in parentheses x times.
Type	g:ttext	Places text on screen starting at current graphics cursor position.
Xmit	g:xn1,n2,n3	Transmits screen-control characters whose ASCII codes are n1,n2,n3 .
eXecute Graphics	gx:map gx:photo2!	Draws named image, step-by-step. Quickly replaces the screen with complete image in named file.
Accept Point	ap:	Accepts student pointing input from game controller or other device.
Type Specify	ts:20(ax\$;d25;wr)	Determines how and where text will appear on screen; background and foreground colors; type styles and line spacings.
Viewport	ts:vl,r,t,b	Sets left, right, top, and bottom boundaries for text viewport. Default is full screen. Columns = 0 to 39 (left to right); rows = 0 to 23 (top to bottom).
Goto	ts:gx,y	Move text cursor to position x,y .
Size	ts:sx	Single (x=1) or double (x=2) size.
Thickness	ts:tx	Sets character thickness, where x=1 is normal, x=2 boldface.
Mode	ts:mx	Sets type style, where x=1 is normal, x=2 is overstrike, x=3 is exclusive-or.
Transmit	ts:xn	Sends any character to screen; n is that character's ASCII number.
Line Spacing	ts:lx	Sets double-space, triple-space, etc., where x is 1 through 15.
Print	ts:p	Sends future characters to printer.
Quiet	ts:q	Turns off flow to printer.
Animate	ts:ax\$	Executes fast, unformatted write of string variable x\$ .
Walk	ts:w[lrud]	Moves text cursor one character space left, right, up, or down.
Delay	ts:dx	Pauses x/60 seconds. Used to pace animation sequences.
Background	ts:bx	Sets background text color, where x is from 0 through 20.
Foreground	ts:fx	Sets foreground text color, where x is from 0 through 20.
Inverse	ts:i	Future characters are printed in background color on a background that is the foreground color.
Normal	ts:n	Cancels Inverse ts:i instruction.
Erase Screen	ts:es ts:esx	Erases text viewport in the set text background color. Erases entire screen in graphics color specified by x (0-10).

Repeat-factor	ts:*x(...)	Executes commands in parentheses x times.
eXecute char-set	tx:cyrillic tx:	Starts using named character set. Goes back to standard character set.
Sound	s:2,12;4,24;3,12	Plays sounds of pitch,duration . Pitch = 1 (low) to 50 (high) Pitch = 0 is silent (rest) Duration = 0 (short) to 255 (long)
eXecute Sound	sx:tuneup	Plays sound effect stored in named diskette file.
External- DeVice	v:[commands]	Activates peripheral equipment such as videotapes and videodisks.

## File Handling Instructions

Keep	k:	Transmits object text or variables to a file named system.log .
Open new file	fox:n,score fox:0,score	Creates and opens new file score , to hold records 0 through n . Finds existing file score , and deletes it from the diskette.
Open existing file	fix:n,score	Opens existing file score , for reading or writing records 0 through n .
File Output	fo:n,high: 37	Stores string high: 37 in record n of the file now open.
File Input	fi:n,x\$	Reads record n of file now open; stores result in variable x\$.

## Appendix E

# Differences From Apple PILOT

- 290 Language Extensions
  - 290 The TS: Instruction
  - 290 Recordkeeping
  - 290 External Device Control
  - 291 Immediate Execution Mode
  - 291 Relative Graphics
  - 291 Additional Language Extensions
- 292 Operating System Extensions
- 293 Converting Apple PILOT Lessons

Apple SuperPILOT is an extension and improvement of Apple PILOT, which has its roots in Common PILOT and earlier versions of the PILOT language. Despite the many differences between SuperPILOT and its predecessors, this new version of the language is compatible with Apple PILOT and Common PILOT: any lesson that runs on either of these PILOT systems will likely run in SuperPILOT without modification. There are only four minor exceptions, discussed at the end of this section.

This appendix summarizes the extensions of SuperPILOT beyond the limits of Apple PILOT. The descriptions are brief, as they are meant to serve only as a quick summary for those who are already familiar with Apple PILOT. For fuller descriptions of the new features and their operation, you are encouraged to read the appropriate sections of this manual or the accompanying Apple II SuperPILOT Editors Manual.

## **Language Extensions**

---

### **The TS: Instruction**

The Type Specify instruction provides a much greater variety in the presentation of text material on the screen, and also allows for greater control over the appearance and even the movement of that text. There are twelve different commands that may appear in the object field of a TS: instruction, and they separately perform functions such as setting the text background and foreground colors, changing character size and style, simulating movement (animation) of shapes created in the Character Set Editor, setting multiple line spacing, and sending text to a printer. See the chapter Text Instructions for details on these commands and their use.

### **Recordkeeping**

It is now possible, by using a new Keep instruction, to keep records of student performance automatically. If you create a special file named system.log, and if that file is on the student's lesson diskette or on a diskette named SYSLOG that is in one of the system's disk drives when a lesson diskette is first booted, that file will automatically store the beginning, end, and link points of the lesson, as encountered, plus additional information you instruct it to keep (for example, response times, intermediate scores, variable values at different points, and so forth). This file can be analyzed and summarized by Apple SuperPILOT LOG, an optional statistical package available from your Apple dealer. The recordkeeping option is explained in full in the chapter File Handling Instructions.

### **External Device Control**

By using another new instruction, V:, you can now communicate with external devices such as videodisks and videotapes. See the Special Effects Instructions chapter.



## Immediate Execution Mode

Your ability to experiment with SuperPILOT instructions and to make temporary patches in lessons you are testing has been increased through the development of a true immediate mode. You can enter Immediate Mode from the Lesson Text Editor or from any response instruction during the running of a lesson in Author Mode. Pressing CTRL-I moves you into and out of this mode. The Apple II SuperPILOT Editors Manual contains a complete section on the use of Immediate Execution Mode.

## Relative Graphics

Apple PILOT relies on absolute coordinate movement in Graphics Mode: you specify the x,y screen position for each command. SuperPILOT, however, has added relative (turtlegraphics) movement, for speed and simplicity. There are two new commands: one sets an absolute angle that is used as the direction of graphics movement, and the other sets a relative angle (or spin) to change that direction in incremental steps. Five of Apple PILOT's existing Graphics commands, Move, Plot, Quit, Draw, and Redraw, can now be given a single numeric value (instead of x,y coordinates) that SuperPILOT executes as a length or distance in the set angle direction. Repeat-factors are allowed with the turtlegraphics commands as well. These new commands are discussed and compared with the absolute coordinate commands in the chapter Special Effects Instructions.

## Additional Language Extensions

Besides the five major improvements listed above, there are several additional extensions worth noting:

1. Negative variables, signed numbers, and mathematical expressions (in parentheses) are now accepted wherever simple variables may be used, with two exceptions: file (fi/fo) record numbers, and subscripts. For example: `g:m0,(y+3)` .
2. A string literal is now acceptable wherever a string variable may be used, with four exceptions: as the object of an Accept, Execute Indirect, or File Input instruction; or when assigning a value to a string variable in a Compute instruction. A string literal is terminated by either a comma or a semicolon, or the end of a statement. For example, `fix:3,myfile` will open the file, `myfile` , but a string variable must be used to retrieve a record from it, as in `fi:l,a$` .
3. Type instructions may now contain expressions in addition to simple string and numeric variables. A left parenthesis immediately after a pound sign ( # ) causes the enclosed expression (up to the corresponding right parenthesis) to be evaluated and printed. For example: `t: #(n+m)` .
4. Coordinates for all graphics commands may be positive or negative. This means, for example, that you can now use the instruction `g:o280,256` to put the offset at the center of the screen, and then use the entire screen for four quadrant Cartesian geometry.

5. Previously edited line-drawn graphics (called by GX: instructions) are now affected by the graphics offset position. Multiple copies of a drawing can therefore be placed at different points on the screen by simply changing the offset. Quick-draw images remain unaffected by the graphics offset, however, since they simply replace the screen contents.
6. An additional system variable has been added %o , to allow control of the game I/O annunciator ports. The instruction c:%o(n)=expr will activate game port n if expr<>0 and will deactivate game port n if expr=0 , where n is the port number, 0-3.
7. Two other system variables have been added to store the text screen coordinates of the text cursor. The %c system variable stores the vertical (or column) position (0-39) and the %r system variable stores the horizontal (or row) position (0-23).
8. The function KEY(0) formerly returned 1 if a key was being pressed; it now returns the decimal ASCII code number (1-127). KEY(0) still returns 0 if no key is being pressed.
9. The function ASC(x\$) now returns 0 if the length of x\$ is 0.
10. The variable memory space has been expanded over twofold. There are now 200 numeric variable locations and 1600 string variable locations available for Dimensioning, plus additional space for use by SuperPILOT.
11. Subroutine nesting is now permitted to ten levels.
12. Multiple commands are now acceptable in Dimension and Compute instructions, as they have been in Graphics instructions. In the same manner, they are separated by semicolons, as in

```
d:a$(20);b$(255);n(3,4)
```

or

```
c:a$="Good";n(0,3)=4;b$="PILOT!"
```

## Operating System Extensions

The significant increase in Graphics capability of SuperPILOT over Apple PILOT is made possible by the addition of a new high-resolution screen driver. In addition, the operating system has been expanded to take advantage of the increased capacity of the language card. SuperPILOT operates only in this 64K environment. Except for the addition of the language card, the system requirements for SuperPILOT are identical to those of Apple PILOT:

Author System:

48K Apple II or Apple II Plus  
16K Language Card  
Two Disk II Drives

Student System:

48K Apple II or Apple II Plus  
16K Language Card  
One Disk II Drive

## Converting Apple PILOT Lessons

---

If you have written lessons under the Apple PILOT system that you want to run under SuperPILOT, you may do so with very little chance that modifications will be needed. There are four minor changes from PILOT that could cause your previous lessons to run differently under SuperPILOT:

1. In a Graphics instruction consisting of multiple commands, Apple PILOT does not move the text cursor to the graphics cursor position until the end of the instruction. SuperPILOT, however, resets the text cursor to the graphics cursor position at the end of each command. So, if your Apple PILOT lesson contains an instruction such as the following,

```
g:m100,200;tmessage
```

Apple PILOT prints the word `message` at the graphics cursor position before the move is executed; SuperPILOT prints it at the graphics cursor position after the move is executed.

2. Apple PILOT allows spaces to be substituted for commas between coordinates in graphics commands, while SuperPILOT does not. For example, the instruction

```
g:m100 200
```

is acceptable in Apple PILOT, but in SuperPILOT it causes an error message in Author Mode and raises the Error Flag in both Author Mode and Lesson Mode. This change was made necessary because SuperPILOT supports relative graphics movements, which are specified by only a single number after the command name.

3. The SuperPILOT graphics cursor now moves to the screen location accepted by an `AP:` instruction. With Apple PILOT, the graphics cursor is not affected by `AP:` instructions.

4. In Apple PILOT three options of the PROblem instruction--the Spaces, Uppercase, and Lowercase options--were ignored by an Accept instruction that was modified by the Single modifier. An as: instruction in SuperPILOT no longer overrides these three options. Note that the S , L , and U options of the Problem instruction will still be overridden by the Accept instruction's eXact modifier, as in ax: or asx: .

## Appendix F

# Apple II Colors

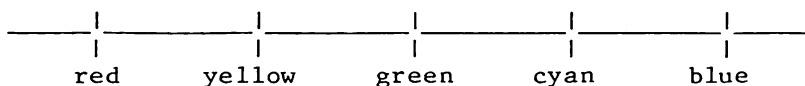
- 296 Natural Colors and Apple II Colors
- 297 Where Colors Are Displayed on the Screen
- 299 Where the Colors Come From

As you wander deeper into SuperPILOT and other Apple II graphics systems, you will begin to notice that the high-resolution graphics screen does not always act as you might expect. It acts almost all right, but not quite: occasionally, carefully drawn vertical lines refuse to be visible; a dainty white line crossing a field of green leaves jagged blocks of orange in its wake. The purpose of this appendix is to explain the subtle pattern of rules underlying what at first glance appeared to be a simple screen.

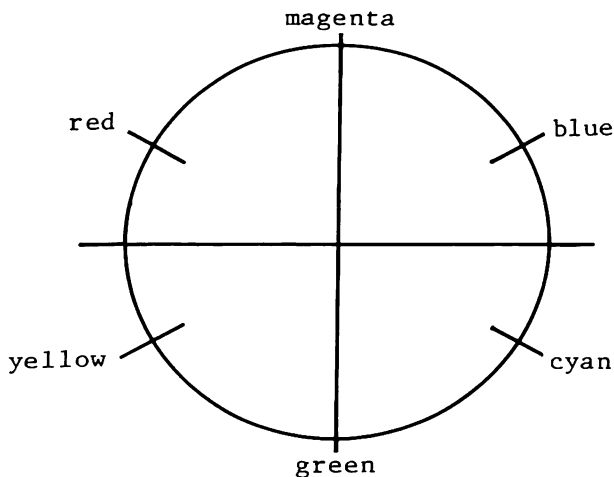
## Natural Colors and Apple II Colors

---

Below is a line representing the color spectrum. Along with the color names that most of us learned in school you may find, perhaps, a new one: cyan. Cyan lies between green and blue and is commonly called blue/green or turquoise.



If we now bend the spectrum up until the two extreme ends meet, we produce a color wheel, as pictured below. The point at which the ends meet is another "color" they never told us about: magenta. Magenta appears to exist where we connected the low-frequency (deep-red) and high-frequency (violet) ends of the visible spectrum. However, it has the distinction of being all in our minds: one cannot isolate magenta with a prism; it splits into its red and violet components.



The ends of the two intersecting lines represent the four Apple II hi-res colors. The SuperPILOT names for these colors are not all exactly correct. SuperPILOT Green is the same as natural green, and SuperPILOT Orange is midway between natural red and yellow, as it should be. But

Violet is used by SuperPILOT instead of the less familiar color name magenta, and Blue is used to describe the color on the right of the wheel, halfway between natural blue and cyan.

So when you see references in this manual to SuperPILOT's four basic colors--Green, Violet, Orange, and Blue--remember that they are actually the four points on this color wheel: green, magenta, red/yellow, and blue/cyan. (Note: Manuals for other products that use the Apple II's colors may refer to magenta as Purple, or by some name other than Violet; regardless of the name used, however, the color being produced is magenta.)

When you set the hue or tint control on your own TV, adjust it while looking at Violet (SuperPILOT color number 2). All four colors are as accurate as possible when the screen color is right between natural blue and red.

Armed with a sufficiently wide-range hue control, you need not be limited by these four particular colors. When you adjust the hue control, it's as if you rotate the two intersecting lines in the above color wheel: the four ends will point to new color values that will be produced when you specify SuperPILOT's Green, Violet, Orange, or Blue. The range of variation depends on the design of your particular TV. (Many newer sets have a "magic button" of one sort or another, one action of which is to limit the range of the hue control--turning off this feature may help you explore extra colors.)

We have discussed this at such length because you may be writing software in an environment where you can control what colors students will see. But when you do not have control over the TVs' settings, keep in mind that their hue controls may be set incorrectly. It would be hazardous to teach children what the color magenta is by preparing what you expect to be a magenta screen: what the child is presented with on her school's ten-year old TV could be anything from blue to pink.

## **Where Colors Are Displayed on the Screen**

---

Now that you know what the Apple II's colors look like in relation to natural colors, we're ready to explore where the graphics screen can and does display the different colors. That is the subject of this section. Later, we will dig even deeper to examine how the Apple II actually tells the screen which colors to display.

The Apple II's high-resolution graphics screen consists of 192 rows of 280 dots each, for a total of 53,760 individual dots. To have the power to make any dot capable of being any of the six Apple II colors (the four described above, plus Black and White) would require over 20,000 bytes of memory. (A byte, as you may already know, is the amount of memory needed to store one typewritten character of information. Each byte of Apple II memory consists of eight bits, the most elemental measure of computer memory.)

If the Apple II used these 20K bytes of memory just for graphics, there would be far less room for SuperPILOT's other features, not to mention your lessons. Fortunately, Apple found a method that would let us keep the full range of colors, the fully-detailed resolution of the screen, and at the same time cut the memory down to a more reasonable 8K bytes.

For each of the 53,760 dots on the graphics screen, there is only one bit of memory. And since a bit can be only a zero (off) or a one (on), it would stand to reason that any individual screen-dot can be only off (not lit up) or on (lit up). Therefore, all hi-res graphics should be black-and-white.

And hi-res graphics would be black and white, if there weren't certain overlapping electronic "rules" being applied. To discover these rules, let us first deal only with the two colors Violet and Green. We will return to Orange and Blue later. It is relevant to note that Violet and Green (like Orange and Blue) are complementary colors: they appear directly opposite each other on the color wheel. (Remember, we are using SuperPILOT's color names to describe the natural colors that are found in these positions on the color wheel.)

The hi-res screen's 280 vertical columns are not created equal: every other column is either a Violet column or a Green column. It is as though there were a filter over the front of a black and white TV, a filter consisting of 280 alternating stripes of Violet and Green material. If a dot is lit behind one of the Violet filters on an even-numbered column (0, 2, 4, 6, 8, ...278), you will see Violet; if a dot is lit behind one of the Green filters on an odd-numbered column (1, 3, 5, 7, 9, ...279), you will see Green.

White is the presence of at least two lit dots next to each other: if two adjacent dots are both lit, both dots appear White. This happens because the complementary colors Violet and Green mix together inside your color TV to make one elongated White "dot."

On the other hand, Black is the presence of at least two unlit dots in a row: one unlit dot is not enough. If, for example, dots 0, 2, and 4 are lit, you might expect to see three Violet dots alternating with two Black dots. Instead, you will see a solid, Violet line. Your eyes are not playing tricks, your color TV set is. A computer monitor (either black and white or color) will clearly show the true on-off pattern.

So far, we've discovered the following rules:

1. Violet exists only on even-numbered columns.
2. Green exists only on odd-numbered columns.
3. Black areas must be at least two dots wide.
4. White areas must be at least two dots wide.

This color scheme gives hi-res graphics several strange traits, many of which you may have discovered already. For example, drawing a Single-dot On line (graphics color number 9) vertically on a Black screen will produce a colored line: Violet on an even column, Green on an odd column. You can see this graphically by using SuperPILOT's Accept



Point instruction to display the vertical and horizontal cross-hairs on the screen. As you turn the knob on hand controller Ø, the vertical line will change from Violet to Green to Violet. The horizontal line will always be White because any two or more adjacent horizontal dots turned on will be White.

On many older sets, drawing a Single-dot Off line vertically through a solid White area will produce a color line. The line will be Violet if the Black was drawn on the Green column; it will be Green if the Black was drawn on the Violet column.

As mentioned above, any time a Green dot and a Violet dot next to each other are both lit, they both are turned to White, appearing as a single, long, White dot. This may be particularly disturbing when using text colors. Many foreground-background combinations produce scattered White dots where foreground meets background. Use of White- or Black-stripe colors (numbered 8-16) as backgrounds and foregrounds alleviates this problem.

We now have accounted for four colors: Green, Violet, White, and Black. The way the Apple II gets the extra two colors Orange and Blue is simple: it rotates its digital equivalent of your TV's hue control 90 degrees until Violet becomes Blue and Green becomes Orange. In effect, it has exchanged the Violet filter for a Blue filter and the Green filter for an Orange filter. Simple. Unfortunately, how the Apple II knows when to make Violet into Blue or Green into Orange is not so simple.

## Where the Colors Come From

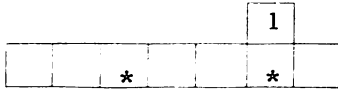
---

The 280 dots on each horizontal line are stored away in 280 bits of memory, held in 40 bytes of memory with seven bits per byte. Those of you who recall that there are eight bits in a byte, not seven, will instantly recognize that we seemed to have dropped a bit. The answer to the mystery of the extra colors lies within this unsung bit. In fact, this extra bit is responsible for the Apple II's ability to rotate its internal equivalent of the hue control, thereby producing Orange and Blue from Green and Violet.

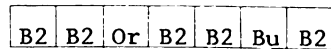
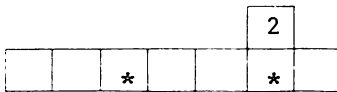
The extra bit is used to select one of two color groups: group 1 is made up of Black1, Green, Violet, and White1 (graphics colors numbered 0-3), and group 2 is made up of Black2, Orange, Blue, and White2 (numbers 4-7).

Every seven bits have a controlling bit that selects color group 1 or color group 2. This bit acts like a switch to select the proper pair of filters for the screen-dots that correspond to that specific byte. For example, a byte is shown below, on the left. The asterisk indicates a bit that is on, while a blank indicates a bit that is off. The controlling bit appears above the other seven, set to 1 (color group 1). On the right is a representation of how the corresponding seven screen-dots would appear, using color abbreviations for Black1,

Green, and Violet (assume that the first bit corresponds to an odd numbered screen-dot).



The color you choose to draw with carries an identification as to its group. Violet, for example, has an identifier that tells the computer it is in color group 1. Every time a single new dot is drawn within a given row of seven screen dots, the controlling bit is set to the same color group as the identifier of the color with which you are drawing. So, if we were to draw an Orange line vertically through this row so that it passes through the Green dot, here's what the result would be (color abbreviations are for Black2, Orange, and Blue):



When the Orange line reached this row, the controlling bit was turned to 2, so that Orange (which is in color group 2) could be displayed. But this action caused all of the Green-Violet filters in that row to be switched to Orange-Blue filters. Therefore, the Violet dot changes to Blue. The Black1 dots also change to Black2, but that is not noticeable, of course.

So, placing a Blue dot (color group 2) on a long Green horizontal line (color group 1) will change all the Green dots within that byte to Orange. Creating an Orange background (color group 2) in the Graphics Editor and drawing a diagonal line across it using White1 (color group 1) will create a jagged line of Green boxes. This problem can be avoided by selecting White2 when traversing a color group 2 area.

Many Apple II graphics systems make an effort to release you from having to worry about all these rules all the time, using techniques such as making Black or White vertical lines two dots wide automatically so they will not show up in color. The Apple SuperPILOT color character generator has a high-degree of intelligence, avoiding color conflicts whenever possible and offering nonpartisan Blacks and Whites which do not interfere with their neighboring color dots.

When you believe you understand the way Apple II graphics colors work, use the SuperPILOT Graphics Editor to purposely create anomalies and then figure out which rules they are displaying. After a short time, you will develop a deep associative understanding of how the screen is functioning. Armed with that, you will find that the potentially restrictive behaviors will simply cease to get in your way: you will be able to create and control virtually any kind of drawing, with the

full power to get any color combinations you want within seven dots of the ideal horizontal location.

Also, keep in mind that nothing on any horizontal line has any effect whatsoever on anything on any other horizontal line. And remember what your extra effort is buying you: the Apple II six-color high-resolution screen is fitting in a remarkably small space, leaving SuperPILOT the space to provide you with powerful Editors and you the space to write sparkling friendly interactive software for your student audience.



# Index

## A

A: Accept instruction 53-63, 160-165, 194, 197, 200, 205, 294

A Absolute angle command, Graphics 117

A Animate command, Type Specify 134, 138

ABS(X) absolute angle function 232

Absolute angle command, Graphics 117

Accept instruction 53-63, 160-165, 194, 197, 200, 205, 294

answer counting 15, 56, 207

effect of Problem instruction 49

entry to Immediate Mode 61

modifiers 60, 160

response editing 55

response timing 54, 161, 241

unlabelled destination 61

variables 57, 205

Accept Point instruction 160-165, 241, 293

addition operator 102, 242

Agnew, Spiro 148

ampersand 65, 245

AND logical operator 245

"and" Match controller

character 65

angle functions 231

angles 231

Animate command, Type Specify 134, 138

animation of text 134, 137-143

Answer Buffer (system variable %B) 53, 57, 194, 198, 226, 279

Answer Count (system variable %A) 16, 56, 195, 198, 225, 279

Answer-Count conditioner 14, 15, 57, 208, 225

answer counting 15, 56, 207

AP: Accept Point instruction 160-165, 241, 293

Apple Pascal see Pascal

Apple PILOT-SuperPILOT differences 2, 174, 231, 290-294

arctangent function 232

arguments of functions 229, 247

arithmetic expressions 246

arithmetic functions 232, 279

arithmetic operators 242

array, numeric

see numeric variable array

ASCII characters and codes 132, 152, 234, 240, 243, 252-258

ASC(X\$) character-to-code function 234

assertion 21, 243

assignment, in Compute 99

assignment indicator (=) 99, 244

asterisk

label indicator 8, 72, 75

multiplication operator 242, 247

Remark lines 36

repeat-factors 120, 136

string literal 85

wildcard symbol 65

ATN(X) arctangent function 232

Audio/visual Device Control Instruction 165

Author Mode 29, 53, 61, 129, 169, 200, 260, 265, 268

Author System 293

## B

B Background Text Color

command, Type Specify 135

backspace 115, 253

bees 134

bell or beep 253

black-and-white monitor 145

black-and-white TV 145

blank lines 29

blank Remark lines 36

blank spaces with animation 140

blank Type lines 38

blocks, diskette 175, 179

boldface type 132

BTN(X) game button function 239

Buffer, Student Answer (system variable %B) 53, 57, 194, 198, 226, 279

button, game control 160, 239

## C

- C Capitalize option, Compute 104
- C Color command, Graphics 116, 126
- C Last-expression conditioner 14, 24
- C: Compute instruction 98-106, 205, 207
- CAI 2
- calculations 98, 242
  - in SYSX Escape routine 198
- capital letters 3, 5, 28, 49, 55, 104, 178, 182
- Capitalize option, Compute 104
- character codes, ASCII 234, 252-258
- Character Set Editor 134, 138, 151, 258
- character sets 115, 151
- character variables 222
- CHR(X) code-to-character function 234
- clear screen 115, 253
- clear to end of line 255
- click 156
- closing a file 176
- colon 2, 3, 25, 284
  - see also continuation colon
- Color command, Graphics 116
- color in graphics 122, 126
- color in text 135, 143
- color spectrum 296
- color TV 145
- color wheel 296
- colors 296-301
  - natural 296
  - unfriendly 144
- column coordinate (system variable %C) 124, 226, 279
- comma
  - between elements in a command 114, 155
  - between object elements 83
  - between numbers in subscript 95, 219, 223
- comments 36, 180
- Common PILOT 2, 175, 185, 290
- compacting a diskette 179
- comparisons of numbers or strings 243
- Compute instruction 98-106, 205, 207
- Computer Assisted Instruction 2
- concatenation 246
- conditioners 6, 13-25, 283
- constants 100, 214
- continuation colon 6, 10, 25, 29, 284
  - Graphic instruction 119
  - Match instruction 64
  - Remark instruction 36, 38
  - Sound instruction 156
  - Type instruction 209
- control characters 115, 252
- control options, PProblem 48
- controller characters, Match 64
- controls, game 160, 239
- conversion
  - ASCII to character 115, 234, 252
  - character to ASCII code 234, 252
  - real number to integer 233
  - number to string 236
  - string to number 235
  - to nonstandard character set 151
- converting Apple PILOT lessons 293
- coordinates, Graphics 61, 116-122, 160
  - absolute (Cartesian) 116
  - relative (Turtlegraphics) 116
- COS(X) cosine function 231
- counting answers 15, 56, 207, 225
- Create and Open New File instruction 176
- cross-hairs, Accept Point 160
- CTRL-A 5, 28
- CTRL-C 32, 33, 53, 200
- CTRL-I 31, 33, 53, 61, 200, 291
- CTRL-Z 5, 28
- cursor
  - graphics 114, 124, 293
  - text 44, 53, 114, 124, 131, 132, 139, 252, 293

## D

- D Delay command, Type Specify 134, 140



D Draw command, Graphics  
     absolute coordinates 117  
     Turtlegraphics 118  
 D: Dimension instruction 41,  
     58, 94-98, 204, 218, 220  
 .DATA file name suffix 174  
 default Graphics color 126  
 Delay command, Type Specify  
     134, 140  
 delays in program 82  
 Delete option, Compute 104  
 deleting data files 178  
 debugging messages 268  
 Dimension instruction 41, 58,  
     94-98, 204, 218, 220  
     with Link 86  
 dimensions of numeric varia-  
     ble array 95, 217  
 disk drives, multiple 260-265  
 disk drive names 261  
 diskette blocks 175, 179  
 diskette files  
     see files  
 diskette names 260  
 display characters 258  
 display modes 255  
 division operator 102, 242  
 DLE indent code 253  
 Dodgers 140  
 dollar sign, with string varia-  
     ble names 28, 42, 58, 94,  
     205, 220  
 double-spacing of lines 133  
 Draw command, Graphics  
     absolute coordinates 117  
     Turtlegraphics 118  
 drawing color, Graphics 114, 126  
 drawing lines 116-118  
 duration, note 155

## E

E Error conditioner 17-20  
 E Escape option, PProblem 33,  
     50, 197  
 E: End instruction 31, 67,  
     75, 77-83, 201  
 E notation 59, 106, 211, 214  
 editing  
     a string with Compute 103  
     of Accepted responses 49, 65  
     of Match text 65

Editor  
     Character Set 151  
     Graphics 129  
     Lesson Text see Lesson  
     Text Editor  
     Sound Effects 156  
 Editor menu 32, 200  
 elements  
     in expressions 247  
     of instructions 7  
     of numeric variable arrays  
         95, 106, 217  
 End instruction 31, 67, 75,  
     77-83, 201  
 equal operator 244  
 equal sign (assignment indica-  
     tor) 99, 244  
 Erase modifier 13, 84  
 Erase Screen in Graphics Color  
     command, Graphics 116, 126  
 Erase Screen in Graphics Color  
     command, Type Specify 135  
 Erase Text Screen command,  
     Graphics 115  
 erasing animation 140  
 error conditioner 14, 17-20  
 Error Flag 17-20  
     lowering 18, 21, 89  
     testing for 61  
 error messages 31, 268-276  
     language 268-272  
     system execution 272-275  
     system failure 275  
 error handling 10, 20, 188  
 Escape command 33, 50, 197-200  
 Escape option, PProblem 50, 197  
 escaping from lesson 200  
 escaping from program loop 201  
 evaluation of expressions 246-  
     249  
 eXact modifier 13, 60  
 exclamation point  
     logical OR operator 245  
     match controller 65  
     quick draw file indicator 129  
     string operator 246  
 eXclusive-OR display mode 132,  
     256  
 eXecute character-seT file  
     instruction 45, 151  
 eXecute Graphics file instruction  
     129  
 eXecute Indirect instruction 88

eXecute Sound file instruction  
158  
execution order, in expressions  
103, 247, 281  
execution-time commands 33-34,  
50, 194-202  
exponentiation 234, 242  
expression conditioner 6, 14,  
21-24  
expressions 43, 101, 246-249,  
278-282  
EXP(X) e-to-the-X function 234  
external device control 290

## F

F Foreground Text Color  
command, Type Specify 135  
false (value is 0) 21, 24, 243  
FI: Input from Open File  
instruction 188-191  
File Input instruction 188-191  
file names 129, 130, 174, 178  
File Output instruction 184-188  
files,  
character-set 151  
data 174  
graphics 129  
lesson 29, 83, 260  
sound 158  
FIX: Open Existing File  
instruction 181-184  
FIX(X) truncation function 232  
FLO(X\$) string-to-number  
function 235  
Foreground Text Color command,  
Type Specify 135  
form of instructions 5-7, 282-  
284  
form of operations and  
expressions 242-249  
formatting screen displays 38,  
123  
FO: Output to Open File  
instruction 184-188  
.FOTO file name suffix 130,  
174  
FOX: Create and Open New File  
instruction 176-181  
frogs 134  
full screen (no viewport) 124

functions 102, 229-241, 279  
arithmetic 232  
input 239  
string 234  
summary 279  
transcendental 231

## G

G Goto option, PROblem 50, 194  
G Goto XY command, Type  
Specify 131  
G: Graphics instruction 114-  
128  
game controls 160, 239  
game port control 227  
goofing off 173  
Goto execution-time command 33,  
67, 194-196  
Goto option, PROblem 50, 194  
Goto XY screen command 255  
Goto XY command, Type Specify  
131  
.GRAF file name suffix 130,  
174  
graph paper 141  
graphics coordinates, Accepting  
160  
graphics cursor 114, 124, 293  
Graphics Editor 129  
graphics files 129  
Graphics instruction 114-128  
colors 126  
command rules 118  
commands 114  
cursors 114, 124, 293  
screens 121-122  
graphics screen 121-122  
high-resolution 122  
SuperPILOT 121  
greater-than operator 244  
greater-than-or-equal operator  
244  
GX: eXecute Graphics instruc-  
tion 129-131, 292

## H

H Hang modifier 12, 44  
Hello lesson 30, 86, 201  
hiding text 145



high-resolution graphics  
  screen 122, 292, 296  
home cursor 254  
Horizontal Coordinate (system  
  variable %X) 118, 124, 226

I Inverse command, Type  
  Specify 135  
Immediate Execution Mode 31,  
  61, 76, 200, 258, 291  
indent 253  
initializing a diskette 260  
Input from open File  
  instruction 188-191  
input functions 239  
instruction  
  format 5-7, 282  
  length 29  
  names 10-12, 283  
  object 26  
  repeating 32  
  summary 284-288  
instruction-modifying  
  expression 21  
INS(N,T\$,P\$) word-in-string  
  function 237  
INT(X) real-to-integer function  
  233  
Inverse command, Type Specify  
  135  
inverse video 253

## J

J Jump modifier 13, 66  
J: Jump instruction 67, 72-74  
Jump instruction 67, 72-74  
Jump modifier 13, 66  
jumping to next section 48  
jumping to labels 50, 72, 194

## K

K: Keep instruction 168-171,  
  290  
Keep instruction 168-171, 290  
Keeping student records 168-  
  171, 290  
KEY(X) keypress function 240

## L

L Line spacing command, Type  
  Specify 133  
L Lowercase option, Compute  
  104  
L Lowercase option, PProblem  
  49, 65  
L: Link instruction 83-88  
label table 9, 51  
labels 8-9, 50, 83, 194, 282  
  number of 50  
language card 292  
language error messages 268-272  
largest-record number 177, 182  
last-expression conditioner 24  
length of instructions 29, 38  
LEN(X\$) length-of-string function  
  239  
lesson diskette 83, 129, 151,  
  158, 174, 260  
Lesson Mode 30, 32, 53, 169,  
  182, 201, 262, 268  
Lesson Text Editor 6, 29, 36,  
  37, 64, 119, 128, 156, 201  
less-than operator 244  
less-than-or-equal operator 244  
limits  
  Accepted numbers 60  
  array-element subscripts 95,  
  character positions 122  
  data file size 177  
  expressions 248  
  functions 229-241  
  graphics coordinates 116,  
  118, 121, 124  
  memory 249  
  numeric variables 216  
  operators 242-246  
  real numbers 60, 214  
  record numbers 177, 182, 185,  
  189  
  response time 49, 54, 162  
  string variables 94, 220  
  subscripts 218, 223  
  variable names 215  
line formatting 38-41  
Line Spacing command, Type  
  Specify 133  
lines, drawing 116-118  
linefeed 253  
Link instruction 83-88

linking with Pascal programs  
84  
LN(X) natural logarithm function  
234  
logical operators 245  
LOG(X) base-10 logarithm function  
233  
loop 201  
lowercase letters 3, 5, 28,  
49, 55, 104, 178, 182  
Lowercase option, Compute 104  
Lowercase option, PProblem 49,  
65

## M

M Modes of Type command, Type  
Specify 132  
M Move Command, Graphics  
absolute coordinates 117  
Turtlegraphics 117  
M: Match instruction 63-69  
Ma 141  
magic 115  
Match instruction 3, 63-69  
editing 65  
Match word 63, 210  
Maxwell 138  
memory space  
for numeric variable arrays  
96  
for variables 84, 292  
minus sign 242  
Modes of Type command, Type  
Specify 132  
modes, screen 255  
modifiers 6, 12, 283  
with Accept 60  
with Graphics 127  
with Link 84  
with Match 66  
with Sound 157  
with Type 44-46  
monitor, black-and-white 145  
Move command, Graphics  
absolute coordinates 117  
Turtlegraphics 117  
multiple disk dives 260-265  
multiplication operator 102,  
242  
music 155

## N

N No conditioner 14, 63  
N Normal command, Type  
Specify 135  
names  
of disk drives 261  
of diskettes 260  
of instructions 11, 283  
of variables 215, 217, 220,  
223  
natural logarithm function  
LN(X) 234  
negative numbers 243  
nesting  
expressions 248  
subroutines 77, 292  
new date files 178  
Nixon, Richard 148  
No conditioner 14, 63  
non-printing control characters  
115, 252  
non-zero (true) 21, 24  
Normal command, Type Specify  
135  
normal display mode 255  
normal video 253  
NOT logical operator 245  
notes, musical 155  
not-equal operator 244  
null characters 179  
number comparisons 243  
numbers 214, 278  
in Accepted responses 60  
in strings 59, 235  
notation 59, 211, 214  
number-storage place 27, 206  
number-to-string function  
FLO(X\$) 235  
numeric argument 230  
numeric constants 100, 214  
numeric variable 27, 41, 57,  
100, 207, 215, 278  
numeric variable array 27, 41,  
58, 94, 95, 100, 217, 278

## O

O Offset command, Graphics 116  
object 6, 26, 284  
Offset command, Graphics 116  
offset position 121, 292

one-dimensional numeric array  
95, 100  
Open Existing File instruction  
181-184  
Open New File instruction 176-  
181  
opening a data file 175, 176,  
181  
operators 102, 242-246, 280  
arithmetic 102, 242  
logical 245  
precedence 103, 246, 281  
relational 243  
summary 280-282  
string 246  
OR logical operator 245  
"or" Match controller character  
65  
order for evaluating operations  
103, 246, 281  
order of elements in instructions  
7  
Output to open File instruction  
184-188  
overprint display mode 256  
overstrike type mode 132

## P

P Pascal modifier 13, 84  
P Plot command, Graphics  
absolute coordinates 117  
Turtlegraphics 118  
P Point modifier, Accept 13,  
61, 160  
P Print command, Type Specify  
133  
paddles 160, 239  
parentheses  
in expressions 103, 247  
with subscripts 95, 217, 223  
parts of an instruction 7  
Pascal filer 130, 153, 174, 261  
Pascal modifier 13, 84  
Pascal programs 84  
Pascal system 130, 153, 174,  
261  
PDL(X) game control function 239  
percent sign 64  
PILOT-SuperPILOT differences  
2, 290-294  
pitch, note 155

Plot command, Graphics  
absolute coordinates 117  
Turtlegraphics 118  
plotting, reverse color 164  
plus sign 242  
in diskette name 260  
Point modifier 13, 61, 160  
point plotting 116-118  
pound sign 42, 43, 58, 206, 215  
precedence among operations  
103, 246, 281  
precision of numbers 214  
PR: PProblem instruction 48-52  
Print command, Type Specify  
133  
printers 133, 134, 145  
printing  
graphics 134  
text 133, 145  
PProblem instruction 48-52,  
197, 210, 294  
as unlabelled branch  
destination 48, 52  
program 3  
program segments, Linking 83  
prompt, triangle 31, 200, 258  
pseudo-variables 58, 224

## Q

Q Quiet command, Type Specify  
134  
Q Quit command, Graphics  
absolute coordinates 117  
Turtlegraphics 118  
quick-draw graphics files 129,  
292  
Quiet command, Type Specify 134  
Quit command, Graphics  
absolute coordinates 117  
Turtlegraphics 118  
quotation marks 214

## R

R Redraw command, Graphics  
absolute coordinates 117  
Turtlegraphics 118  
R: Remark instruction 29, 36  
radian measure 231  
random number function RND(X)  
233

real numbers, limits 60, 214  
 record keeping 168-171, 290  
   diskette 171, 260, 290  
   file 168, 170  
   option 168, 290  
 records, number 175, 185, 189  
 Redraw command, Graphics  
   absolute coordinates 117  
   Turtlegraphics 118  
 reformatting text 40  
 relational operators 243  
 relative branch destinations 67  
 relative coordinates (Turtle-  
   graphics) 116-118, 291  
 relative graphics 116-118, 291  
 Remark instruction 29, 36  
 removing data files 178  
 repeat-factors 120, 135  
 repeating instructions 32  
 replace option, Compute 104  
 replacing variables with value  
   41  
 replacing expressions with  
   value 43  
 RESET 34, 201  
 Resource diskette 260  
 Response Buffer  
   see Student Answer Buffer  
 response editing 49, 55, 103  
 response timing 49, 54, 161,  
   241  
 response-timing option, PProblem  
   49, 54  
 retrieving  
   stored character sets 151  
   stored data records 174, 188  
   stored graphics images 129  
   stored sound effects 155  
 RETURN 7, 64, 253  
 Reverse, Graphics color 126  
 reopen existing file 181  
 RND(X) random number function  
   233  
 rounding of numbers 214  
 row coordinate (%R system  
   variable) 124, 226, 279  
 running a lesson 29-34  
 run-time commands 194-202

S Size of Type command, Type  
   Specify 132  
 S Spaces option, PProblem 49,  
   65  
 S Specify modifier 45, 131-151  
 S Spell modifier 13, 66  
 S Spin command, Graphics 117  
 S: Sound instruction 155-158  
 Save modifier 13, 170  
 saving  
   see storing  
 scientific notation 59, 106,  
   211, 214  
 screen  
   command characters 252  
   coordinates, Accepting 160  
   graphics 121  
   modes 255  
   text 37, 114  
 segments, Linking 83  
 semicolon  
   in Animate command 139  
   in Compute instruction 99,  
     292  
   in Dimension instruction 97,  
     292  
   in Graphic instruction 118,  
     292  
 SGN(X) sign function 232  
 sign function 232  
 significant digits 214  
 Silentype 133  
 simple numeric variable 27,  
   41, 57, 100, 207, 215, 278  
 single-dot off color 126  
 single-dot on color 126  
 single-spacing of lines 133  
 Single modifier 13, 60  
 SIN(X) sine function 231  
 Size of Type command, Type  
   Specify 132  
 slash (/)  
   with Animate command 139  
   with Compute instruction 103  
 .SONG file name suffix 174  
 sound effect 155-160  
 Sound Effects Editor 158  
 sound effects files 158  
 Sound instruction 155-158

## §

S Save modifier 13, 170  
 S Single modifier 13, 60



- spaces
  - in Accepted responses 48, 55, 65
  - in Dimension instruction 97
  - in Control options 48
  - in Graphics instruction 118
  - in Match instruction 64
  - in response editing 55, 65
  - in Type instruction 205, 207
  - with assignment indicator 99
- Spaces option, PProblem 49, 65
- Specify modifier 45, 131-151
- Spell modifier 13, 66
- Spin command, Graphics 117
- SQR(X) square root function 233
- square brackets 5
- square root function 233
- stopping the program 201
- storing
  - character sets 151
  - graphics images 129
  - numbers 57, 99, 174, 206, 214
  - screen positions 160
  - sound effects 158
  - strings 57, 99, 174, 204, 220
  - data records 174, 184
  - lesson data 174
  - student records 168
  - student responses 57, 160, 226
- string comparisons 244
- string constants 100, 214, 291
- string editing with Compute 103
- string expressions 246
- string functions 234
- string operator 246
- string pseudo-variables 58
- string variables 28, 41, 57, 94, 100, 204, 220, 278, 291
  - as objects of FI instruction 189
  - in file instructions 175
  - editing 103
  - execution as instruction 88
- strings in animation 140
- string-to-number function 236
- STR(X) number-to-string function 236
- Student Answer Buffer (system variable %B) 53, 57, 194, 198, 226, 279
- student responses 53, 160, 194-202, 226
- student system 293
- student's name, using 204
- student's numbers, using 206
- subroutines 74-83
  - nesting 77, 292
- subscripts 27, 95, 217, 223
- substring variables 43, 58, 100, 222, 279
- subtraction operator 102, 242
- sub-expressions 103, 247
- summary of SuperPILOT language 278-288
- SuperPILOT differences from Apple PILOT 290-294
- SuperPILOT LOG package 168, 170, 174, 290
- SuperPILOT Technical Support Package 165
- SX: eXecute Sound file instruction 158
- SYSLOG diskette 171, 260, 290
- system execution error messages 272-274
- system failure errors 275
- system variables 28, 57, 100, 225-229, 279
  - Answer Count %A 16, 56, 195, 198, 225, 279
  - Column Coordinate %C 124, 226, 279
  - Game Port Control %O 227, 279
  - Row Coordinate %R 124, 226, 279
  - Signed V parameter (audio/visual) %V 165, 227, 279
  - Spin Angle %S 117, 226, 279
  - Student Answer Buffer %B 53, 57, 194, 198, 226, 279
  - Unsigned V parameter (audio/visual) %W 165, 227, 279
  - X-Coordinate %X 61, 118, 124, 226, 279
  - Y-Coordinate %Y 61, 118, 124, 226, 279
- system.log file 168-171, 175, 290
- SYSX Escape subroutine 33, 50, 197

## T

- Thickness of Type command, Type Specify 132

T Time option, PROblem 49, 54, 162

T Type on graphics screen  
command, Graphics 115

T: Type instruction 37-46, 291

television 145

terminating  
lesson 53, 77, 83  
subroutine 77

testing a lesson 29, 195, 268

text, printing 133, 145

text animation 137-142

text characters  
alternate sets 151  
ASCII codes 258  
in graphics displays 115, 122  
in text displays 38

text colors 135, 143-147

text cursor 44, 53, 114, 124  
123, 131, 132, 252, 293

text displays 38, 115, 122

Text Editor  
see Lesson Text Editor

text instructions 36-46

text screen 122

text viewport 40, 114, 122,  
125, 131, 252

Text Viewport command 114

Thickness of Type command,  
Type Specify 132

Time option, PROblem 49, 54,  
162

timed Wait 92

timing of responses 49, 54,  
91, 161

TIM(X) response time function  
54, 161, 241

transcendental functions 231

trailing 140-142

Transmit ASCII command, Type  
Specify 132

triangle prompt 31, 200, 258

trigonometric functions 231

true (value is not 0) 21, 24,  
243

truncation function 232

TS: Type Specify instruction  
45, 131-151, 290

turnkey system 30

Turtlegraphics 116-118, 291

two-dimensional numeric array  
95, 100

TX: eXecute character-seT file  
instruction 45, 151-154

Type on Graphics Screen  
command, Graphics 115

Type instruction 37-46, 291  
length 38  
modifiers 44  
variables 41

Type Specify instruction 45,  
131-151, 290

## U

U Uppercase option, Compute  
104

U Uppercase option, PROblem  
49, 65

U: Use instruction 67, 74

unary plus and minus operators  
242

unequal operator 244

unfriendly colors 144

unlabelled destinations 9, 48,  
52, 61, 67, 72, 75, 78, 194  
Accept 61  
Match 67  
PROblem 48, 52

unused diskette blocks 179

upper- and lowercase letters 3,  
5, 28, 49, 55, 104, 178, 182

Uppercase option, Compute 104

Uppercase option, PROblem 49,  
65

Use instruction 67, 74

## V

V Viewport command, Graphics  
114

Viewport command, Type Specify  
125, 131, 135

V: Audio/Visual Device Control  
instruction 165, 290

variables 27, 41, 204, 215, 278  
erasing 84  
in Accept 57, 215  
in Compute 98  
in Dimension 95  
in eXecute Indirect 88

- in file instructions 174
  - FIX: 182
  - FI: 188
  - FOX: 178
  - FO: 184
- in Graphics 118, 121
- in PProblem 50
- in Sound 156
- in Type 41
- in Type Specify 131
- in Wait 91
- passed by Linking 84
- variables, system 225-228
- Vertical Coordinate (system variable %Y) 118, 124, 226
- video, normal and inverse 253
- videodisks and videotapes 165, 290
- viewport 40, 114, 122, 125, 131, 252
- Viewport command, Graphics 114, 125, 135
- Viewport command, Type Specify 125, 131, 135

## W

- W Walk command, Type Specify 134, 139
- W Wipe-labels option, PProblem 9, 50
- W: Wait instruction 91
- Wait instruction 91
- waiting operations 248
- Walk command, Type Specify 134, 139
- wildcard character, Match 65
- window, text 40, 114, 122, 125, 131, 253-254
- Wipe-labels option, PProblem 9, 50
- word-in-string function 237
- word-storage place 27, 204

## X

- X Erase modifier 13, 84
- X eXact modifier 13, 60
- X eXecute modifier 13, 45
- X Xmit command, Graphics 115, 252

- X Transmit ASCII command, Type Specify 132
- XI: eXecute Indirect instruction 88
- Xmit command, Graphics 115, 252
- X-Coordinate (system variable %X) 61, 118, 124, 226, 279
- X-or type mode 132, 256

## Y

- Y Yes conditioner 14, 63
- Y-Coordinate (system variable %Y) 61, 118, 124, 226, 279

## Z

- zero (false) 21, 24, 243

## Symbols

- ! logical OR operator 245
- Match "or" controller 65
- quick-draw file indicator 129
- !! string operator 246
- " around string constants 214
- # with simple numeric variable names 42, 43, 58, 206, 215
- \$ with string variable names 28, 42, 58, 94, 205, 220
- % Match "start-word" controller 64
- & logical AND operator 245
- Match "and" controller 65
- () parentheses
  - in expressions 103, 247
  - in subscripts 95, 217, 223
- [] square brackets, optional element 5

*	in remark lines 36	@a	72, 78
	label identifier 8, 72, 75, 282	@p	52, 72, 78
	Match "wildcard" controller 65	@m	67, 72, 78
	multiplication operator 102, 242	%A	Answer Count 16, 56, 195, 198, 225, 279
	repeat-factor 120, 136	%B	Student Answer Buffer 53, 57, 59, 194, 198, 226, 279
	string literal 85	%C	Column Coordinate 124, 226, 279
**	exponentiation operator 242	%O	Game Port Control 227, 279
+	addition operator 102, 242	%R	Row Coordinate 124, 226, 279
	in diskette name 260	%S	Spin Angle 117, 226, 279
	unary plus sign 242	%V	Audio/visual (signed) 165, 227, 279
,	between object elements 83	%W	Audio/visual (unsigned) 165, 227, 279
	between elements	%X	X-Coordinate 61, 118, 124, 226
	in Graphics command 114	%Y	Y-Coordinate 61, 118, 124, 226
	in Sound command 155		
	between numbers in a subscript 95, 219, 223		
-	subtraction operator 102, 242		
	unary minus sign 242		
/	Compute string-editing symbol 103		
	division operator 102, 242		
	spacing in Animate command 139		
^	logical NOT operator 245		
:	before object of instruction 25		
	to continue an instruction 25		
	in file names 263		
;	between commands in a list		
	Compute 99, 292		
	Dimension 97, 292		
	Graphics 114, 292		
	in object of Animate command 139		
<	less-than operator 244		
<=	less-than-or-equal operator 244		
<>	unequal operator 244		
=	equal operator 244		
	assignment indicator 99		
>	greater-than operator 244		
>=	greater-than-or-equal operator 244		
@	Escape command 33, 50, 197		



# SuperPILOT Language Quick Reference Guide

---

## The Parts of an Instruction

These are all the possible elements of a SuperPILOT instruction, in correct order:

Element	Comment
*label	Usually on separate line.
instruction name	Required; 1 to 3 letters.
modifier	Usually just one; one letter.
conditioners (expression)	Up to 4; one character each. One allowed; in parentheses.
:	Required colon.
object	Depends on instruction.

### Label

Optional; identifies a place in the lesson. The first six characters of the label are significant; preceded by an asterisk and followed by at least one space or end of line. When label appears in the object field of a branching instruction, asterisk is omitted.

### Modifier

Optional; usually only one is used. Modifiers affect specific instructions. Here are the instruction-modifier pairs:

TH:	(Type-Hang)	No carriage-return after a Typed line.
AX:	(Accept-eXact)	Accept exact response, with no editing.
AS:	(Accept-Single)	Accept a single-character response.
MS:	(Match-Spell)	Allow up to one wrong character on a Match.
MJ:	(Match-Jump)	Jump to next Match if this Match is unsuccessful.
LX:	(Link-Erase)	Start a new lesson without preserving the old lesson's variables.
LP:	(Link-Pascal)	Link to the Pascal program named in object.
KS:	(Keep-Save)	Write data accumulated from Keep instructions onto system.log file.



# SuperPILOT Language Quick Reference Guide

## The Parts of an Instruction

These are all the possible elements of a SuperPILOT instruction, in correct order:

Element	Comment
*label	Usually on separate line.
instruction name	Required; 1 to 3 letters.
modifier	Usually just one; one letter.
conditioners (expression)	Up to 4; one character each. One allowed; in parentheses.
:	Required colon.
object	Depends on instruction.

## Label

Optional; identifies a place in the lesson. The first six characters of the label are significant; preceded by an asterisk and followed by at least one space or end of line. When label appears in the object field of a branching instruction, asterisk is omitted.

## Modifier

Optional; usually only one is used. Modifiers affect specific instructions. Here are the instruction-modifier pairs:

TH:	(Type-Hang)	No carriage-return after a Typed line.
AX:	(Accept-eXact)	Accept exact response, with no editing.
AS:	(Accept-Single)	Accept a single-character response.
MS:	(Match-Spell)	Allow up to one wrong character on a Match.
MJ:	(Match-Jump)	Jump to next Match if this Match is unsuccessful.
LX:	(Link-Erase)	Start a new lesson without preserving the old lesson's variables.
LP:	(Link-Pascal)	Link to the Pascal program named in object.
KS:	(Keep-Save)	Write data accumulated from Keep instructions onto system.log file.

## Conditioners

Optional; up to four, plus an expression, may be useful on one instruction. Conditioners may be used with any instruction. An instruction is skipped unless all of its conditions are met:

Y	(Yes)	Execute if last Match was successful.
N	(No)	Execute if last Match was unsuccessful.
1-99	(Answer-Count)	Execute if conditioner number equals current Answer-Count.
E	(Error)	Execute if the Error Flag has been raised.
(..)	(Expression)	Execute if expression's value is non-zero.
C	(Last-expression)	Execute if last evaluated instruction-modifying expression was true (non-zero).

## The SuperPILOT Instruction Set

### Text Instructions

Remark r: THIS IS A COMMENT.  
r: Version 3, 18 April '81

Comments to author; not executed.

Type t:What is an apple?  
t:#n is right, \$\$ \$

Displays and formats object text on student's screen. If text includes variable names, their stored values are displayed, instead.

### Response Instructions

Problem pr:egt20  
pr:us

Starts new section and sets options (if PR: sets any option, all options not restated are turned off).

E	(Escape)	pr:e	Response starting @ tells SuperPILOT to Use subroutine sysx .
G	(Goto)	pr:g	Response goto label tells SuperPILOT to Jump to label.
L	(Lowercase)	pr:l	Converts responses to small letters.
U	(Uppercase)	pr:u	Converts responses to capital letters.
S	(Spaces)	pr:s	Removes all spaces from responses.
Tx	(Time)	pr:t15	Sets maximum response time to x seconds ( pr:t0 resets unlimited response time).

W (Wipe-labels) pr:w  
"Forgets" all labels previous to this point in lesson.  
(no options) pr:  
Starts new section without changing any set options.

Accept a:  
a:#n \$\$ \$  
Accepts student's response. May assign response to variables.

Match m:COMPUTER  
m:yes!ok!right  
Looks for object text in student's last response. Result of search used by Yes and No conditioners. Object text may include any number of controller characters:

\* (Wildcard) m:comput\*r  
Matches any response character.  
% (Start-word) m:%apple  
Matches space or beginning of response.  
! (Or) m:apple!pear  
Separates object texts. Match successful if response contains either of given texts.  
& (And) m:apple&computer  
Separates object texts. Match successful if response contains both texts in the given order.

### Control Instructions

Jump j:next  
j:@m  
Branches to the specified label, or to the next pr: ( j:@p ), the next m: ( j:@m ), or the last a: ( j:@a ).

Use u:score  
u:@p  
Branches like J: , usually to a labelled subroutine that ends with e: , but "remembers" where to return.

End e:  
e:@a  
Returns to instruction following the u: that called this subroutine, or (if there is an object) terminates subroutine and then branches like J: . If not reached by a u: , ends lesson.

Link l:lesson2  
l:lesson2.review  
Starts new lesson, keeping all the variables from the old lesson. May start at a specified label.

eXecute Indirect xi:q2\$  
xi:s\$  
Executes contents of the specified string variable as a SuperPILOT instruction.

Wait w:5  
w:n3  
Pauses the specified number of seconds or until any key is pressed, whichever comes first. Object may be a simple numeric variable name.

## Computation Instructions

Dimension d:q2\$(40)  
d:r(4,5)  
Sets maximum string size for string variable or maximum subscript for each dimension (one or two) of a numeric variable array.

Compute c:q2\$="APPLE"  
c:m=417.95  
c:h3=cos(flo(a\$)-2)

Evaluates an arithmetic or string expression, and stores it in the variable to left of equal sign.

Compute c:/q2\$ /@/ c  
(used for string editing)  
Edits contents of specified string variable, with these options:

U (Uppercase) c:/s\$ u  
Converts all letters to capitals.  
L (Lowercase) c:/s\$ l  
Converts all letters to lowercase.  
C (Capitalize) c:/s\$ c  
Capitalizes first character if it is a letter.  
/x/ (Delete) c:/s\$ /x/  
Removes every character x .  
/xy (Replace) c:/s\$ /xy  
Replaces every character x with the character y .

## Special Effects Instructions

Graphics g:c6;p23,40;d98,13  
g:es2;m0,0;dx1,y1;dx2,y2  
Executes the graphics commands that appear in the object field:

Vl,r,t,b (Viewport) g:v29,39,0,23  
Sets left, right, top, and bottom character positions for text viewport. Columns = 0 (left) to 39 (right), and rows = 0 (top) to 23 (bottom). Default is to full screen, 0,39,0,23.  
ES (Erase Screen) g:es  
Erases the text viewport to the set text background color.  
ESx (Erase Screen in Graphics Color) g:es2  
Erases screen to color x (0 to 10) without changing drawing color:  
Cx (Color) g:c6  
Sets drawing color to x (0 to 10).  
Ox,y (Offset) g:o99,230  
Defines absolute x,y as reference point for relative locations, and moves pen there. Graphics x = 0 (left) to 559 (right), y = 0 (bottom) to 511 (top).  
Ax (Angle) g:a45  
Sets the direction, in degrees, for relative graphics commands.  
Sx (Spin) g:s90  
Increments the set angle direction by the specified number of degrees.  
Mx,y (Move) g:m20,77  
Moves pen to point at relative x,y .  
Mx (Move) g:m100  
Moves pen specified distance in set angle direction.  
Px,y (Plot) g:p300,450  
Plots point at relative x,y .



**Px** (Plot) g:p300  
Plots point at specified distance in set angle direction.

**Qx,y** (Quit) g:q300,450  
Plots black point at relative x,y .

**Qx** (Quit) g:q240  
Plots black point at specified distance in set angle direction.

**Dx,y** (Draw) g:d9,200  
Draws line to relative x,y .

**Dx** (Draw) g:d360  
Draws line of specified length in set angle direction.

**Rx,y** (Redraw) g:r300,450  
Draws black line to relative x,y.

**Rx** (Redraw) g:r210  
Draws black line of specified length in set angle direction.

**Ttext** (Type) g:tOhio River  
Places text on screen starting at current graphics cursor position.

**Xx1,x2,x3** (Xmit) g:x8,8,7,18  
Transmits screen-control characters whose ASCII codes are x1,x2,x3 .

**\*x(...)** (Repeat-Factor) g:\*3(d50;s90)  
Executes the commands in parentheses the specified number of times.

**eXecute Graphics** gx:map  
gx:mirage  
Redraws image from named file, step by step on student's screen.  
Does not erase screen before drawing.

**eXecute Graphics** gx:photo2!  
(Quick-Draw) gx:monster!  
Quickly replaces the screen with complete image from named file.  
File name includes ! character.

**Type Specify** ts:\*20(ax\$;d25;wr)  
Determines how and where text will appear on the screen;  
background and foreground colors; type styles and line spacings.

**Vi,r,t,b** (Viewport) ts:v0,39,0,4  
Sets left, right, top, and bottom boundaries for text viewport.  
Default is full screen. Columns = 0 to 39 (left to right), and rows = 0 to 23 (top to bottom).

**Gx,y** (Goto) ts:g10,15  
Move text cursor to x,y coordinates.

**Sx** (Size) ts:s1  
Sets single (x=1) or double (x=2) size.

**Tx** (Thickness) ts:t2  
Sets character thickness, where 1 is normal, 2 is boldface.

**Mx** (Mode) ts:m2  
Sets type style, where x=1 is normal, x=2 is overstrike, x=3 is exclusive-or.

**Xx** (Transmit) ts:x8  
Sends any character to the screen, by that character's ASCII number.

**Lx** (Line Spacing) ts:l2  
Sets spacing between lines, multiples from 1-15.

**P** (Print) ts:p  
Sends future characters to printer.

**Q** (Quiet) ts:q  
Turns off character flow to printer.

**Ax\$** (Animate) ts:ax\$  
Executes a fast, unformatted write of string variable x\$ .

**Wlrud** (Walk) ts:wrrd  
Moves text cursor one character position left (wl), right (wr), up (wu), or down (wd).

**Dx** (Delay) ts:d15  
Pauses for specified 60ths of a second.

**Bx** (Background) ts:b5  
Sets the background text color, 0-20.

**Fx** (Foreground) ts:f4  
Sets the foreground text color, 0-20.

**I** (Inverse) ts:i  
Future characters are printed in the background color on a background that is the foreground color.

**N** (Normal) ts:n  
Cancels Inverse ts:i instruction.

**ES** (Erase Screen) ts:es  
Erases text viewport in the set text background color.

**ESx** (Erase Screen in Graphics Color) ts:es6  
Erases screen to graphics color x .

**\*x(...)** (Repeat-Factor) ts:\*4(ab\$;wr)  
Executes the commands in parentheses the specified number of times.

**eXecute character-seT** tx:cyrillic  
tx:chess-set  
Starts using characters in named diskette file.

**eXecute character-seT** tx:  
(no object)  
Goes back to standard ASCII character set.

**Sound** s:2,20;4,55;3,20  
s:30,200;0,20;40,200  
Plays sounds of pitch,duration . Pitch = 1 (low note) to 50 (high note)  
Pitch = 0 is silent (rest)  
Duration = 0 (short) to 255 (long)

**eXecute Sound** sx:tuneup  
sx:oursong  
Plays sound effect stored in named diskette file.

**Accept Point** ap:  
Accepts x,y coordinates of graphics screen position indicated by student using game controllers or other pointing device.

**Video Control** v:[commands]  
Provides access to peripheral devices such as videotapes and videodisks.

## File Handling Instructions

**Keep** k:#a tries, #w wrong  
k:Name: \$n\$  
Transfers object to lesson file system.log .

**Open new file** fox:25,p\$  
fox:j5,new  
Creates and opens new file with the given name or with the name stored in the specified string variable, to contain records 0 through the specified largest-record number.

**Open new file** fox:0,x\$  
(with 0 records) fox:0,new  
Finds existing file with given name or name stored in the specified string variable, and deletes that file from the diskette.

**Open existing file** fix:m2,q\$  
fix:25,new  
Opens existing file with given name or name stored in the specified string variable, for reading or writing records 0 through the specified largest-record number.

**File Output** fo:n,x\$  
fo:13,good times  
Stores given string or contents of specified string variable in the specified record of file now open.

**File Input** fi:n,x\$  
fi:13,p\$  
Reads specified record of file now open, and stores result in specified string variable.

## SuperPILOT Colors

**Graphics Colors**

0= Black1	4= Black2	8= Reverse
1= Green	5= Orange	9= One Dot On
2= Violet	6= Blue	10= One Dot Off
3= White1	7= White2	

**Text Colors**

0= Black	8= White/Black	13= Green/White
1= Green	9= Green/Black	14= Violet/White
2= Violet	10= Violet/Black	15= Orange/White
3= White	11= Orange/Black	16= Blue/White
4= Black	12= Blue/Black	17= Green/Orange
5= Orange		18= Green/Blue
6= Blue		19= Violet/Orange
7= White		20= Violet/Blue

## Variables and Functions

### Variables

User Variables	Typical Names	Example
Simple numeric	f , q3	c:f=15
Numeric array	p8(13), m(3,4)	c:p8(7)=33
String	r\$ , u5\$	c:r\$="Apple"
Substring	r\$(3) , u5\$(4,7)	c:u5\$(2,5)=r\$

## System Variables

%A	Answer-Count	(number of times last a: executed)
%B	Answer Buffer	(response to last a: )
%X	X-Coordinate	(response to last
%Y	Y-Coordinate	AP: instruction)
%C	Column-Coordinate	(Vertical and horizontal
%R	Row-Coordinate	text cursor position)
%S	Spin Angle	(current angle value)
%O	Game Port	(peripheral access)
%V	V: Parameter	(signed A/V access)
%W	V: Parameter	(unsigned A/V access)

## Functions

### Transcendental Functions

**SIN(X)** Returns sine of angle x radians.

**COS(X)** Returns cosine of angle x radians.

**ATN(X)** Returns radian measure of angle whose tangent is x .

### Arithmetic Functions

**SGN(X)** Returns -1 if x<0 , 0 if x=0 , or 1 if x>0 .

**ABS(X)** Returns absolute value of x .

**FIX(X)** Returns integer found by discarding any decimal portion of x , without rounding (-32767.9 < x < 32767.9).

**INT(X)** Returns largest integer less than or equal to x (-32767.9 < x < 32767.9).

**RND(X)** If x<1 , returns random number from 0 to .999969.  
If x is from 1 to 32767, returns random integer from 0 through x-1 .

**SQR(X)** Returns square root of x (x>0).

**LOG(X)** Returns base 10 log of x (x>0).

**LN(X)** Returns natural log of x (x>0).

**EXP(X)** Returns e (2.71828) to power x .

### String Functions

**ASC(X\$)** Returns decimal ASCII code for first character in x\$ .

**CHR(X)** Returns character whose decimal ASCII code is x .

**FLO(X\$)** Returns first number in string x\$ .

**STR(X)** Returns number x expressed as a string of characters.

**INS(N,T\$,P\$)** Searches from n-th character of t\$ for first occurrence of p\$ . Returns 0 if no occurrence found, or returns position in t\$ of matching sequence's first character.

**LEN(X\$)** Returns number of characters in x\$ .

### Input Functions

**PDL(X)** Returns an integer from 0 to 255 indicating position of knob ("paddle") on game controller number x .

**BTN(X)** Returns 1 if button on game controller x is being held down, or 0 if that button is not being pressed.

**KEY(X)** Returns 1 if any key has been pressed since last a: or w: , or 0 if no key has been pressed.

**TIM(X)** Returns response time in seconds for last a: or ap: or 0 if pr:tn timeout ended response.