COLOR PILOT

by

George Gerhold and Larry Kheriaty

Tape Version

Cat. No. 26-2709

First Edition

This manual was written by George Gerhold and Larry Kheriaty and edited by Karen McGee.

TABLE OF CONTENTS

INTRODUCTION - CAI AND PILOT

"Computer-Assisted Instruction" is a term used to include all applications of the computer to teaching except for the teaching of computer programming. The essential feature of CAI is the facility for carrying on a dialogue with someone who may have absolutely no knowledge of or interest in computers. While computers expect precision and perfection, people seldom deliver either. Therefore, if the computer is to carry on a dialogue with a human being, it must be taught (programmed) to allow for flexibility, ambiguity, and errors in responses. While the ability to handle a variety of responses is essential in CAI, it is also extremely useful wherever there is to be interaction between machine and unsophisticated user.

A skilled programmer usually can make the computer do anything computers can do using almost any computer language, but any programmer can do more in a given time using a language designed for the application at hand. Computer languages are tools; good tools free humans for more creative tasks. The computer languages designed for CAI are called author languages. Of these author languages, PILOT is the most widely used for small computers.

There is a possibility of confusion between what are called "author languages" and "authoring systems." An author language is a computer language which allows an author to control the computer completely, and includes features which make the programming of dialogues relatively easy. An authoring system is a program which allows a teacher to generate instructional materials according to one of a number of pre-made formats. In an authoring system the author need have little knowledge of the computer; however, the author has little control over the format of the instruction. Color PILOT is an author language, not an authoring system, but it could be used to write a variety of authoring systems.

PILOT, which stands for Programmed Inquiry, Learning, or Teaching (which came first, the acronym or the name?), was originated by Dr. John A. Starkweather, Dr. Marty Kamp, and associates in the medical school at the University of California, San Francisco. Many people worked on extensions to what is now called "Core PILOT" or "PILOT 1973." The most successful and most powerful of these extensions was developed at Western Washington University by Mr. Larry Kheriaty and Dr. George Gerhold. The end result of that development was a language called "COMMON PILOT." COMMON PILOT is now available on many computers, ranging from most microcomputers to the largest mainframes, although not all versions are marketed under the COMMON PILOT name. Color PILOT is based on COMMON PILOT, with added features like color and sound which are available on the TRS-80 Color Computer, and without those features which are beyond the computer's capacity. The end result is a powerful language suitable for the programming of graphic and textual dialogues.

Remember that in CAI we have a dialogue between the computer and the user. The computer is really a stand-in for the creator of the program in this dialogue. We will call the creator of the program "the author" or "the programmer" in this manual; we will call the user of the program "the student." Using those terms, we can say that one of the indicators of quality CAI is the extent to which the students forget that they are interacting with a computer and think that they are "talking" to the author. While working your way through this manual, you will have to play two roles. When entering or editing programs, you will be functioning as an author or programmer. While testing the programs, you will be functioning as the student, since an author must always play student to test CAI programs.

GETTING STARTED

MAKING CONNECTIONS

Color PILOT requires the following pieces of equipment:

TRS-80® Color Computer marked 16K RAM
CTR-80 Cassette Recorder (or equivalent)
Standard Color TV Receiver
Antenna Switch and Cables

You may wish to refer to the chapters on "Installation" and "Operation" in your copy of the **TRS-80 Color Computer Operation Manual**, but all you really need to do is the following:

1. Select a location where you have three power outlets handy.

2. Connect the antenna switch (a metal box about 1 1/2 by 3 inches labeled **COMPUTER**) to the VHF antenna terminals on the TV set.

3. Select the cord which has a single phono plug on each end.Plug one end into the antenna switch (position labeled **TO GAME CONSOLE**), and plug the other end into the back of the computer (position marked **TO TV**).

4. Select the cord which has a five-prong connector on one end and three separate plugs on the other end. Plug the five-prong connector into the back of the computer (position marked **CASSETTE**). Then plug the black-covered plug into the socket marked **EAR** on the recorder; plug the gray-covered plug with the larger metal center into the socket marked **AUX** on the recorder, and finally plug the remaining plug (gray-covered with the smaller metal center) into the small diameter MIC socket on the recorder.

5. Plug in the power cords on the TV, the computer, and the cassette recorder.

6. Set the antenna switch to **COMPUTER.**

This completes the interconnection of your computer.

LOADING COLOR PILOT

The following steps will get your computer ready to run preprogrammed PILOT programs and to create new PILOT programs:

1. Turn on the TV set.

2. Select channel 3 or 4 on the TV set. Then select the same channel on the channel switch on the back of the computer.

3. Turn on the computer. The switch is on the back and is marked **POWER**. The screen should now display the message:

   ```
   EXTENDED COLOR BASIC 1.Ø
   COPYRIGHT (C) 198Ø BY TANDY
   UNDER LICENSE FROM MICROSOFT

   OK
   ```

   -Color PILOT will not work with Disk Color BASIC. If the message on your screen reads "DISK EXTENDED COLOR BASIC," turn the power off on the computer and then pull out the ROM pack at the right rear of the computer. Begin again with Step 3 above.

At this stage you may want to adjust the Sharpness, Brightness and Contrast controls on your TV set to give the most legible display. You may also wish to try the other of the two possible channels (3 or 4). You will adjust the color controls later. Once you get the above message, you're ready to load Color PILOT via cassette:

4. Insert the Color PILOT cassette into the recorder.

5. Set the volume control to about 5.

6. Press **REWIND**.

7. When the tape is completely rewound, press **STOP**.

8. Press **PLAY**.

9. On the computer keyboard, type C L O A D M and press ENTER.

The cassette recorder should now start and a slowly flashing **F** in reverse video followed by the name PILOT should appear in the upper left corner of the screen.

1Ø. When the message OK appears, type E X E C and press ENTER.

The prompt

PILOT:__

will appear at the top of the screen.

11. When the PILOT: prompt appears, press **STOP**, then **REWIND**, then **STOP** on the recorder and remove the Color PILOT tape. If the prompt does not appear. . .

    - Press the RESET button on the back of the keyboard;

    - Adjust the volume control on the recorder a little lower or higher;

    - Continue with step 3 above.

The computer is now ready to understand Color PILOT.

TWO WAYS TO PROCEED

There are two ways to get your computer to run Color PILOT instructions: In **IMMEDIATE** mode you enter an instruction which is run immediately; in **EDIT-RUN** mode you enter a series of instructions which can then be run as a program.

> **IMMEDIATE** mode is useful for learning Color Pilot and for experimenting with the effect of certain PILOT instructions.
>
> **EDIT-RUN** mode is the standard mode for the creation of Color PILOT programs.

In the first five chapters of this manual, we will use the **IMMEDIATE** mode to explore some of the Color PILOT instructions. This is probably the easiest way for a beginner to learn to program. However, some of the features of Color PILOT can be illustrated only in sequences of instructions. If you get impatient and want to see how to use Color PILOT for instruction, skip ahead to Chapter 6.

At many places in this manual we have inserted comments in the margins where emphasis is needed or where another way of stating something might help. We suggest that you add your own marginal comments anytime that you have to stop to figure something out. Explain it to yourself in your own words for future reference.

LEARNING COLOR PILOT

1. TYPING MESSAGES

After you've loaded Color Pilot onto the computer (see page 6), the screen should display the prompt line:

```
COLOR PILOT copyright 1982
MicroPi, License to Tandy Corp.
All rights reserved.
PILOT:__
```

-If something else is on the screen, press **BREAK**.

To enter **IMMEDIATE** mode, type **I**. This will produce the new prompt:

```
Immediate Mode:
```

on the screen. This indicates that the computer is in **IMMEDIATE** mode and that it is ready to receive and run single Color PILOT instructions.

Type in the following line:

```
T:WHEN I PRESS ENTER I WILL HAVE FINISHED MY
PILOT PROGRAM.
```

Don't worry about overfilling the line; in **IMMEDIATE** mode the computer will cope with that automatically. If you make a mistake just backspace by using the left arrow key (←) and retype. The screen should now contain the following:

```
Immediate Mode:
under license to Radio Shack
T:WHEN I PRESS ENTER I WILL HAVE
FINISHED MY PILOT PROGRAM.__
```

-The underline following the period is called the "cursor." The cursor indicates where the next character will be typed.

Press the **ENTER** key. The screen will show:

```
Immediate mode:
T:WHEN I PRESS ENTER I WILL HAVE
FINISHED MY PILOT PROGRAM.
WHEN I PRESS ENTER I WILL HAVE F
INISHED MY PILOT PROGRAM.
```

Let's analyze what happened. You instructed the computer to type a message by giving the operation code for a **TYPE** instruction (**T**). The operation code **T** is separated by the colon (:) from the actual message which is to be typed. In running the **TYPE** instruction (after you pressed [**ENTER**]), the computer typed everything following the colon. The computer packed as much as it could on one line, and then continued typing on the next line. We can control the spacing to avoid splitting words, but that comes later. Now try instructions that type out some other messages, for example:

```
HELLO
```

and:

```
NOT ALL ONE LINERS ARE FUNNY.
```

By now the screen is cluttered with text. Type in the instruction:

```
TS:LET'S MAKE A FRESH START
```

-In **IMMEDIATE** mode always press [**ENTER**] to run an instruction.

After you press [**ENTER**] the screen will display only:

```
LET'S MAKE A FRESH START.
```

Why did that happen? The instruction was a **TYPE** instruction as indicated by the **T** op code (op code is short for operation code). The **S** in the **TS:** is called a modifier because it modifies the way the instruction works; here the modification is to clear the screen before typing the message. As usual, the colon separates the op code from the message.

No doubt you've noticed that the computer can type out in lower case. How do we make it do that? We might want to type the instruction:

```
TS: Let's make a fresh start.
```

When you load Color PILOT the keyboard is locked in upper case. By holding a [**SHIFT**] key down and typing [**Ø**], you can turn the shift lock on and off. Turn off the shift lock.

- [**SHIFT**] [**Ø**] switches the [**SHIFT**] lock on and off.

As soon as we switch off the [**SHIFT**] lock, we have an opportunity to make case errors. Try the following instruction:

```
ts:Let's make a fresh start.
```

This confuses the computer, so it sends us an error message, in this case the message Ø-ERR. The computer is trying to tell us that it did not understand the first letter of the line as an op code. The computer tries to assist us in figuring out what is wrong by displaying the offending line and an error message. Notice that the cursor is at the end of the line, not at the start of the next line. We must press any key before we can proceed.

Of course, we see that the error is the lower case "t" in the op code, so we try:

```
Ts:Let's make a fresh start.
```

Again we get the same error message, so we now know that all op codes and modifiers must be upper case.

-All op codes and modifiers must be upper case.

One last error you might try is the omission of the colon. Again the computer gives the same error message. Remember that all op codes must be followed by a colon!

Many beginners are unduly worried about making errors. Don't worry about it; the computer will find the errors quickly and will point them out to you. Let the machine do the boring work!

You've probably noticed that after the computer has completed a **TYPE** instruction the cursor is at the beginning of the next line. Suppose we want the student to fill in a blank; that is, instead of the screen display:

```
2 + 3 =
-
```

we want:

```
2 + 3 = -
```

To get this, we suppress the automatic cursor movement to the beginning of the next line by attaching the **HANG** modifier (**H**) to the **T** op code. Try the following instruction:

```
TH:2 + 3 =          (2 spaces at end)
```

One problem is that it is hard to see spaces at the end of messages. It is easier to check spaces in **EDIT-RUN** mode.

To complete this exploration of the features of the **TYPE** instruction, compare the effects of the following three instructions. Pay careful attention to the position of the cursor.

```
TS:
TSH:
THS:
```

Notice that more than a single modifier can be included in the op code and, further, that the order of modifiers is unimportant. You may want to include some message on these **TYPE** instructions to make the effects of the modifiers clearer.

2. DOTS, LINES, AND WINDOWS

In the first chapter we used the screen like an endless sheet of paper. Now let's learn to use it as a piece of graph paper. For this we use the **GRAPH** (G:) op code. To get a clean piece of graph paper, we must erase the screen. Try typing in:

```
G:E
```

E stands for the **ERASE** operation.

The graphics screen has a resolution of 256 dots across (numbered Ø-255) and 192 up (numbered Ø-191). Like graph paper, the origin (Ø,Ø) is in the lower left corner. Let's locate the origin on the screen by plotting a dot there. Try:

```
G:DØ,Ø
```

The D tells the computer to plot a dot at the position which follows - in this case Ø,Ø. The position of the dot is given in the order horizontal, vertical (in standard math terms, X,Y).

The upper right corner of the screen should be at the position 255,191. Put a dot there.

```
G:E,D255,191
```

OK, now what happens if the coordinate given is too big? Try this:

```
G:E,D255,192
```

-We can put many graphic operations in a single **G** instruction.

If you watched carefully, you saw the dot disappear and reappear in exactly the same spot. Now try:

```
G:E,D256,191
```

A bit of explanation is in order! The screen wraps around every 256 dots. Thus 256,191 is equivalent to Ø,191. The screen wraps around every 256 dots in the vertical direction as well, and positions between 192 and 255 in the vertical direction are all placed at the top of the screen (at 191).

Now that we can put a dot anywhere on the screen, we can move on to lines. To draw a line we plot a dot at one end of the line and tell the computer to draw a line to the other end. Thus:

```
G:E,D1Ø,1Ø,L12Ø,12Ø
```

will draw a line at about a 45 degree angle. Try it.

If we want to draw two connected lines, we can use the end of the first line as the dot at the start of the second:

```
G:E,D1Ø,1ØØ,L5Ø,1Ø,L9Ø,1ØØ
```

By extension, a square:

```
G:E,D5Ø,5Ø,L2ØØ,5Ø,L2ØØ,2ØØ,L5Ø,2ØØ,L5Ø,L5Ø
```

or a triangle:

```
G:E,D125,16Ø,L185,1ØØ,L65,1ØØ,L125,16Ø
```

or any shape bounded by straight lines can be drawn. It may be helpful here to think in terms of a graphics cursor which remains at the position specified by the coordinates of the last used graphics operator. The graphics cursor is not visible on the screen, and it is not necessarily at the same position as the text cursor (the text cursor is visible at times).

"But," you ask, "what good is a triangle without a label?" One of the very useful features of Color PILOT is the ability to mix text and graphics in the same display. Redraw the triangle with one addition:

```
G:E,D125,16Ø,L185,1ØØ,L65,1ØØ,L125,16Ø,W1ØØ,8Ø
```

The addition is the operation W1ØØ,8Ø, which establishes a text window with the specified position as the upper left corner and the lower right corner of the screen as the lower right corner of the window. We can label our triangle by writing in the window with the instruction:

```
TS:TRIANGLE
```

Notice that the **TS:** cleared only the text window, not the whole screen. To erase the whole screen we could either restore the window corner to its full screen position (Ø,191), or we could use the erase operation with the **G:** op code.

Can we place the label above or inside the triangle? Yes, but not easily in **IMMEDIATE** mode. We'll return to this question later.

Our example earlier used the window operator to position text. The corner of the window is not necessarily exactly at the position specified, but it is close - within the size of one character. That's accurate enough for now.

The window operator is very useful for designing screen displays called "frames." A diagram can be drawn on one part of the screen and discussion can be positioned elsewhere. A changing text can be combined with a fixed diagram, etc. To illustrate the effect of the window operator, try:

```
G:E,W247,1ØØ
```

and then try:

```
T:ABCKEFGHIJKLMNOPQRSTUVWXYZ
```

Notice how it squeezes text towards the lower right corner and how the scrolling works.

Before going on to the next chapter, reopen the window by running the following instruction:

```
G:WØ,191
```

3. BIG LETTERS AND SCREEN MODES

There are a number of ways that the appearance of text on the screen can be changed. We can choose these different display modes via the mode operator on the G op code. For example,

```
G:M2
```

changes the background (and other) color. Try it. Depending on your TV, this may make characters easier or harder to read. To change back, type:

```
G:MØ
```

So, mode Ø is the standard mode for starting Color PILOT, and M2 uses the alternate color set.

Now let's try some other modes. Type:

```
G:M1
```

Nothing visible happened yet, so type:

```
G:M1,E
```

Aha! Now we can see that the display is reversed. To see the effect on graphics, try:

```
G:D5Ø,5Ø,L15Ø,15Ø
```

Again nothing! We just drew a black line on a black background. If we change to a different pen color, we'll see something.

```
G:P1,D5Ø,5Ø,L15Ø,15Ø
```

We'll come back to pen colors later, but pen color 1 (P1) will always be visible. We can try our old friend to see how lower case text looks in reverse video.

```
TS:Let's make a fresh start.
```

As you can see, it's a little hard to read along the edges.

Mode 1 is the reverse of mode Ø and, as you may have guessed, mode 3 is the reverse of mode 2. Try the line and the text in mode 3. To get there type:

G:M3

You must admit the letters are colorful.

The table below summarizes what we've discovered about display modes so far.

MODE	COLOR SET	VIDEO
Ø	Normal	Normal
1	Normal	Reverse
2	Alternate	Normal
3	Alternate	Reverse

There are four more modes. Let's experiment further.

G:M4,E

Again we can try:

TS:Let's make a fresh start.

The characters are four times as big. This can be useful for small children, titles, and special effects.

All combinations of color set, video, and characters are possible, as is indicated by the complete display mode table below:

DISPLAY MODE	COLOR SET	VIDEO	CHARACTER
Ø	Normal	Normal	Normal
1	Normal	Reverse	Normal
2	Alternate	Normal	Normal
3	Alternate	Reverse	Normal
4	Normal	Normal	Large
5	Normal	Reverse	Large
6	Alternate	Normal	Large
7	Alternate	Reverse	Large

Try all the display modes. Legibility is fine for all combinations with the large characters. Notice that the character size does not effect the graphic operators (dot and line).

Display modes can be mixed on a single screen, but it is difficult to illustrate that effectively in **IMMEDIATE** mode. Color sets cannot be mixed; a change of color set always effects the whole screen.

If we give a display mode operation outside of the range listed in the table, the computer interprets the number as equal to the excess over some multiple of 8. For example, the instruction:

```
G:M34
```

will have the same effect as:

```
G:M2
```

because 34 = 4 * 8 + 2. In computer terminology, we say that 34 is 2 modulo 8, and the display mode is calculated modulo 8.

4. SIMPLE COMPUTATIONS

After all, we are learning to program a computer. We'd better learn to make the computer do arithmetic. The op code for arithmetic is **COMPUTE** (C:). A typical **COMPUTE** instruction is:

```
C:X = 2 + 3
```

In English this instruction is best translated as:

```
Set the variable X equal to 2 plus 3
```

Therefore, after the instruction has been run, the variable X will contain the number 5.

If you are familar with the concept of a variable from algebra, that definition for a variable will do fine here. If not, just think of the variable X as a convenient name for a spot in memory where a number can be stored. Color PILOT allows you to use a maximum of 26 variables, named A,B,C,...Y,Z.

-Variable names must be upper case.

How can we verify the effect of a **COMPUTE** instruction in **IMMEDIATE** mode? Run the instruction:

```
C:X = 2 + 3
```

How do we find out what the value of X is? We can embed any variable in the message portion of a **TYPE** instruction. However, we must somehow help the computer distinguish between the letter "X," which might occur in any number of words, and the Variable X. We do this by preceding the variable with the number sign (#). Thus:

```
T:#X
```

will cause the computer to type out the current value of X, which is 5.

At this point, you might check the result of a number of common errors. If you type:

```
T:#x
```

the computer types #x because lower case x is not a variable. If you type the wrong variable name:

```
T:#W
```

you get the initial value of the variable W, which is Ø.

-All variables have an initial value of Ø.

We may want to use variables in longer messages. Try:

```
T:Well, #X is the answer.
```

Notice that a single space after the variable is ignored (for agreement with the COMMON PILOT syntax), so if we want a space to appear after the variable, we must put in two spaces.

```
T:Well, #X  is the answer.
```

The four basic arithmetic operations are indicated by the symbols +,-,*, and /. Try:

```
C:X = 2 * 3 + 5
```

and to find the result try:

```
T:#X
```

Now try typing:

```
C: X = 5 + 2 * 3
T:#X
```

From these examples, we see that multiplication (and division) are done before addition (and subtraction). Of course, we can use parentheses to change the order of operations. Try:

```
C:X = (5 + 2) * 3
T:#X
```

As you can see from this example, Color PILOT handles only integer arithmetic. Now try this:

```
C: X = 8/3
T:#X
```

The result is 2, which shows that the decimal portion of the number is simply thrown away; the result is not rounded off. Try:

```
C:X = (-8) / 3
T:#X
```

Again notice that the decimal is discarded.

Now look at the order of operations with equal priority. The following equation:

```
C:X = 4 * 3 / 2
T=#X
```

gives 6, but:

```
C: X = 4 * (3/2)
T:#X
```

gives 4. In general, the order that the equation is worked is from left to right. In the second example, the parentheses force the division to be done first; the integer result of the division is 1, and the end result is 4.

-Any time you are not sure of the effect of a numerical **COMPUTE** instruction, try it with simple integers.

There are a few more characteristics of integer arithmetic which we should cover. Try:

```
C: X = 32,ØØØ
```

The C-ERR message appears because the computer does not like the comma in the number. Try:

```
C: X = 32.Ø
```

The computer doesn't know what to do with the decimal point when it is restricted to integers.

There is a highest and a lowest integer which Color PILOT can handle. The highest is 32767 and the lowest is -32768. The integers wrap around. To see the result, try this series of steps:

```
C: X = 32767
T:#X
```

No surprises so far, but now try:

```
C: X = X + 1
T:#X
```

-Remember the equal sign means replace, so X=X+1 makes perfect sense.

Adding 1 to the highest number gives the lowest number. This is what is meant by "wrap around."

5. STRINGS

A sequence of digits is called a "number"; that's not news. A sequence of letters is called a "string"; that may be news. In this chapter we're going to learn how to manipulate strings.

Just as with a number, we need a named place in memory to store a string, and we name the place by using a variable name - one of the letters from A to Z. The computer always uses the same amount of space for all numbers, but the length of strings can vary greatly. The longest string that Color PILOT can handle is 255 characters, but it would be very wasteful of valuable space in the memory to make all strings that long. Instead, we tell the computer how much space to reserve for each string variable. Obviously this space - really a length - should be the maximum length that the particular variable will ever hold in the program.

-Maximum length = Maximum number of characters

Space for a string variable is reserved by a **DIMENSION** instruction. Try:

```
D:Y$(1Ø)
```

Nothing visible appears on the screen, but the computer now has reserved room for 1Ø characters in memory under the name Y$. The variable Y$ can now hold anything from Ø to 1Ø characters. The **DIMENSION** instruction also converted the variable Y from a numeric variable to a string variable for the remainder of the session (later for the remainder of the program).

Let's briefly review the implications of the way variables are handled in Color PILOT. There is a maximum of 26 variables available with names A...Z. Any one variable can be either a number or a string. Color PILOT assumes that all variables are numbers with an initial value of Ø. We tell Color PILOT that a particular variable is to be a string instead of a number by using the **DIMENSION** instruction, which also specifies the maximum length the string variable can hold. We have been using Y$ for a string variable name instead of simply Y. In Color PILOT, the $ in Y$ is optional. In this manual, we consistently use the $ with string variables for three reasons: In complex programs, the string sign helps keep variables straight; the syntax then agrees with other versions of PILOT; and the syntax agrees with most versions of BASIC.

We can now set the string variable to some value. Type:

```
C:Y$ = "ABC"
```

The right side of the assignment is called a "literal"; the computer assigns literally what is inside the quotation marks to the string variable. Now check the result by typing:

```
T:$Y$
```

Notice again that we must warn the computer that it is to substitute for the current value of a variable; here we do this by preceding the variable with a $.

The variable names must be upper case, but the contents of a string variable can be either upper case, lower case, or a combination of the two. Try:

```
C: Y$ = "Abcd"
T:$Y$
```

The contents can also be digits; try:

```
C: Y$ = "Abcd23y"
T:$Y$
```

If we try to load too much into a string variable (i.e., if we try to exceed the maximum length), the variable will just take as much as it can. Try:

```
C: Y$ = "ABCDEFGHIJKLMNOP"
T:$Y$
```

One common programming error is to forget to tell the computer that a variable is to be a string variable. Try this:

```
C: W$ = "ABC"
T:$W$
```

The result is zero because the literal "ABC" contains no number. In the absence of a **DIMENSION** instruction which would tell the computer that the variable W is a string instead of a number, the variable is converted to zero. But now try:

```
C: W$ = "ABC4"
T:$W$
```

Here the result is 4 because, as before, the computer treats W as a numeric variable and does an automatic conversion of any numeric characters in the string to a number. The number can occur anywhere within the string. The computer searches for the first digit or a minus sign and begins the conversion there.

The point of introducing string variables is that we can manipulate the strings. We can "add" strings:

```
C: Y$ = "ABC"!!"defg"
T:$Y$
```

The operator !!, called the "concatenation operator," causes the second string - here "defg" - to be tacked on to the end of the first string. Concatenation is the one place where auto-conversion between strings and numbers does not work. The two items to be concatenated must both be strings.

Strings can be altered in whole or in part, and parts of strings can be extracted. These kinds of changes require that we tell the computer where in the string the changes should occur. Let's first set up a string:

```
C: Y$ = "ABCDEFGHI"
```

and check the current contents:

```
T:$Y$
```

Now let's change the characters "DEF" to "xyz." We have to tell the computer where to start the change and how far to go in making the change. We do this as follows:

```
C: Y$(4,3) = "xyz"
```

and check the result by typing:

```
T:$Y$
```

Notice that we specified that the change should start in the fourth position (originally a "D"), and that the change should continue for three characters.

-Specify a portion of a string as (start,length) after the variable name.

We use the same notation to pick out pieces from a longer string. First create a second string variable with a maximum length of five:

```
D:Z$(5)
```

Then pick out a piece of Y$:

```
C: Z$ = Y$(5,2)
```

and check the result:

```
T:$Z$
```

If we don't specify a length in the parentheses, the computer assumes a length of one. Therefore:

```
C: Z$ = Y$(5)
```

is legal and is equivalent to:

```
C:Z$ = Y$(5,1)
```

6. ENTERING PROGRAMS

So far we've learned quite a bit about programming computers, but we've not really written any programs, other than "one-liners." Color PILOT has a built-in editor which can be used for writing multi-line programs (and for other text editing). After the program is written it can be run. In this chapter we will concentrate on learning to use the editor by repeating some of the examples from previous chapters. From now on you don't need to worry about typing carefully, as the editor makes it easy to correct errors.

To enter **EDIT** mode, first press **BREAK** to get the system prompt:

PILOT:

If you've been running in **IMMEDIATE** mode or if you've been running other programs, there is already a program in the program space which we must erase.

To erase a program - **SHIFT** **CLEAR**

That is, while holding down the **SHIFT** key, press **CLEAR**. To enter **EDIT** mode, simply type **E**. The screen will be blank except for the text cursor in the lower left corner. The computer is now ready to accept a program from the keyboard. For our first program using the editor, we'll use the program which draws a triangle and labels it. Earlier we did this by the two instructions:

```
G:E,D125,16Ø,L185,1ØØ,L65,1ØØ,L125,16Ø,W1ØØ,8Ø
TS:TRIANGLE
```

Now that we're using the editor, we make a few minor changes. The editor can handle only instructions which fit on a single line. Therefore we break the **GRAPH** instruction into two instructions on the screen.

Type in these two lines; press **ENTER** to end each one.

```
G:E,D125,16Ø,L185,1ØØ,L65,1ØØ
G:L125,16Ø,W1ØØ,8Ø
```

-The graphics cursor stays at the end of the last line drawn, even between instructions.

The **S** modifier will not be needed on the **T** op code because we don't have to erase the PILOT instructions from the text window as we did when we were running in **IMMEDIATE** mode. So the next line should be:

```
T:TRIANGLE
```

That completes our program; now we want to run it. To stop editing (leave **EDIT** mode), press [BREAK]. As always, this gives the system prompt. To enter **RUN** mode, press [R] - but watch closely because it is FAST!

Did you see it? In fact, it is too fast for us to use, so we need to make it wait. That's easy enough. At the moment, the computer is displaying the system prompt. Enter **EDIT** mode by pressing [E]. The first line of your program will appear at the bottom of the screen. By pressing [ENTER] or [↓] you can scan through your program line by line. Press [ENTER] three times, which gets you to a blank line at the end of your program, and add the instruction:

```
W:2ØØ
```

This is a **WAIT** instruction. It tells the computer to wait either until a specified amount of time has passed or until the user presses any key. The number after the colon is the number of tenths of seconds to wait; the instruction W:2ØØ says to wait for 2Ø seconds.

Run the edited program by typing [BREAK], then [R].

Now let's make some changes in your program to practice using features of the editor. Press [E] to enter **EDIT** mode, and press [↑] four times to get the whole program onto the screen. We want to change the program to the following:

```
G:E,D125,16Ø,L185,1ØØ,L65,1ØØ
G:L125,16Ø,W8Ø,135
T:A
G:W165,135
T:B
G:W13Ø,8Ø
T:C
W:12ØØ
```

Let's do this by altering the second line, the third line, and the last line; and by adding four more lines before the **W** instruction. Move the cursor to the indicated position on the second line by using [↑] and [→] keys.

```
G:L125,16Ø,W1ØØ,8Ø
```

Now type the characters 8Ø,135.

-Overtyping replaces characters.

Next move the cursor to the indicated position in the third line.

```
T:TRIANGLE
```

Delete the letters "TRI" by holding down the [SHIFT] key and pressing the [←] key three times. Skip over the A by pressing the [→] key once, and delete the rest of the line using [SHIFT] [←].

- [SHIFT] [←] deletes a character.

Next we need to insert four lines before the W:2ØØ instruction. Move the cursor to the start of the W:2ØØ instruction; hold down the [SHIFT], and type [↓] four times. You can check that this has added four blank lines in the desired position by using the arrow keys. Type the four lines in the blank lines.

- [SHIFT] [↓] inserts a blank line

```
G:W165,135
T:B
G:W13Ø,8Ø
T:C
```

Finally, we want to insert the 1 in the last line. Move the cursor to the indicated position.

```
W:2ØØ
```

Use [SHIFT] [→] to insert a space and type in the 1. The resulting program should draw a triangle and letter the sides. Try it by pressing [BREAK] and then [R].

- [SHIFT] [→] inserts a space which can then be overtyped with any other character.

In the above exercise, we have illustrated the basic functions of the editor: change characters, insert characters, delete characters, and insert lines. A complete list of all the control functions in the editor follows:

Move Cursor	[↑] [↓] [←] [→]
Delete Character	[SHIFT] [←]
Insert Space	[SHIFT] [→]
Insert Line	[SHIFT] [↓]
Move to 1st Line	[CLEAR]
Auto Scroll until any keystroke	[SHIFT] [↑]

Once we have completed editing a program, there are a number of things we can do with it. We already know how to run the program; we press [BREAK] to get back to the system prompt and then press [R] to run the program. If the program is complete and the run indicates that there are no errors in it, we will probably want to save the program on tape for future use. Make sure that you have a fresh cassette in the recorder. Before you give the instructions to save the program, you must make sure of two things:

1. Either the cassette is a leaderless tape or the tape has been run past the leader. If that is not the case, press the **RECORD** and **PLAY** buttons at the same time and pull the plug from the **MIC** jack. This will allow the tape to advance and effectively erase whatever was on the tape. Allow the tape to run for about five seconds and then replace the **MIC** plug. This will stop the tape.

2. **RECORD** and **PLAY** buttons must be pressed at the same time.

You can now save the program by typing an [S] in reply to the PILOT: prompt. When the save is complete, the light on the recorder will go off and you will get another system prompt.

It is possible to record more than a single program on a tape, but it takes planning. You must run the tape to the end of the program already on the tape; run some blank tape to separate the programs (see Step 1 above), and then record your program. This means that you must position the tape before you type in the program that you plan to save. All in all, it is much easier to save one program per tape.

You might want to get a printed listing of the program. This can be done either before or after it is saved, as saving does not destroy the program. To print your program, plug the printer into the back of the computer (marked **SERIAL I/O**), position paper in the printer, turn on the printer, and type [P] in reply to the system prompt. If the printer is not hooked up or not ready, the computer will respond with a question mark.

Of course, we want to be able to load programs from the tapes after we've saved them there. Typing an [L] in reply to the prompt will cause the computer to load the next program from the cassette. The **PLAY** button on the recorder should be down before typing [L] . Once the computer starts reading the program, a small flashing dot will appear above the prompt. It may take a few seconds for this to appear if you've recorded some blank tape to lead into your program.

These, then, are the ways in which we preserve and retrieve our programs. The table below lists all the operations we can call from the system prompt:

Clear Program Space	[SHIFT] [CLEAR]
Enter **EDIT** Mode	[E]
Enter **RUN** Mode	[R]
Enter **IMMEDIATE** Mode	[I]
Print Program	[P]
Save Program on Tape	[S]
Load Program from Tape	[L]

-When saving programs on tape, press down the **RECORD** and **PLAY** buttons before typing [S]

The editor provides a simple mechanism for entering material into the computer. The editor has no way of knowing whether what you are entering is a Color PILOT program, a letter to Grandma, or the grocery list. It will work equally well for all of them. Thus you can use the editor as a simple text editor for other purposes. Once the text is correct, you can save it on tape, and you can send it to the printer. If you don't like 32-character lines, then type an @ as the last character of a line. The @ at the end of the line will suppress the carriage return. Thus a paragraph with alternate lines ending in @ will print out double width.

7. COLOR

Now that we know how to write a multi-line program, we can write a program which will be useful for adjusting the color controls on the TV set.

Your Color Computer can produce eight different colors on a properly adjusted set. Two sets of four colors are available. When you start up Color PILOT you are in what we will call the "normal" set. We will refer to the other set as the "alternate" set.

We've already seen in Chapter 4 that the color set is selected by picking a mode in a **GRAPH** instruction (e.g., G:M2). Within each mode there are eight pen states which can be set. The effect of a pen state can be seen clearly by drawing blocks of color. Enter and run the following program:

```
G:P4,DØ,Ø,B5Ø,5Ø
W:1ØØ
```

This should produce a black box. The operation P4 specifies the pen color (color 4 = black); DØ,Ø sets the graphic cursor to one of the left corners of the block, and B5Ø,5Ø specifies that a block is to be drawn with 5Ø,5Ø as the opposite right corner.

-One of the left corners of a block must be given before the opposite right corner.

Let's look at all the colors in the normal set. Enter and run the program:

```
G:E,P4,DØ,Ø,B5Ø,5Ø
G:P5,B1ØØ,1ØØ
G:P6,B15Ø,15Ø
W:2ØØ
```

You should get three blocks of distinct color; the fourth color is the background color. The exact colors will depend on the color settings on your TV set. Before touching those settings, look at the alternate set. Change the first line to:

```
G:E,M2,P4,DØ,Ø,B5Ø,5Ø
```

Again we should get three colored blocks; the background is the fourth color.

Now let's make this into a program useful for adjusting the color controls. All we need to do is to change the color set back and forth once the blocks are drawn. We'll have the program show each color set twice to give us a chance to adjust the color controls.

```
G:E,MØ1P4,DØ,Ø,B5Ø,5Ø
G:P5,B1ØØ,1ØØ
G:P6,B15Ø,15Ø
W:2ØØ
G:M2
W:2ØØ
G:MØ
W:2ØØ
G:M2
W:2ØØ
```

Run this program several times and adjust the TV to give the best colors. Then complete the following table according to your adjustment of your TV set:

Pen Color	Normal	Alternate
4	Black	Black
5		
6		
7		

It would be useful to have this program handy whenever you want to readjust your TV set. Simply place a blank tape in the cassette recorder (not the Color PILOT tape), press down the **RECORD** and **PLAY** buttons simultaneously, and then type [S] in response to the system prompt. If you are using tape with a leader, then pull the plug in the **MIC** connection and let the tape run long enough to bypass the leader before typing [S] . You can then reload this program at any time the computer is displaying the system prompt by rewinding the tape and typing [L] .

The two colors given by pen states 5 and 6 have one rather unfortunate characteristic: They occasionally switch, but never in the middle of a program. This switch in colors is a consequence of the way in which Color PILOT generates color on this computer. It is unavoidable, so use color for emphasis, but don't count on the shade.

There are four more pen states available. Pen state Ø has no effect; it is not used. Pen state 1 reverses whatever color it is drawing over. Try typing in these instructions:

```
G:E,MØ,P4,D3Ø,3Ø,B8Ø,8Ø
G:P5,B13Ø,13Ø
G:P6,B18Ø,18Ø
G:P1,DØ,Ø,L19Ø,19Ø
W:2ØØ
G:M2
W:2ØØ
```

Pen states 2 and 3 are used for drawing lines. Pen state 2 is the same color as pen state 4, the normal foreground color. Pen state 2 will give a sharper line than pen state 4. Try:

```
G:E,MØ,P2,DØ,Ø,L5Ø,5Ø,L15Ø,9Ø
G:P4,D1Ø,Ø,L6Ø,5Ø,L16Ø,9Ø
W:2ØØ
```

Notice that the lines on the right are thicker. Pen state 3 bears the same relationship to pen state 7 as pen state 2 does to pen state 4. Pen state 3 is used to draw sharp lines on a reverse video screen. Of course, pen states 4-7 can be used to draw lines, if the extra thickness is desirable or color is needed.

Numbers greater than 7 for pen states are treated modulo 8.

Color TV sets do not all interpret microcomputer color signals in the same way. Therefore we have to avoid describing colors and effects in too much detail. We recommend that you try all combinations of color sets, pen colors, and normal and reverse video to find what works best on your TV. Remember that colors on a TV set are never truly accurate. Use different colors for variety and emphasis, but do not assume that everyone will see the same colors you see on your set.

8. ACCEPTING ANSWERS

Up to this point we have been concentrating on those Color PILOT instructions which can be tested on their own or in combination with one other instruction. We now introduce instructions which only make sense in longer combinations.

Color PILOT is a language designed for programming dialogues. Therefore we need an instruction which allows the user of the program to respond. The instruction with this capacity is the **ACCEPT (A:)** instruction. The **ACCEPT** instruction tells the computer to accept input from the keyboard and to store it for future reference. The keyboard input must end with a press of the **ENTER** key.

The user's response is stored in a special string variable named "%B." %B can be used in the same way as other string variables (it is already dimensioned to length 8Ø). Try the following program fragment:

```
TS:What shall I call you?
A:
T:OK, $%B, press ENTER to start.
A:
T:The first....
W:2Ø
```

Try it several times using different names.

Notice that the second **A:** is used as a user-controlled pause, similar but not identical to the **WAIT** instruction. A **WAIT** instruction will pause for a specified maximum time, but an **ACCEPT** instruction will pause forever - until **ENTER** is pressed.

You may have noticed that the name typed back in the message is always in upper case, regardless of how it was typed in. The **ACCEPT** instruction automatically converts the user response to upper case for reasons which will become clear later. If we don't want that conversion to occur, we add the **HOLD (H)** modifier to the **ACCEPT** code. Try this version of the program fragment:

```
TS:What should I call you?
AH:
T:OK, $%B, press ENTER to start.
A:
T:The first.....
W:2Ø
```

Now we move on to a more complex example. Type in this program:

```
TS:The box below is 1ØØ units
TH:high.
G:P2,D1ØØ,Ø,L15Ø,Ø,L15Ø,1ØØ
G:L1ØØ,1ØØ,L1ØØ,Ø
W:2Ø
T: To make the box half
T:full, how many units
T:high should you fill it?
A:
G:P5,D1Ø1,1,B149,%B
T:Is that half full?
A:
```

There are a number of things worth noting in this program. The second line (TH:high.) leaves the text cursor hanging. The following two **G:** instructions draw a box without moving the text cursor so that the next **TYPE** instruction continues typing on the same line. That is why there is a space at the start of the second message (T: If...). After the response, the user's value is inserted into the last **G:** instruction using a variable, here %B. Although %B is a string variable, the computer knows that a number is needed here, and the conversion is automatic. Note also that the auto-conversion to upper case with the **ACCEPT** instruction has no effect on numbers.

-Variables can be used to specify graphic coordinates.

With a 32-character limit on a line, it is not unusual to see several **TYPE** instructions in a row. For convenience, Color PILOT allows you to omit the op code on all but the first of a series of **TYPE** instructions. The following program will do the same thing as the first version:

```
TS:The box below is 1ØØ units
TH:high.
G:P2,D1ØØ,Ø,L15Ø,Ø,L15Ø,1ØØ
G:L1ØØ,1ØØ,L1ØØ,Ø
W:2Ø
T: To make the box half
:full, how many units
:high should you fill it?
A:
G:P5,D1Ø1,1,B149,%B
T:Is that half full?
A:
```

The instructions with just the colon are called "continuation instructions" or "continuation lines." We could not use a continuation line for the second instruction because we wanted to change modifiers on the second line. The continuation line assumes that everything before the colon on the last **TYPE** instruction applies to the continuation as well. The exception is the **S** modifier; it would make no sense to reuse it in successive lines.

Remember that the computer is only a machine; it is not intelligent and it does not really understand English. Try running the program again, but respond "fifty" instead of "5Ø". The computer does not know that the five letters f-i-f-t-y mean the same thing as some digits. Because the computer finds no digits, it assumes the answer is zero in the conversion.

There is another modifier which can be attached to the **ACCEPT** instruction: the **SINGLE (S)** modifier. The **ACCEPT SINGLE** instruction makes the computer wait until any one key is pressed. The response is stored in %B with no case editing. The **AS:** instruction is useful in multiple choice programs such as the following:

```
TS:The space shuttle is named
:
:     A. Mercury
:     B. Columbia
:     C. Gemini
:
:Answer A, B, or C.
:
AS:
```

Note the use of blank continuation lines to space out the text on the screen.

Students usually find it confusing when **A:** and **AS:** instructions are mixed in a single program ("Do I press **ENTER** now, or is the computer computing?"). It is best to decide whether all your answers are going to be single keystroke answers before you begin using **AS:**.

9. PROCESSING ANSWERS

Once we have a student answer, we want to be able to analyze and take various actions based on that answer. The instruction most frequently used for this purpose is the **MATCH** (**M:**) instruction.

If we give the computer the instruction:

```
M:MINUS
```

the computer compares the portion of the **MATCH** instruction following the colon with the current contents of the answer buffer (%B) and determines whether or not the answer buffer contains the text of the M: instruction. An example will help illustrate the process. Assume that at the last **ACCEPT** the student answered "It's minus this time." Therefore the answer buffer contains (remember the internal conversion to upper case):

```
IT'S MINUS THIS TIME.
```

The **MATCH** instruction establishes a window five spaces wide containing the letters M I N U S. The comparison proceeds in steps.

```
STEP 1
       IT'S MINUS THIS TIME.
       MINUS                        Result - no match

STEP 2

       IT'S MINUS THIS TIME.
        MINUS                       Result - no match

STEP 3
       IT'S MINUS THIS TIME.
         MINUS                      Result - no match

                   etc.

STEP 6
       IT'S MINUS THIS TIME.
            MINUS                   Result - match
```

If the window reaches the end of the answer buffer without finding a match, the match has failed. The process is called a "window string match." Notice that the computer ignores the rest of the student answer in the search. Thus the technique of processing answers is basically a search for key words.

In making the comparisons, the computer considers upper- and lower-case letters as different. The auto-conversion to upper case by the standard **A:** instruction eliminates this difficulty. Whatever the student types, the computer can make comparisons with upper case. Of course, if we are writing a lesson in proper capitalization of words, then we use the **AH:** instruction and adjust the field of the **MATCH** instruction accordingly.

-Most examples assume use of the **A:** instruction, so the fields of the **MATCH** instructions will be in upper case.

Naturally the students are going to make spelling errors and typing errors. There are a number of features which can be used with the **MATCH** instruction to help the computer recognize misspelled words. The easiest to use is the **SPELLING** modifier **MS:**. The instruction:

```
MS:MINUS
```

will count a match as successful if it finds a window position where all but one of the characters match or where all the characters are present but any two of them are reversed. Thus:

MS:MINUS

will match	will not match
MINAS	MENAS
MENUS	MUMIS
MUNIS	

The ability to handle character reversals covers many common spelling errors, for example handle-handel and their-thier. One-character errors also cover many spelling and typing errors.

Another way to handle some spelling errors is to use the wild card character (*) in the field of the **MATCH** instruction. The * matches any one character in the answer buffer. Thus:

M:M*NUS

will match	will not match
MINUS	MINAS
MENUS	MENES
MANUS	
MXNUS	
M3NUS	

Any number of *'s can be used in the field of a **MATCH** instruction, and *'s can be used in the fields of **MATCH** instructions with the spelling modifier.

Often there are equivalent suitable answers for a question. We can allow for alternate answers by means of the **OR** operator (!). If we wished to treat the answers "minus" and "negative" as equivalent, we could use the **MATCH** instruction:

```
M:MINUS!NEGATIVE
```

or with some provisions for spelling errors:

```
M:M*N*S!N*G*TIV
```

or perhaps:

```
MS:M*NUS!NEG*TIV
```

If the match using the window for "minus" fails, then the computer tries again with a second window for "negative." The match fails only if neither window string match succeeds.

Now that we know how to process student answers by key word searches, we need to learn how to do something with information. That is, we need to learn how to tell the computer to take alternate actions depending on whether the match succeeds or fails. The **MATCH** instruction sets a **Y** flag (in the computer's memory) if the match succeeds and an **N** flag if the match fails. These two symbols, **Y** and **N**, can be attached to op codes as the conditioners. Likewise, any instruction whose op code includes a **Y** conditioner will run if and only if the **Y** flag is set; that is, if the last match attempted was successful. If the last match failed, then the instruction will be skipped. Any instruction whose op code includes an **N** conditioner will run if and only if the **N** flag is set;that is, if the last match attempted failed. If the last match succeeded, the instruction will be skipped.

To see this effect enter the following program.

```
TS:Have you read Chapter 3?
A:
M:YES!YEA
TY:Fine, we'll go ahead.
TN:I guess you'd better do that.
W:1Ø
```

Now run the program and answer "yes." Then run it again and answer "no." Notice that the **TY:** runs when the match succeeds, and the **TN:** runs when the match fails.

The conditioners can be combined with the **MATCH** op code in a sequence of instructions to give the effect of an **AND** operator. Say we had asked the following question:

```
TS:What are the colors in the
:Canadian flag?
A:
```

We want the answer to contain the two words "red" and "white". We check for this by the following sequence.

```
M:RED
MY:WHITE
```

If the student's answer contains "red" ("RED" after auto-conversion), then and only then is the second **MATCH** instruction attempted. A usable program based on this example follows.

```
TS:What are the colors in the
:Canadian flag?
A:
M:RED
MY:WHITE
TY:Good.
TN:No, they are red and white.
W:1Ø
```

The **MATCH** instruction is an essential tool for the writing of dialogue. It is very difficult to do all the equivalent things in languages intended for computation, like BASIC. Mastery of the features of the **MATCH** instruction will allow you to introduce real flexibility into your programs. Don't hesitate to experiment, and don't hesitate to watch over the shoulders of students who are trying out your programs. You'll often get ideas as to how your **MATCH** instructions could be changed to improve your programs.

10. REVIEW - MULTIPLE CHOICE

We've been learning many new things in the previous chapters, and we have more to learn. Let's pause to review what we've learned by writing a short multiple choice test. This will show how some of the instructions can be combined into instructional sequences.

The test will be a check on the ability of students to correctly name simple polygons. Of course, you could substitute your own questions on any other subject in this format. We will introduce one new instruction, the **REMARK** instruction (**R:**). The computer ignores **REMARK** instructions; they are included in programs to provide information for the programmer.

We have now reached the stage where the sample programs will be rather long. We will interweave instructions with explanations of what the instructions are doing. To try out the programs, enter all the instructions; obviously the explanations are not to be entered.

Enter the following program:

```
D:N$(2Ø)
TS:What name shall I call you
:today?
AH:
C:N$=%B
```

We use an **AH:** instruction so that the student name will not be all upper case. We must move the name to another variable (N$) because %B will be overwritten by the next **ACCEPT**.

```
T:
:Today I'm going to give you a
:short quiz on naming shapes.
:
:Press ENTER when you are ready
:to start.
A:
R:Draw a triangle
G:E,P5,D125,185,L185,11Ø,L65,11Ø
G:L125,185,WØ,85
T:This is a
:
:            A.  Square
:            B.  Triangle
:            C.  Pentagon
:
:Pick A, B, or C.
:
AS:
M:B!b
```

Remember that **AS:** does not cause auto-conversion to upper case.

```
TY:Good.
TN:No, it's a triangle.
CY:R=R+1
```

The variable R is used to count correct answers. The next instruction gives the student time to read the response.

```
W:3Ø
R:Draw a square
G:E,P6,DØ,75,LØ,115,L4Ø,115
G:L4Ø,75,LØ,75,W9Ø,13Ø
T:This is a
:
:        A.  Square
:        B.  Pentagon
:        C.  Hexagon
:
Pick A, B, or C.
:
AS:
M:A!a
TY:Right.
TN:It's a square.
CY:R=R+1
W:3Ø
R:Draw a hexagon
G:E,P2,DØ,75,LØ,115,L34,135
G:L68,115,L68,75,L34,55,LØ,75
G:W11Ø,13Ø
T:This is a
:
:     A.  Pentagon
:     B.  Trapezoid
:     C.  Hexagon
:
:Pick A, B, or C.
:
AS:
M:C!c
TY:Good.
TN:It's a hexagon.
CY:R=R+1
W:3Ø
G:WØ,191,E
C:P = R*1ØØ/3
```

The variable P represents the percent correct.

```
T:You got #R  out of 3 right,
:$N$. That is #P%.
```

There must be two spaces after the variable R so that there will be one in the message as typed.

```
W:3Ø
G:E,M4,P6,D5Ø,5Ø,B225,14Ø
G:P7,D7Ø,7Ø,B2Ø5,12Ø,W9Ø,9Ø
T:THE END
W:2Ø
G:M6
W:5Ø
```

The last few lines provide an eye-catching ending and review the ideas of pen colors and screen modes.

The program given above is functional and could be extended to include many more questions and more choices per question. If we were going to make this much longer, we could save ourselves some typing and save some program space by loading the prompting message "Pick A, B, or C." into a string variable. For example,

```
D:M$(16)
C:M$="Pick A, B, or C."
```

should appear early in the program, before the first test item. Then instead of:

```
:Pick A, B, or C.
```

we use:

```
:$M$
```

The saving in routine typing and in program space becomes worthwhile as the test becomes longer.

11. JUMPING

While a simple test can be run directly from start to finish, true instruction requires a more complex path through a program. If a student makes an error, you may want to point out the nature of the error and then jump back to give the student one or more additional chances.

The Color PILOT instruction for jumping around in the program is the **JUMP** instruction (**J:**). The **JUMP** instruction has the general form:

```
J:destination
```

There are two kinds of destinations: the last **A:** run and a label. Jump to the last **A:** run (including, of course, the modified **ACCEPT**s **AS:** and **AH:**) is specified by:

```
J:@A
```

This frequently is used to give the student another try at answering a question.

Enter and run the following program:

```
TS:What is the most heavily used
:language on microcomputers?
A:
M:PILOT
TY:Not yet, but maybe soon. Try
:again.
JY:@A
M:BAS*C
TY:That's right.
TN:I'm not sure what you typed,
:but it isn't the answer to
:this question. Try again.
JN:@A
W:2Ø
```

There are three points worth noting in this example. We can attach **Y** and **N** conditioners to **J** op codes. Continuation lines operate under the conditioners attached to the preceding **TYPE** instruction. Finally, it is not necessary or even desirable to match for the correct answer first.

The other type of destination for a **JUMP** instruction is a label. Labels consist of from one to six characters. The characters must be upper-case letters or digits, but the first character must be an upper-case letter. We use the **JUMP-to-label** to transfer control to some specific line in the program. We indicate the target line by starting it with a label. The computer assumes that each line starts with an op code. We don't want a language which does not allow labels to start with letters used for op codes, and we don't want to confuse the computer. So we warn the computer that what follows is a label by starting the line with an asterisk:

```
J:PART2
....
*PART2
```

-The asterisk is not part of the label; it must not be included in the destination portion of the **JUMP** instruction.

The label may stand alone on the line, or it may share a line with an instruction. If it shares a line, the label must be separated from the op code by at least one space. The two examples below are equivalent:

```
*PART2                  *PART2 T: Next we will...
T: Next we will...
```

Now that we have the **JUMP** and **MATCH** instructions, we can make our examples teach something. Enter the following:

```
TS:Let's use some information
:from track and field meets to
:compare the English and metric
:measures of length.
:
*Q1
T:In North America the shortest
:sprint event is the 1ØØ yd
:dash; in Europe it is the 1ØØ
:meter dash. If the same runner
:ran both events under equiva-
:lent conditions, would the
:time in the 1ØØ meters be
:longer or shorter than the
:time in the 1ØØ yd dash?
:
```

```
A:
T:
M:L*NG
TY:Of course, because 1ØØ meters
:is the longer distance.
WY:3Ø
JY:Q2
```

The pause is to give time to read the response to a correct answer before jumping ahead to the next question (Q2).

```
M:SH*RT
TN:It must be either longer or
:shorter. Type one or the
:other.
JN:@A
```

This section is included to help the student who is unaware of how to respond, or the student who makes numerous typing errors or is trying to outsmart the computer. If the computer gets to the next instruction, the student must have typed a shorter answer (or some misspelling thereof).

```
TS:We're comparing meters with
:yards. Which one is longer?
:
A:
T:
M:MET
TY:Right, and if 1 meter is
:longer than 1 yard, then
:1ØØ meters is longer than
:1ØØ yards. When you press
:ENTER I'll let you try the
:original question again.
AY:
TSHY:
JY:Q1
```

Combinations of modifiers and conditioners are legal. The **JY:** makes any **N** conditioner on following instructions redundant (if the **MATCH** for "meter" had not failed, the **JUMP** to Q1 would have occurred).

```
T:Look
:
:              1 meter = 39.4 inches
:              1 yard  = 36.Ø inches
```

```
:
:Now try again.
J:@A
```

The **JUMP** is to the last **A** run, which is the **A:**, not the **AY:**. Although the **AY:** is closer, if the **AY:** runs, then the **JY:**Q1 will run too, and we will never reach this part of the program:

```
*Q2 TS:OK, now the metric mile
:is contested at 15ØØ meters.
:       etc.
W:2ØØ
```

Once you have entered the program, run it several times with various answers to make sure you try out all the **JUMP**s. For example,

```
Run 1   Answer - nonsense, longer
Run 2   Answer - shorter, nonsense, yards,
        meters
```

12. GIVING HINTS

One way to help a student who is having difficulty is to give a series of hints, each one coming a bit closer to the answer. Color PILOT includes a conditioner which makes hints easy to program.

The computer automatically counts the number of times in a row it uses the same **ACCEPT** instruction. Every time the computer comes to an **ACCEPT** intruction it asks itself, "Is this the same **ACCEPT** as the last **ACCEPT** I ran?" If it is not the same **ACCEPT** instruction, the last **A** counter is reset to one. If it is the same one, the last **A** counter is increased by one. Then if we attach a digit as a conditioner to the op code of any instruction, that instruction will run only if the digit is the same as the value of the last **A** counter.

```
TS:Who was the first President
:of the United States?
A:
MS:W*SHINGT*N
TY:Right.
JY:NEXT
T1:No, his first name was
:George.
T2:No, his picture is on
:the dollar bill.
T3:No, a western state is
:named after him.
T4:Still wrong, the state
:is the location of the city
:of Seattle.
T:                    Try again.
J:@A
*NEXT
W:2Ø
```

Run this program giving a sequence of at least five wrong answers. Notice first that the only way to reach any of the instructions with digit conditioners is to bypass the JY:NEXT instruction - that is, to give a wrong answer. Also notice that the last message in the sequence does not have a digit conditioner on the op code. Without this line, a fifth or subsequent wrong answer would produce no message. The computer would be expecting a response, and so would the student. The usual student reaction would eventually be to press the **ENTER** key a few times (which would lead to no visible effect), and then to give up. Always make sure that the student is furnished with enough hints to continue with the program.

-When using digit conditioners, make provision for a number of tries greater than the largest conditioner.

13. REVIEW - A DIALOGUE

It's time to pull together what we've covered so far. We'll write a one-question dialogue to illustrate the integration of the features:

```
TS:What is the adjective which
:describes this triangle?
G:MØ
G:P5,D1ØØ,112,L125,16Ø,L15Ø,112
G:P2,L1ØØ,112,WØ,96
```

Draw a triangle using color to emphasize the critical feature:

```
T:The two colored sides are the
:same length.
G:WØ,8Ø
A:
M:ISO
MY:C*LES
```

The sequence of two **MATCH** instructions is used instead of M:ISO*C*LES to cover misspellings like "isoceles". Remember that this sequence looks for ISO and C*LES, but ignores anything (or the fact that there is nothing) between.

```
TY:That's it.
WY:3Ø
EY:
```

In a longer program this **END** would be a **JUMP** to the next section:

```
M:RIG
TY:No, a right triangle includes
:a right angle.
GY:P5,D13Ø,Ø,L1ØØ,Ø,L1ØØ,4Ø
GY:P2,L13Ø,Ø
```

Draw a right triangle using color to emphasize the right angle.

```
WY:4Ø
TYS:
JY:@A
```

Clear the answer and the hint; then return for another chance:

```
M:EQUI
MY:LATER
TY:The bottom is shorter than
:the other sides. Equilateral
:means all sides equal.
WY:4Ø
TYS:
JY:@A
M:SCAL
TY:A scalene triangle has no
:equal sides.
WY:4Ø
TYS:
JY:@A
M:OBT*S
TY:I'll draw an obtuse triangle
:to show you the difference.
GY:P5,D13Ø,Ø,L1ØØ,Ø,L8Ø,4Ø
GY:P2,L13Ø,Ø
WY:4Ø
TYS:
JY:@A
M:AC*T
TY:That's true, but it does not
:describe the triangle as having
:two sides of the same length.
WY:4Ø
TYS:
JY:@A
```

The preceding instructions check for possible predictable errors and give an appropriate message. Now we take care of unanticipated errors:

```
T1:I didn't understand that. The
:adjectives used to describe
:triangles in my dictionary are
:acute, equilateral, isosceles,
:obtuse, right, and scalene.
:
:Try one of those.
W1:2Ø
```

This response is longer than the others, so we give more time.

```
T2:Two prefixes mean equal
:equi- and isos-.
T3:The answer is isosceles. Type
:it so you'll remember it.
T4:You're asleep; I quit.
W:2Ø
E4:
W:3Ø
```

Response 4 will get a two-second wait; response 2 and 3 will get five-second waits, and response 1 will get seven seconds.

```
TS:
J:@A
```

Enter this program and try running it. Try more complex answers than simple one-word replies. The result should be a dialogue somewhat like a conversation between a student and a teacher.

14. NUMERICAL QUESTIONS

In the last few chapters we have concentrated on processing textual answers. Now we turn to numerical answers which are different in one important respect. A number has a value which can be different from the correct answer but still close enough, and greater than or less than the correct answer. If the correct answer to a question is 99, the answer 1ØØ may be acceptable even though the characters typed are completely different. For this reason, we often want to handle numbers in a way which makes judgements of "close", "greater", and "less" easy to program. To make these kinds of judgements, we use relational conditioners.

The following are examples of relations:

```
X = 3
X + Y > 5       (read > as "is greater than")
Y < 1Ø          (read < as "is less than")
Z >= 3          (read >= as "is greater than or equal to")
Y + Z <= 15     (read <= as "is less than or equal to")
P < > 1Ø        (read < > as "is not equal to")
```

A relation has a truth value; it is either true or false. For example, the first relation on the list is true if the current value of X is 3, and it is false otherwise.

Relations can involve strings too:

```
X$ = "ABC"
X$ <> "xyz"
```

Relations can be compound:

```
X > 5 & M$ = "ABC"
X = 6 ! M$ = D$
```

where & means AND and ! means OR.

A relation can be attached to any op code as a conditioner. The instruction will run if the relational conditioner is true, and it will be skipped if the relational conditioner is false. The syntax for a typical relational conditioner is:

```
T(X=3):That's correct.
```

Relational conditioners can be combined with modifiers and other conditioners (e.g., Y and N). The only syntax rules are that spaces are allowed only within the parentheses, and that the relational conditioner must come last.

It's time for an example. We'll do a simple arithmetic drill; large characters are probably best for potential students at this level.

```
G:M4,P2
TS:Simple Addition
:
:    5
:  + 3
G:D3Ø,13Ø,L8Ø,13Ø
TH:                (four spaces)
```

This TH: instruction should contain enough spaces to align the cursor under the 5 and 3. Space at the ends of lines are not visible in printouts.

```
A:
C:X=%B
T(X=8):Good.
J(X=8):NEXT
T(X>8):Too high; try
T(X<8):Too low; try
T:again.
J:@A
*NEXT W:2Ø
```

As usual, we urge you to enter this program and run it trying all reasonable errors. The use of the relational conditioners is straightforward here; the only step that is not obvious is the **COMPUTE** instruction. The instruction C:X=%B forces the conversion of the answer into numerical form for future reference in the other relational conditioners. In fact, this instruction is unnecessary because the conversions in the relations are automatic. Because of the automatic conversion from string to number, the following program will do exactly what the previous one does:

```
G:M4,P2
TS:Simple Addition
:
:    5
:  + 3
G:D3Ø,13Ø,L8Ø,13Ø
TH:              (four spaces)
A:
```

```
T(8=%B):Good
J(8=%B):NEXT
T(8<%B):Too high; try
T(8>%B):Too low; try
T:again.
J:@A
*NEXT W:2Ø
```

There is one subtle point here: Notice that we reversed the order from X=8 to 8=%B. There is no difference between X=8 and 8=X, because the quantities on both sides of each relation are numbers; however, there may be a difference between 8=%B and %B=8, because there is a conversion between string and number implied. Which way will the computer convert? The computer converts to the form on the left. Thus if we use the form 8=%B, the computer converts %B to a number. This relation will then be true if the answer is "8" or "It's 8." If we use the form %B=8, the computer will convert the string, which will be the same as the answer "8" but different from the answer "It's 8."

Which of our two ways of programming the problem is better, the one using X in the relational conditioners or the one using %B in the relational conditioners? It depends on your definition of "better." The first version will run a little faster, but so little as to be undetectable. The second version requires a bit less typing, but again so little as to be insignificant. The best definition of "better" - here and almost everywhere in computer programming - is the way which is clearest to the programmer.

-When there are two or more ways to program the same effect, pick the one which will be the easiest to understand a month from now.

There is one change we can make in the program which is clearly an improvement; the program will run faster, take less typing, and be easier to understand. The two instructions which handle the correct answer,

```
T(8=%B):Good.
J(8=%B):NEXT
```

tell the computer to check the truth of the same relation twice in a row, obviously getting the same result both times. The computer remembers the truth value of the last relational conditioner evaluated, and by using the C conditioner this truth value can be reused again and again without reevaluation. The above two instructions can be replaced with:

```
T(8=%B):Good.
JC:NEXT
```

This can save typing, space, and computation in a complex program. More important, it can clarify the structure of a program by grouping together those instructions which all depend on a single relational conditioner.

Arithmetic drill is an obvious educational application for the computer, and Color PILOT makes programming of such drill very easy. The rest of this chapter is mainly concerned with showing how the simple program above can be extended into a general addition program, similar to those arithmetic drill programs costing hundreds of dollars.

The key to the extension is the use of the random number generator to generate many problems. We can ask the computer to generate random numbers between Ø and some largest integer by using the **RND** function.

-A single die (from a pair of dice) is an example of a random number generator for the integers between 1 and 6.

If we use the **COMPUTE** instruction:

```
C: X = RND(4)
```

then, after the instruction is run, X will have one of the four values, Ø, 1, 2, or 3. If we use the **COMPUTE** instruction:

```
C: X = RND(8)
```

then, after the instruction is run, X will have one of the eight values, Ø, 1, 2, 3, 4, 5, 6, or 7. So the function **RND** causes the computer to choose at random one of the integers from a list starting with Ø and ending at the "number in parentheses minus 1". It's time for an example; enter and run the following program:

```
TS:Look at random numbers.
*MORE C: X = RND(1Ø)
TH:  #X
C: Y = Y+1
J(Y<2Ø):MORE
W:2ØØ
```

Notice that the numbers produced are always less than 1Ø, and are always Ø or greater.

We can make use of this random number generator to generate the two addends. To present a problem for one-digit addition, we might proceed as follows:

```
TS:Simple Addition
:
C: Y = RND(1Ø)
T:  #Y
C: Z = RND(1Ø)
T: +#Z
```

Here Y will be a number between Ø and 9, perhaps 8, and Z will also be a number between Ø and 9, perhaps 7. So the sum could come out to be a number greater than 9, which will introduce the complication of carrying and two-digit answers. Carrying might not be appropriate for the students in question. There are at least two ways to avoid this complication. One is to reject any set of numbers which leads to a sum that is too large:

```
TS:Simple Addition
:
*ANOTH C: Y = RND(1Ø)
C: Z = RND(1Ø)
J(Y+Z>9):ANOTH
T:   #Y
:  +#Z
etc.
```

A second way is to choose one addend and the sum, and to compute the second addend:

```
TS:Simple Addition
:
*ANOTH C: Y = RND(1Ø)
C: S = RND(1Ø)
C: Z = S - Y
J(X<Ø):ANOTH
T:    #Y
:   +#Z
etc.
```

There is little basis for choice between the two methods here. (In long division problems without remainders, it makes sense to pick the divisor and the answer instead of the dividend.) We'll stick with the first method.

Now we complete the program for one problem:

```
G:M4,P2
*PROB TS:Simple Addition
:
*ANOTH C: Y = RND(1Ø)
C: Z = RND(1Ø)
J(Y+Z>9):ANOTH
T:    #Y
:   +#Z
G:D3Ø,13Ø,L8Ø,13Ø
TH:          (four spaces)
A:
T(Y+Z=%B):Good.
JC:NEXT
T(Y+Z<%B):Too high; try
T(Y+Z>%B):Too low; try
T:again
W:2Ø
J:@A
*NEXT W:2Ø
```

Notice that the randomly generated variables, Y and Z, are used in relational conditioners and on the left side to force proper conversion.

The payoff for using the random number generator is reached when we use this code in a program which generates a number of problems. Let's say we want the student to complete twenty problems. We can do this by the following addition:

```
*NEXT W:2Ø
C: P = P+1
J(P<2Ø):PROB
T:You've completed
:#P  problems
W:2Ø
```

In this manual, we are mainly concerned with the mechanics of programming in Color PILOT, not with instructional design. But this is a good place for an aside on design. We must always be careful that the messages on the screen do not overreach the likely reading ability of the typical student who might use the program. In creating this program, we should ask if a student who needs drill on one-digit addition will be able to read messages like "Too high; try again." If we decide that they may not be able to read these kinds of messages, we will have to find some other way to communicate the message: for example, with up- and down-arrows. In our example we'll stick with the textual messages because they are easier for the beginning programmer to keep in mind than a series of **GRAPH** instructions. If we were preparing this for use with children, we would replace the text messages with graphics.

We now have an arithmetic drill, but the format is not ideal. If the student makes an error, the new answer does not go into the right position. We can fix this by erasing the incorrect answer and positioning the cursor for the new answer. Replace the sequence:

```
G:D3Ø,13Ø,L8Ø,13Ø
TH:          (four spaces)
A:
...
J:@A
```

with:

```
G:D3Ø,13Ø,L8Ø,13Ø
*ERR G:W2Ø,12Ø
THS:         (four spaces)
A:
...
J:ERR
```

Here we use the G:W2Ø,12Ø to position the text cursor, and the following THS: to erase the previous answer by overtyping with spaces.

Before going further, let's pull this together into a useful drill for one-digit addition:

```
G:M4,P2
*PROB GWØ,191
TS:Simple Addition
:
*ANOTH C: Y = RND(1Ø)
C: Z = RND(1Ø)
J(Y+Z>9):ANOTH
T:    #Y
:   +#Z
G:D3Ø,13Ø,L8Ø,13Ø
*ERR G:W2Ø,12Ø
THS:         (four spaces)
A:
T(Y+Z=%B):Good.
JC:NEXT
T(Y+Z<%B):Too high; try
T(Y+Z>%B):Too low; try
T:again.
W:2Ø
J:ERR
*NEXT W:2Ø
C: P = P+1
G:WØ,144
J(P<2Ø):PROB
TS:You've completed
:#P  problems.
W:3Ø
```

There are many elaborations we could add to this program, most of which raise questions of teaching strategy. We will limit ourselves to two possibilities for the purpose of illustration. In the first extension we will allow for problems with carrying.

Enter the following program but, as usual, not the comments:

```
G:M4,P2
TS:Simple Addition
:
W:2Ø
```

The window position will be shifted between the units column, the tens column, and the message space. These two instructions restore the window position and clear it for a new problem:

```
*ANOTH G:WØ,191
TS:
```

We make no provision for the case Y + Z > 9 because we're going to allow problems with carries here.

```
C: Y = RND(1Ø)
C: Z = RND(1Ø)
```

Erase the possible error - a two-digit response - by typing over with blanks. This time we moved the line under the addends up a bit to avoid the erase:

```
T:  #Y
: +#Z
G:D2Ø,14Ø,L7Ø,14Ø
*ERR G:W12,12Ø
THS:        (four spaces)
```

Position the cursor to accept the units digit first:

```
G:W5Ø,12Ø
```

Transfer the units digit to the variable U so that there is room in %B for the tens digit. The variable T, which will hold the tens digit, is given a value of Ø in case there is no carry. (Otherwise it would keep some value left over from a previous problem with a carry.)

```
AS:
C: U = %B
C: T = Ø
```

Get and convert the tens digit, if appropriate. The teaching strategy here is open to debate. Are we giving too much away by not allowing the student to report a false carry? Should the student have to fill in a Ø or a blank if there is no carry?

When we are beginning to worry about teaching strategy instead of programming details, we are indeed making progress!

```
G(Y+Z>9):W35,12Ø
ACS:
CC: T = %B
```

Move to the message space below the problem:

```
G:WØ,9Ø
```

Here at last is an example of a student error we can diagnose and give a remedy for:

```
T(Y+Z=1Ø*T+U):Good
JC: NEXT
T((Y+Z>9)&(T<>1)):Good grief,
TC: you forgot the
:carry; try again.
```

We use the **HANG** modifier on the first line because the length of the relational conditioner prevents us from putting much text on the line. Time to read the message, then clear it:

```
WC:2Ø
TCS:
```

The rest of the program provides for other types of student errors and calculates the student's score:

```
JC:ERR
T(Y+Z<1Ø*T+U):Too high; try
T(Y+Z>1Ø*T+U):Too low; try
T:again.
W:2Ø
J:ERR
TS:
*NEXT W:2Ø
C: P = P+1
J(P<2Ø):ANOTH
T:You have done
:#P  problems.
W:3Ø
```

As we increase the number of times the student uses this program, we may begin to worry about repetitions of the same problems. One strategy is to make a list of problems, and to remove problems from the list when the student gets them right. There are 1Ø possible values for Y (Ø-9) and 1Ø possible for Z. Thus there are:

1Ø * 1Ø or 1ØØ

possible different problems of this type (counting 4 + 5 and 5 + 4 as different problems).

There are many different ways we could handle the record keeping in this part of the program. One thing we don't want to do is to actually keep the problems themselves in the list because this would make the list very long. In fact, in order to make sure that each problem is only used once, all we need is one character space which can hold one of the symbols to represent each possible problem, one symbol for USE and another for DON'T USE. We'll make a list of 1ØØ U's (for use) and turn them into Ø's when that particular combination is used. We start by setting up the string of 1ØØ U's in the string variable R$:

```
D:R$(1ØØ)
C:R$="UUUUUUUUUUUUUUUUUUUUUUUUU" (25 U's)
C:R$=R$!!R$
C:R$=R$!!R$
```

The second **COMPUTE** instruction doubles the length of the list to 5Ø, and the third **COMPUTE** instruction doubles it again to 1ØØ.

Most of the program is the same as our previous program, but we'll use small characters so we can display the list on the screen for testing:

```
TS:Simple Additions
:
W:1Ø
*ANOTH G:MØ,P2,WØ,191
TS:
C: Y = RND(1Ø)
C: Z = RND(1Ø)
```

We must check whether this combination has been used before we write it on the screen. We assign each combination of Y and Z a unique position in the list of 1ØØ entries by the formula:

```
position = 1Ø * Y + Z + 1
```

We arrive at this formula by noting first that there are 1Ø possible values for Z, which gives the factor of 1Ø in 1Ø * Y (for every value of Y there must be room for 1Ø different values of Z). Second, we note that the list starts at position 1 and that the lowest set of Y and Z are the values Ø,Ø. Therefore we must add 1 to start at position 1 for the lowest set.

```
J(R$(1Ø*Y+Z+1)<>"U"):ANOTH
```

If the combination has been used before, then go get another combination.

We continue for a while as in our earlier program:

```
T:   #Y
:  +#Z
G:D2,16Ø,L3Ø,16Ø
*ERR G:W12,15Ø
THS:   (four spaces)
G:W24,15Ø
AS:
C: U = %B
C: T = Ø
G(Y+Z>9):W16,15Ø
ACS:
CC:T = %B
G:WØ,9Ø
T(Y+Z=1Ø*T+U):Good
CC: R$(1Ø*Y+Z+1)="O"
```

If the student got the problem right, then remove it from the list. We use the character "O" instead of the number zero because a number occupies two character positions. Next we put in two instructions which would not be in the final program but which will allow us to see what is happening to the list as we go:

```
TC:$R$
WC:2Ø
```

From here on we continue as before:

```
JC:NEXT
T((Y+Z>9)&(T<>1):
TC:You forgot the
:carry, try again.
WC:2Ø
TCS:
JC:ERR
T(Y+Z<1Ø*T+U):Too high; try
T(Y+Z>1Ø*T+U):Too low; try
T:again.
W:2Ø
TS:
J:ERR
*NEXT W:2Ø
C: P = P+1
J(P<5Ø)*ANOTH
```

We could use this for more problems, but if we get too close to 1ØØ (say over 9Ø), it may take the computer a while to stumble randomly onto the unused combinations.

```
T:You've completed
:#P problems.
W:3Ø
```

Within the examples in this chapter, we have made a number of assumptions about teaching strategies, some of which already have been pointed out. In addition to those assumptions, we must consider how many problems of a particular type should be given to a student at a sitting, and whether a problem should be removed from the list if the student doesn't answer it correctly on the first try. Our purpose here is to show how the Color PILOT language can be used by example. If you prefer other strategies, it's easy to alter the programs to match them. Try it.

Throughout the chapter we stuck with simple addition problems. However, contained within the examples are all the techniques you would need to develop a complete set of programs for integer arithmetic. For example, the addition program could be extended to cover multi-digit addition or subtraction. Multiplication and division require only different expressions in the relation conditioners and more elaborate cursor movement for multi-digit problems. In these kinds of applications, it is easy to get carried away and write THE all-purpose program. Remember that at the level these basic arithmetic programs will be used for, the students will probably have a very limited attention span. They will not be able to cover everything from simple addition to long division with remainders in just one sitting.

By restricting our examples to integer addition, we have illustrated "greater than" and "less than," but not "close enough." Let's finish with a short example which covers this concept. Suppose we were asking the student to estimate the number of yards in a mile, and we would accept any answer between 16ØØ and 18ØØ as close enough. We could do the following:

```
TS:Guess how many yards in
:a mile.
:
A:
C: X = %B
T(X>16ØØ & X<18ØØ):Close enough.
JC:NEXT
T(X<16Ø1):Too low.
T(X>1799):Too high.
T:Try again.
J:@A
*NEXT W:2Ø
```

Notice the use of the **AND** operator (&) in the relational conditioner.

Another and perhaps preferable way to handle "close enough" is to use the **ABS** function. The **ABS** function takes the absolute value of whatever is inside the parentheses. The instruction:

```
C: X = ABS (3 - 5)
```

sets X equal to 2. Another way to think of **ABS** is that it throws away the minus sign on a negative number. Using the **ABS** function, we can change the above program to:

```
TS:Guess how many yards in
:a mile.
:
A:
C: X = %B
T(ABS(X-17ØØ)<1ØØ):Close enough.
JC:NEXT
etc.
```

The relational conditioner looks for a number within 1ØØ of 17ØØ; the **ABS** function is used to say that we don't care if the variable is above or below. In most cases, use of the **ABS** function is more easily understood by the programmer than relational conditioners.

15. FRAME DESIGN

We have introduced most of the features of the **GRAPH** instruction in Color PILOT. We can use these features to manage the screen display in a variety of ways - as an endless scroll, as a series of frames, or as a single frame with changing entries. In this chapter we will work through an example of a single frame with changing entries, as this will illustrate the various techniques we have for screen management.

Our example will be a sample check register drill. At the top of the screen we will have a check record page, and we will split the rest of the screen into a question-answer space and a response-message space.

This application will make heavy use of the window operation to position text. We could proceed as before by trial and error, but instead let's be systematic. The screen holds 24 lines of text. Each character on a line occupies a block eight dots high. (Notice that 8 * 24 = 192, the vertical dimension of the screen). A window coordinate which lies within a block is treated as being at the lower left corner of the block. From that information, or by trial and error, we can build the following table:

LINES UP	LINES DOWN	POSITION	LARGE CHARACTER
24	1	184-255	
23	2	176-183	176-255
22	3	168-175	
21	4	16Ø-167	16Ø-175
2Ø	5	152-159	
9	6	144-151	144-159
18	7	136-143	
17	8	128-135	128-143
16	9	12Ø-127	
15	1Ø	112-119	112-127
14	11	1Ø4-111	
13	12	96-1Ø3	96-111
12	13	88-95	
11	14	8Ø-87	8Ø-95
1Ø	15	72-79	
9	16	64-71	64-79
8	17	56-63	
7	18	48-55	48-63
6	19	4Ø-47	
5	2Ø	32-39	32-47
4	21	24-31	
3	22	16-23	16-31
2	23	8-15	
1	24	Ø-7	Ø-15

Large characters occupy double-sized blocks. You may prefer to remember the formula instead of consulting the table. Using the numbers from the bottom up, you can see that the formula is:

corner position = (line number - 1) * 8

A similar table for the 32 columns will be useful late at night when your brain has turned to putty and your eyes will no longer focus:

Column	Position
1	Ø - 7
2	8 - 15
3	16 - 23
4	24 - 31
5	32 - 39
6	4Ø - 47
7	48 - 55
8	56 - 63
9	64 - 71
1Ø	72 - 79
11	8Ø - 87
12	88 - 95
13	96 - 1Ø3
14	1Ø4 - 111
15	112 - 119
16	12Ø - 127
17	128 - 135
18	136 - 143
19	144 - 151
2Ø	152 - 159
21	16Ø - 167
22	168 - 175
23	176 - 183
24	184 - 191
25	192 - 199
26	2ØØ - 2Ø7
27	2Ø8 - 215
28	216 - 223
29	224 - 231
3Ø	232 - 239
31	24Ø - 247
32	248 - 255

Again, the formula is:

corner position = (line number - 1) * 8

and large characters occupy double blocks (1,3,5,etc).

The program starts with a title page that says "CHECKBOOK" in large letters and gives the author's name in small letters. We'll put it all in a colored box. Type:

```
G:M7,E,P5,D2Ø,2Ø,B235,171
G:P7,D3Ø,3Ø,B225,161
W:2ØØ
```

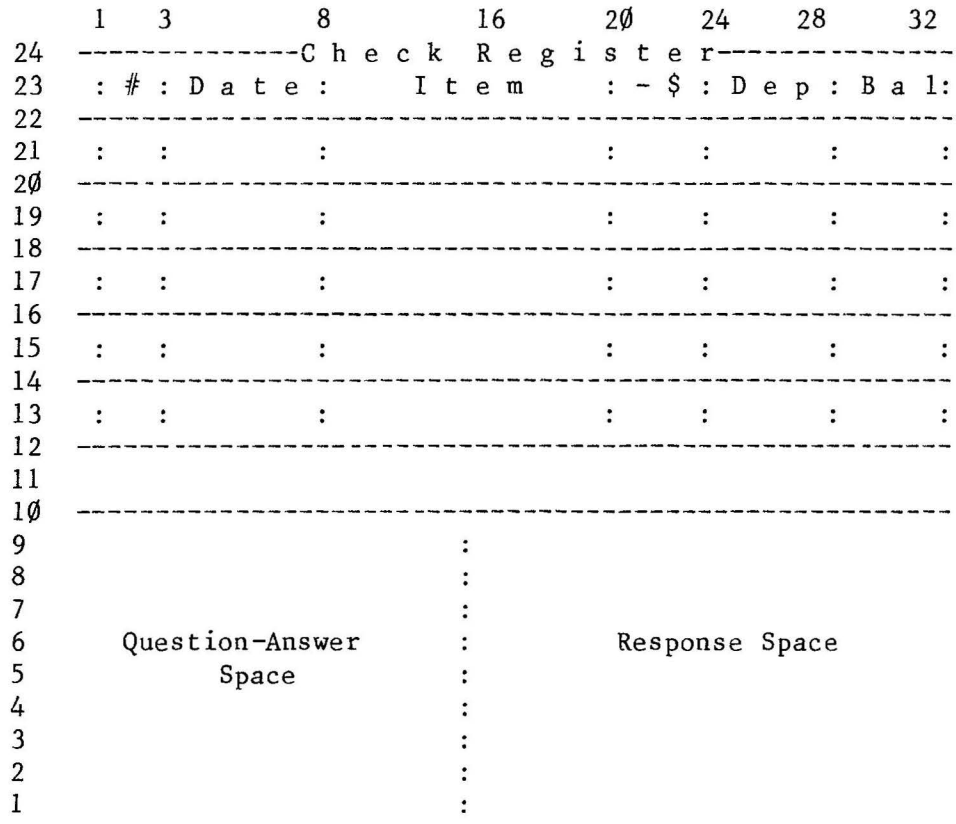
We also could draw a frame as a series of four blocks, but the time this would save is not really worth doing the extra programming.

Next, we insert the title slightly above the center of the frame. Remove the **WAIT** instruction (it was just there to give us time for admiration) and add:

```
G:M4,W48,96
T:CHECKBOOK
G:MØ,W112,8Ø
T:by
G:W88,64
T:Your name
W:3Ø
```

Coordinates for the window were selected from the tables so that "by" appears in line 11, column 15 and "Your name" appears in line 9, column 12. You may want to adjust the horizontal coordinate of the last window if your name has a different length than the phrase "Your name."

The rest of the program will use a single frame. First we design the frame by picking lines and columns. The frame design we'll use is on the next page.



We have to live with the limitations of the display and the computer; we have to abbreviate column headings, and we have to use only dollars (no cents). Once the form is designed, we can write the program to display it.

First we outline the check register, using a double line to give more color:

```
G:E,P6,D4,92,L4,188,L251,188
G:L251,92,L2,92,L2,189,L253,189
G:L253,91,L2,91
```

The coordinates are selected so as to put horizontal lines in the centers of lines 24 and 12 and vertical lines in the centers of columns 1 and 32.

The next section draws horizontal lines in the centers of lines 14, 16, 18, 2Ø, and 22 and writes the title:

```
G:P2,D5,1Ø8,L251,1Ø8,D5,124
G:L251,124,D5,14Ø,L251,14Ø
G:D5,156,L251,156,D5,172
G:L251,172,W64,184
T: Check Register
```

Vertical lines are drawn in columns 3, 8, 2Ø, 24, and 28. For example, to draw a vertical line in the center of column 2Ø, the X coordinate must be at:

```
(2Ø - 1) * 8 + 4 = 156
```

```
G:D2Ø,187,L2Ø,92,D6Ø,187,L6Ø,92
G:D156,187,L156,92,D188,187
G:L188,92,D216,187,L216,92
```

A series of window-type operations supply the column headings. The coordinates are taken from the table using line 23 as the vertical position:

```
G:W8,176
T:#
G:W24,176
T:Date
G:W88,176
T:Item
G:W168,176
T:-$
G:W192,176
T:Dep
G:W224,176
T:Bal
```

An alternative to this approach would be to insert the headings with a single **TYPE** instruction, using appropriate spacing in the message space. This would have to be done before the vertical lines were drawn; otherwise, the spaces would erase the tops of the lines. In this example we are not trying for the most compact program; if we were, we might use the variable Y=176 in each of the preceding window instructions.

The last step in creating the form is to block off two areas at the bottom of the screen:

```
G:P5,D4,76,L251,76,D124,76
G:L124,Ø
```

This draws a horizontal line in text line 1Ø, so the window coordinates for these two spaces will be:

Ø,64	128,64
Left	Right

Now we're ready to begin using the form:

```
G:WØ,64
T:Your paycheck
:is for $ 9ØØ,
:and you keep
:$ 1ØØ cash. How
:much are you
:starting the
:account with?
```

Notice that we keep the lines short enough to fit in the left area. We will describe the reason for the space after the $ in Chapter 18.

```
A:
M:8ØØ
GN:W128,64
```

Move to the response (right) area:

```
TN:Now wait.
: 9ØØ - 1ØØ
:is not $%B.
:Try again.
WN:3Ø
TSN:
GN:P7,DØ,Ø,B118,15,WØ,8
JN:@A
```

The last **GRAPH** instruction shows a way to erase selectively; draw a block over the text to be removed using the background color. The following code erases the original question in the left area:

```
G:W128,64
T:Good.
*BACK
G:P7,DØ,Ø,B118,75,WØ,64
T:In which
:column should
:we put this?
:
G:W128,64
TS:
```

This erases the "Good." from the response space:

```
G:WØ,24
A:
G:W128,64
```

Move to the response space:

```
M:DEP
TY:Right, I'll
:put it there.
*ENT GY:W192,16Ø
TY:8ØØ
JY:ON
M:BAL
TY:You're rushing
:things. It's a
:deposit.
WY:2Ø
JY:ENT
M:ITEM!#!DATE!-$
TN:That's not a
:column.
WN:2Ø
TNS:
JN:BACK
T:No, this is
:a sum of money
:you're putting
:    in.
W:2Ø
TS:
J:BACK
*ON W:2Ø
```

This completes one question. We clean up both spaces and continue.

```
G:DØ,Ø,B123,75,W128,64
TS:
G:WØ,64
T:This makes the balance what?
:
A:
```

This is obviously just the start of a program which could be greatly expanded. We will not do so here. The most important techniques to absorb from this example are the uses of windows for positioning text and selective clear, and the use of blocks of background color for selective erase. While we have not used variables or expressions in this example, remember that they can be used in place of numbers in **GRAPH** instructions.

16. SUBROUTINES

In some applications, we may want to repeat the same series of computer steps a number of places within a program. One approach would be to repeat the series of instructions at each place in the program that they are needed, but this is wasteful of memory and programmer time. Color PILOT, like most computer languages, has an instruction which handles this kind of repetition easily. It is the **USE** subroutine (**U:**) instruction.

Before we get into the details of subroutines, let's choose an application where they might be useful. We're going to develop a map-reading exercise in which the student has to direct an object through the streets of a city by giving directions. To have the largest possible map on the screen, we'll have to alternate displays of the instructions and the map. The following instructions will draw a map:

```
G:M4,WØ,15Ø
T:A Map Reading
:
:    Exercise
W:2Ø
TS:
G:MØ,W1ØØ,18Ø
T:North
G:WØ,1Ø5
T:W                    E  (maximum length
T:e                    a   lines are here)
T:s                    s
T:t                    t
G:W1ØØ,Ø
TH:South
```

-An easy way to get maximum length lines is to enter the start and end, and then to insert spaces in the middle.

The **HANG** modifier is needed to keep "South" on the bottom line; a **T:** instruction would produce a carriage return which would push "South" up a line. The last three instructions below draw a horizontal row of blocks across the screen:

```
C: X = 15
C: Y = 15
*BLOCK G:P5,DX,Y,BX+2Ø,Y+2Ø
C: X = X+35
J(X<22Ø):BLOCK
```

These two instructions reset X at the start of another row and move Y up one vertical row:

```
C: X = 15
C: Y = Y+35
```

Enter the following code; add a long **WAIT** instruction on the end, and try it out.

```
J(Y<17Ø):BLOCK
```

That's a pretty good display for a relatively small number of instructions, but that doesn't mean that we want to retype them over and over. So let's plan what we want to happen and learn to use the **USE** instruction. We organize the program in the following way:

1. Title Page
2. Tell about map.
3. Draw map.
4. Tell about start position.
5. Draw map with position.
6. Tell about destination.
7. Draw map with position and destination.
8. Explain how to move.
9. Show map and allow moves.

Notice that we are going to have to draw the map at least four times.

We separate out those instructions which actually draw the map and give them the label MAP.

```
*MAP G:MØ,W1ØØ,18Ø
T:North
G:WØ,1Ø5
T:W                    E
T:e                    a
T:s                    s
T:t                    t
G:W1ØØ,Ø
TH:South
C:X=15
C:Y=15
*BLOCK G:P5,DX,Y,BX+2Ø,Y+2Ø
C:X=X+35
J(X<22Ø):BLOCK
C:X=15
C:Y=Y+35
J(Y<17Ø):BLOCK
E:
```

This set of instructions was included in what we had before, but there are two important additions: the subroutine has a beginning - a label (in this case MAP), and the subroutine has an end - the **END** (**E:**) instruction.

-A label to begin and an **END** instruction to end must be present in every subroutine.

To see how the subroutine is used, we must write a portion of the main program:

```
G:M4,WØ,15Ø
T:A Map Reading
:
:    Exercise
W:2Ø
TS:
G:MØ,WØ,1ØØ
T:When you push ENTER, I'll
:show you a simple map of the
:center of a city.
A:
TS:
U:MAP
W:3Ø
G:E,WØ,1ØØ
TS:Your current position is
:marked with an X.
W:2Ø
TS:
U:MAP
G:W4Ø,4Ø
T:X
W:3Ø
```

The map-drawing subroutine is used twice in this part of the program. The form of the **USE** instruction is:

```
U:label
```

in this case:

```
U:MAP
```

This instruction tells the computer to jump to the label MAP, but to remember where it was in the program before the jump. The computer runs the instructions starting at the label MAP until it finds an **END (E:)** instruction. Then the computer jumps back to the instruction following the **USE** instruction. In our sample program, the sequence for the first **USE** instruction is:

```
TS:         -Clear screen
U:MAP       -Go draw the map and then return
W:3Ø        -Pause three seconds
```

and the sequence for the second **USE** instruction is:

```
TS:         -Clear screen
U:MAP       -Go draw the map and then return
G:W4Ø,4Ø    -Set the window
```

Before we can try this out, we have to have both the main program and the subroutine in the memory. Where should we put the subroutine? While we can put it anywhere in the program, the performance will be better if we put it close to the beginning. But there is a complication related to the **END** instruction; the **END** instruction also is used to end a program. If the computer finds an **END** instruction when it is not under the influence of a **USE** instruction (i,e., when it does not have a return point stored), then the computer stops running the program. Therefore, if the computer were to get into a subroutine by simply working its way down a list of instructions, instead of jumping in via a **USE** instruction, it would quit when it hit the **END** instruction of the subroutine.

The problem is easily avoided. Probably the best arrangement is the following:

```
J:PAST
*MAP G:MØ,W1ØØ,18Ø
. . .
J(Y<17Ø):BLOCK
E:
*PAST
G:M4,WØ,15Ø
etc.
```

The first instruction and the label PAST are needed to prevent accidental entry into the subroutine.

One other change will be needed to make the program run smoothly. We want to be able to move the X to any intersection, but we can type the letter X only at one of the 32 character positions on each of the 24 lines. Instead of typing a letter, we'll draw an X using the variable U and V for positions (we set the values of U and V before this instruction is run):

```
G:DU,V,LU+8,V+8,DU,V+8,LU+8,V
```

Because we are learning about subroutines, let's make this a short subroutine as well. Put it up at the front, all subroutines together:

```
J:PAST
*MAP G:MØ,W1ØØ,18Ø
. . .
J(Y<17Ø):BLOCK
E:
*X
G:P2,DU,V,LU+8,V+8,DU,V+8,LU+8,V
E:
*PAST
```

and alter the last few lines of the program:

```
T:Your current position is
:marked with an X.
W:2Ø
TS:
U:MAP
C:U=37
C:V=39
U:X
W:3Ø
```

Now continue with the main program:

```
G:E,WØ,1ØØ
T:Your destination is marked
:with a box.
W:2Ø
TS:
U:MAP
U:X
G:P6,D177,142,B185,15Ø
W:3Ø
G:E,WØ,1ØØ
T:You can move by typing one
of the four directions.
:
:To move - type N, S, E, or W.
:
:Press ENTER to begin.
A:
TS:
*RE U:MAP
U:X
G:P6,D178,144,B186,152,W15Ø,Ø
```

This really completes the illustration of subroutines, but we may as well turn the program into something useable. First, add instructions to set the position for the new X:

```
*MOVE TH:N,S,E, or W?
AS:
M:N!n
CY:V=V+35
M:S!s
CY:V=V-35
M:E!e
CY:U=U+35
M:W!w
CY:U=U-35
```

Then restart if the last move puts us off the edge:

```
G(V<Ø!V>19Ø!U<Ø!U>23Ø):E,M4
GC:W6Ø,1ØØ
TC:   Lost
:
:Off the map!
WC:5Ø
TSC:
CC:U=37
CC:V=39
JC:RE
```

Type X in the new position:

```
U:X
```

Quit if we reached the destination:

```
G(V=144&U=177):E,M4,W3Ø,1ØØ
TC:Congratulations
:
:You made it.
WC:5Ø
EC:
```

And go back for another move:

```
J:MOVE
```

As it stands, this program is complete and functional. However, improvements to programs are usually possible, especially when graphics are involved. One improvement would be to display only one X for current position, but to leave a track showing where we've been. This can be done by making the following changes. First, add the ERASE: subroutine to the subroutines:

```
*ERASE
G:P3,DM,N,LM+8,N+8,DM,N+8,LM+8,N
G:P2,DM+4,N+4,LU+4,V+4
E:
```

This draws over the X with the background color, thus erasing it. It then draws a track. We must alter the latter part of the main program, starting after the label MOVE:

```
*MOVE TH:N,S,E, or W?
AS:
M:N!n!S!s!E!e!W!w!
JN:MOVE
C:M=U
C:N=V
```

The variables M and N keep track of the old position so that U and V can get the new position. Notice that nonsense answers are rejected first, as before:

```
M:N!n
CY:V=V+35
 etc.
```

Also insert the instruction:

```
U:ERASE
```

immediately before the U:X instruction.

-By now you may be having trouble keeping track of all the changes. There is a listing of the final version of this program in Appendix III.

A different extension of the program would be to add another level of use, one in which the student gives a complete set of moves at one time. The computer then runs the moves in the sequence given. Instead of ending the program (the **EC:** instruction), use a **JUMP** instruction (JC:PART2).

We continue, starting at PART2:

```
*PART2 G:E,MØ,WØ,1ØØ
T:Now you tell me all the
:moves with one line of letters.
:
:Example,  4 Norths and 1 East
:           by
:NNNNE
:
:                Push ENTER.
A:
*RE2 TS:
U:MAP
C:U=142
C:V=74
U:X
D:B$(2Ø)
G:P6,D2,39,B1Ø,47,W15Ø,Ø
TH:List moves:
*AA C:C=KEY(Ø)
J(C=Ø):AA
C:B$=CHR(C)
TH:$B$
A:
C:B$=B$!!%B
```

These lines require a bit of explanation. The message "List moves:" comes right at the end of the screen. The next key pressed will trigger a carriage return; it will not appear on the screen, although it would be in the answer buffer. Therefore, to get a complete list displayed after the carriage return, we use the **KEY** function.

The **KEY** function asks whether or not a key has been pressed. The zero after the word **KEY** has no particular significance but must always be included in the **KEY** statement. The **COMPUTE** statement above sets the variable X to Ø if no key has been pressed. Or, if a key has been pressed, X is given the ASCII value of the key: For instance, the "A" key gives a value of 65 (see Appendix I). Unlike the **ACCEPT** statement, which always waits for the user to enter a response, the **KEY** function just returns a value based on the present state of the keys and then continues immediately with the next statement, whether or not a key has been pressed. The **KEY** function can be used to check the keyboard occasionally without stopping the progress of the program. In the example above, the **KEY** instruction simply holds the program in a loop until a key is pressed, but there are many other ways to use this function. For example, the **KEY** function can be used with another statement to interrupt the program once a key is pressed. Or the **KEY** function could work as a timer which would count down until the student responded and give the number of seconds that the student took to respond to a particular question.

Copy the list of moves into B$; find out how long the list is (the **LEN** function gives the current length of the string mentioned in parantheses), and use A as a position marker in the list:

```
C:L=LEN(B$)
C:A=1
```

These instructions pick one character off the list and treat it as the current move:

```
*MOVE2 C:%B=B$(A)
C:A=A+1
```

At this point, copy in all the instructions from the instruction M:N!n!S!s!E!e!W!w to the last instruction (J:MOVE). Make the following replacements:

```
JC:RE ------------> JC:RE2
G(V=144 & U=177)--> G(V=39 & U=2)
CC:U=37 ----------> CC:U=142
CC:V=39 ----------> CC:V=74
J:MOVE-----------> J(A<=L):MOVE2
```

And add the following new line at the bottom of this list:

```
J:AA
```

Notice that we have had to retype a number of lines of code in this last addition. Could we make this a subroutine? Not in Color PILOT because there are **USE** instructions included in the code in question. In the cassette version, it is not possible to call subroutines from within other subroutines.

We will close this chapter by suggesting, but not completely documenting, another extension. Both the starting position and the destination could be selected at random. The starting position can be picked from the following values of U and V which correspond to intersections within the map:

U	V
37	39
72	74
1Ø7	1Ø9
142	144
177	

For U, we want to pick one of five values:

```
C: U = RND(5) * 35 + 37
```

and:

```
C: V = RND(4) * 35 + 39
```

The position of the destination can be picked in similar fashion. The one other change needed is to make the relational conditioner which detects arrival at the destination depend on the variables selected for the position of the destination.

Finally, we should ask, which version of the program is the best? The answer is not necessarily the most general or the most elaborate. Here, and in every instructional program, we should answer a number of questions:

1. Who is the program for?
2. What is the program supposed to help the student accomplish?
3. How long should the student work at this task?
4. Is it reasonable to expect a student who is stimulated by the simplest level in the program to also be able to handle the most complex level?

17. EXECUTE INDIRECT

Perhaps the most unusual instruction in the Color PILOT language is the **EXECUTE INDIRECT (X:)** instruction. This instruction executes the contents of a string variable as a Color PILOT instruction. The advantage of this is that the contents of a string variable can be changed while the student is running a program. The **EXECUTE INDIRECT** allows us to actually change parts of a program while it is running, and to base those changes on student input.

The **EXECUTE INDIRECT** instruction is useful in many situations; we'll illustrate a few. We might write a program which contained ten questions, and we might want to give the student a choice of which question to try. One way to do this is:

```
TS:Which question do you want?
A:
J(1=%B):QUES1
J(2=%B):QUES2
J(3=%B):QUES3
. . .
J(1Ø=%B):QUES1Ø
```

A more elegant way to handle the same task is:

```
D:S$(8)
TS:Which question do you want?
A:
C: S$ = "J:QUES"!!%B
X:S$
```

The **COMPUTE** instruction tacks the number given onto the end of the **JUMP** instruction in S$, and the **EXECUTE INDIRECT** actually does the jump.

Another example is provided by the **IMMEDIATE** mode of Color PILOT. **IMMEDIATE** mode is really just the following short program, as you can see if you enter **IMMEDIATE** mode and them enter **EDIT** mode without clearing the program.

```
AH:
X:%B
J:@A
```

You might want to include this program at the beginning of a program you're developing. It will give you a quick way of trying out graphics without erasing your program (entering **IMMEDIATE** mode will erase your program). If you have a long program, it may take time to run through to where you're trying something new.

To use it at the start of a program, begin:

```
AH:
X:%B
J:@A
*START
. . . the real program
```

For regular trials of the program, you would first type J:START to bypass the **IMMEDIATE** section. Of course, you would remove the first four lines when the program was finished.

With the **EXECUTE INDIRECT** instruction we can write programs which use random selection of verbal parameters in much the same way that we used random selection of numbers in our addition example. Our example here is fairly complex, so let's first make it clear what we want the program to do.

We are going to write a program which will drill the student on subject-verb agreement. We want to randomly select subjects from a list, and we want to randomly select the rest of the sentence to provide some variety. We also want to use **MATCH** instructions to allow for spelling errors.

The following table lists the possibilities. The variable A will be used in the program to index the subjects and the verb forms.

A	Subject	Correct Verb
1	I	am
2	He	is
3	She	is
4	You	are
5	They	are

We begin the program by creating the needed string variables:

```
D:S$(2Ø)
D:T$(4)
D:D$(69)
D:E$(23)
D:R$(2Ø)
D:F$(2Ø)
D:W$(25)
D:M$(6)
D:N$(6)
D:O$(7)
```

Now we make a number of lists - first, a list of subjects:

```
C:S$="I   He  She You They"
```

We make each entry on the list the same length as the longest entry ("They") by adding spaces. In the same way, we make a list of destinations for variety in completing the sentences:

```
C:D$="going to the circus.   "
C:D=D!!"coming through the rye."
C:D=D!!"getting saturated.    "
```

We deviated from our usual practice of using the $ on string variables so that we could fit each entry on a single line. Each of the three entries in D$ is 23 characters long. We make all entries in a list the same length for ease in choosing single entries (see below).

-If all entries in a list are the same length, it is easy to predict where a new one starts.

We're going to have a number of **MATCH** instructions to cover correct answers and various errors. In the following table we list the fields of the **MATCH** instructions we'll use:

A	Subject	Correct	Singular-Plural Error	Person Error
1	I	am	*re	is
2	He	is	*re	am
3	She	is	*re	am
4	You	*re	am!is	--
5	They	*re	am!is	--

Next we make lists of these fields, using exactly the same order as we used for the list of subjects. However, we avoid using spaces for fillers because a space in the field of a **MATCH** instruction requires that there be a space in the same position in the student answer. We could use any of a number of other filler characters; we'll use *:

```
G:R$="AM**IS**IS***RE**RE*"
C:W$="*RE***RE***RE**AM!ISAM!IS"
C:F$="IS**AM**AM**xxxxxxxx"
```

That looks confusing, but if you count carefully you'll see that we have just listed the entries in the table using * as a filler. The one exception is the last two entries in F$. The forms of English verbs are such that it is impossible to make a person error with a plural subject. We never want a message about a person error to appear with a plural subject, so we make the field of the **MATCH** instruction into something which will never match - in this instance, lower case (we'll be using an **A:**).

We want the program to randomly select subjects from the list in S$ and entries from the list in D$. To avoid double use of combinations, we make a list for record keeping, just as we did in Chapter 13. There are five subjects and three destinations, so there are 15 combinations:

```
D:L$(15)
C:L$="111111111111111"
```

-Although we're using 1 and Ø as symbols for record keeping, we use them as characters instead of numbers because a character occupies only half as much memory as a number.

That completes the setup. Now pick the index A and one for destination (B). We reject combinations which have been used already. I is the index for the list of combinations.

```
*PICK C: A = RND(5)
C: B = RND(3)
C: I = 3*A + B + 1
J(L$(I)="Ø"):PICK
C:L$(I)="Ø"
```

From these index numbers, we can compute the starting positions for entries in the lists, S$, D$, R$, W$ and F$:

```
C: X = 4*A + 1      (for S$, R$, and F$)
C: Y = 5*A + 1      (for W$)
C: Z = 23*B + 1     (for D$)
```

In every case the starting point is calculated by the formula:

(entry length) * (random number) + starting position

Now pick out the entries from the five lists:

```
C:T$ = S$(X,4)
C:E$ = D$(Z,23)
C:M$ = "M:"!!R$(X,4)
C:N$ = "M:"!!F$(X,4)
C:O$ = "M:"!!W$(Y,5)
```

After all that, the rest of the program is very short and simple:

```
TS:Give the missing verb.
:
:$T$       $E$
A:
M:AINT!AIN*T
TY:Very funny, but not very
:correct. Now be serious.
JY:@A
```

Someone is sure to try "ain't" as an answer. Next we check for the right answer:

```
R:Check for correct
X:M$
TY:Right!
WY:2Ø
CY:P=P+1
JY(P<1Ø):PICK
EY:
```

Use 1Ø of the 15 possibilities, and then quit.

```
R:Check for person error
X:N$
TY:The subject is singular, but
:you need the other singular
:form of the verb. Try again.
JY:@A
R:Check singular-plural error
X:O$
TY:Think about the number of
:people that are
:$E$
:Try again.
JY:@A
```

These instructions provide for the standard errors. Finally, we take care of unpredictable errors:

```
T:The verb must be one
: of these:
:
:   are  am  is
:
:Try again
J:@A
```

As usual, there are many elaborations and extensions we could add to this program. But our purpose here is to illustrate the **EXECUTE INDIRECT** instruction, so we'll resist the temptation to continue. This example is typical; the **EXECUTE INDIRECT** is used infrequently, but the whole program is based on its use.

Instead of continuing, we'll give as a parting present a minor extension of the **IMMEDIATE** mode program, which will turn the computer into a calculator:

```
TS:Type your expression.
D:Z$(3Ø)
A:
C:Z$ = "C:H="!!%B
X:Z$
T:Result = #H
J:@A
```

18. NEW CHARACTERS

In many applications, we will have to use special characters if we are going to use the standard notations of those fields. Geometry programs might need symbols like ϕ,Θ,∝ and ß. Physics programs might need vectors, advanced math programs might need integration signs, and English programs might need special phonetics symbols. Foreign languages might need a whole new alphabet. The list is endless. Color PILOT enables the programmer to add characters and redefine existing characters by means of the **NEWCHAR** (N:) instruction.

To master the **NEWCHAR** instruction, we must know something about binary numbers and the internal computer representation of characters. A binary number is simply a list of Ø's and 1's. (A decimal number is a list of digits taken from the larger group Ø-9). Binary numbers are important here because Ø and 1 can represent a dot off and a dot on, and the computer makes characters by turning some dots off and other dots on. To begin, let's get comfortable with the first 16 numbers in three number systems--decimal (base 1Ø), binary (base 2), and hexadecimal (base 16). The following table should help:

Decimal	Binary	Hexadecimal	Dot Pattern
Ø	ØØØØ	Ø	. . . .
1	ØØØ1	1	. . . x
2	ØØ1Ø	2	. . x .
3	ØØ11	3	. . x x
4	Ø1ØØ	4	. x . .
5	Ø1Ø1	5	. x . x
6	Ø11Ø	6	. x x .
7	Ø111	7	. x x x
8	1ØØØ	8	x . . .
9	1ØØ1	9	x . . x
1Ø	1Ø1Ø	A	x . x .
11	1Ø11	B	x . x x
12	11ØØ	C	x x . .
13	11Ø1	D	x x . x
14	111Ø	E	x x x .
15	1111	F	x x x x

Each character occupies an 8 x 8 block of dots. We define a character by telling the computer which dots should be off (background color) and which dots should be on (foreground color). The patterns of Ø's (off) and 1's (on) in a horizontal row of eight dots is transmitted to the computer as two hexadecimal digits. A list of sixteen hexadecimal digits gives the computer all eight rows.

If we wanted to define a character to be a small box, we would first fill out the 8 x 8 grid of dots and translate the pattern into hexadecimal digits:

Pattern	Hexadecimal Digits
x x x x x x x x	FF
x x	81
x x	81
x x	81
x x	81
x x	81
x x	81
x x x x x x x x	FF

This pattern is transmitted to the computer by the list of 16 hexadecimal digits:

FF818181818181FF

The other thing the computer needs is a key or character designated as the box. Your Color Computer, like almost every computer, uses the ASCII convention. In the ASCII convention, eight binary digits (called collectively a "byte") are used to represent each character. Only seven of the eight digits are actually used; the eighth is ignored, in effect. This means that there is room for:

$$2^7 = 128$$

different characters. The first 32 (numbers Ø - 31) are reserved for unprintable display functions (like carriage returns, line feeds, and backspaces), so that leaves us with 96 characters (numbers 32 to 127) to play with.

The **NEWCHAR** instruction uses the format:

N:ASCII number,hexadecimal list

The ASCII number can be a value between 32 and 127. Before we define the box, we'd better make sure we know what character we're replacing. Our regular text messages will look strange if we replace upper case "A" or lower case "e" with a box! The ASCII convention for characters is listed in Appendix I; from the list we find that upper case "A" is number 65 (decimal), and lower case "e" is number 1Ø1 (decimal).

Now let's actually replace a character. This will work fine in **IMMEDIATE** mode (remember to press [BREAK] [I]). Let's replace the % with a box. The % character is character 37:

N:37,FF818181818181FF

To see the result try typing:

T:%%%%

Surprised? Remember that we've changed % to a box, so that when we push the key marked %, we get a box.

What if we want to get the % sign back? Either we give another **N:** instruction with the original dot pattern, or we reload Color PILOT from the tape.

-A **NEWCHAR** instruction changes the character until the computer is turned off.

The large characters work with new characters in the expected way. Try:

G:M4

then:

T:%%%%

Reverse video also works; try:

G:M1,E

and then:

T:%%%%

This also shows another potential use of the special characters; they can be used to create elements of more complex graphic displays. We'll return to that later.

If you count the gray keys on the keyboard of your computer, you'll find that there are 44 keys. All but three - the @, space, and Ø keys - give different characters with the shift key, so there are:

44 * 2 - 3 = 85

characters which can be entered from the keyboard. What about the remaining 96 - 85 = 11 characters? They can be defined by an **N:** instruction, but how do we get them into our programs and onto the screen? We embed the character number in the message portion of a **TYPE** instruction, but we must precede the number with the number sign. Try:

T:#37

We get the box again. This is obviously an inefficient way to get something for which there is a key on the keyboard, but it is a useful way to get something for which there is no key.

Enter the following program:

```
D:T$(2Ø)
*START TS:Character check
TH: N =
A:
C:T$ = "T:#"!!%B
```

This instruction, which will look strange on the screen because of what we've previously done to the % symbol, is necessary because if we just programmed T:#%B, the computer would type out the number (the current contents of the variable %B). We must force the contents of the variable into the **TYPE** instruction.

```
X:T$
W:6Ø
J:START
```

We can use this program to find out what the current characters are, especially those 11 which do not correspond to keys on the keyboard. Run the program for the numbers between 32 and 127, and note especially those which do not appear on the keyboard. Caution: the first character is a space, so don't think the program isn't working when nothing appears on the first try.

This experiment gives the following table for Color PILOT:

Non-keyboard Characters

Decimal	Character
91	[
92	\
93	]
94	^
95	-
96	□
123	{
124	:
125	}
126	~
127	☺

These characters will be especially useful for special characters, as they do not interfere with any characters accessible from the keyboard. Therefore the student will never enter them accidently.

-Characters numbered 91 - 96 and 123 - 127 cannot be entered from the keyboard.

Parallel vertical lines can give some surprising effects. In **IMMEDIATE** mode try:

```
N:91,5555555555555555
```

followed by:

```
T:#91
```

That's probably not what you expected! Instead of parallel vertical lines, you got a colored block. Now you have some idea of how Color PILOT fools the computer into mixing color with text. The effect requires that the vertical lines be on alternate columns of dots. Try:

```
G:M4
```

to double the character size, and then:

```
T:#91
```

That's probably what you expected the first time.

To get the other color, we switch the 1's and Ø's:

```
G:MØ
N:92,AAAAAAAAAAAAAAAA
```

and:

```
T:#92
```

Horizontal lines give no surprises. Try this:

```
N:93,FFØØFFØØFFØØFFØØ
```

and then:

```
T:#93
```

Where could we make use of the color effect? Think of the patterns as colored blanks. One application could be for emphasizing blanks in fill-in-the-blank questions. Try the following program:

```
N:91,5555555555555555
G:WØ,12Ø
THS:The spoon goes on the #91
T:#91#91#91#91
:side of the plate.
```

```
:
TH:What word belongs in #91
T:#91#91#91#91?
G:W18Ø,12Ø
A:
W:1ØØ
```

The **NEWCHAR** instruction can be used to create elements which may be used in complex diagrams. The following pattern will make a decent circle on the screen:

```
. . x x x x . .
. x . . . . x .
x . . . . . . x
x . . . . . . x     N:91,3C4281818181423C
x . . . . . . x
x . . . . . . x
. x . . . . x .
. . x x x x . .
```

Try this pattern in both normal and large character size. The advantage of this is that it replaces drawing lines (in this case eight). It is also much faster, requiring much less typing.

A related set of characters can be typed in sequence to supply a form of simple animation. First we design two characters based on the above circle:

```
. . x x x x . .          . . x x x x . .
. x . . x . x .          . x . . . x x .
x . . . x . . x          x x x . . x . x
x x x x x . . x          x . . x x . . x
x . . x x x x x          x . . x x . . x
x . . x . . . x          x . x . . x x x
. x . x . . x .          . x x . . . x .
. . x x x x . .          . . x x x x . .
```

N:91,3C4A89F99F91523C N:92,3C46E59999A7623C

Our program is:

```
G:E,M4
N:91,3C4A89F99F91523C
N:92,3C46E59999A7623C
C:I=Ø
*LOOP G:W11Ø,9Ø
T:#91
W:1
G:W11Ø,9Ø
T:#92
W:1
C:I=I+1
J(I<1ØØ):LOOP
```

If that isn't smooth enough, try adding:

```
N:93,3C62939DB9C9463C
```

making the program:

```
G:E,M4
N:91,3C4A89F99F91523C
N:92,3C46E59999A7623C
N:93,3C62939DB9C9463C
C:I=Ø
*LOOP G:W11Ø,9Ø
T:#91
W:1
G:W11Ø,9Ø
T:#92
W:1
G:W11Ø,9Ø
T:#93
W:1
C:I=I+1
J(I<1ØØ):LOOP
```

(To speed it up, remove the W:1 instructions.)

Here, as with the use of color, it is wise to keep the limitations of the computer in mind. This is not a suitable device for creating Saturday morning cartoons, but some limited animation can provide visual stimulation and emphasis.

Before leaving the topic of user-defined characters, we should list some of the results of using numbers outside the range of 32 - 127. Numbers over 127 will cause an error message to be displayed. Numbers less than 32 are ignored, except for 12 (Carriage return) and 13 (Clear screen).

The following program can be a great help in designing new characters. The program allows you to experiment with the dots in a large grid, displaying the current design in both large and regular sizes. It also gives the proper sequence of hexadecimal characters for the current design. Make a copy of this program on tape for future use:

```
D:D$(4)
D:E$(4)
D:P$(64)
D:H$(16)
D:N$(16)
TS:
C:I=12
G:P2
*BOX1
```

```
G:D12,192-I,L14Ø,192-I
G:DI,18Ø,LI,52
C:I=I+16
J(I<=14Ø):BOX1
N:94,ØØ3C7E7E7E7E3CØØ
N:95,FFBDFFFFFFFFBDFF
N:96,FF818181818181FF
C:D$(1)=" "
C:D$(2)=CHR(94)
C:D$(3)=CHR(96)
C:D$(4)=CHR(95)
C:H$="Ø123456789ABCDEF"
C:N$ = H$
C:I=1
*CL1
C:P$(I)=Ø
C:I=I+1
J(I<65):CL1
G:W144,175
C:E$=CHR(127)
N:127,1Ø3854Ø1Ø1ØØØØØ
TH:$E$
T:  Move up
N:127,1Ø1Ø1Ø5438ØØØØØ
TH:$E$
T:  Move down
N:127,ØØ2Ø4ØFCØ2ØØØØ
TH:$E$
T:  Move left
N:127,ØØ1ØØ8FCØ81ØØØØ
TH:$E$
T:  Move right
T: SPACE BAR
T: CHANGES DOT
G:W16,32
T:Code for char:
C:C=3
C:X=Ø
C:Y=Ø
U:MOV
C:C=1
U:CPAT
G:WØ,Ø
AS:
U:MOV
C:A=ASC(%B)
C(A=9):X=X+(X<7)
C(A=8):X=X-(X>Ø)
```

The logical operation X<7 is either true, which gives a value of 1 to the expression, or it is false, which gives a value of Ø to the expression.

```
C(A=1Ø):Y=Y+(Y<7)
C(A=94):Y=Y-(Y>Ø)
C:Z=Y*8+X+1
C(A=32):P$(Z)=1-P$(Z)
U(A=32):CCEL
C:C=P$(Z)+3
U:MOV
C:C=C-2
G:WØ,Ø
J:@A
*MOV
G:WX*16+16,175-Y*16,MØ
C:E$=D$(C,1)
TH:$E$
E:
*CPAT C:I=1
*CP1
C:K=P$(I)*8+P$(I+1)*4+P$(I+2)*2
C:K=K+P$(I+3)
C:I=I+4
C:N$(I/4)=H$(K+1)
J(I<65):CP1
*CP2
G:WØ,16
T:  $N$
X:"N:127,"!!N$
G:W144,96,M4
T: #127 #127 #127
T: #127 #127 #127
T: #127 #127 #127
E:
*CCEL
C:I=(Z-1)/4*4+1
C:K=P$(I)*8+P$(I+1)*4+P$(I+2)*2
C:K=K+P$(I+3)
C:N$(I/4+1)=H$(K+1)
J:CP2
```

19. LEFTOVERS

There are a number of features of the Color PILOT language which we have not illustrated. We will now cover them briefly.

The **SOUND** instruction (**S:**) can be used only with one of the two modifiers - **SS:** for **SOUND START** and **SH:** for **SOUND HALT.** These two instructions start and stop the cassette recorder; whatever comes off the recorder between an **SS:** instruction and an **SH:** instruction is passed through the computer to the speaker in the TV set. Of course, the PLAY button on the recorder must be down for this to work. You can make a normal audio sound track to go with the PILOT program, and the sound can be any combination of voice, music or sound effects.

The **SOUND** instruction is of limited utility because of two characteristics of tape recorders. There is no synchronization between the computer and the recorders. If you want to play a 2Ø-second message, you might program:

```
SS:
W:2ØØ
SH:
```

The computer could be doing something besides waiting during the interval; timing would have to be set by trial and error. If the timing is off (it can vary slightly from recorder to recorder), there is nothing you can do. The timing errors can accumulate if you're trying to play a series of, for example, ten 2Ø-second messages. Starting and stopping are particularly troublesome because the times vary between recorders. The second limitation is that we have no way of reversing tape. Our PILOT programs are likely to have many jumps, both forward and backward, but the tape can only run forward. We would be discarding much of the power of the computer if we make all our programs correspond to the limitations of the cassette recorder.

For the above-mentioned reasons, the major application of the **SOUND** instruction is at the beginning of a program. Most programs begin with some instructions to the students as to how they are to proceed. Often these instructions are text heavy. Perhaps it would be better to present them aurally instead of visually. This would be one solution to the problem of explaining what is to be done to a weak reader or a young child.

We have made use of a number of the functions included in Color PILOT in earlier chapters. In particular, we have used and discussed **ABS** (absolute value), **RND** (random number), **LEN** (current length of a string), and **KEY** (check keyboard for input).

Color PILOT offers a number of additional functions, including the following two:

CHR (number) - This converts the number into the corresponding character using the ASCII convention (Appendix I). It can be used to load a character not on the keyboard into a string.

ASC (string) - This is the inverse of the **CHR** function; it gives the number which corresponds to the position of the first character of the string using the ASCII convention (Appendix I).

The following program illustrates the use of these two functions:

```
TS:I have a way to turn messages
:into a secret code. Your task
:is to discover the code.
:
:You give me a word, and I'll
:change it into the code and
:give it back. When you think
:you have it, type DONE in code.
:If the message you get back
:makes no sense, then you
:don't have it yet.
:
:    Press ENTER when ready.
A:
C: X = RND(5)
D: A$(2Ø)
*WORD TS:Give me your word.
:
A:
C:A$=""
```

This instruction gets rid of the previous contents of A$:

```
C:I=1
C:L = LEN(%B)
```

Set up for conversion of the word into the code (offset by X in the alphabet):

```
*DONE J(L<>4):CON
C:Y=ASC(%B(I)) - X
```

Convert any four-letter response backwards from offset to see if it is correctly coded "DONE":

```
C(Y<65): Y = Y+26
```

This wraps around the alphabet (A = character 65):

```
C: A$ = A$!!CHR(Y)
```

Build translated letter onto the string:

```
C: I = I+1
J(I<5):DONE
J(A$="DONE"):END
```

Successful solution, so quit:

```
C: I=1
C: A$ = ""
```

Unsuccessful, so reset for encoding:

```
*CON C:Y=ASC(%B(I)) + X
C(Y>9Ø): Y = Y-26
C: A$ = A$!!CHR(Y)
C: I=I+1
J(I<=L):CON
```

This sequence converts into code by offset of X:

```
T:
:$%B = $A$
:
:
:   Press ENTER to try another.
A:
J:WORD
*END TS:You got it!
W:3Ø
E:
```

The functions **STR** and **FLO** form a pair similiar to the **CHR** and **ASC** functions:

STR (number) - This converts a sequence of digits into a string of characters.

FLO (string) - This finds the first digit or minus sign (-) in a string and converts that and all following digits into a number. Conversion stops when a non-numeric character is encountered.

There is one operator which we have not discussed. This is the negation operator (**NOT**). This operator reverses the truth value of any expression and is primarily a convenience for advanced programmers.

The op code which we have not discussed is the **VIDEO** op code (**V:**). This op code is used in the following format:

V:string variable

The program control is transferred to the contents of the string variable, which must contain the compiled version of an assembly language subroutine. Use of the **V:** instruction is beyond the scope of this manual. The instruction is included so that control of video disk players and/or video tape players is possible from Color PILOT, but the instruction could be used for many other purposes.

20. DEBUGGING

Occasionally programmers make mistakes. You've probably already discovered that. In this chapter we'll try to give you some hints on tracking down errors in your programs, a process often called "debugging."

The computer will give you quite a bit of help in locating "bugs." You've already seen that the computer will detect and flag many syntax errors. A list of error codes is given in Appendix II, Section 9. These are the messages which the computer prints out while running a program. The computer prints the error code and the offending line and then pauses. The instruction is not run; the program can be continued by pressing any key but **BREAK** . Appendix II, Section 9 gives additional information as to what the error codes actually represent.

A more difficult type of error is one which is not flagged by the computer. In this case the program runs without stopping; there are no syntax errors. The problem is that the program doesn't do what we want it to. There are two basic techniques that we use in these situations; both involve temporary insertion of instructions into the program. For example, does the computer take or ignore a particular conditional **JUMP** instruction? A test like the following might be useful:

```
JY(X>4):PART2
T:No jump
. . .
*PART2 T:Jump
```

where the two **TYPE** instruction are the temporary additions.

A similar approach is to add **WAIT** instructions (or to increase the time on existing **WAIT**s). This is particularly useful in cleaning up complex screen displays when you may have to search through and make notes on listings before the display disappears or changes.

Usually these kinds of problems are traceable by examining the current values of variables. When a variable has a value other than what we expect, we usually can work backward to find where things went wrong. Again the technique is to insert temporary **TYPE** instructions, but now the **TYPE** instructions should cause variables to be printed. If several variables are needed, then it is a good idea to have them labeled.

```
T:X = #X, Y = #Y, A$ = $A$
```

This may not seem to be a very impressive list of things to try, but it's usually enough. In difficult cases or in long programs, you may want to solve the problem by writing a short test program. That's a very good way to test the effect of various fields in **MATCH** instructions.

21. STYLE AND PROGRAMMING AIDS

In this final chapter, we want to pass along a few suggestions which will help you write efficient and clear programs with a minimum of drudgery. Some of these suggestions have been mentioned earlier, but we'll collect them all here.

Every time the computer jumps to a label or uses a subroutine, it begins the search for the label at the start of the program space. Therefore labels at the start of the program will be found more quickly than labels towards the end of the program. For this reason, frequently-used subroutines should be placed as close as possible to the start of the program. Loops, which of course must involve a jump to a label, will also run faster if close to the start. (A "loop" is a series of instructions used repeatedly within a program.) In extreme cases you might put a loop into a subroutine which is called only once, just to move the loop to the start of the program.

DIMENSION instructions should be run only once. Otherwise they will reserve additional sections of memory, thus wasting earlier sections. The program will run, but memory used by **DIMENSION** instructions comes out of the program space. Since the program space is limited to just over 4ØØØ characters, there is no room to waste.

In general, it is a good idea to collect all the **DIMENSION** instructions for the whole program into one section of the program. Then if you need a string variable, you'll be able to check which variable names have been used easily and reliably. If the **DIMENSION** instructions are scattered throughout the program, it is easy to miss one. Among the hardest errors to find are those where the same variable name has been used unwittingly for two different purposes. Collecting the **DIMENSION** instructions together will minimize occurrence of such errors. Of course, it is often a good idea to use the same variable name for different purposes, as this saves space. (We did not do this in our examples because we were building the programs in very small steps.) Just be sure that you've finished with the variable for the first purpose before you use it for the second.

Naming variables is a troublesome point. When working on a large computer with unlimited memory, programmers use long and meaningful variable names (e.g., pressure and temperature). When using small computers, many authors choose to use variable names which are at least suggestive of the quantity in question (for example, P for pressure). This does make programming some expressions easier and this is the system we have used in the examples in the manual. The trouble is that too many quantities start with the same letter (e.g., time and temperature). If you are going to need many variables, then you may prefer to start with A and work through the alphabet. That will at least force you to keep some documentation of which variable stands for which quantity. Whatever approach you use, it is helpful to be systematic and to keep track of what has been used, and for what purpose.

This leads to the topic of documentation. It is usually obvious what the purposes of **TYPE** and **MATCH** instructions are; the contents of the field portions of these two instructions are informative. It is less obvious what the purpose of most **COMPUTE** instructions is (especially with the short variable names), and it is always difficult to figure out what **GRAPH** instructions do (except window setting). One way to avoid confusion would be to insert **REMARK** instructions to document what is going on every time there is a **GRAPH** instruction or a sequence of **COMPUTE** instructions. The trouble is that the **REMARK** instructions also use up valuable program space. Perhaps the best solution is to write a separate "program" using the text editor. This program should document the Color PILOT program and should be stored on the same cassette after the real program so that it can be printed when the real program is printed. The exact form of the documentation is a matter of personal style and preference; it is a text file and will never be run as a program. If you want some ideas on ways to do this, consult any programming text which covers Program Development Languages (PDL). A PDL is one efficient way to outline the sequences and logic of a program.

Use of a PDL, or some other way of outlining the sequence and logic of a program, is a great aid to programming. To get maximum benefit, complete the documentation before starting to program. You'll find that as time goes by, the design of the program takes much more thought and effort than the actual programming. That's what good computer languages are all about!

In addition to a programming text, we suggest that you get two other items to facilitate some of the more routine programming tasks. Graph paper with an 8 x 8 grid is available; this will make designing and programming new characters much easier. And if you are going to do much programming, a sheet of plastic which corresponds to the size of the graphic grid on the screen of your TV will be very useful. To make one, tape a piece of Mylar to the front of your TV and run the following program:

```
G:E,MØ,P1
*LOOP G:DX,Ø,LX,191
C: X = X+1Ø
J(X<26Ø):LOOP
W:2ØØØØ
```

Mark the end points of the lines on the plastic sheet. Then run the program:

```
G:E,MØ,P1
*LOOP G:DØ,Y,L255,Y
C: Y = Y+1Ø
J(Y<2ØØ):LOOP
W:2ØØØØ
```

Again mark the end points of the lines on the plastic sheet. Then draw the grid of lines with permanent ink on the plastic. If you can print numbers backwards, label the grid lines.

You can draw on the reverse side of the plastic with marker pens intended for use with overhead transparencies (available from most bookstores). Then, on the plastic, design a frame, transfer the coordinates into your program, and erase the plastic for reuse with a damp cloth.

You have now completed the introduction to Color PILOT. We hope that you had fun reading it and experimenting with Color PILOT. More to the point, we hope that you find Color PILOT to be as useful a tool in the creation of worthwhile programs as we have found it to be.

APPENDIX I

ASCII Code Table

CHARACTER NUMBER	DISPLAY
32	(Blank)
33	!
34	"
35	#
36	$
37	%
38	&
39	'
4Ø	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	Ø
49	1
5Ø	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
6Ø	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
7Ø	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M

78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	□
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	:
125	}
126	~
127	☺

APPENDIX II

SUMMARY OF CASSETTE COLOR PILOT FOR THE TRS-80 COLOR COMPUTER

1. GENERAL

Upward compatible with COMMON PILOT.

Allows high resolution graphics (256 x 191) in four colors and text on the screen simultaneously.

Allows normal size text (24 lines of 32 characters) and double size text (12 lines of 16 characters) under program control. Has full upper- and lower-case text (96 printable ASCII characters). All character patterns may be re-defined under program control.

Allows text to be anywhere on the screen or only within a user-defined text window.

Allows use of two color sets, normal or reverse video characters, and four colors.

Draws points, lines, boxes and filled areas.

Contains a complete screen-oriented editor for program entry.

Versatile expressions, string processing, and automatic type conversions.

2. PILOT MODES

PILOT is loaded from cassette by entering [C][L][O][A][D][M] . Once loaded, type [E][X][E][C] .

PILOT displays both upper and lower case. Holding down the [SHIFT] key and then pressing [Ø] (zero) turns the shift lock on or off (same as BASIC).

COMMAND MODE is signified by the prompt:

PILOT:

on the screen. At this point any of the following keys may be pressed:

[L]	Load a PILOT program from cassette (or from disk).
[S]	Save a PILOT program on cassette (or on disk).
[R]	Start running the current program.
[E]	Enter the text **EDIT** mode.

[P]	Print the current program on the serial printer.
[I]	Enter **IMMEDIATE** mode.
[SHIFT] [CLEAR]	Clears the program area.
[BREAK]	Causes the **COMMAND** mode to be entered from **RUN**, **EDIT** or **IMMEDIATE** mode.

IMMEDIATE mode enables the user to type in any PILOT statement which will b executed immediately. This is useful in trying out features of, among other things, the **GRAPHICS** statement. To return to command mode, the [BREAK] key is pressed.

RUN MODE is the mode in which a PILOT program is executed. To cancel the execution of a program, the [BREAK] key is pressed.

EDIT MODE is described below.

3. PILOT STATEMENTS

REMARK	**R:**any text	Statement is ignored by PILOT.
TYPE	**T:**text **TH:**text **TS:**text	Types text to screen; text may contain variables as "$V" or "#V", or special characters may be typed by "$number" or by "#number". The **H** (hang) modifier (**TH:**) will suppress the automatic new line after the type. The **S** (screen) modifier (**TS:**) clears the text window before the text is displayed.
CONTINUE Text		A **TYPE** statement can be followed by one or more continuation lines. A continuation line is just a colon followed by more text in the same format as on the **TYPE** statement.
MATCH	**M:**pattern **MS:**pattern	Pattern may contain "*" as a wild card character or "!" to separate multiple possible answers. The student answer is matched against the pattern and the "Y" or "N" flag is set. The "S" (spelling) modifier (**MS:**) may be used to allow for arbitrary spelling errors on the part of the student.

ACCEPT	**A:** **AS:** **AH:**	Allows the student to enter an answer into the answer buffer (%B). Unless otherwise specified, all input is automatically converted to uppercase before being stored in the buffer. The **H** (hold case) modifier specifies that input is not to be automatically converted to upper case. The **S** (single) modifier specifies that only one character is to be typed by the student. The one character can be any printable or non-printable character on the keyboard and it is not capitalized or otherwise modified.
JUMP	**J:**label **J:@A**	Causes a program jump to the specified label or the last **ACCEPT** executed.
USE	**U:**label	Causes a subroutine call to be made to the specified label. Only one level of call is allowed.
END	**E:**	If a **USE** has been executed, then a return is made to the statement after the **USE.** If there is not an unended **USE** in effect, the PILOT program is ended.
WAIT	**W:** expression	The expression specifies the number of tenths of seconds to pause.If a key is depressed before the time expires, the **WAIT** is terminated.
EXECUTE	**X:**V$	The string variable "V$" is executed as the next statement. It may be any PILOT statement except another "X".

NEWCHAR N: number,XXX

The number selects a character number from 32 to 127 to be redefined. "XXX" represents 16 hexadecimal characters which define the 64 bits that make up the new character pattern. The character pattern is made up of 8 rows of 8 bits each. Each pair of hex characters specifies the bit pattern for one row of the character cell. The rows are specified top to bottom. The bit values for the hex characters are:

```
Ø= ....   1= ...*   2= ..*.   3= ..**
4= .*..   5= .*.*   6= ..*.   7= .***
8= *...   9= *..*   A= *.*.   B= *.**
C= **..   D= **.*   E= ***.   F= ****
```

For example,
N:65,3C42427E4242420Ø defines character 65 (upper case "A") to look like this:

```
. . * * * * . .     (3C)
. * . . . . * .     (42)
. * . . . . * .     (42)
. * * * * * * .     (7E)
. * . . . . * .     (42)
. * . . . . * .     (42)
. * . . . . * .     (42)
. . . . . . . .     (ØØ)
```

COMPUTE C:target=expression

The expression is evaluated and assigned to the target. The target may be a variable or a subscripted variable of the form V$ (position) or V$ (position, length). If the expression type (numeric or string) does not match the target type, then it is automatically converted to the correct type.

DIM	**D:**V$(length)	Reserves string space for the variable specified. The maximum length is 255. Once dimensioned, the string variable may contain any length, from Ø to the maximum allocated.
SOUND	**SS:** **SH:**	**SOUND START (SS:)** turns on the cassette tape and plays the recorded sound through the TV set. This assumes that the PLAY button is down on the recorder and that it is connected for computer control of the motor. **SOUND HALT (SH:)** turns off the cassette tape. It is up to the program to use the Wait statement or other means to time the starting and halting of the tape recorder.
VIDEO	**V:**variable	This statement is for the advanced programer only. Improper use can crash the program. "V" is used to control video tape, video disk and various other specialized interfaces. The variable should be a string which contains 68Ø9 machine language codes. The "V" statement executes a jump to subroutine to the first byte of the string. The X-reg will point to the start of the string. The A,B,X and Y registers may be used without saving. The subroutine should end with an RTS op code.
GRAPHICS	**G:**G-list	Executes the graphics commands in the "G-list". The commands in the G-list may be separated by comma or semi-colon. The possible commands follow:

ERASE E — Erase the entire screen.

MODE M expression — Sets the screen mode as shown:

MODE	COLOR-SET	CHAR-SIZE	CHAR-TYPE
Ø	Ø	single	dark on light
1	Ø	single	light on dark
2	1	single	dark on light
3	1	single	light on dark
4	Ø	double	dark on light
5	Ø	double	light on dark
6	1	double	dark on light
7	1	double	light on dark

PENSTATE P expression — Sets the type of dots, lines and boxes drawn per the following chart:

PENSTATE	EFFECT
Ø	NO CHANGE is made on the screen
1	pixels are COMPLEMENTED on screen
2	a one-bit wide DARK mark is made
3	a one-bit wide LIGHT mark is made
4	a two-bit wide BLACK mark is made
5	a two-bit wide RED mark is made
6	a two-bit wide GREEN mark is made
7	a two-bit wide LIGHT mark is made

The exact colors made by these commands vary according to the color set selected and the particular TV in use.

DOT D X-EXP,Y-EXP — A dot is made at the specified location and the beam is moved to that location.

LINE L X-EXP,Y-EXP — A line is made from the beam location to the specified location and the beam is moved to that location.

BOX B	X-EXP,Y-EXP	A box is filled in with one corner at the beam location and the opposite corner at the specified location. The beam is moved to that location. Boxes must be drawn from left to right only.
OFFSET O	X-EXP,Y-EXP	A screen offset is set which will be added to all (X,Y) locations given in D,L,B commands.
WINDOW W	X-EXP,Y-EXP	The upper left corner of the text window is set to the character cell which contains the specified point, and the text cursor is moved to that point. Once a text window is defined all text output from the program will be confined to that window. The window can be moved at any time.

The spaces shown after each command above are optional and may be omitted.

All coordinates are given as Ø-255 for X and Ø-191 for Y. (Ø,Ø) is the lower left corner of the screen; (255,191) is the upper right. Any value over 255 is treated as modulo 256. Any Y value between 191 and 255 is treated as 191.

The BEAM location is initially at (Ø,Ø). Each D, L, B moves the beam to a new location.

The initial conditions in PILOT are **MODE Ø**, **PENSTATE 1**, **WINDOW** (Ø,191), and **OFFSET** (Ø,Ø).

5. STATEMENT LABELS

Any PILOT statement may be preceded by a label of the form:

*label

The label may be on a separate line, or may precede a statement on a line. In the latter case, the label must be followed by a space. The label may be any length but may contain no spaces.

6. STATEMENT MODIFIERS AND CONDITIONALS

H-modifier	**TH: . . . TYPE HANG** **AH: . . . ACCEPT HOLD** **SH: . . . SOUND HALT**
S-modifier	**TS: . . . TYPE WITH SCREEN CLEAR** **MS: . . . MATCH WITH SPELLING CORRECTION** **AS: . . . ACCEPT SINGLE** **SS: . . . SOUND START**
Y-conditioner	May follow any op code (example: TY:Correct). Statement is executed only if last MATCH was "YES".
N-conditioner	May follow any op code (example: TN:Wrong). Statement is executed only if last MATCH was "NO".
digit-conditioner	A digit from 1 to 9 may follow any op code (example:T3:Let me give you a hint.). Statement is executed only if the digit matches the **ACCEPT COUNTER**. The **ACCEPT COUNTER** is the number of times in a row the last **ACCEPT** statement has been executed.
(relational-exp)	A relational expression may follow any op code (and optional modifiers and conditioners). Statement is executed only if the expression has a non-zero (true) value. For example: T(X + 1=Y):text
C-conditioner	May follow any op code (example: JC:X123). Statement is executed only if the last relational expression was true.

The above modifiers, conditioners and relational expressions may be used in any combination. Normally, modifiers follow the op code, followed by conditioners, followed by relationals. For example:

```
TSY3(X<1):Very good.
```

In the above, the op code is **T**. The **TYPE** will clear the screen and output "Very good." only if the last MATCH was "YES," it is the third time through this accept, and the variable "X" is less than 1.

7. EXPRESSIONS

PILOT allows variables from A to Z. "%B" is also treated as a string variable of length 8Ø. %B is the student answer buffer and will contain the student response after each **ACCEPT**.

The variables A to Z may each be treated as a number or as a string. To be treated as a string, a variable must be dimensioned by a **D** statement. When using a variable as a string, a "$" may be coded after the variable name. The "$" is optional. The variable is a string if and only if it has been dimensioned; otherwise, it is a number.

In Cassette PILOT, all arithmetic is in 16-bit integer form. Numbers may be from -32768 to 32767.

If any value within an expression is of the wrong type (string or number) for the context, then it is automatically converted to fit the context. This allows, for example, the storing of numbers within a string, thus using the string as a numeric array. If the two arguments of a relational operator differ in type, then the second argument is converted to the type of the first, and then the comparison is made.

Subscripting may occur on dimensioned variables only. There are two forms of subscript, and either may be used within an expression or as the target of a **COMPUTE**. The two forms are:

```
V$(position)    and    V$(position,length)
```

The first form designates a string of length 1 at the given position. The first position of a string is position 1. When an unsubscripted string variable is the target of a **COMPUTE**, the variable takes on the length of the expression unless the expression is longer than the maximum allocated length of the string. If the expression value is too long, it is truncated. When a subscripted string is the target of a **COMPUTE**, the expression value is padded with spaces or truncated on the right to the specified substring length.

PILOT evaluates expressions in the same manner as BASIC with respect to parentheses and operator precedence. In addition, PILOT allows the use of logical and relational operators anywhere within expressions. These operators always produce Ø for false or 1 for true.

ARITHMETIC OPERATORS: + - * /

RELATIONAL OPERATORS: < > = <= >= <>

LOGICAL OPERATORS: & (and) ! (or) NOT (negation)

STRING CONCATENATION: ! !

FUNCTIONS:

ABS (X)	absolute value of X
KEY (Ø)	Ø if no key down, otherwise gives key value as a number
RND (X)	gives random number for Ø to X-1
CHR (X)	gives a one-character string with given value from Ø-255
ASC (X$)	gives number from 1-255, value of first character of X$
LEN (X$)	gives current length of string X$
STR (X)	gives string representation of number X
FLO (X$)	gives numeric representation of string X$

STRING LITERAL: "any text"

8. EDIT MODE

The PILOT **EDIT** mode is entered by pressing [E] from **COMMAND** mode. This allows editing of the currently loaded program. To start with a blank program area, press [SHIFT] [CLEAR] in **COMMAND** mode before pressing [E].

The editor is very easy to use. It works on the principle that "what you see is what you get." The first line of text (if there is one) is displayed on the bottom line. To enter lines of text, just type them on the screen. The cursor will always appear on the bottom line, but the text may be moved up or down the screen at will. The following keys cause special actions to take place:

[ENTER]	Moves the text up one line on the screen, or if it's already on the last line, then a new line is added to the text end.
UP ARROW	Moves the text up one line unless it's already on the last line.
DOWN ARROW	Moves the text down one line unless it's already on the first line.
LEFT ARROW	Moves the cursor left one character unless it's already in column 1.
RIGHT ARROW	Moves the cursor right one character unless at the line end.

[SHIFT] UP ARROW	Scrolls text up until a key is pressed or the last program line is reached.
[SHIFT] RIGHT ARROW	Adds a space before the character which the cursor underlines.
[CLEAR]	Moves to the top line of the text.
[SHIFT] DOWN ARROW	If the cursor is in column 1, a blank line is inserted in front of the current line (the current line will bump down, off the screen). If the cursor is not in column 1, the current line is split at the cursor location into two lines.
[SHIFT] LEFT ARROW	Deletes the character under the cursor and moves the remainder of the line left to close the gap. If the line has no characters then the blank line is removed.
[BREAK]	Exits the **EDIT** mode and returns to the **COMMAND** mode.

In general, to enter new lines just type each line followed by an [ENTER] key. To modify a line, move the cursor into place with the arrow keys and then modify text by typing the new text over the old, or by inserting or deleting characters as shown above.

Note: If the editor quits accepting new text, it is because the program area is full.

The editor is general enough to be used not only for writing PILOT proagrams but also for simple word processing applications. After editing a text file, it may be printed or saved on cassette for later use. One such use would be writing documentation for lessons written in PILOT. Since the editor has a maximum line length of 32 characters, a facility is provided to allow for printing of longer text lines on the printer. If a line is ended with an "@" character, then no RETURN is output at the end of the line. The result will be that the following line on the screen will be printed on the same printer line.

9. ERROR CODES IN PILOT

When loading a module from cassette, these errors may occur:

1??	Tape checksum error
2??	Memory error
3??	Improper type of module
4??	Module too long for program area

When executing a PILOT program, these errors may occur:

C-ERR	Invalid syntax on **COMPUTE** statement
D-ERR	Invalid syntax of **D** statement or insufficient memory for string

E-ERR	Invalid expression syntax
G-ERR	Invalid syntax on **GRAPHICS** statement
J-ERR	Jump or Use of a non-existent label
K-ERR	Internal stack overflow, expression too complex
N-ERR	Attempt to redefine character less than 32 or greater than 127, or not 16 hex digits on "N" OP code
O-ERR	Invalid PILOT op code or modifiers
P-ERR	Unbalanced parentheses
Q-ERR	Unmatched quotes
R-ERR	Ran out of memory space for expression processing
S-ERR	Subscripts out of bounds
U-ERR	Attempt to execute a **USE** when one is already in effect
X-ERR	Invalid value on "X" statement, or another "X" statement attempted from an "X"

When an error occurs, PILOT will display one of the above error messages and then the line in error. Processing will then pause until any key is pressed. If **BREAK** is pressed, the program is cancelled and command mode is entered. If any other key is pressed, PILOT will resume execution of the program on the statement after the one in error.

APPENDIX III

Program Listing From Chapter 16

```
J:PAST
*MAP G:MO,W1ØØ,18Ø
T:North
G:WØ,1Ø5
T:W       E
T:e       a
T:s       s
T:t       t
G:1ØØ,Ø
TH:South
C:X=15
C:Y=15
*BLOCK G:P5,DX,Y,BX+2Ø,Y+2Ø
C:X=X+35
J(X<22Ø):BLOCK
C:X=15
C:Y=Y+35
J(Y<17Ø):BLOCK
E:
*X
G:P2,DU,V,LU+8,V+8,DU,V+8,LU+8,V
E:
*ERASE
G:P3,DM,N,LM+8,LM+8,N
G:P2,DM+4,N+4,LU+4,V+4
E:
*PAST
G:M4,WØ,15Ø
T:A Map Reading
:
:
:   Exercise
W:2Ø
TS:
G:MØ,WØ,1ØØ
T:When you push ENTER, I'll
:show you a simple map to the
:center of a city.
A:
TS:
U:MAP
W:3Ø
G:E,WØ,1ØØ
TS:Your current position
:is marked with an X.
W:2Ø
```

```
TS:
U:MAP
C:U=37
C:V=39
U:X
W:3Ø
G:E,WØ,;1ØØ
T:Your destination is marked
:with a box.
W:2Ø
TS:
U:MAP
U:X
G:P6,D177,142,B185,15Ø
W:3Ø
G:E,WØ,1ØØ
T:You can move by typing one.
:To move - type N,S,E,or W.
:
:Press ENTER to begin.
A:
TS:
*RE U:MAP
U:X
G:P6A,D178,144,B186,152,W15Ø,Ø
*MOVE TH:N,S,E, or W?
AS:
M:N!n!S!s!E!e!W!w!
JN:MOVE
C:M=U
C:N=V
M:N!n
CY:V=V+35
M:S!s
CY:U=U-35
M:E!e
CY:U=U+35
M:W!w
CY:U=U-35
G(V<Ø!V>19Ø!U<Ø!U>23Ø):E,M4
GC:W6Ø,1ØØ
TC:    Lost
:
:Off the map!
WC:5Ø
TSC:
CC:U=37
CC:V=39
```

```
JC:RE
U:ERASE
U:X
G(V=144&U=177):E,M4,W3Ø,1ØØ
TC:Congratulations
:
:You made it.
WC:5Ø
J     C:PART2
J:MOVE
*PART2 G:E,MØ,WØ,1ØØ
T:Now you tell me all the
:moves with one line of letters.
:
:Example, 4 Norths and 1 East
:           by
:NNNNE
:
:             Push ENTER
A:
*RE2 TS:
U:MAP
C:U=142
C:V=74
U:X
D:B$(2Ø)
G:P6,D2,39,B1Ø,47,W15Ø,Ø
TH:  List moves:
*AA C:C=KEY(Ø)
J(C=Ø):AA
C:B$=CHR(C)
TH:  $B$
A:
C:B$=B$!!%B
C:L=LEN(B$)
C:A=1
*MOVE2 C:%B=B$(A)
C:A=A+1
M:N!n!S!s!E!e!W!w!
JN:MOVE2
C:M=U
C:N=V
M:N!n
CY:V=V+35
M:S!S
CY:V=V-35
M:E!e
CY:U=U+35
M:W!w
CY:U=U-35
```

```
G(V<Ø!V>19Ø!U<Ø!U>23Ø):E,M4
GC:W6Ø,1ØØ
TC:Lost
:
:Off the map!
WC:5Ø
TSC:
CC:U=142
CC:V=74
JC:RE2
U:ERASE
U:X
G(V=39&U=2):E,M4,W3Ø,1ØØ
TC:Congratulations
:
:You made it.
WC:3Ø
EC:
J(A<=L):MOVE2
J:AA
```

IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE LICENSED ON AN "AS IS" BASIS WITHOUT WARRANTY

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs

NOTE Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory

RADIO SHACK SOFTWARE LICENSE

A Radio Shack grants to CUSTOMER a non-exclusive, paid up license to use on CUSTOMER'S computer the Radio Shack computer software received Title to the media on which the software is recorded (cassette and/or disk) or stored (ROM) is transferred to the CUSTOMER, but not title to the software

B In consideration for this license, CUSTOMER shall not reproduce copies of Radio Shack software except to reproduce the number of copies required for use on CUSTOMER'S computer (if the software allows a backup copy to be made), and shall include Radio Shack's copyright notice on all copies of software reproduced in whole or in part

C CUSTOMER may resell Radio Shack's system and applications software (modified or not, in whole or in part), provided CUSTOMER has purchased one copy of the software for each one resold The provisions of this software License (paragraphs A, B, and C) shall also be applicable to third parties purchasing such software from CUSTOMER

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U.K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

PRINTED IN U.S.A.