Cassette PILOT User's Manual

Describes Cassette PILOT, Release 2.2

Processor Technology Corporation
7100 Johnson Industrial Drive
Pleasanton, CA 94566
Telephone (415) 829-2600

First Edition, First Printing, June, 1978
Manual Part No. 727111

IMPORTANT NOTICE

Cassette PILOT was developed by John A. Starkweather of the University of California at San Francisco. Portions of this manual and a related PILOT, Version 1.1, were prepared under contract to the Lister Hill National Center for Biomedical Communications, National Library of Medicine, Bethesda, Maryland. PILOT Release 2.2 is an extensive revision for the Processor Technology Sol Terminal Computer or a similar computer using CUTER software.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

SECTION 1

INTRODUCTION

1.Ø INTRODUCTION

PILOT is a programming language for interactive programs, i.e., for those programs in which the user participates in a kind of conversation with the computer by typing responses to questions posed by the computer. PILOT stands for Programmed Inquiry, Learning Or Teaching, and has been used most commonly as a language for computer-assisted instruction. It was first developed at the University of California in San Francisco and has been implemented on a variety of large and small computers.

This guide provides a description and operating procedures for a version of PILOT that runs on the Sol Terminal Computer manufactured by Processor Technology Corporation, or on an equivalent 8Ø8Ø microcomputer that uses CUTER software and has a video display and CUTS format cassette tape controlled by CUTER software.

1.1 WHAT CAN I DO WITH PILOT? WILL I BE ABLE TO DO IT?

PILOT enables a person without any prior computer experience to develop and test dialogue programs for use in an instructional context. The simplicity of the syntax, as well as the conversational structure of the system, make it easy for the teacher to prepare a program and easy for the student to use it, even without any training.

PILOT makes it possible, for example, to present the student with a reading passage, give him time to study it, and then ask him a series of multiple-choice questions to check on his comprehension. The program might include computer responses keyed to the answer a student has given, or it might ask him for his reaction to the passage, scan his response, and comment or provide suggestions based on that response.

A teacher can prepare a vocabulary quiz that keeps track of a student's score and then moves on or repeats a lesson, depending on the student's performance or choice.

A PILOT program can introduce a mathematical word-problem and offer the solution, step by step, or it can give the student the opportunity to discover as many of the steps as possible, with the computer hinting along or even revealing a step, when necessary.

But PILOT has more than traditional academic applications.

PILOT is also a vehicle for learning about computers. You can write a program that teaches how to write a PILOT program and let the student save the program that he writes, so that it can be executed or revised on another occasion. (Or he can write, edit, and run the program in the same sitting.)

A user can have considerable control over the order of events in a program, and is therefore not likely to find the experience intimidating. (If he's clever, he can even cheat on the math test by "peeking" at the answers.) He can select what he wants to read, or decide whether to repeat an exercise or execute another program. He can even determine where his response will appear on the screen and put it in a funny place or make it appear with reverse video (i.e., white letters on black, for a screen set to black letters on white, or vice-versa) to emphasize a particular word in his answer. He can keep his answers, list them, and review the exercises with which he had difficulty.

PILOT is also suitable for writing conversational games.

Section 3 of this manual introduces the elements of PILOT by proposing a simple program design and showing how the PILOT language can be used to realize and expand it.

Just as in a spoken language, you can express many ideas and accomplish many tasks by combining a handful of simple statement types.

1.2 THE PILOT SYSTEM, AND HOW TO USE IT

To take advantage of PILOT and have your programs understood by the computer, you will need to use a program called PILOT, which is recorded on your cassette tape (see Section 2). The purpose of this important program is to interpret the statements of a program written in PILOT language, so that they may be executed by the computer. A program that fulfills this function is called an INTERPRETER. The PILOT interpreter is written in a more complex language than you will be using to write your own programs. The interpreter makes it possible for the PILOT language to be as straightforward as it is. (The dirty-work has to happen somewhere.)

Obtaining access to the interpreter by "playing" it into the computer from cassette is the first step involved in using PILOT. Here is a typical sequence of steps for using PILOT:

1) Make contact, "playing" the interpreter into the computer;

2) Prepare a program, using PILOT language statements;

3) Enter the program, using the EDITOR, a special part of PILOT;

4) Save the program on cassette tape;

5) Use a program that has been recorded on cassette tape; and

6) Make changes in the program, if necessary.

It should already be clear that, since PILOT is so highly interactive, the person sitting at the keyboard and playing the program from a cassette tape will be able to make certain choices as to the way in which the program will behave. To some degree, you can provide for this decision-making in the program, itself; to some degree, PILOT provides for user choices. In general, one of PILOT's primary features is that a given program may not provide exactly the same experience every time that it is used; rather its operation will be conditioned by the needs and reactions of different users on different occasions.

The order of events outlined above is not hard and fast. For example, you may use the PILOT EDITOR to edit a form-letter, or you may not want to enter a program of your own, but simply to use one prepared and saved in advance. Another very common practice would be bouncing back and forth between the interpreter and the EDITOR, saving the program only when you were satisfied with its form.

WHY NOT BASIC?

It is possible to write interactive programs in a general purpose programming language such as BASIC, APL, FORTRAN, or PL/1 instead of in PILOT. The reverse is certainly not true, for PILOT is a specialized language oriented toward dialogues, drills, tests, etc., rather than toward some kinds of computation handled well by general purpose languages. The advantage of PILOT over general purpose languages such as BASIC is that interactive programs are easier to write in PILOT. If a BASIC program is to handle free-response dialogue, the programmer must often make unwieldy arrangements for processing input and comparing words or portions of words that the program must recognize. In PILOT this kind of processing is easy, and the programs are more readable by human beings than they might be if they were written in some other computer language.

1.3 HOW THIS MANUAL IS ORGANIZED

This manual attempts to give complete written information about PILOT programming and the operation of 8080 PILOT.

Section 1 - introduction, definitions and conventions
Section 2 - procedures for "making contact" with PILOT;
Section 3 - a tutorial introduction to the fundamental PILOT statement types;
Section 4 - a detailed description of the syntax of the PILOT language;
Section 5 - a description of the PILOT EDITOR;
Section 6 - comments on the two test programs distributed with PILOT; and
Section 7 - a survey of errors and error messages.

We encourage you to explore the actual operation of PILOT instructions by using the test programs described in Section 6 and recorded on the same tape as PILOT, itself. Many PILOT functions are demonstrated by these programs. After reading this manual, you might even want to try guessing what combinations of commands were used in the test programs. Then list the programs--you will know how--and see whether you were correct! (Of course, as with any language, even given a limited number of statement types, there will be several ways of expressing the same idea "correctly." You will probably want to approach problems in the ways that seem to you to be the clearest and most convenient.)

1.4 TERMS AND CONVENTIONS

The following terms will appear frequently in our discussion. The definitions given here are not technical, nor even very complete, but should provide some insight into the kinds of concepts that are being described when these words and expressions are used.

A BUFFER is a temporary storage space set aside inside the computer for a collection of possible items. A PROGRAM BUFFER, for example, is space set aside for the information necessary to execute a certain program. (Think of setting up a room in a theatre as a temporary dressing-room for a group of visiting actors. Ideally, you provide a place where an actor, or several actors, can go to change clothes, if necessary, without having to worry about being disturbed. Actually, for the plan to work, you must also take precautions against anyone's accidentally walking into the room. There are similar precautions to be taken when using computers.)

A CHARACTER may be a letter, a punctuation mark, a number, or a symbol. Note that CHARACTERS, even those which represent numbers, are not involved in computations, i.e., the symbol "3" plus the symbol "2" does not yield the number "5," the symbol "5," or, in fact, anything meaningful for our purposes. A CHARACTER is not a quantity, but a representation. A CONTROL CHARACTER is formed by hitting the control key and another key simultaneously. Some CONTROL CHARACTERS function as commands to the computer and are not displayed on the screen. Others are displayed as special graphic symbols. (There is a chart of these graphic symbols in Section 5.)

CODE has two meanings. In its more standard usage, the CODE for something is a sign that is used to stand for the thing. A more specialized meaning of the word is: that sequence of instructions, in a computer language, that makes up a computer program, or part of a computer program. When you write a PILOT program, you will be writing CODE which is translatable into activity by the computer.

A COMMAND is a direct instruction to the computer. A STATEMENT, by contrast, is indirect. It is entered as a line in a program,

and initiates no immediate activity. Almost every "sentence" in the PILOT language can be used both as a program STATEMENT and as a direct COMMAND. The convention we will adopt for this manual is to use the word STATEMENT, except in the context of immediate execution. (As we shall see, the COMMANDS which belong to the PILOT EDITOR are not part of the PILOT language, and always result in immediate execution.)

A CONSTANT is an item of data that has a fixed, or constant, value. 32 is a CONSTANT.

A CURSOR is the rectangle that is visible on the video display screen. It serves to show the screen position at which the next CHARACTER will be displayed.

EDITING a FILE is modifying that FILE in any way, including typing it into the computer for the first time. The EDITOR is that program, or that part of a program, which enables you to use certain direct COMMANDS to EDIT a FILE.

EXECUTION is the actual "running" of a program. When execution occurs, the computer follows programmed instructions.

A FILE is a collection of related information, usually named and referenced as a unit. Think of a FILE as a way of saving such a body of information for subsequent use. A FILE can contain a program, text, or data of various kinds. A DATA COLLECTION FILE is a FILE that you set up to store information that the computer receives, in this case, from the keyboard, and that you designate as something that you want to keep. A FILE is very different from a BUFFER, in that 1) a BUFFER is a temporary storage area that exists only during the operation of a program, whereas a FILE (at least for our present purposes) is a permanent record, 2) a FILE is usually recorded on some kind of device (e.g.,tape) outside the "memory" of the computer, whereas a BUFFER does not exist, theoretically, when the computer is inactive, and 3) a FILE consists of information, whereas a BUFFER is a place where information can temporarily reside. (A FILE is analogous to the group of visiting actors, all staying at the same motel, and therefore reachable at the same telephone number.) Recording a FILE on an external device is called WRITING the FILE; retrieving a FILE from storage is called READING it.

An INTEGER is a number that can be divided by one without leaving a fractional remainder: 5,-5, and Ø are all INTEGERS.

LOADING a program is reading into the computer the instruction set that makes up the program. You cannot execute a PILOT program that you have saved on cassette until you have LOADED it.

MEMORY is the storage area of the computer. One of the important facts to remember about computer MEMORY is that it is limited. It is possible for a program, or a DATA COLLECTION FILE, to be too large for the computer to accommodate. We will discuss ways

of making sure that the programs we are using are not too big to fit into MEMORY. (A BUFFER is one way of allocating a portion o MEMORY.)

A STRING, or CHARACTER STRING, is a set of consecutive CHARACTERS. "DOG," "23," and "The cat is on the mat" are all STRINGS.

A VARIABLE, by contrast to a CONSTANT, is a name to which different values can be assigned. The equation x = 3 causes a value of 3 to be assigned to a variable named x. Computer languages have rules for designating VARIABLES. We will discuss rules for PILOT variables in Section 3.4.

The following conventions will be used in this manual:

1. PILOT statements and commands will be represented in capital letters, even though you do not really have to enter them in capital letters. EDIT, LOAD, and MC are all examples of PILOT commands.

2. Where an expression is enclosed by angle brackets <like this>, the expression tells what kind of item should be typed as part of a statement. For example, <number> indicates that a number should be entered (without the brackets). <cr> indicates that a carriage return should be struck.

3. Square brackets [like these] enclose an optional element in a statement. For example, [<filename>] indicates that the use of a filename is optional. The brackets, again, are not literal.

4. In examples that contain PILOT code, two rows of dots are used to indicate code that was omitted because it was irrelevant to the example.

```
<statement>
  ....
  ....
<statement>
```

5. "cond" is an abbreviation for "conditional expression." A conditional expression is used in cases where a statement should be executed only if a specific condition is met. Conditional expressions are discussed at length in Section 3.2.

6. Regard the video display screen as divisible into 16 horizontal ROWS and 64 vertical COLUMNS. When you type in a PILOT command, the characters that you type will fall in the same ROW, but in consecutive COLUMNS. It is not necessary to begin typing a PILOT statement in any particular COLUMN within a ROW. In the examples in this manual, indentation may be used for the sake of readability; it has no effect, whatever, on the operation of the program.

7. Quotation marks (" ") are used to enclose a message that appears on the screen, e.g., "PREPARE TAPE 1."

8. Control characters, those formed by depressing the control key in conjunction with another key on the keyboard, are denoted by ctrl-<character>; for example, ctrl-A.

SECTION 2

MAKING CONTACT

The first step for using PILOT is to load the PILOT interpreter into your computer. Unless you are very well-versed in the use of cassette recorders, you might want to look at the appendix called "Using Cassettes." (If you plan to be reading and writing files, you can decide to use two tape recorders, one for input and one for output. If you use separate recorders for the two functions, use unit 1 for reading programs and unit 2 for collecting data. See Section 7 of your Sol manual or the appropriate section of your own system manual to be sure that you correctly connect the recorders to the interface device.) For our introduction to PILOT we will need only one cassette recorder and the PILOT program tape.

Sit at your terminal or imagine yourself seated there. (The former alternative is recommended.) Type the SOLOS/CUTER CA command, rewind the tape and position it past the leader. Now hit the MODE SELECT key and wait for the SOLOS/CUTER prompt (>). Put the recorder in PLAY mode, type the SOLOS/CUTER XEQ command or the GET command (don't forget the carriage return), and wait for about a minute while the tape is being read. If you used the GET command, you will have to enter EX 100 to start the program.

EXAMPLES:

XEQ <cr>	This command plays the program from the cassette and starts the program.
or	
GET <cr>	This command plays the program from the cassette.
PILOT O 0100 1B57	When the tape has been read, the computer displays this message on the same line as the GET command.
EX 100 <cr>	This command starts the program.

If PILOT has been accessed properly, you should see the following message on your video screen.:

```
8080 PILOT 2.2   (MOD 000)
Copyright (C) 1978, Processor Technology Corp.
```

Immediate commands are:

LOAD	GET	SAVE	COPY	READ
RUN	EDIT	INFO	CLEAR	SCRATCH
SET	LIST	REWIND	BYE	CUSTOM

Now that you have made contact with PILOT, you can enter any of the PILOT immediate commands. We need to consider, at this point, only two of the possible commands: LOAD and BYE. If you want to try either, or both, of the test programs, PLTST and WAPP, you can do so by typing LOAD or pressing the LOAD key, and following whichever of these you entered with a space, the program name, and a carriage return. (PLTST and WAPP are described in Section 6.)

EXAMPLE:

LOAD WAPP <CR>

If you omit the program name, you will load and execute whatever program is recorded next on the cassette tape; if you enter only a carriage return, nothing will happen.

You will notice that you started PILOT at "address" 100. Between 100 and 103, where the message regarding immediate commands arises, the PILOT system is INITIALIZED: that is, storage space is allocated, various parameters are set, and the computer is made ready for subsequent operations. Sometimes you will want to discontinue or interrupt a PILOT program and return to PILOT. For example, you may be thinking of adding some questions to a spelling test program that you have LOADed into the program buffer from a cassette tape. To to discover exactly how large the program already is and whether there will be enough room to make the desired addition, you want to leave the program and ask PILOT for that information. If you were to restart PILOT at 100, it would reinitialize the whole system, erasing your spelling test program, and any variables that you had stored in the memory of the computer. The way to avoid this disgruntling occurrence is to restart PILOT at 103, bypassing the initialization of the system. This point, at which the "Immediate commands..." message again appears (this time, though, without title or copyright information) will be called PILOT RESTART, and can be reached, as we will see, in a variety of ways.

When you have finished with the test programs, or if you want to wait until later to try them, leave PILOT by typing BYE (an immediate command) and rewind your tape.

SECTION 3

PREPARING A SIMPLE PILOT PROGRAM

3.1 INTRODUCTION

A student is sitting at the computer keyboard. He is about to begin executing a PILOT program that he has been told consists of a reading comprehension exercise. You have showed him how to load the program, or he has read Section 2 of this manual and found the instructions for himself. He types LOAD <program name> <cr>, and soon sees on the screen the question:

"Do you like to read? (Type in an answer.)"

He types , "Not much," and the computer replies:

"You don't? Well, this exercise isn't too long."

Or he types , "Yes, most of the time," and the computer replies:

"Oh, good. I hope you enjoy this exercise!"

The reading exercise program has done four things so far: 1) typed text, 2) accepted an answer, 3) matched that answer with some kind of vocabulary data, and 4) replied accordingly (by again typing text). Let us use a shorthand for the three basic functions involved:

T: for "type text"
A: for "accept an answer"
M: for "match"

Let us also suppose that, by appending a Y or an N to one of the codes, we can "condition" the performance of the function, i.e., we can make performance conditional upon a successful match of the student's last response with the vocabulary data. Here is the hypothetical program:

CODE	FUNCTION
T:Do you like to read?...	Type this text on the video display.
A:	Accept an answer from the keyboard.
M:no	Check for a match with this text.
TY:You don't? That's...	Type this text if a match was found.
TN:Oh,good. I hope...	Type this if a match was not found.

The sequence of statements in the column labeled "CODE" is, as you have probably guessed, valid PILOT language.

A PILOT program consists of a series of statements that give a step-by-step description of what the computer must do to interact with a person sitting at a keyboard. An instruction code, consisting of one or more letters followed by a colon, defines the statement type and determines its function.

Single letter codes define "core" instructions that occur in all versions of PILOT. Of these, the three most important are the ones we have already used (T, A and M).

The letters Y and N are "conditioners" that can be appended to another code, causing it to be effective or not depending upon the success of the last attempted match.

PILOT is designed for the development of conversational programs that allow relatively free response by the user. As indicated in the example above, recognition of user responses is accomplished by searching the responses for specific words or word stems. In ordinary conversation, one or more words in a sentence often carry most of the sentence's meaning. The M-statement is used to define those words, portions of words, or word groups that we want recognized in an answer. In the example we have been using, the M-statement will find the desired match in the answers "no," "not much," "nope," "not really," "I don't know," and "Not unless it's about sports." It will not respond properly to "only comic books," "very seldom," or "Yes, especially novels."

Conversation in a PILOT program is facilitated both by the structure of the language, and by the programmer's ingenuity in designing questions and matching replies. For example, it is more effective to search for a negative answer than to search for a positive one, because the letter combination "no" will occur in almost all negative answers. It is permissible to use more than one word or word stem in an M: statement.

Detailed explanations of all of the statements in the PILOT language are found in Sections 4.1-4.7.

3.2 CONDITIONAL EXECUTION, CONTINUATION LINES, LABELS

If we were limited to asking a question, scanning for a particular word in the answer, and replying accordingly, PILOT would not be a very useful language. Fortunately there are other statements and options that we can use to accomplish more complicated tasks. For our reading comprehension program, we want to be able to present a reading passage more than one line long, ask related questions, and, perhaps, move to one of several possible points in the program, depending upon the fulfillment of a certain condition. To do these things, we need provisions (some of which we have already discovered) for conditional execution, ways to continue text beyond the first line entered, and a way of labeling parts of a program so that control can be directed to any one of them by name. (To direct control control is to determine which statement of the program will be executed next.) PILOT has all of these capabilities.

CONDITIONAL EXECUTION

As we have already seen, Y or N can be appended to any statement, as a "conditioner" of that statement. Operation will then depend upon whether the last M statement was successful. Y: by itself is a shorthand form of TY: Likewise, N: by itself is a shorthand form of TN:

The execution of a PILOT statement can also be made dependent upon the value of a numeric variable or expression; the variable or expression should appear in parentheses between the command code and the colon. (Rules for variables are given in Section 3.3; rules for expressions are in Section 4.2.) If the value of the variable or expression is greater than Ø, then the statement will be executed. Otherwise it will be ignored.

TY(X): will be executed if X=1Ø, but not if X=Ø

If both a numeric condition and a Y-N condition are used in the same statement, the statement will be executed only if both of the prescribed conditions are satisfied. For example, the statement TY(X): will type subsequent text only if the most recent match was successful AND the value of X is positive. (For instance, you can determine that a student may proceed to the next exercise only if 1) he has no questions on the previous exercise, and 2) his score on that exercise was acceptable.) T(X)Y: and TY(X): are interchangeable.

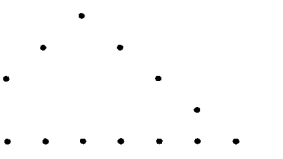
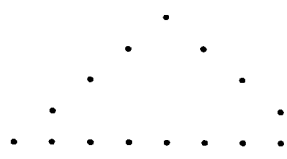
CONTINUATION

If you want to type a number of lines of text, only the first line must be preceded by a T: All subsequent lines may be preceded simply by a colon, or, in fact, by no introductory character at all.

```
T:THIS TEXT WILL BE DISPLAYED
 :AND THIS WILL ALSO BE DISPLAYED
  AND SO WILL THIS.
```

In fact, because PILOT will simply display any unexecutable text, you can even "draw" a picture by arranging characters on the screen:

EXAMPLE:

(PILOT program)	(display)
T: What shape is this?	What shape is this?
	
A:	(user enters "triangular")
M:triang	

PILOT LABELS

Any PILOT statement may include a label, either as the first element of the statement or on the line above it. A label serves as a kind of landmark in the code, so that you can say in your program, "Go to this point right away, and do whatever you are told when you get there." (As we shall see, you can also say, "and be sure to come back, when you're finished!") PILOT labels begin with an asterisk (*) and end with a blank. They may have a maximum of ten characters. If a label is the only item in a line, it refers to the statement or group of statements following it.

No complaint is made if there are duplicate labels, but only the first in sequence will be found when the label is referenced.

EXAMPLE:

(PILOT program)	(display)
*LABEL	
T:DISPLAY THIS	DISPLAY THIS
.....	.....
.....	.....

3.3 JUMPS AND SUBROUTINES

Now that we have some additional tools, we can expand and develop our reading exercise program. For the sake of convenience, let us say that the reading passage that we want to use is a popular fairy tale.

```
T: Once upon a time....
   ....
   ....
 : And they lived happily ever after.
```

The student will be considering the text in screenfuls, that is, in segments a maximum of sixteen lines long. By inserting an A statement after every screenful of information, we give the student an opportunity to indicate, by his response, that he has finished reading that portion of the text and is prepared to continue with the exercise. If we did not arrange for a pause, the text would appear and then disappear off the top of the screen much too fast for anyone to read it.

Thus, we might display the first part of "Beauty and the Beast," in which a father takes leave of his three daughters before setting out on a business trip. He asks each daughter, in turn, what she would like him to bring her ....

We have already used one easy method of programming question and answer sequences. Using the T, A, and M statements, we can ask a question and analyze the answer in either of the following ways:

```
T: What did the youngest daughter want her father
 : to bring her when he returned from his journey?
A:
M: rose
TN: No, she wanted him to bring her a rose.
```

or

```
T: What did the youngest daughter want her father
 : to bring her when he returned from his journey?
 : Enter the number of the correct response.
 :
 : 1) a necklace
 : 2) a rose
 : 3) a frog
A:
M: 2
TN: No, she wanted him to bring her a rose.
```

After the student has answered the first set of questions, we can give him another choice. We can use a T statement to ask him, "Do you want to read the last story again?" Presumably, if his answer is negative, we will want to go on to the next story; if his answer is affirmative, we will want to jump backward in the program and repeat execution of those statements that cause the story to be displayed.

To effect the jump, we need two elements of PILOT language: a label to which to direct the action of the program, and a statement that directs the action to that label. The PILOT statement that initiates a jump to a specified label is the J: statement. Let us go back and insert a label at the beginning of the original series of T statements:

```
*STORY1
T: Once upon a time...
 ....
 ....
```

and then let us insert the J statement, after the first group of questions:

```
 ....
 ....
T: Do you want to read the last story again?
A:
M: yes
JY: *STORY1
```

Notice the use of the conditioner as part of the J statement.

With the program arranged this way, the student who asks to have the story repeated must answer all of the questions again. The J statement does not provide for any return to the original statement. What if you wanted the student to be able to review the story without answering the questions again or to look back at the story between questions to correct a wrong answer before proceeding? We need a way to isolate a group of statements to which we can jump from wherever we are, without losing our place in the execution of the program. (We also need a way to jump ahead in a program, without necessarily having to skip all of the intervening code.) A group of statements that we reference as a unit and after whose execution we automatically return to our prior position in the program, is called a SUBROUTINE. A PILOT statement that calls for the USE of a certain labeled group of statements as a subroutine is the U statement.

To use a U statement to enable our student to review what he has just read, we can to mark the beginning and end of the set of statements that we want regarded as a subroutine. The label *STORY1, which we inserted to mark the destination of a J statement, will mark the beginning of the subroutine STORY1. The statement to mark the end of a subroutine, or of an entire program, is called the End, or E, statement.

In our example, the E statement should be placed at the conclusion of the text:

```
*STORY1
 ....
 ....
T: And they lived happily ever after.
E:
 ....
 ....
```

The U statement can replace the current J statement, or it can be put after any one of the individual questions, or it can be inserted repeatedly, wherever you want the story to be displayed again. At any of these points, you can have the student "detour" to a vocabulary or grammar exercise within the same program by marking a subroutine accordingly and calling it. The subroutine can be called either unconditionally or depending upon fulfillment of a specified condition. The syntax of a U statement is identical to that of a J statement.

```
*STORY1
 ....
 ....
T: And they lived happily ever after.
E:
 ....
 ....
T: Do you want to read the last story again?
A:
M: yes
UY: *STORY1
 ....
 ....
```

3.4 PILOT VARIABLES

A variable was defined earlier as "a name to which different values can be assigned." In PILOT, we can concern ourselves with two types of variables: numeric variables, and string variables.

STRING VARIABLES

PILOT allows the input to an Accept (A:) statement to be saved as a character string and retrieved later in the program. The string variable name, with which the string will henceforth be associated, is written after the colon in the A statement. A string variable name begins with "$" and ends with a blank or carriage return.

Later on, if the string variable name appears in any portion of a TYPE (T:) or REMARK WRITE (RW:) statement, it will be replaced with the value most recently assigned to it. If there has not been a value assigned to it, the variable name, itself, will be used by default.

EXAMPLE:

```
    (PILOT program)                    (display)

T: What is your favorite story?  What is your favorite story?
A:$TITLE                         (User enters "Pinocchio")
T: Who is the hero in $TITLE?    Who is the hero in Pinocchio?
   ....
   ....
T: This is $unknown.             This is $unknown.
```

A string variable name may have a maximum of ten characters in addition to the dollar sign. The maximum length of a character string to be stored as a string variable is dictated by the parameter INMAX, which can be set within a PILOT program and will be described in Section 4.4.

NUMERIC VARIABLES

Numeric variables may be assigned integer, or whole number, values in the range of -32768 to 32767. Numeric variable names begin with the pound sign character (#) and consist of one letter (A-Z). Like string variable names, they are inserted after the colon in an Accept (A:) statement, so that whatever legal value is entered becomes associated with the named numeric variable. In the execution of an ensuing Type (T:) or Remark Write (RW:) statement, the current value of a variable will be substituted for the variable name. If no value has been assigned to the variable, the current value is assumed to be zero.

Any time that you return to PILOT restart to execute an immediate command, you have the option of using the CLEAR command to reset all of your numeric variables to zero. Otherwise, the last values assigned to them are retained. String variables are unaffected by a return to restart UNLESS you load a program or program segment longer than the previous one, in which case you risk losing the variables most recently set. The INFO command, discussed later, will allow you to avoid such a problem. There is also a NEW$ command, which permits you to clear away away string variables so that you can reclaim the space which they have occupied in the program buffer.

EXAMPLE:

```
    (PILOT  program)                ( display )

        A:#X                        "23" entered
        T:THE VALUE IS #X           THE VALUE IS 23
```

You can change the value of a numeric variable by accepting a new value in another A statement (thereby causing the new entry to supersede the former value), or by assigning a new value as part of a Compute (C:) statement.

3.5 COMPUTE AND REMARK

In our fairy tale example, we might want to dictate that a jump be made, a subroutine used, or a message displayed, depending on how well the student performed on the last set of exercises. In anticipation of this need and other arithmetic applications, PILOT provides a Compute (C) statement.

EXAMPLE:

```
      (PILOT program)                    (display)

T: How old are you?                  How old are you?
A:#N                                 (User enters "6")
C: N=N+1
T: Then you'll be #N next year.      Then you'll be 7 next year.
```

To keep track of a student's score on an exercise, we can insert, after every M statement following a question a command that adds 1 to the value of a numeric variable ON THE CONDITION that the previous answer was correct:

```
....
....
T: What did the youngest daughter...
A:
M: rose
CY: N=N+1
....
....
```

The conditions governing the use of compute statements in PILOT are discussed more explicitly in Section 4.2.

The final "core" statement in this initial overview of PILOT is the Remark (R:) statment, which might, at first glance, seem almost useless, since its function is to tell PILOT, "Ignore this remark." Actually, the R statement can be very useful: as you develop bigger and more complicated PILOT programs, you might want to write yourself or another programmer notes about what is happening in the code at a particular point. Say, for example, you have a very long program, so long that you have divided it into several PROGRAM SEGMENTS to be executed in succession. At the beginning of a certain segment, you might want to insert some documentation for yourself.

EXAMPLE:

```
R: This is the second segment of an exercise program
R: intended for students who performed below average
R: on the diagnostic test.
 ....
 ....
```

3.6 IMMEDIATE OPERATION

Whenever PILOT is awaiting input from the keyboard, whether in response to an A statement, or at the PILOT restart point, the user is able to submit a command of his own for immediate execution. The syntax of such an "immediate command" is important: any statement within PILOT can be entered as a command, as long as it is preceded by a backslash (\) and, where there is not already a colon in the syntax of the statement, terminated by a colon (:). For example,"\U:*LABEL" will cause immediate execution of the referenced subroutine; \LOAD: will cause a new program to be loaded from the cassette tape.

The interactive possibilities of PILOT are greatly enhanced by the use of these immediate commands. For example, there is the option of letting the student specify a detour to the subroutine of his choice, or of allowing him to calculate his own score on an exercise, in order to determine whether to continue.

After the execution of an immediate command, PILOT returns to the A statement in reply to which the immediate command was entered and awaits an answer. (The J and U commands entered for immediate execution work slightly differently, in that the answer to the last question must be supplied for the immediate command to take effect. In the other cases, the user will be expected to answer the last question only AFTER the execution of the immediate command.) Exceptions to the rule of return to the A statement are the \LOAD:, \READ:, and \GET: commands, whose execution obliterates the program in memory, and \<cr>, which stops the current program and returns PILOT to its restart point.

When PILOT begins operation, it displays commands that are often used in an immediate mode. These are LOAD, GET, SAVE, COPY, READ, RUN, EDIT, INFO, CLEAR, SCRATCH, SET, LIST, REWIND, BYE, and CUSTOM. These commands move and alter files, make some determinations as to how your equipment will function, and provide information that is usually more useful to the programmer than to the user. At this point of control, defined in Section 2 as "PILOT restart," these commands may be entered for immediate operation without an initial backslash (\) or a terminal colon (:).

At PILOT restart, typing RUN will cause program execution to begin. Thus, if you have just entered a program in the PILOT EDITor (see section 5), you will initiate the execution of that program. Otherwise, you will execute the last program stored (using LOAD or GET or READ) in the program buffer (unless you left PILOT in the meantime). Pressing the LOAD key is equivalent to typing the word LOAD, and causes another program or program segment to be read from the cassette tape into the program buffer. LOADing also initiates execution of any executable file: that is, of a PILOT program.

IMPORTANT NOTE ON IMMEDIATE COMMANDS

If you enter an immediate command from within a PILOT program and you neglect to include the colon following the letter code for the command, the line which you have entered will be displayed on your video screen, without any other effect on the operation of the program.

SECTION 4

PILOT STATEMENT DESCRIPTIONS

4.1 STATEMENT SUMMARY

The following section describes each of the PILOT instructions provided in this version of PILOT for the 8080 microprocessor. The descriptions include syntax, function, and examples. Where appropriate, error messages that might be generated by invalid statements or invalid user response are described. In the examples below, the column marked "display" shows text and messages IN THE ORDER THAT THEY APPEAR ON THE SCREEN. This order does not necessarily correspond to the order of statements in the PILOT program

Section 3 of this manual provided a brief overview of the PILOT core instructions, which will be treated in greater detail on the pages indicated:

PILOT CORE INSTRUCTIONS

Note that the core instructions are limited to single letter codes. These instructions are standard in syntax and operation for many different implementations of PILOT. It is therefore advantageous to use core instructions as much as possible, so that users of other versions of PILOT may benefit from your work. Multi-letter codes represent "keyword" instructions that have been added to PILOT to meet special needs. When you use them, you should keep in mind that they do not necessarily exist in all other versions of PILOT.

8080 PILOT KEYWORD EXTENSIONS

* usually used in immediate mode
** ONLY used in immediate mode

For purposes of our discussion, we divide PILOT instructions into six categories, corresponding, roughly, to the type of instruction or to the service performed. We hope this method of organization will make it easier for you to use this manual as a reference. The groupings are:

1) Core instructions--those which perform all of the most basic functions in PILOT (already discussed).

2) Cursor and video control instructions--those which enable you to determine where text will appear on the video screen,

3) Aids to conversation--those instructions which facilitate dialogue in PILOT by extending the available core instructions,

4) Instruction that set various kinds of parameters

5) File manipulation instructions--those related to storing and retrieving programs and data, and

6) Commands usually used in immediate mode from PILOT restart. (They may actually be executed from within a PILOT program, but might cause complications if not used carefully.)

The EDIT command, which makes it possible for you to enter and alter text or program files, has its own extensive treatment in Section 5.

Before continuing, you may want to review the terms and conventions introduced in Section 1.4.

4.2 CORE INSTRUCTIONS

These are the statements that you will be using most frequently in your PILOT programming. They let you type text on the video screen, accept and analyze responses, and alter the order in which statements are executed. There are a number of "keyword" instructions, which are discussed in connection with with the core instructions to which they are closely related. These are TH:, RW:, and the variations on J: and M:. Remember that such instructions do not share the standardization of of normal core instructions.

```
                ****
STATEMENT       * TYPE (T:), YES (Y:), and NO (N:)
                * TYPE AND HANG (TH:)
                ****
SYNTAX:
-------------------------------------------------------------
          [<label>]  T [<cond>] : <message>
          [<label>]  Y [<cond>] : <message>
          [<label>]  N [<cond>] : <message>
          [<label>] TH [<cond>] : <message>
-------------------------------------------------------------
```

DESCRIPTION: Display a message to the PILOT user. A message consists of a character string that may include one or more variable names. All character positions to the right of the colon are reproduced literally except that the values of variables are inserted as replacements for their names. String variable names are a maximum of ten characters long and are preceded by "$." Numeric variable names are one letter long and are preceded by "#."

```
$abacus is a string variable.
#a is a numeric variable.
```

Typing a ctrl-DELete character in the text of a Type statement causes the remainder of the line to be displayed on the screen in reverse video. This capability is described under the heading, "Cursor and Video Control."

YES (Y:) and NO (N:) statements are abbreviated forms of TY: and TN: and are entirely equivalent in operation. It is acceptable to use the arithmetic conditional with either of these forms. For example, Y(X): will be executed on the condition that 1) the last match was successful, AND 2) the value of X is positive. (See the discussion of conditional execution in Section 3.2.)

The TH statement is like the T statement, except that the typing position "hangs" after the message is displayed. The usual progress to the next line is suppressed, so that the user's next response is displayed immediately after the message on the screen, rather than on the next line. Thus,

```
TH: 6 + 7 =
A:
```

allows the student to type his answer where it seems most natural for the answer to appear.

ERROR MESSAGES: Reference to a string variable that has not yet been assigned a value will cause display of the string variable name,including the "$." You may deliberately arrange for a string variable name, rather than a value, to appear. If the problem arises unexpectedly, it is usually caused by a misspelling of the variable name. A numeric variable to which no value has been assigned is displayed with a value of zero.

EXAMPLE:

(PILOT program)	(display)
*START	
T: Please tell me your $name.	Please tell me your $name.
A:$NAME	(user enters "Little Red Hen")
T: Hi, $NAME!	Hi, Little Red Hen!
: How old are you?	How old are you?
A:#a	(user enters "8")
T:Is it fun to be #a ?	Is it fun to be 8 ?

STATEMENT

```
****
* ACCEPT (A:)
****
```

SYNTAX:

```
[<label>] A [<cond>] :
[<label>] A [<cond>] : $<string variable>
[<label>] A [<cond>] : #<numeric variable>
```

DESCRIPTION: Makes it possible for the user to communicate with PILOT from the keyboard. While entering a response, the user may cancel the last character entered by depressing the DELete key, and cancel the current line by depressing ctrl-X. ("!" is displayed on the screen whenever ctrl-X is used.) A A carriage return signals the termination of a line. If the line being entered exceeds the maximum length allowed for a response (i.e., if the present or default value of the parameter INMAX is exceeded), PILOT supplies its own carriage return and regards the line as terminated.

If there is nothing to the right of the colon in an A statement, the response will be retained in a special "temporary entry buffer," so that it may be considered by subsequent Match and WRite commands (see later).

If a variable name is used, then the response will be stored as a value for that variable.

If a numeric variable name is included and a non-numeric response is entered, an error message will be displayed and another response will be accepted.

LIMITATIONS: The length of a response is limited to 80 characters or to a lower limit set by the keyword statement INMAX. (The syntax of the INMAX statement will be given in Section 4.5.) INMAX has an initial value of 64 in this version of PILOT. A numeric variable response must be an integer between -32768 and 32767.

ERROR MESSAGES: "*NUMERIC RESPONSE REQUIRED" occurs if a non-numeric entry is attempted in response to a command requesting a value for a numeric variable. The user is expected to enter another response.

"*NO ROOM" indicates that the area available in memory for string variable storage has been exhausted. (See discussion of NEW$, in section 4.3, for some ideas about how to cope with this problem.)

EXAMPLE:

```
(PILOT program)                    (display)

T:WHO ARE YOU?                     WHO ARE YOU?
A:                                 (user enters "your son")
.....                              .....
.....                              .....
T:WHAT IS YOUR NAME?               WHAT IS YOUR NAME?
A:$NAME                            (user enters "Timothy")
.....                              .....
.....                              .....
T:WHAT IS YOUR AGE?                WHAT IS YOUR AGE?
A:#A                               (user enters "10")
.....                              .....
.....                              .....
```

STATEMENT

****
* MATCH (M:)
* MATCH FOR COMMA (MC:)
* MATCH JUMP (MJ:)
****

SYNTAX:

```
[<label>] M  [<cond>] : <pattern>[,<pattern>...,<pattern>]
[<label>] MC [<cond>] : <pattern>[^<pattern>...^<pattern>]
[<label>] MJ [<cond>] : <pattern>[,<pattern>...,<pattern>]
```

DESCRIPTION: The response received by the last A statement is scanned for occurrences of ANY of the character patterns that follow the colon in the Match statement. If the response is found to contain one or more such strings, a subsequent statement including a Y condition will be executed. In this form of the M statement, commas are used to separate patterns in the list, with blanks considered part of the pattern (except those following the last item. (Any response is regarded as as though it begins with a blank, however, so that the statement M: YES will match a "yes" response to the previous question, whether or not the user preceded the response with a blank.) If you want to specify a pattern ending with a blank as the last pattern in your list, follow that pattern with a comma, as in the third example below. Contiguous blanks are reduced to one. For example, the pattern "M r." will be found to match with "M r." "Mr." will match with neither "M r." nor "M r.", however. Here is a more technical description of how the Match statement operates:

1. Each pattern in the M statement has multiple blanks reduced to one.
2. The user's last response has a blank added to each end.
3. The user's last response has multiple blanks reduced to one.

4. A moving window scan of the response is made with each pattern, until either a match is found or the input is exhausted.

MC: is a keyword extension that allows a search for text containing commas. In this case, a caret (^) is used as a separator between patterns. On some printers this character (5EH) will appear as an up arrow or a vertical line.

The MATCH JUMP statement (MJ:) causes a jump to the next M, MC, or MJ) statement, if the present attempt to match a response is unsuccessful. Look at the example in the discussion of the Jump statement MJ: YES would be exactly equivalent in execution to M: YES followed by JN:@M.

EXAMPLES:

```
M:A,B,C
        Matches A or B or HAT or ALICE or JOB
        Does not match TENT or X

M:  A, B, C
        Matches A or B or ALICE
        Does not match JOB or HAT

M:  A , B , C ,
        Matches only A or B or C
```

```
****
* JUMP (J:)
****
```

STATEMENT

SYNTAX:

```
[<label>] J  [<cond>] : [*] <destination label>
[<label>] J  [<cond>] : @M
[<label>] J  [<cond>] : @P
```

DESCRIPTION: Causes a jump to the specified destination in the current program. The most common form of the statement is that which results in a jump to a particular label. (The asterisk is optional.)

The other two forms of the statement, illustrated above, cause a jump to the next Match statement (M, MJ, MC), or PR statement, respectively. The PR statement will be treated in Section 4.4. In general, it serves only as a destination for a J or U statement.

In case it is not immediately evident why one would want to use the alternative forms, let us consider an example. The student has just finished reading a passage about color and color groupings. As a test of his comprehension, we ask him to name a warm primary color. We want to give him an opportunity

to review any material that he does not seem to understand. Thus we want not only to check whether his answer is correct, but to determine in what respect his answer is incorrect, and proceed accordingly. He need not be asked to review what he already understands. Using the different forms of the Jump instruction, we can generate the code in the example below.

The Match Jump statement (MJ:) provides a shorthand for the sequence of Match (M:) followed by JN:@M. The syntax is shown under our description of the Match statement (M:).

ERROR MESSAGES: The designation for a destination that can not be found will be displayed, followed by:

"-NOT FOUND"

EXAMPLE:

```
    (PILOT program)                    (display)

    T: Name a warm primary color.   Name a warm primary color.
*TEST1 A:$color                     (user enters "green")
    M: red,yellow,blue              (match not found)
    JY:@M
T:$color is not a primary color;   Green is not a primary color;
:Let's review the primary colors.  Let's review the primary ....
     ....                           ....
     ....                           ....
     : Now name a primary color.    Now name a primary color.
    J:*TEST1                        (user enters "blue")
    M:blue
    JN:@P
T: $color is not a warm color;     Blue is not a warm color;
:Let's review the warm and cool    Let's review the warm.....
:colors.
     ....                           .....
     ....                           .....
    PR:
     ....                           .....
     ....                           .....
```

```
                    ****
STATEMENT           * USE SUBROUTINE (U:)
                    ****
```

SYNTAX:

```
        [<label>] U [<cond>] : [*]<destination label>
        [<label>] U [<cond>] : @M
        [<label>] U [<cond>] : @P
```

DESCRIPTION: Causes a jump to a specified subroutine in the current program, and returns control, after the execution of that subroutine, to the statement following the Use instruction.

The end of a subroutine is marked by an End (E:) statement; the beginning of a subroutine is indicated by the label or instruction to the right of the colon in the Use statement.

Usually, the destination of this kind of jump is a label (with optional asterisk). The other possible destinations are @M and @P, just as in the J statement.

If you neglect to mark the end of a subroutine, the next End statement in your program will be regarded as the subroutine terminator. If the next End statement is the final statement in your program, control will return to the statement following the Use. Therefore, it is a good idea to be careful about marking subroutines, and to check for this kind of error when you find that some unlikely portion of your program is being repeated. (Consider what must happen if your U statement occurs between the desired subroutine and the next E statement in your program: the calling statement will be encountered AS PART OF THE SUBROUTINE, and will continually reinitiate the execution of that subroutine!)

LIMITATIONS: A subroutine may be used within another subroutine with the limitation that no more than seven Use statements may be pending at once. (The example on the next page has a maximum of three Use statements pending at any particular point in the program.)

ERROR MESSAGES: The name of a nonexistent destination will be displayed, followed by:

"-NOT FOUND"

If too many subroutines are pending, you will receive the message:

"*USE DEPTH EXCEEDED"

If a subroutine is being re-executed within itself because the programmer failed to include a subroutine terminator (see above), the Use depth will eventually be exceeded.

EXAMPLE:

```
      (PILOT program)                    (display)

*START  T:THIS IS WHERE WE START.    THIS IS WHERE WE START.
     T:DO YOU NEED INSTRUCTIONS?     DO YOU NEED INSTRUCTIONS.
     A:                              (user enters "yes")
     M:  YES                         (match found)
     UY:*INSTR                       (use subroutine INSTR)
       .....                          .....
       .....                          .....
*INSTR
T:THIS IS WHAT YOU NEED TO KNOW     THIS IS WHAT YOU NEED TO KNOW
       .....                          .....
       .....                          .....
     E:                              (return from subroutine)
```

```
            ****
STATEMENT   * END OF SUBROUTINE OR PROGRAM (E:)
            ****
```

SYNTAX:

```
                [<label>] E [<cond>] :
```

DESCRIPTION: Indicates the end of a subroutine or the end of the current program. The first E-statement encountered during the execution of a subroutine will be regarded as the end of that subroutine, and will return control to the statement following the calling (Use) statement for that subroutine. If an End statement is encountered with no Use statement pending, control is returned to PILOT restart.

Upon termination of the PILOT program, a data collection file left open will be closed. Data collection files will be discussed in Section 4.6.

LIMITATIONS: A maximum of seven subroutines may be be in execution at a given point in a PILOT program. See the error message in connection with the Use statement.

EXAMPLE:

```
(PILOT program)          (display)
*START  U:*FIRST
        T: TIME           NOW
*END    E:                IS
*FIRST  U:*SECOND         THE
        T: THE            TIME
        E:
*SECOND U:*THIRD
        T: IS
        E:
*THIRD  T: NOW
        E:
```

STATEMENT

****
* COMPUTE (C:)
****

SYNTAX:

```
[<label>] C [<cond>] : <num variable> = <num expression>
```

DESCRIPTION: Evaluates the numeric expression and assigns the result to the numeric variable.

PILOT lets you add (+), subtract (-), multiply (*), divide (/), return the remainder of a division (%), or call for a random number (RND(n)). There are some general rules to remember when you compute with PILOT:

1) There are no fractions or decimals: all numbers containing fractions or decimals are truncated to their integer part. 4 divided by 5 is zero, -1Ø divided by 3 is -3.

2) The % operation returns only the remainder of a division. If you wanted to execute a certain statement on condition that n be an odd number, you could write:

```
....
....
C: A= n % 2
T(A):#n is an odd number.
....
....
```

because if n is odd, the remainder of the division of #n by 2 will be greater than zero.

3) The expression RND(n) represents a random number in the range of 1 to n, with n having a maximum value of 32767.

4) The order of operations is very important. In a numeric expression containing multiple-operations:

a) The unary negative (that which reverses the sign of a number by negating it) is evaluated first.
b) Anything in parentheses is evaluated second. (Within parentheses the normal order of operations applies.)
c) Multiplication, division, and the return-remainder operation are evaluated third. (In the absence of parentheses, two operations of the same rank are performed from left to right, so that 2*2/3=1, not Ø.)
d) Addition and subtraction are evaluated last. (3-2+5=6)

LIMITATIONS: The expression to the left of the equals sign must be a numeric variable name, with the # omitted; it may or may not be a variable to which a value has already been assigned in the PILOT program. (If there has been no previous assignment within your program, the initial value is zero.) Neither the result of a computation, nor any of its elements, may be outside the acceptable range of integers -32678 to 32767: an attempt to store too high or too low a value will result in the assignment of -32768 or 32767, (whichever is closer to the number indicated).

ERROR MESSAGES: An offending expression is displayed, followed by the message:

"*ILLEGAL EXPRESSION"
or
"*VALUE OUT OF RANGE(-32768 TO 32767)"

The message "*ILLEGAL EXPRESSION" can occur as a result of bad syntax or in response to an illegal numeric variable name. After the message is displayed, PILOT awaits an entry from the keyboard. If <cr> is entered, execution proceeds with the next PILOT line. (You can, alternatively, supply an immediate command.)

The second message occurs if there is an attempt to set a value that is either too high or too low to be acceptable in PILOT (not between -32768 and 32767).

EXAMPLES:

(PILOT program)	(display)
C: x=(a+b)*(b-c)	(computations are made)
C: q=(RND(n)+a)%1Ø	

STATEMENT

```
****
* REMARK  (R:)
* REMARK WRITE (RW:)
****
```

SYNTAX:

```
[<label>] R  : <text>
[<label>] RW : <text>
```

DESCRIPTION: The Remark statement is a way of including useful descriptive information in the program listing without having the computer "do" anything about it. A Remark may be placed at any point in a PILOT program.

How easy was it to read the code given in relation to the Jump statement? Does the insertion of R statements make any difference?

EXAMPLE:

(PILOT program) (display)

```
R:Test on Color                (Same as for code in Jump section)
T:
 :Name a warm primary color.
*TEST1
A:$color
R:Is $color primary?
M:red,yellow,blue
R:If the student named a primary color, jump to the next test.
JY:@M
R:If control reaches this point, $color is not primary.
R:  Review lesson.
T:$color is not a primary color;let's review the primary colors.
....
....
:Now name a primary color.
R:Repeat the first test.
J:*TEST1
R:The color is primary. Test for warm or cool.
*TEST2
M: blue
R: If there is no match, then the student has named a warm
R: primary color.  Proceed to the next problem.
JN:@P
R: $color is not warm. Review lesson.
T: $color is not a warm color;let's review warm and cool colors.
....
....
PR:
....
....
```

There are only two differences between this program and the earlier version of it. The differences are 1)that the second version might be a bit easier to read, and 2)that the second is somewhat longer than the first. Both of these factors must be taken into consideration when you write a program. In the example above, we have probably incorporated more comments than are really necessary. A user who already knows PILOT will not need very many of these. It is also important not to clutter a program with unnecessary material, because the length of a program is limited by the capacity of the computer. Every character of Remark, although PILOT will not act on it in any way, contributes to the length of the program.

The RW statement is a Remark statement that writes remarks on a cassette file. The RW statement is similar to the T statement, in that it involves a string of text, with substitution of current values for any string or numeric variables included in the text. The major difference is that whereas the T statement types text on the screen, the RW statement writes text into an opened data collection file on your cassette tape. Data collection files will be discussed in Section 4.6.

4.3 CURSOR AND VIDEO CONTROL INSTRUCTIONS

These instructions control the presentation of text on the video display. Recall the following two concepts introduced in Section 1.5:

1) The cursor marks the current position on the screen, and,

2) The screen may be divided into horizontal rows and vertical columns.

The position of the cursor on the video screen is called its "address."

The cursor control statements are:

```
                  ****
                  * CURSOR ADDRESS (CA:r,c)
                  * CLEAR AND HOME (CH:)
STATEMENT         * CLEAR TO END OF LINE (CL:)
                  * CLEAR TO END OF SCREEN (CE:)
                  * ROLL DISPLAY (ROLL:n or RL:n)
                  ****
SYNTAX:
-------------------------------------------------------------
                [<label>] CA [<cond>] : [<row>] [,<col>]
                [<label>] CH [<cond>] :
                [<label>] CL [<cond>] :
                [<label>] CE [<cond>] :
                [<label>] ROLL [<cond>] : [<num>]     or
                [<label>] RL   [<cond>] : [<num>]
-------------------------------------------------------------
```

DESCRIPTIONS: (Remember that the display is composed of 16 horizontal rows and 64 vertical columns. For the purpose of these commands, the first row on the screen should be designated row 1, and the first column should be designated column 1.)

CA: sets the cursor address by row and column at which the next text is displayed or the next input is accepted. The row and column may be indicated by either an integer constant or a numeric variable name. If column indication is omitted, it is set to 1, and if row is omitted, it is set to the last used row + 1. To omit the row designation, follow the colon with a comma and your desired column number. A statement of the form CA:6 moves the cursor to row 6, column 1; a statement of the form CA:,6 moves the cursor down one line, and positions it at the sixth column of that line.

CH: clears the screen and sets cursor address to row 1 and column 1.

CL: clears from the current cursor position to the end of the current line. The cursor address is not changed.

CE: clears from the current cursor position to the end of the screen. The cursor address is not changed.

ROLL: or RL: rolls screen information upward. Lines disappear off the top of the screen and are filled in from the bottom by blanks. The number of lines may be indicated by either an integer constant or a numeric variable name. If no number is given, the screen is rolled up one line.

EXAMPLES:

(PILOT program)	(display)
CA:2,n	(causes the next text to begin in the nth column of the second row on the screen)
CA:,6	(causes the next text to begin in the sixth column of the next row on the screen.)
RL:5	(causes the first 5 lines to be rolled off the top of the screen, and all subsequent lines to move upward. Blank lines count as lines.)

IMPORTANT NOTE:

None of these "cursor control" statements make the cursor appear as a character on the display screen. Outside of the the PILOT editor (section 5), it is probably best to consider the cursor as a position, rather than as a visible marker of that position.

In addition to providing control of the cursor address and making it possible to clear all or portions of the screen, PILOT provides an easy way to emphasize portions of text with reverse video (reversal of black and white in the display). A ctrl-DELete character in the text of a PILOT Type statement (T:) causes characters from that point to the end of the line to be displayed in reverse video. Each ctrl-DELete character encountered causes the display to reverse again; thus, a single word or letter can be reversed for emphasis. If the text of a T statement consists of only a ctrl-DELete character at the beginning and end separated by blanks, a line of solid cursor blocks will be displayed. If a line is empty, with the exception of a ctrl-DELete character in its rightmost position, the line will be defined as a series of blanks. (In this way, you can easily "erase" a word or phrase from an existing display by "typing" blanks over it.)

The TH statement in allows the user's next keyboard entry to appear immediately following the given line of text, instead of on the next line. There is an example in Section 4.1, where the T and TH statements are described.

4.4 AIDS TO CONVERSATION

There are a number of PILOT statements that make it easier to converse within the PILOT system. They are

```
          ****
STATEMENT * FOOT OF SCREEN HALT AND PROMPT (FOOT:)
          ****
```

SYNTAX:

[<label>] FOOT [<cond>] : [<text>]

DESCRIPTION: Causes the text provided after the colon to appear on the bottom line of the screen and executes an A statement. If there is no text after the colon, the message is:

'PRESS "RETURN" TO GO ON..'

Execution will resume when the user hits the RETURN key. (Even if the message is not "PRESS RETURN TO GO ON.")

When we thought about writing the reading exercise program, two of the problems were how much text could be displayed on the screen at once, and how to give the student enough time to read it. We decided to display a screenful of text and then ask a question at the bottom of the screen. FOOT is a combination of three commands:

```
CA:16
T: <text>
A:
```

A possible reason to use FOOT, as opposed to the three other statements, is that FOOT takes up less space in the program buffer; a disadvantage is that FOOT is not included in all versions of PILOT.

EXAMPLE:

```
        (PILOT program)              (display)

        ....                         ....
        ....                         ....
T:Now we'll go on ...                Now we'll go on...
  FOOT:                              PRESS "RETURN" TO GO ON
                                     (at line 16)
T:NEXT QUESTION:                     (user enters <cr>)
        ........                     NEXT QUESTION
```

```
                  ****
STATEMENT         * PAUSE (PAUSE: or PA:)
                  ****
```

SYNTAX:

```
              [<label>] PAUSE [<cond>] : [<num>]
              [<label>] PAUSE [<cond>] : [<num>]
```

DESCRIPTION: Causes program operation to halt for a specified length of time, 1 to 99 seconds. The length of time may be indicated by either an integer constant or a numeric variable name. If no time is given, the wait is approximately one second. (The figures given are for a normal Sol. Your computer may differ slightly.)

The PAUSE statement permits you to regulate the amount of time a student has to study text on the video screen. This feature of PILOT has an obvious application to timed exercises of various kinds. Consider how you could change the reading exercise program, if you did not want to give the student unlimited time to study each screenful of material.

Although PAUSE is theoretically available as an immediate command, there is no reason for a user ever to enter it, because at any time that he is free to enter such a command, he has unlimited time to study whatever information is displayed on the screen. (Remember that an immediate command may be entered in response to any A statement, and that PILOT always waits for the response to an A statement before resuming execution.)

EXAMPLE:

```
     (PILOT program)                (display)

      PAUSE:3                (results in a 3 second pause)
```

```
                ****
STATEMENT       * NEW STRING VARIABLES (NEW$:)
                ****
```

SYNTAX:

```
-------------------------------------------------------------
                [<label>] NEW$ [<cond>] :
-------------------------------------------------------------
```

DESCRIPTION: The NEW$ statement clears all string variables defined earlier in the program, and thereby reclaims the storage space that these occupied. You can use the $NEW statement to avoid cluttering the program buffer with unnecessary data whenever you have accumulated a lot of variables that are no longer in use.

One of the most important conversational features of PILOT is the ability to retain a user's response, so that it can be retrieved later in the program. The NEW$ statement lets the programmer deal with one of the major complications inherent in programming: the limited storage capacity of the computer.

Every time that you Accept a value for a string variable, that value must be stored in the program buffer area in the memooey of the computer. In PILOT a new value assigned to a previously defined string variable name does not supersed the old value for that variable; rather, successive entries exist side by side. Although only the most recent value is used by PILOT during the execution of a program, any former values continue to take up memory space. Eventually, if you keep assigning more values to more variables-and particularly if your program requires a considerable amount of room for its own storage--you will receive the message, "NO ROOM."

Note that this consideration does not apply to numeric variables. Each value assigned to a numeric variable actually replaces the old one, so that there are always 26 numeric variables, corresponding to the letters of the alphabet, stored in the memory of the computer. Each of these numeric variables always has a value (assumed to be zero where no other has been assigned) and none of them are stored with the program in the program buffer area. Because of these distinctions, there is really no storage problem with numeric variables. Such values are initialized, i.e., set to zero, rather than actually cleared from memory, in response to a CLEAR command. (CLEAR is discussed in Section 4.7, below.)

EXAMPLE:

```
     (PILOT program)                    (display)

  T:GIVE YOUR ANSWER NOW.        GIVE YOUR ANSWER NOW.
    *ANSWER A:$TEXT              (user enters "jhgblj")
    M:PATTERN                    (match not found)
    N:SOMETHING'S WRONG.         SOMETHING'S WRONG.
    N:PLEASE TRY THAT AGAIN.     PLEASE TRY THAT AGAIN.
    NEW$N:                       ....
    JN:*ANSWER                   ....
   T:FINE, LET'S GO ON.          FINE, LET'S GO ON.
     .....                       .....
```

```
                    ****
STATEMENT           * EXIT FROM PILOT (BYE:)
                    ****
```

SYNTAX:

```
                [<label>] BYE [<cond>] :
```

DESCRIPTION: Causes PILOT to discontinue operation, and returns control to SOLOS or CUTER.

This instruction is normally used in the immediate mode at the PILOT restart point.

EXAMPLE:

```
                  BYE        (at PILOT restart)
```

```
                    ****
STATEMENT           * PROBLEM START (PR:)
                    ****
```

SYNTAX:

```
                  [<label>] PR:
```

DESCRIPTION: Provides a destination for the J:@P or U:@P command.

The PR: statement is very much like a label, in that it initiates no activity but functions as a landmark to be considered by other operations. It can also have the mnemonic advantages of a label, insofar as it can designate the beginning of the next exercise, set of questions, or general phase of a program. There are two reasons that you might decide to use a PR statement, rather than a label:

1) Unless you use labels only two characters long, PR: statements are shorter.

2) By using PR: to signify the beginning of a new section or procedure, you can avoid cluttering your program with a lot of extra labels that don't have very much significance. (You might want to label every SET of questions, but you probably would not want to label every question.) Using too many labels can be as confusing as using none: imagine trying to find someone who has told you, "In case you don't recognize me, I'll be wearing a hat," in a crowd in which many people are wearing hats. If you have any idea how the person looks, you will very likely find him, but not nearly as easily as you would if not many other people in the crowd were wearing hats.

STATEMENT

```
****
* CALL PROGRAM ELSEWHERE IN
*       MEMORY (CALL:)
****
```

SYNTAX:

```
[<label>] CALL: <address> [,<argument>]
CALL: <address>    (at PILOT restart)
```

DESCRIPTION: Calls the assembly language routine whose address in memory is indicated by the first parameter following the colon, and stores an argument, if one is given, in registers D and E. The address must be a decimal number; the argument must be a number greater than -32768 and less than 32767.

This statement permits you to call an assembly language program that you have stored in memory outside the PILOT program buffer. If the assembly language program includes a RETurn instruction, control will return to the PILOT program statement which immediately follows the CALL.

EXAMPLE:

```
(PILOT program)                    (function)

....                               ....
....                               ....
CALL: 16385,-25                    (executes a program at
                                    address 16385 and puts
                                    a value of -25 into
                                    registers D and E)
```

4.5 SET PARAMETERS

The statements described in this section specify the way in which PILOT will accept input and present output. To set a few of these parameters, you will need to know something about the equipment you are using. If you do not intend to use any output device other than the video display, you will not need to SET O or N at all.

Notice that the SET statement corresponds to the SOLOS/CUTER SET commands. PILOT enables you to set these parameters without exiting to SOLOS/CUTER.

```
                ****
STATEMENT       * SET SOLOS PARAMETERS (SET:)
                ****
```

SYNTAX:

```
---------------------------------------------------------------
                [<label>] SET [<cond>] : S=n
                [<label>] SET [<cond>] : O=n
                [<label>] SET [<cond>] : N=n
                [<label>] SET [<cond>] : M=n
                SET S=n (etc.)   (at PILOT restart)
---------------------------------------------------------------
```

DESCRIPTION: Sets the SOLOS parameters of DISPLAY SPEED, OUTPUT PORT, NUMBER OF NULLS, or memory setting, either during the execution of a PILOT program or as a direct action at PILOT restart.

SET S=n sets display speed of the screen, where n determines the speed with Ø fastest and 9 slowest. You don't have to set this parameter unless you are dissatisfied with the default display speed.

SET O=n determines where output is to be sent by the computer. Port Ø is the screen, 1 is the serial port, 2 is the parallel port, and 3 allows use of a user routine previously set in SOLOS. The nature of your equipment determines how these parameters should be set. For example, your printer MAY require a serial port, but not necessarily. Find out exactly what specifications are appropriate to your equipment. If you do not set O at all, the output will appear on your video screen.

SET N=n sets the number of nulls provided after a carriage return. Again, a particular printer will have a particular requirement in this respect. The only way to establish a fit value for N is to know what your equipment requires. The default is no nulls.

SET M=n determines the upper limit of the program buffer. If your computer has more than 16K of memory, you might find it advantageous to increase the size of the program buffer, so that it can accommodate larger programs and text files.

Any program or text that is stored in the program buffer at the time of this command is lost, so it is a good idea to change this parameter either before the intended file is put into the buffer, or after it has been saved.

M must be a decimal number which points to a memory location LOWER than the lowest memory location occupied by SOLOS or CUTER. (For SOLOS, this memory location is CØØØ in base 16, or 49152 in base 1Ø. The location for CUTER is whatever you indicate when you load CUTER into memory.) Remember that the value you specify for M must be in base 1Ø, not in base 16! The default value for M is 16383.

The greatest value possible for M is 32767 Decimal, or 7FFF Hexadecimal. If you want to set an upper bound higher than 7FFF, use the following formula: for every 1ØØ Hex (or 256 Dec) greater than 7FFF, subtract 256 from 32767, and enter the result as a negative number. Thus, to set an upper bound of 8ØFF, enter SET: M=-32511.

EXAMPLE: (Assumes that your printer uses the serial port)

```
(PILOT program)
   SET: O=1        At PILOT restart, means "Now use printer.
   LIST            Command to produce a listing (on printer)
   SET: O=Ø        Means "Now use video display, again."
```

```
          ****
STATEMENT * INPUT MAXIMUM NUMBER OF CHARACTERS (INMAX:)
          ****
```

SYNTAX:

```
[<label>] INMAX [<cond>] : <integer>
[<label>] INMAX [<cond>] : <numeric variable>
```

DESCRIPTION: Limits the number of characters to be considered by subsequent A statements. The limit must be expressed as either an integer (1-8Ø) or a numeric variable name, and must appear immediately to the right of the colon. (The maximum number of characters includes blanks; e.g., the strings "123 123" and "1231231" are both seven characters long. The maximum number does NOT include the carriage return.)

If the user supplies the maximum number of characters, PILOT will add its own carriage return, and regard the response given as complete. Any additional text that the user types will be disregarded. (It will not even be visible on the screen.) If INMAX is set to 1, for instance, PILOT will react immediately to the first character entered, without the user's having to press the carriage return key. Of course, if the user gives a carriage return before the character limit is reached, the response is regarded as complete at that point.

LIMITATIONS: INMAX values should be set between 1 and 80. If you do not set INMAX, PILOT allows a length of 64 characters, corresponding to the width of the video screen. A value greater than 64 causes a harmless overflow to the next line on the screen. INMAX may be set and altered any number of times during a PILOT program.

The entry of an immediate command will temporarily override INMAX; upon return to program execution, the former value of INMAX will be restored.

EXAMPLE:

```
(PILOT program)                    (display)

T:Enter a five digit number.       Enter a five digit number.
  INMAX:5                          (User enters 123456.)
A:#N                               12345
T:Thank you.                       Thank you.
T:You entered #N.                  You entered 12345.
```

4.6 FILE MANIPULATION INSTRUCTIONS

The next series of statements pertain to the collection, storage, and retrieval of information on files. (The definition and a few of the basic characteristics of a file were discussed in Section 1.4 of this manual.) In general, you will be writing files to serve the following purposes:

1) To make a record of data accumulated during the execution of a PILOT program, or
2) To save a program or data that you have just entered or altered using the EDITor.

Actually, every that you LOAD a program from the cassette, you are reading a file. The two test programs are program files exactly like those PILOT will permit you to create. You can read a data file to EDIT it, LIST it, or execute it. (This last option is available only if the data in the file happens to be a collection of program statements.)

All files read and written by PILOT are in standard SOLOS/CUTER format. A file created using PILOT WRite statements has a "byte access" structure, whereas a file that has been SAVEd, using the PILOT SAVE command, has a "block access" structure. (Look at Section 5 of your SOLOS/CUTER manual for further details regarding file structures. WRite and SAVE will be introduced in this and the next section.) In Appendix B you will find an example of a BASIC program that reads data from a "byte access" file written in PILOT. BASIC can be used to read only those files that have "byte access" structure; "block access" files can, however, be read from SOLOS/CUTER. If you want to read a PILOT file (of the "byte access" type) in PTDOS, you must use the CTAPE1 driver described in your PTDOS manual.

STATEMENT

```
****
* OPEN DATA FILE (OPENF:)
* CLOSE DATA FILE (CLOSEF:)
****
```

SYNTAX:

```
[<label>] OPENF [<cond>] : [name] [/u]
[<label>] CLOSEF [<cond>] :
```

DESCRIPTION: To store data in a file, it is necessary to "open" that file, just as, to store groceries in a cupboard, it is necessary to open the cupboard door. It is also desirable to open the correct cupboard door, in order not to store the tea with the linens and risk unpleasant surprises. The OPENF statement should be before the first WRite or RW statement in a PILOT program. You will need more than one OPENF statement in a program if you have closed one file and want to open another one. ONLY one data file can be open at one time.

A file can be given a name one to five characters long, with no blanks or slashes. Assigning a name to a file, and using that name when you open, read, or load the file, is a practice that you are not likely to regret. (Think of trying to locate a particular, unlabeled file in a large filing cabinet.) When you give a file a name, the name literally becomes a part of that file, so that when you try to read or load, the computer can look for the filename. If a file does not have a name, the only way to read or load it is to know exactly where that file is recorded on the cassette tape, and to position the tape accordingly.

A unit number may be affixed to a filename. (The /u does not count as part of the five-character name.) If you do not specify any unit number, unit 1 will be assumed. You may, however, specify unit 2, so that tape unit 1 can be used for reading PILOT program segments, and unit 2 used for data collection. (The possibility of using two cassette recorders was mentioned in Section 2.)

The CLOSEF statement is used to close the current file, ensuring that all data from prior WRITE statements is actually entered on tape, and making it possible to open another file. (To continue the previous analogy, there is now a closed cupboard door between the dog and the dog food.) PILOT will close any file that is open at the end of a program. One of the primary uses of CLOSEF is to close a file during the execution of a program, so that data collection can begin on another file.

EXAMPLE:

```
(PILOT program)                 (display)

.....                             .....
.....                             .....
OPENF:FRED/2                    (file "FRED" opened on unit 2)
  WR:                           (user's last response written
                                 in FRED)
CLOSEF:                         (FRED closed)
  E:
```

****
STATEMENT * WRITE INTO DATA FILE (WRITE: or WR:)
****

SYNTAX:

```
[<label>] WRITE [<cond>] :
```

DESCRIPTION: The last entry resulting from an A statement is written to cassette tape.

A WRITE statement will be executed only if it has been preceded in the PILOT instruction sequence by an OPENF statement. The OPENF statement determines the file name and tape unit in use.

If you Accept a value for a numeric variable, and then alter the value of the variable in a Compute statement, a subsequent WRite statement will store the last value entered by the user, not the new value for the variable. If you want to record anything other than the user's last entry to an A statement, you must use the RW command, which records text, including the current value for any named variable, in the open file. The RW statement need not contain text other than the desired variable name.

EXAMPLE:

```
(PILOT program)                 (display)

OPENF:CITY                      ("CITY" opened on unit 1)
T: PLEASE TYPE YOUR ANSWER.     PLEASE TYPE...
A:                              (user types "Pittsburg")
WRITE:                          (Pittsburg is written into
                                 file CITY on tape unit 1)
```

```
                    ****
STATEMENT           * REWIND TAPE (REWIND:)
                    ****
```

SYNTAX:

```
[<label>] REWIND [<cond>] : [u]
REWIND [ u]    (at PILOT restart)
```

DESCRIPTION: Turns on the motor for tape unit 1 or 2 and displays the message:

"REWIND DECK 1" or
"REWIND DECK 2"

If you hit the MODE SELECT key, the motor will be turned off. If you hit the MODE SELECT key while the reading a file, you will get the message, "TAPE READ ERROR." Once the file has been read, you MUST use the MODE SELECT to return control to PILOT before you can enter any other command.

EXAMPLE:

(PILOT program)	(display)
REWIND:2	REWIND DECK 2 (The motor for tape unit 2 is turned on.)

```
                ****
STATEMENT       * LOAD NEW PROGRAM OR PROGRAM SEGMENT (LOAD:)
                ****
```

SYNTAX:

```
[<label>] LOAD [< file name>] [/u] [<cond>] :
LOAD [< file name>] [/u]    (at PILOT restart)
(LOAD KEY,RETURN KEY)       (at PILOT restart)
```

DESCRIPTION: Loads a PILOT program, or any file that has been SAVEd, into the program buffer and initiates execution if the file is executable. If the file you are attempting to LOAD is not a PILOT program, it is merely displayed on the screen. The program or program segment just LOADed will replace the program previously in operation. If a filename is not supplied, the first file found on the tape will be LOADed. If a tape unit number (e.g. /2) is not supplied, tape unit 1 will be used.

This command is used in the immediate mode to load a new program and also in a program sequence to replace the current segment of a long program with the next segment.

Any time that you return to PILOT restart to LOAD a new segment of a large program, you have the option of using the CLEAR command to set all numeric variables to zero; otherwise, the values for these variables will be retained. In the case of a program segment larger than the previous one, there is a good chance that LOADing the larger segment will cause the loss of some string variables. The way to avoid this difficulty is to arrange your program segments so that each is shorter than the last. (A program segment is just a program that is intended to be executed in sequence with other programs.) The INFO command, which will be described in Section 4.7, will enable you to determine the size of any program or program segment.

If you want to EDIT or LIST a file that has been SAVEd, it is more convenient to use the GET command than to use LOAD. (EDIT is discussed in Section 5 of this manual; GET and LIST are described later in this section.)

ERROR MESSAGES: "*TAPE READ ERROR" indicates that there has been a bad read of the tape file or that MODE SELECT (ctrl-@) has caused termination of the loading process.

MODE SELECT (ctrl-@) from the keyboard will abort the load.

EXAMPLE:

```
(PILOT program)                    (display)

     .....                          .....
     .....                          .....
     LOAD:                         PREPARE TAPE 1  (or 2)
                                   (user puts recorder in "play";
                                    file loaded from tape unit 1)
```

```
                  ****
STATEMENT         * READ FROM DATA FILE (READ:)
                  ****
```

SYNTAX:

```
            [<label>] READ [<file name>] [/u] [<cond>] :
```

DESCRIPTION: Reads a data file in "byte access" format (i.e., a file written by PILOT but not SAVEd) from the cassette into the program buffer. Initiates execution, if possible. If the data in the file is not executable PILOT code, for example, if it is a collection of student scores and responses, it will simply be displayed. It is this feature of PILOT, the attempted execution of a data file, that makes it possible for a PILOT program to assist a user in developing another PILOT program!

If no filename is specified, the file recorded next on the cassette tape will be READ. If no unit number is specified, unit 1 is used.

Why would you want to READ a data file?

1) To obtain a LISTing of it. Use the SET command to send the output to the appropriate printing device; then use LIST (see below) to obtain your listing.

2) To EDIT it. For example, you or the user might want to change a PILOT program because it does not "run" in the expected fashion. If you want to EDIT a PILOT program that has been SAVEd, it is better to GET it, rather than READ it. If you READ it, you will have to enter 1) an immediate \EDIT: or 2) a \ <carriage return> followed by EDIT, in response to an Accept statement in the program. If you intend to EDIT a file containing only unexecutable data, you will initiate the EDIT from PILOT restart. (EDIT is described in Section 5, below.)

3) To SAVE it. The SAVE command is treated below; it causes a "byte access" file created with WR: and RW: statements to be converted to "block access" format.

EXAMPLE:

(PILOT program)	(display)
READ:	PREPARE TAPE 1 (or 2) (user puts recorder in "play"; file read from tape unit 1)

4.7 COMMANDS USUALLY USED IN IMMEDIATE MODE

Although all of these commands except SCRATCH can be entered as program statements, they will probably be most useful from PILOT restart. After a command is executed from the restart point, PILOT expects to receive a carriage return. When you type the carriage return, you will be back at restart and from there can enter another command or return to program execution.

STATEMENT

```
****
* GET PROGRAM (GET:)
****
```

SYNTAX:

```
GET  [< file name>] [</u>]
                    (at PILOT restart)
```

DESCRIPTION: Loads the named program from cassette into the program buffer, without initiating execution. If a file name is not specified, the next file found on the tape is loaded. If no unit number is specified, unit 1 is used.

This command provides a convenient way of loading a SAVEd file into memory for a purpose other than that of running the program, for example, to EDIT or LIST. Only a "block access" file, one that has been SAVEd either in PILOT or in SOLOS/CUTER, can be loaded in this fashion. Once the file has been loaded, PILOT returns to restart. If you decide that you want to execute the program after all, type RUN.

EXAMPLE:

(command form)	(display)
GET BOX/2	PREPARE TAPE 2 (user puts recorder in "play"; a file called BOX is loaded from unit 2)

```
                ****
STATEMENT       * RUN PROGRAM (RUN:)
                ****
```

SYNTAX:

RUN	(at PILOT restart)

DESCRIPTION: Starts execution of the program in the program buffer. If there is no program in the program buffer, nothing happens.

If you have loaded a file using GET, or if you have discontinued the execution of a program, you can give this command from restart to execute the program, beginning at its first executable statement, i.e., not wherever you left off.

EXAMPLE:

(command form)	(function)
RUN	Starts execution of the program in memory

```
                    ****
STATEMENT           * LIST PROGRAM (LIST:)
                    ****
```

SYNTAX:

[<label>] LIST [<cond>] :	
LIST	(at PILOT restart)

DESCRIPTION: Lists the program in memory on whatever device is specified in the most recent SET O= statement. This command will most often be used in the immediate mode, either to review the program on the display screen or to create a printed listing.

The LIST command may be given from within a program or from PILOT restart. If you want to stop a LISTing that is in progress, hit the ESCape key on your keyboard. Control will return to the program in memory or to PILOT restart, depending on the point from which the command was given. If you want to halt a LISTing momentarily, you can hit the space-bar. The LISTing resumes when another key is pressed. If you are LISTing on the screen, you can reset the video display speed by hitting number key: Ø is fastest, 9 is slowest.

KEYS Ø-9: Reset video display speed; Ø=fast,9=slow.

EXAMPLE:

(command form)	(function)
LIST	lists program in memory

```
                    ****
                    * SAVE PROGRAM (SAVE)
STATEMENT           * COPY PROGRAM (COPY)
                    * CUSTOM COPY PILOT AND PROGRAM (CUSTOM)
                    ****
```

SYNTAX:

SAVE [< file name>] [/u]	(at PILOT restart)
CUSTOM [<file name>] [/u]	(at PILOT restart)
COPY [n]	(at PILOT restart)

DESCRIPTION: SAVE writes the PILOT program currently in memory on cassette tape; CUSTOM writes both the PILOT interpreter and the program in memory on cassette tape. The resulting program files are in "block access" format. SAVEing a file is the only way to record it after it has been EDITed, unless you also want to record the interpreter. COPY copies n files from tape unit 1 to tape unit 2.

If a file name is not supplied as part of the SAVE or CUSTOM command, the file name written on the tape will be blank. If a /2 is not supplied, tape unit 1 will be used. The file type will be "P."

To load a file written with the CUSTOM command, follow the instructions for making contact with PILOT (Section 2). Once the tape has been read, the program that was in the buffer when you gave the command will begin to execute immediately.

If you want to EDIT the third of five program segments, without having to LOAD or GET each of them in succession, use COPY to copy the first two files from tape unit 1 to tape unit 2. Integer n specifies the number of files that you want to COPY. If there is no n included in the command entered, one file will be copied. (If you are not using two cassette recorders, you will not use COPY.)

LIMITATIONS: SAVE, CUSTOM, and COPY are generally used only in immediate mode at PILOT restart.

EXAMPLE:

(command form)	(function)
SAVE TEST1	Saves a file called "TEST1" on unit 1
COPY 4	Copies 4 files from unit 1 to unit 2
CUSTOM TEST1	Saves the PILOT interpreter and "TEST1" on unit 1

****
STATEMENT * SCRATCH CURRENT PROGRAM (SCRATCH)
****

SYNTAX:

SCRATCH (at PILOT restart)

DESCRIPTION: Deletes the current program from memory, along with all string variables. If there is no program in the program buffer, nothing happens. SCRATCH can be entered only from PILOT restart.

The PILOT interpreter is not deleted by this command.

STATEMENT

```
****
* FILE SIZE INFORMATION (INFO:)
****
```

SYNTAX:

INFO	(at PILOT restart)

DESCRIPTION: Tells where your PILOT program is in the memory of the computer. Four items of information will be displayed on the screen:

1) MEMORY SETTING - This number, which is in base 16 (10 through 15 are denoted by the letters A through F), tells where the program buffer ends.

2) FILE BEGINNING - This number, also in base 16, indicates where the program buffer starts.

3) CHARACTERS IN FILE, including blanks - This number, in base 10, indicates the number of characters in the program.

4) SPACE REMAINING - This number, also in base 10, indicates the number of unused characters remaining in memory.

After this information has been displayed, PILOT awaits a carriage return before going back to PILOT restart.

The INFO command helps you to decide whether your program is too long relative to the amount of memory allocated to it. (Remember that string variables are stored along with the program in the program buffer.) If your program almost fills the buffer, you should consider shortening the program, dividing it into segments, or assigning more memory (with the SET M= statement.)

EXAMPLE:

(command form)	(display)
INFO	5FFFH Memory setting. 1D69H File beginning. 2142 Characters in file. 14905 Space remaining ... Press RETURN

****

STATEMENT * CLEAR NUMERIC VARIABLES (CLEAR)

****

SYNTAX:

CLEAR (at PILOT restart)

DESCRIPTION: Re-initializes all numeric variables to zero. (If you have returned to PILOT restart to execute a new program, you may want to make use of this command to prevent using an old value accidentally.)

SECTION 5

ENTERING AND EDITING YOUR PILOT PROGRAM

5.1 THE EDIT COMMAND

SYNTAX:

EDIT	(At PILOT restart)

DESCRIPTION: Makes available a new set of commands that can be used to create and alter text and program files. In EDIT the cursor may be positioned anywhere on the screen, lines may be scrolled up and down, and characters and entire lines may be inserted or deleted. There are also provisions for searching the file for strings, and for moving quickly to any one-tenth portion of the file from Ø to 9.

EDIT is different from all other PILOT commands, in that when you enter it, or execute it within a program, you move into an area of PILOT that is really quite separate from the interpreter. The EDITor is a kind of subsystem of PILOT. It enables you to make changes in any SOLOS file (whether or not it was created by PILOT, or for use with PILOT.) The EDITor does not not know or care whether you are working with a PILOT program, a BASIC program, a list of student responses, or a letter to your veterinarian.

The EDIT command may be entered, with a slash preceding and a colon following, in response to any A statement, or it may be entered from restart without the slash and colon. If you want to enter your PILOT program for the first time, give the EDIT command from PILOT restart, without LOADing or READing anything into the program buffer. If you want to EDIT a file that you have created in PILOT with WRite statements and never SAVEd in "block access" format, you must READ it into memory. If you have SAVED a file and then want to EDIT it, use the GET command to load it into memory without executing it.

When you enter the EDIT command from PILOT restart, the first 16 lines of the file in memory are displayed on the Sol video display or VDM-1, with the cursor at line one and position one (column Ø). If you give the command in response to an A statement, or if EDIT: is actually a statement in your PILOT program, the first line on the display will be the A statement or the EDIT statement, respectively. If there are fewer than 15 lines of text to follow the first line on the display, or if the program buffer is empty, the remaining portion of the screen

will be filled with # signs. If you are entering a file for the first time, just begin typing; otherwise, the file displayed on the screen is ready for EDITing. The next few pages tell how to go about changing a file by using control characters. (Remember, a control character results from your pressing the control key simultaneously with another key on the keyboard.)

Press MODE SELECT to exit from the EDITor and return control to the PILOT restart point. If you want to execute the program you have just EDITed, press the carriage return and execution will begin. (Maybe you just wanted to look at a file before executing it to make sure that it was the right program, or maybe you entered or changed a program, and now want to see whether it really works.)

The EDITor in PILOT bears a great deal of similarity to the video editor in PTDOS (the disk operating system available with the Sol System III or IV). The two editors are NOT, however, identical!

Below is a list of the control keys used by the EDITor. A more complete description of each command is given after the list.

5.2 COMMAND CONTROL KEY LIST

CONTROL KEYS - FUNCTION:

CONTROL	W	- move cursor up one line
CONTROL	Z	- move cursor down one line
CONTROL	A	- move cursor left one character
CONTROL	S	- move cursor right one character
CONTROL	E	- move file up one line
CONTROL	X	- move file down one line
CONTROL	R	- scroll file up 16 lines
CONTROL	C	- scroll file down 16 lines
CONTROL	T	- toggle insert character mode; ON/OFF
CONTROL	H	- delete character under cursor
CONTROL	B	- insert line above cursor
CONTROL	P	- delete line
CONTROL	V	- initiate string search mode
CONTROL	I	- continue search for string
CONTROL	U	- move cursor to indicated column of this line
CONTROL	M	- same as RETURN (below)
CONTROL	J	- same as LINEFEED (below)
CONTROL	DEL	- cause subsequent characters to be in reverse video

OTHER KEYS	-	FUNCTION:
TAB	-	same as control I
MODE SEL	-	exit editor
RETURN	-	insert line below cursor
LINE FEED	-	delete all text to the right from the cursor & position cursor one line down
CURSOR UP	-	move cursor up one line (same as ctrl-W)
CURSOR DOWN	-	move cursor down one line (same as ctrl-C)
CURSOR LEFT	-	move cursor left one character (same as ctrl-A)
CURSOR RIGHT	-	move cursor right one character (same as ctrl-S)
HOME CURSOR	-	move cursor to upper left corner of screen
DELETE	-	backspace and erase a character
REPEAT	-	re-enter whatever other key, or combination of keys, is depressed; continue while REPEAT key is held

NOTE: The cursor keys on a Sol are on either side of the space-bar.

The control-DELete character is recorded in the file and displayed on the screen. On the Sol or VDM-1 it is a rectangle made up of diagonal lines. There are also a number of control characters which may be entered as text, without having any effect on program or EDITor operation. Here is a list of those characters, with the screen representation given to them by the 6574 Character Generator ROM. (If your system includes, instead, the 6575 Character Generator ROM, some of the representations will be different.

ctrl-D	≷	ctrl-L	↡	ctrl-Y	↾	ctrl-^	◱
ctrl-F	✓	ctrl-N	⊗	ctrl-[	⊖		
ctrl-G	♌	ctrl-O	⊙	ctrl-\	◰		
ctrl-K	↓	ctrl-Q	⊖	ctrl-]	◱		

5.3 DETAILED COMMAND DESCRIPTION

5.3.1 Cursor Positioning Commands

NOTE: Moving the cursor does not change the text.

The keys A,S,W,Z form a diamond on the input keyboard. When pressed simultaneously with the 'CTRL' (control) key, they move the cursor as indicated below:

CONTROL W	move cursor up one line
CONTROL Z	move cursor down one line
CONTROL A	move cursor left one character
CONTROL S	move cursor right one character

The Sol keyboard also contains four cursor control keys to the left and right of the space bar. These keys may be used to move the cursor in the direction indicated on the top of the key. The HOME CURSOR key moves the cursor to the upper left corner of the screen.

5.3.2 Screen Scroll Commands

Screen scroll commands are provided to allow the file to be "rolled" through the screen area until the desired file line is reached.

CONTROL E	scroll up one line
CONTROL X	scroll down one line
CONTROL R	scroll up sixteen lines
CONTROL C	scroll down sixteen lines

5.3.3 Direct File Positioning Commands

In addition to cursor positioning controls, the EDITor offers a way of searching for a specific string of text within your file. The search command is CONTROL V.

CONTROL V editor text search

When you type control V, lines 15 and 16 are cleared, line 15 is marked with a row of dashes, and the cursor is placed at the first position in line 16. At this point the EDITor is waiting for you to enter either: 1) An input line consisting of one or more characters, or 2) a single digit. The input is terminated by a carriage return.

If the CLEAR key is depressed in this mode, line 16 is erased.

1. Character entry:

Any occurrence of the string entered, regardless of preceding or following characters, will represent a find. Therefore, only enough characters to define the desired text uniquely need be supplied. As an example, "the qu" can be used to locate a line in the file containing "the quick brown fox."

Upon receiving a carriage return, the EDITor searches the file, beginning one line below the current cursor position, until a string match is made or until the end of the file is reached. At the beginning of a file, the search begins at the first line. If a match is found, the EDITor positions the line containing the match at the top of the screen. Make your intended changes--you have cursor control in search mode--and then issue a carriage return if you want to look for more occurrences of the same string later in the file. If no occurrence of the string is found before the end of the file is reached, the first 16 lines will be displayed again. Press the MODE SELECT key to get out of search mode. (In this situation, issuing a MODE SELECT will not result in an exit from EDIT, but only from the mode initiated by ctrl-V.)

2. Digit entry:

If you enter a digit from Ø to 9 at line 16, the file will be scrolled so that the top line on the video display screen marks the end of that tenth of the file which corresponds to the number entered. Thus, if the number is 5, the file will be positioned at the half-way point. If Ø is entered, the file will be positioned at the beginning. MODE SELECT will cause an exit from control V mode.

CONTROL I continue search

If the user wishes to continue searching for text matches after having left the control V mode, control I may be used to continue searching for the string that was last designated. The EDITor resumes the search at the first line following that in which the cursor resides at the time of the command and continues until a match is made or until the end of file is reached. The control I command may be given as often as is desired.

NOTE: The 'TAB' key generates the same code as control I, and may be used for the continue command.

5.3.4 File Modification Commands

CONTROL T character insert mode switch (on-off-on....)

Normal file characters input from the terminal are placed in the file in either of two modes. These modes, normal and insert, are alternately selected using the insert mode control.

When ctrl-T is OFF (default mode when EDITor is entered), each character that you type replaces what was formerly at the current cursor position, and the cursor moves to the right one place. When ctrl-T is ON, characters are actually inserted BEFORE the current cursor position, moving the character at that location, and any characters to the right of it, one position to the right. The cursor also advances one position. A line consists of a maximum of 64 characters, so you may begin to lose text that is pushed off the screen by the insertion.

CONTROL H delete character mode

The delete character command removes the character at the current cursor position and moves each character to the right of the cursor one position to the left.

CONTROL B insert line command

The line insert command puts a new blank line at the present cursor position, and moves each subsequent line of the file one row down. The cursor is moved to the first character position of the new line. Use this command to insert a new line 'above' the current line.

CONTROL P delete line command

This control removes the current cursor line from the file.

CONTROL J or LINEFEED blank remaining line

Linefeed deletes all characters to the right of the current cursor position (on the cursor line). The cursor appears at the beginning of the next line in the file.

CONTROL M or CARRIAGE RETURN scroll up & insert line

Carriage return scrolls up one line and inserts a blank line in the file. The cursor is moved to the first character position of the new line. Use RETURN to insert a line 'BELOW' the current cursor position. No characters on the current line are deleted. The exception to this rule is that, if a file contains fewer than 16 lines of text, RETURN will open a new blank line below the last line of text but will not scroll the file.

CONTROL U move cursor to indicated column of current line

After entering ctrl-U, enter a character, corresponding to the column at which you want the cursor to be positioned. Each character in the line below is the one to be entered to move the cursor to the column in which that character appears; the character "9" will cause the cursor to be moved to the tenth column of a line of text, so in the sample line, the character in that column is a "9." The character "[" moves the cursor to column 44.

Ø123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^-'abcdefghijklmno

SECTION 6

USING THE TEST PROGRAMS, PLTST AND WAPP

In Section 2, about making contact with PILOT, it was suggested that you try the two test programs recorded on your PILOT cassette. PLTST is a program that lets you watch PILOT in action. PILOT statements and some PILOT error messages are introduced. You will not be asked to "express yourself in PILOT." The information imparted by the program (mostly by using T statements to type on the screen) is generally an indication of what will be going on next in the PLTST program, itself. (If numeric variables have just been introduced, PLTST is probably about to use a numeric variable.) Notice that the program will sometimes tell you how to form a certain mark of punctuation, e.g., a caret (^). To some extent, such information is keyboard dependent: if you have a Sol, for example, you happen to have a key representing a caret on your keyboard.

In WAPP, or Write A PILOT Program, you will be asked to supply one or more multiple choice questions, a selection of answers--you choose how many--and responses keyed to each answer. Again, you will not have to do any of the formatting and you will not need to use the standard PILOT syntax. WAPP is an example of how one PILOT program can serve as a vehicle for another to be written. You can actually save, and later execute, the multiple-choice program that WAPP writes for you.

Both PLTST and WAPP are written in PILOT, so there is nothing that either of these programs accomplishes that you could not also accomplish with the statements described earlier in this manual. If you cannot figure out how a program does something, either LIST the program or display it in the EDITor, and take a look at the PILOT code.

Once you have successfully loaded one of the test programs, you will find that it is self-explanatory.

SECTION 7

ERROR MESSAGES AND HOW TO DEAL WITH THEM

Most of these error messages have been introduced, either in PLTST or at other points in this manual. Here we present a summary which you might find useful as a reference. In each case, an example of the message, as it might appear to a user, is followed by a brief explanation of its likely implications.

*<label> or PR: or M: -NOT FOUND

The PILOT program in memory specifies a jump to a statement or a subroutine that does not exist. Check for a possible misspellling of the label in the J or U statement. The expression enclosed by the angle brackets is the label as it appears spelled in that statement. If the item not found is a PR or M statement, there is probably no such statement between the J or U statement and the end of the program.

<C statement>
*ILLEGAL EXPRESSION

A compute statement has incorrect syntax. Maybe it specifies an operation that is not allowed, e.g. C: R= A OR B is syntactically incorrect, since "OR" is not a recognized operation.

<C statement>
*VALUE OUT OF RANGE(-32768 TO 32767)

The compute statement in the angle brackets has either an element or a result that is not in the possible range of numbers allowed in PILOT. In such a case, the value assigned to the designated variable is -32768 for a negative number or 32767 for a positive number.

<variable name>

If you refer to a string variable to which no value has been assigned, and your immediate intention is not to accept a value for that variable--say, for example, you wanted to print or write it--that variable name, itself, is displayed. You may want to make such a reference intentionally. If not, check for a spelling error, make sure that the variable you are naming is the one that you really intended to name, or find out if your statement referring to a given variable really follows the assignment of a value to that variable. Did you put a NEW$ statement in your program and then try to retrieve a value that you had erased?

*NUMERIC RESPONSE REQUIRED

The user of a PILOT program receives this message if he gives a non-numeric reply to an A statement that requests a value for a numeric variable. The solution is generally to give a numeric reply.

*USE DEPTH EXCEEDED

The program being executed has more than seven subroutines in execution.

*TAPE READ ERROR

Either the cassette tape has been read badly by the computer or you hit the MODE SELECT key while the tape was being read. If you really want the tape to be read--that is, if you did not hit the MODE SELECT on purpose--try to rewind and play the tape again. If you continue to get the message, there is probably something wrong with your tape or other equipment.

*NO ROOM

There is insufficient space in memory for the string variable that you are trying to store. You may want to use a NEW$ statement to clear the area occupied by variables that you are no longer using.

If you have not received one of the above messages, but something seems to be going wrong, consider the following possibilities:

1) You have used more than one label with the same name; control has been passed to the first such label to occur in the program, rather than to the label to which you intended to jump. Rename one of the labels.
2) You have loaded a program that is too big or a text file that is not executable as a program.
3) You have attempted to enter a response longer than the present value of INMAX will allow. (This problem does not apply to an immediate command. Immediate commands override INMAX.)
4) You have misspelled or mispunctuated something.
5) You have LOADed or READ something into memory and obliterated the former program (and possibly some of your string variables.)
6) You have entered something that is not a command, and it has simply been displayed.
7) You have neglected to mark the end of a subroutine with an End statement.
8) You have relied on a zero value for a numeric variable, but you have not re-initialized numeric variables since the last program was executed.

APPENDIX 1

USING CASSETTES

Successful and reliable results with cassette recorders and cassette files requires a good deal of care. You need to use consistent and careful methods, and you need to know what to expect, when you try to read a manufacturer's tape, or your own. The following methods are recommended:

1) Use only a recorder recommended for digital usage. For use with the Processor Technology Sol or CUTS, the Panasonic RQ-413AS or Realistic CTR-21 is recommended.

2) Keep the recorder at least a foot away from equipment containing power transformers or other equipment which might generate magnetic fields, picked up by the recorder as hum.

3) Keep the tape heads cleaned and demagnetized in accordance with the manufacturer's instructions.

4) Use high quality brand-name tape, preferably low noise, high output tape. Poor tape can give poor results, and rapidly wear down a recorder's tape heads.

5) Bulk erase tapes before reusing. It can be hard to find the file you want in a jumble of old file pieces. Bulk erasing also decreases the noise level of the tape.

6) Keep cassettes in their protective plastic covers, in a cool place, when not in use. Cassettes are vulnerable to dirt, high temperature, liquids, and physical abuse.

7) Experimentally determine the most reliable settings for volume and tone controls, and use these settings only.

8) On some cassette recorders, the microphone can be live while recording through the AUX input. Deactivate the mike in accordance with the manufacturer's instructions. In some cases this can be done by inserting a dummy plug into the microphone jack.

9) If you record more than one file on a side, SAVE an empty file, named "END" for example, after the last file of interest. Once you read its name, you will know not to search beyond it for files you are seeking. To avoid having to search for files, you can record only one file per cassette, at the beginning of the tape, if you can afford the extra cassettes.

10) Do not record on the first or last minute of tape on a side. The tape at the ends gets the most physical abuse. Do not be impatient when trying to read the first file on a tape. You, or the manufacturer of a pre-recorded program, may have recorded a lot of empty tape at the beginning.

11) Record a file more than once before it leaves memory. This redundancy can protect you from bad tape, equipment malfunction, and accidental erasure.

12) Most cassette recorders have a feature that allows you to protect a cassette from accidental erasure. On the edge of the cassette opposite the exposed tape are two small cavities covered by plastic tabs, one at each end of the cassette. If one of the tabs is broken out, then one side of the cassette is protected. An interlock in the recorder will not allow you to depress the record button. A piece of tape over the cavity will remove this protection.

13) Use the tape counter to keep track of the position of files on the cassette. Always rewind the cassette and set the counter to zero when first putting a cassette into the recorder. Time the first 30 seconds and note the reading of the counter. Always begin recording after this count on all cassettes. Record the beginning and ending count of each file for later reference. Before recording a new file after other files, advance a few counts beyond the end of the last file to insure that it will not be written over.

14) The SOLOS/CUTER command CATalog can be used to generate a list of all files on a cassette. Exit to SOLOS/CUTER, type CAT <CR>, rewind to the beginning of the tape, and press PLAY on the recorder. As the header of each file is read, information will be displayed on the screen. If you have recorded the empty file called END, as suggested, you will know when to search no further. If you write down the the catalog information along with the tape counter readings and a brief description of the file, you will be able to locate any file quickly.

15) Before beginning work after any modification to the system, test by SAVEing and GETting a short test program. This could prevent the loss of much work.

In addition to using the above procedures methodically, you need to know the various ways in which programs may be recorded on tapes you have purchased:

1) If you cannot read a file consistently, and suspect the tape itself, do not despair. The same file may have been recorded elsewhere on the tape. Processor Technology often records a second version, later on the same side of the tape. When you first get a tape, CATalog it with SOLOS or CUTER so you will know exactly what it contains. Write down the tape counter readings at the same time.

2) An empty file named END is sometimes placed at the end of the recorded portion of a tape, or between other files. When SOLOS/CUTER CATalogs a file, the file header information is displayed as soon as the beginning part of the file passes the tape head, but nothing is displayed when the end of the program passes by. If another filename such as END is displayed, you know you have just passed the end of the previous file.

3) Some of the programs supplied by Processor Technology contain a checksum test. When a program containing this test is first executed after loading from tape, the checksum test reads all of the program in memory, and calculates a checksum number which is compared with a correct value. If the numbers match, this assures that the program in memory is correct before it is used. Nothing is displayed when the numbers match, but if they do not, the message CHECKSUM TEST FAILED, or a similar message, is displayed. The message may be followed by two numbers, representing the correct and incorrect checksum numbers.

Even though the checksum test failed, it may be possible to enter the program anyway, often by typing the carriage return key. The bad data may not even be apparent, if it is in a portion of the program you do not use. It is best, however, to try to find and correct the problem causing the error so the checksum test is passed. The error can be caused by the cassette inteface circuitry, bad memory locations, bad tape, a faulty recording, improper adjustment or settings on the cassette recorder, or other equipment problems.

APPENDIX 2

WRITE IN PILOT, READ IN BASIC

Here is a sample PILOT program that you can type into the EDITor. The program writes the student's name and responses into a file called WHO? on tape unit 2. The responses can then be read and printed by the BASIC program which follows. When you enter the program, you might want to try inserting DELete characters (video reverse) to emphasize titles.

When you write a data collection file, it is a good idea to keep in mind that different users might be answering questions in a different order, skipping questions, answering questions different numbers of times, etc. It is therefore necessary to arrange your file so that it is possible to tell which answer was given to which question, or by whom the answer was given. In the very crude example supplied below, the student's name is recorded before any other information is requested. You will also see that each guess is recorded along with the number of guesses that have been made, and the number of clues that have been given. Note that the words "guess number," "after ... clues," and "final score" are not really necessary; it is clear that whenever a line consists of two numbers followed by a name, the first number is the guess number, and the second number is the number of clues given. When a line has only one number in it, that number always represents the student's final score.

This example is therefore not intended to be a model, nor is the BASIC program which follows the only way in which you can read or display a data collection file, even in BASIC. BASIC permits you to isolate the numbers that appear in a character string, so that you can use them in arithmetic. As you continue to experiment with PILOT, and with PILOT in relation to your own system, you will certainly devise your own methods of setting up and dealing with data collection files.

Notice that the last statement in the PILOT program directs PILOT to load another program segment from tape unit 1. If you do not have another program segment in mind, you can change the last statement to BYE. There is a reason that you should NOT change the last statement to E. Can you tell what the reason is? If you can't, try changing the statement to E and executing the program.

```
T:Put tape unit 2 into "record" mode. Then press "RETURN."
A:
CH:
OPENF:WHO?/2
CA:7
T:What is your name?
A:
R:Record the player's name.
WR:
T:I am thinking of a famous person, and I want you to try to
:guess who the person is.  There are some clues to help you.
:After each clue you will be able to choose whether to guess or
:to accept another clue.  A correct answer is worth 100 points,
:minus 10 points for each additional clue, after the first one,
:and 20 points for each incorrect guess.
:Obviously, negative scores are possible.
FOOT:
CH:
T:If you want to know your score at any point during the game,
:you can request it by typing
:
:            \U:*SCORE
:
:(Of course, you can do this only if it is your turn to talk to
:PILOT.  Type the request, then a carriage return, and then the
:answer to the original question.)
FOOT:
CH:
T:The first clue is:  The person was a famous writer.
:Do you want to try to guess before seeing another clue?
:(Answer Y or N)
A:
M:Y
UY:*GUESS
R:Numeric variable C contains the number of extra clues given.
C:C=C+1
T:This writer lived in England in the 18th century.
:Do you want to guess? (Y or N)
A:
M:Y
UY:*GUESS
C:C=C+1
T:This writer wrote A Modest Proposal.
:Do you want to guess? (Y or N)
A:
M:Y
UY:*GUESS
C:C=C+1
T:This writer wrote Gulliver's Travels:Do you want to guess? (Y or N)
A:
M:Y
UY:*GUESS
T:There are no more clues.
PA:5
```

```
J:*OUT
*GUESS T:What is your guess?
A:$WHO
R:Numeric variable G contains the number of guesses made.
C:G=G+1
R:Record the guess number, the number of clues given, and the
R:guess made.  Report the same information to the screen.
RW:Guess number #G, after #C clue(s) is $WHO.
T:Guess number #G, after #C clue(s) is $WHO.
PA:5
M:Swift
JY:*OUT
T:$WHO is not the correct answer.
U:*SCORE
E:
*SCORE
R:Numeric variable S contains the point score.
C:S=-10*C-20*G
R:Numeric variable L contains the number of points lost.
C:L= -S
T:You have lost #L points.
E:
*OUT CH:
T:Swift is the correct answer.
CY:G=G-1
U:*SCORE
CY:S=S+100
R:Record the final score.  Report the final score to the player.
RW:Final score is #S.
T:  Your final score is #S.
:
:
:
:                          END OF GAME
PA:5
R:LOAD a new program segment, called NEW, from tape unit 1.
LOAD NEW:
```

This is the BASIC program that will read the file WHO? from tape unit 2, and display the contents of that file on the video screen.

```
10    DIM  A$(100)
20    FILE #1;"WHO?/2",1
30    READ #1;A$ : GOTO 60
40    PRINT A$
50    GOTO 30
60    PRINT "ALL DONE"
70    END
```

APPENDIX 3

REFERENCES

The following references were not used in the preparation of this manual. They are included here in case you are interested in reading about other versions and implementations of PILOT.

Black, Don. The computer in the schoolroom. Conference Proceedings, Second West Coast Computer Faire. Box 1579, Palo Alto, CA 94302, 232-238.

Starkweather, John A. Guide to 8080 PILOT, Version 1.1. Dr. Dobb's Journal of Computer Calisthenics & Orthodontia. April, 1977, 17-29.

Yob, Gregory. PILOT-73. Creative Computing. May/June 1977.

Processor Technology

Processor Technology Corporation

7100 Johnson Industrial Drive
Pleasanton, CA 94566

(415) 829-2600
Cable Address - PROCTEC

Cassette PILOT Update 731Ø68

Please incorporate the following note near the top of page 2-2 and at the bottom of page 6-1 of your PILOT manual:

Before trying to LOAD PLTST or WAPP, type this command from PILOT restart:

SET M=2Ø48Ø <CR>

(If you have already read about the SET command in Section 4.5 of your manual, you know that when you give this form of the command, you make room in memory for a larger program than would otherwise fit in the program buffer.)

No ECN