Programming in BASIC

Wang Laboratories, Inc.

Programming in BASIC

PREFACE

This is a beginner's introduction to programming in BASIC on Wang 2200 computer systems. Starting at the most elementary level, it introduces the reader to the fundamentals of BASIC, and the mechanics of creating programs on a Wang 2200 system. It covers all the concepts and statements needed for competent fundamental programming in BASIC, in commercial and non-commercial environments. It presumes no prior knowledge of BASIC or of programming in general.

The reader is urged to try out new statements and programming concepts as they are introduced. As many as possible of the example programs should be keyed in and executed. One should experiment with the example programs by making changes to them, predicting the effects of the changes, and then confirming or correcting one's knowledge based on the observed effects. As soon as possible, one should begin to write programs that solve simplified problems within one's particular area of interest. Only in this manner can the concepts and capabilities introduced here become practical knowledge.

There is nothing that can be done from the keyboard that can damage a Wang 2200 system. Cautions are included in the text when statements that might destroy data files are introduced. Thus, the reader can feel free to experiment at every stage of learning.

In general, the reader should follow the sequence of presentation in the text; however, there are some suggested alternatives. Chapters 4 and 5 discuss saving programs and the elementary use of a printer, respectively. Since they deal with these relatively more "mechanical" subjects, they have been written so that they may be read at any time between Chapters 2 and 9. By deferring them until the reader wishes to save a program or use a printer, or until Chapter 9, the continuity of presentation of BASIC may be better preserved.

Since this is intended to be an introduction to BASIC for those who will program commercial applications as well as for those who will program non-commerical or "technical" applications, example programs are drawn from both areas. Sections 6-3, 6-4, 15-6, and 19-3 may be omitted by the commercial applications programmer, if desired.

Though this volume is specifically designed for the person who wishes to learn to program in BASIC on Wang 2200 systems, it can also serve as a general introduction to programming in BASIC on any system. However, the reader must be aware that the BASIC language, like most programming languages, has many forms. When going from one implementation of BASIC to another, the user must first become familiar with the idiosyncrasies of the new version. Nevertheless, there is a common core that is a part of Wang BASIC and most other versions of the language. If the reader wishes to read this volume focusing on this common core, the following sequence is suggested:

        Sections 2-1, 2-2
        Chapters 3, 6, 7, 8, 10, 11, 12, 13
        Sections 14-1, 14-2
        Chapters 15, 16, 19

At the end of each of the chapters in Part I of this volume, there is a review of the main points of the chapter. The reader who is already familiar with BASIC, or with another programming language, may wish to skim through Part I reading the review sections first, then reading the text as needed.

Only the most commonly used peripheral devices are discussed in this volume. Specifically, these include:

Keyboard
CRT Display
Printers
Cassette Tape Drives
Disk and Diskette Drives

TABLE OF CONTENTS

TABLE OF EXAMPLES

PART I

THE
FUNDAMENTALS
OF
BASIC

CHAPTER 1
INTRODUCTION TO THE EQUIPMENT
IN YOUR WANG SYSTEM

1-1   THE PRINCIPAL COMPONENTS OF A WANG 2200 SYSTEM

At the heart of a Wang 2200 system is a Central Processing Unit, a keyboard, and a CRT display. Your system may include any of the several different models of each of these items. It may also include such additional items as printers, tape cassette drives, disk or diskette drives, plotters, card readers and a wide variety of other devices. The purpose of this chapter is to introduce you to the functions of a few of these items before we begin to discuss programming.

The Central Processing Unit

The Central Processing Unit (CPU) is at once the very heart of any Wang system and its least conspicuous item. Figure 1.1 shows a 2200T CPU. During program execution, the Central Processing Unit gets program instructions from its memory, interprets and executes them, or if they involve other devices, initiates their execution. It controls the entire system. It contains the system memory, where programs and variables are kept during program execution.
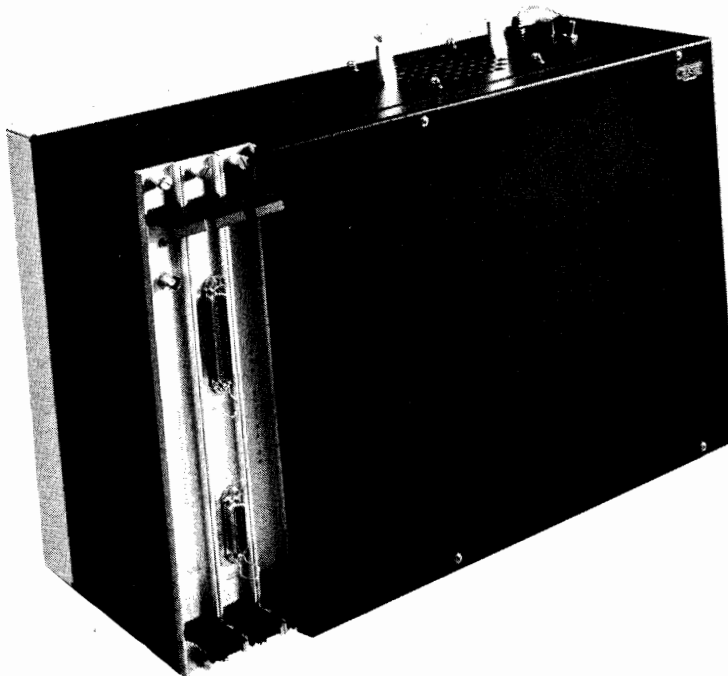
Figure 1.1   A 2200T Central Processing Unit

By itself, the Central Processing Unit has no means of receiving information or of communicating results. It is designed to be able to use a wide variety of other devices for these input and output operations. Two such devices, which are a part of almost every Wang system, are a keyboard and a CRT display. These two items form the heart of the system from the operator's point of view.

Keyboards

Though there are several different keyboard models, they all have the same essential functions. They allow you to enter commands, program lines, and data to the CPU. The Model 2223 keyboard is shown in Figure 1.2. You should consult your Reference Manual or Introductory Manual for information about your specific keyboard. However, a few notes which pertain to programming are in order here.
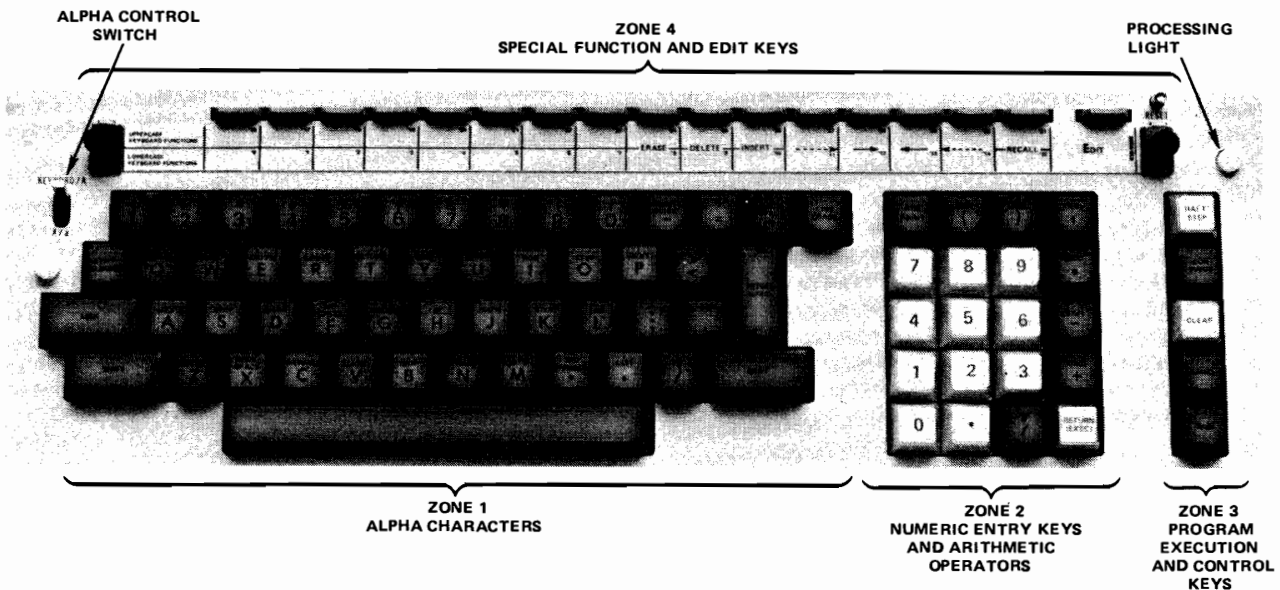
Figure 1.2 The Model 2223 Keyboard

On all keyboards there is at least one key labeled

    RETURN
    (EXEC)

or

    EXECUTE
    (CR/LF)

In many operations this key has special significance. For example, when keying in a program line, this key is used to tell the system that the line is complete. In this manual the symbol (EXEC) is used to refer to this key. (If your keyboard has two or more keys labeled RETURN(EXEC), the keys do the exact same thing; they appear in several locations just for your convenience.)

3

CRT Displays

The CRT is the television screen device that sits directly behind the keyboard, or is incorporated with the keyboard into a single console unit. Figure 1.3 shows the Model 2216 CRT display. (CRT stands for cathode ray tube, the technical name for the television screen tube.) The CRT provides a very fast and easy means for the Central Processing Unit to get information to you. The standard 2200 CRT can display 16 lines of information, each up to 64 characters wide.
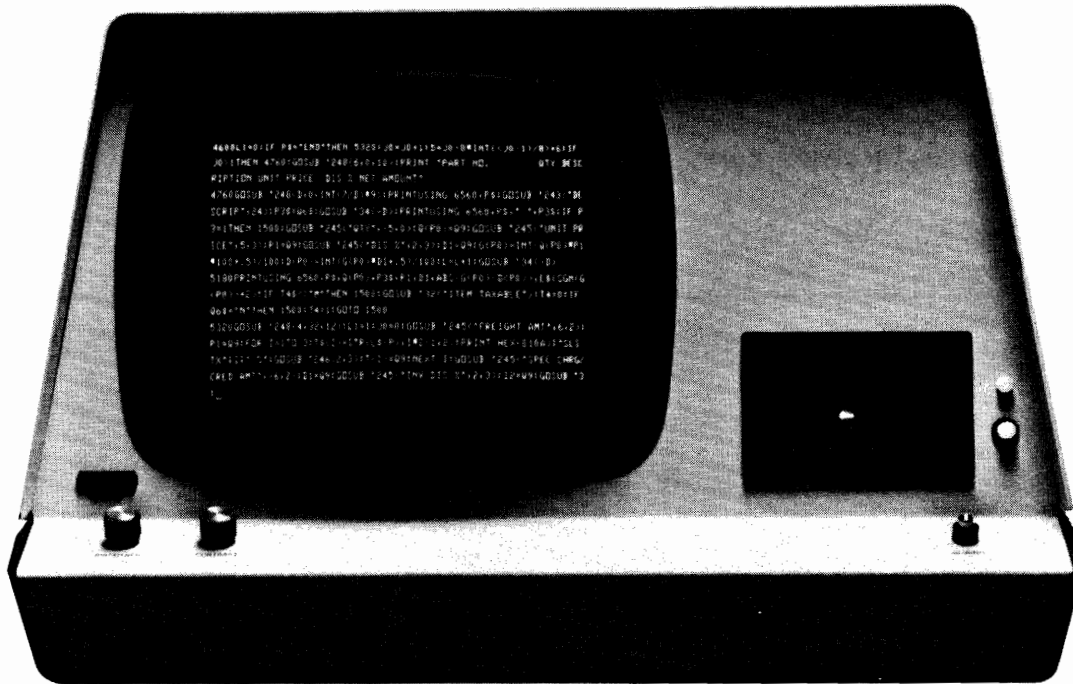
Figure 1.3 The Model 2216/17 Combined CRT Display Cassette Drive

Whether your system is equipped with a separate keyboard and CRT or with a one piece console, the keyboard and CRT work together as if they are in direct communication with each other. Strictly speaking, this is an illusion. There is no direct electronic link between the keyboard and the CRT. The keyboard and CRT, even if they are physically a single console, act as two separate units attached to the Central Processor. It is the control function of the Central Processing Unit which is responsible for the illusion of direct connection.

Other Devices

Most Wang systems include other devices besides the CPU, keyboard and CRT. For example, nearly all of them include some fast and easy means of saving programs and data outside of the CPU memory. Usually tape cassette drives or magnetic disk drives perform this function. These devices offer two key features: they save programs and data in a form that allows very rapid loading into memory, when compared to the alternative of entering it from the keyboard, and, unlike memory, they preserve all data and programs saved on them, even after the system power is turned off.

4

Another device very commonly included in a Wang system is a printer. Printed output from the CPU is often called "hardcopy" output, as distinguished from output that appears on the CRT.

1-2 TURNING ON YOUR WANG SYSTEM

Each of the devices discussed in the last section has its own power switch, except the keyboard. However, many 2200 systems, and all the WCS systems, have a central power switch that activates one or more devices. For the location of these switches, see your Introductory Manual.

When the power is turned on for the CPU, the system very quickly goes through a procedure known as Master Initialization. Master Initialization performs certain internal operations to prepare the processor, and then displays on the CRT the symbol

    READY
    :_

The contents of memory are lost when the CPU power is turned off.

CHAPTER 2
GETTING STARTED

2-1   PROGRAMS AND YOUR WANG SYSTEM

A program is a set of step-by-step instructions which, when carried out, accomplishes a specific task. Though programs are often associated with digital computers and similar devices, the concept of a program does not apply only to computers. When you give a guest directions for driving to your home, explain how to make an omelet, or lead a child through long division, you are providing a kind of program. The logical process you go through in developing these kinds of instructions is no different than that used in developing step-by-step instructions for a computer.

Similarly, there is nothing about a computer program which requires that it be carried out by a computer. Given enough paper, pencil, and time, any computer program can be carried out by someone who can read the instructions of the program. This is not to say that it is efficient to carry out computer programs this way, just that the instructions given in a computer program are equally effective outside of an electronic device.

This means that the careful step-by-step approach you use in everyday problem solving also applies to solving problems with a computer. However, the unique characteristics of a computer do have a bearing on the way you give it instructions, and on the most practical and efficient means of getting the job done.

One of the characteristics of a computer is that it does not inherently understand any human language. Nevertheless, a computer can be made to act on instructions given in a language that is well suited to the tasks the computer can accomplish and to the various needs of its human programmers. BASIC is such a language. As the language of your Wang system, BASIC allows you to create instructions using familiar terms of English and elementary algebra.

Another important characteristic of a computer is that it can carry out instructions extremely quickly. This fact, together with the observation that many tasks involve repeating similar operations, has led to one of the now essential concepts of a computer program: that it consists of a set of instructions provided to the computer in advance of actual execution. By first giving a computer all the instructions needed to accomplish a specific task, and only then telling it to start executing the instructions, the computer can work at its own tremendous speeds, repeating particular operations as many times as necessary to get the job done.

When you give your Wang system a program, the program goes into the system memory located in the Central Processing Unit. Normally just one program is in memory at a time. Old programs, those which are not about to be executed, are cleared out of memory before new programs are loaded in. Thus, memory is not a storage bin, but, rather, a kind of staging area.

There are several different ways of putting a program into memory. When a program is being entered into memory for the very first time, it usually must be entered via the keyboard. (New programs can also be entered via mark sense, or punched, cards, though these devices are beyond the scope of this manual.) Once you have keyed in your first long program, though, you will be very glad that it does not have to be keyed in each time it is to be run.

After a new program has been keyed in, it can be preserved on any of several different kinds of magnetic recording devices. The two principal such devices are tape cassettes and disks. Once saved on one of these devices, a program is always available for quick and easy loading into memory.

In Section 2-2 we introduce two simple computer tasks and the programs that accomplish them. The remaining sections of this chapter will show you how these programs can be keyed into memory. In Chapter 3 we take up the question, "Why do these programs work?"

2-2 TWO SIMPLE PROBLEMS AND THEIR SOLUTIONS IN BASIC

Suppose you have a very simple inventory problem. You have one product, coal, which you buy from a mining company and sell to end users. You always want to know how many tons are on-hand at your yard. You want to be able to post changes to the inventory when coal is received or sold, and you want to be reminded to reorder coal if your inventory drops below 100 tons. When you start using your system, your inventory stands at 42,500 tons.

Your inventory system should continuously display on the CRT an up-to-the-minute inventory report. The report should look like this:

TONS ON HAND = 25055
NUMBER OF TONS RECEIVED (+) OR SOLD (-)?

In this report 25055 is the number of tons of coal currently on-hand. The question mark at the end of the second line indicates that the system is awaiting an entry by the operator. The operator will enter a sale as a negative value and a receipt as a positive value.

Figure 2.1 shows the output from the series of inventory transactions that may have preceded this report. Notice that at the next-to-last report the inventory was below 100 tons, and, therefore, the reorder message appeared. In the last report, the inventory is above 100 tons; therefore, the reorder message does not appear. Notice that a blank line appears between each report.

7

```
OPENING INVENTORY = 42500
NUMBER OF TONS RECEIVED (+) OR SOLD (-)?-6000

TONS ON HAND = 36500
NUMBER OF TONS RECEIVED (+) OR SOLD (-)? 10000

TONS ON HAND = 46500
NUMBER OF TONS RECEIVED (+) OR SOLD (-)? -46370

TONS ON HAND = 130
NUMBER OF TONS RECEIVED (+) OR SOLD (-)? -75

TONS ON HAND = 55
REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS
NUMBER OF TONS RECEIVED (+) OR SOLD (-)? 25000

TONS ON HAND = 25055
NUMBER OF TONS RECEIVED (+) OR SOLD (-)?
```

Figure 2.1   Reports on a Series of Inventory Transactions

A BASIC program that produces this inventory system is shown in Example 2.1.

Example 2.1 A Simple Inventory Program

```
10   LET  I=42500
20   PRINT  "OPENING INVENTORY="; I
30   INPUT  "NUMBER OF TONS RECEIVED (+) OR SOLD (-)", T
40   LET  I=I+T
50   PRINT
60   PRINT  "TONS ON HAND ="; I
70   IF   I >=100   THEN   30
80   PRINT "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS"
90   GOTO 30
```

Before we analyze this inventory program, let's briefly look at another type of problem and a BASIC language program which helps solve it. Then, with two simple programs in hand, we can see what they have in common.

Suppose you would like to know the values of all the factorials of P, from P=0 to some upper limit which you can specify at the time of calculation. You take the factorial function to be defined for positive integer values of P, that is, 0,1,2,3, ... N. For any positive integer P, P factorial, written P!, is equal to the product of all the integers up to and including P. For example,

```
if  P=4 then P! = 1X2X3X4 or 24
if  P=7 then P! = 1X2X3X4X5X6X7 or 5040
```

For P=0, P! is defined as 1.

The program should ask you to enter the highest value of P for which P! is to be calculated. It could ask you this by displaying:

COMPUTE P! FOR P = 1 TO P = ?

If you enter 12, for example, the result should look like this:

```
COMPUTE P!  FOR P=0 TO P=? 12
P= 0              P!= 1
P= 1              P!= 1
P= 2              P!= 2
P= 3              P!= 6
P= 4              P!= 24
P= 5              P!= 120
P= 6              P!= 720
P= 7              P!= 5040
P= 8              P!= 40320
P= 9              P!= 362880
P= 10             P!= 3628800
P= 11             P!= 39916800
P= 12             P!= 479001600
***** DONE *****
```

A BASIC program that produces this output is shown in Example 2.2.

Example 2.2 Computing a Table of Factorials

```
10 INPUT "COMPUTE P! FOR P=0 TO P=",N
20 LET F = 1
30 PRINT "P="; P, "P!="; F
40 LET P = P+1
50 LET F = P*F
60 IF P < = N THEN 30
70 PRINT "***** DONE *****"
```

Look over the two sample programs. You will notice that their lines are numbered, and that the lines begin with words which are English verbs or closely resemble them. These words are some of the keywords of the BASIC language. (If you have a BASIC keyword keyboard, you will find that these words appear on your keys.) Following the keywords you find, among other things, expressions and letter variables that look like simple algebra. You also find phrases in quotation marks. One of the virtues of BASIC is that its most fundamental and frequently used instructions are composed of familiar English words and simple expressions which closely resemble the actions they cause.

Though not apparent from mere inspection of the programs, spaces appearing within BASIC lines are totally ignored by the system, unless they appear within quotation marks. They are put into a program wherever they will make it easier to read.

With the clue provided below and a little detective work comparing the program and the displayed result, you may be able to get an idea of how these BASIC programs work. The clue is: the system always executes the instructions in the sequence of the numbers at the left (the line numbers)

9

unless an instruction tells it to go to some other line and begin executing there. Don't try to figure out every little detail, and don't worry if the programs seem a complete enigma. You should have plenty of questions about them going into Chapter 3, where we will discuss each statement in detail, and consider how the programs as a whole are made up.

Now, though, we must concentrate on how to get these programs into the memory of your Wang system, via the keyboard. Only when they are in the memory of your system can they be executed.

2-3   HOW TO KEY A PROGRAM INTO MEMORY

Clearing Memory

When you turn on your Wang system, its memory is completely clear of all BASIC statements and variables. The display

```
READY
:_
```

appears in the upper left corner of the CRT. The system is now ready to accept a program.

If you haven't just turned on your Wang system, you should always clear the memory before entering a new program. To do this, first depress the RESET key. Depressing RESET stops the execution of any program, clears the screen, and displays

```
READY
:_
```

It does not clear memory of any BASIC statements or variables, though. To clear memory, simply type the capital letters CLEAR. Each letter appears on the screen as you enter it. If you make a mistake, depress the BACKSPACE key, and re-key the correct character. Notice that as each key is depressed, the underscore mark, _ , called the "cursor", moves to the right to show the position the next character will occupy. (If your system is equipped with a BASIC keyword keyboard, the word CLEAR can be displayed with a single keystroke.)

After keying CLEAR, the display should look like this

```
READY
:CLEAR_
```

Now depress (EXEC). The system memory and the CRT display are cleared, and the READY display reappears. This indicates that execution of the CLEAR command has terminated, and the system is once again ready for a keyboard entry. Now you can begin to enter a program.

Entering Statement Lines

Look at Example 2.1, the inventory program. The eight lines that comprise the program are called statement lines, and the numbers which appear at the left of them are called statement line numbers, or simply line numbers. The words and symbols which appear to the right of the line numbers make up the statement itself. Statements in BASIC, like sentences in English, are the fundamental unit of instruction.

When you enter a program into the memory of your Wang system, you enter it one statement line at a time. You begin each statement line with a line number; then key the statement itself. Characters appear on the CRT as you key them. After checking the line for accuracy, you enter it into memory by depressing (EXEC). A colon and cursor then appear on the next line of the CRT, signifying that another statement line can be entered.

On the BASIC keyword keyboard a special key labeled STMT. NO. can be used to generate line numbers. Whenever the display indicates with a :_ that a new line can be entered, depressing STMT. NO. causes a number to appear which is 10 greater than the highest line number already in memory. Alternatively, on any keyboard, line numbers can be created by simply keying whatever number is desired. You will learn more about line numbers and their significance in Chapter 3.

Now, try entering the simple inventory program which appears in Example 2.1 and is reproduced below for your convenience.

Example 2.1   A Simple Inventory Program

```
10   LET  I=42500
20   PRINT  "OPENING INVENTORY=";  I
30   INPUT  "NUMBER OF TONS RECEIVED (+) OR SOLD (-)", T
40   LET I=I+T
50   PRINT
60   PRINT  "TONS ON HAND =";  I
70   IF   I >= 100  THEN  30
80   PRINT  "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS"
90   GOTO 30
```

NOTE:

From this point on we will not distinguish between entry operations using BASIC keyword keyboards and character only keyboards, except when some special circumstance warrants it. If you have a BASIC keyword keyboard, you should be aware that a single keystroke can be used to enter a keyword. Since on any keyboard, a keyword can always be entered by keying it character by character, the text is worded to reflect this type of entry.

To enter the first line of the inventory example, first be sure that the colon and cursor are in the display (:_). Now key

```
10 LET I = 42500 (EXEC)
```

11

The display appears as

        :10 LET I = 42500
        :_

indicating that you can now enter another line.

        When you enter a line, you can put spaces wherever you wish, and as long as they are not added within quotation marks, they have no effect on the execution of the program. If you had entered

        10LETI=42500

the program would work just the way the program in Example 2.1 works. In this text we use spaces liberally in the examples, and recommend that you do the same in your programs.

        If you notice an error in a statement line before depressing (EXEC), correct it by depressing the backspace key until the erroneous character is removed. Then, key the correct characters to finish the line. If you wish to avoid reentering the correct characters that follow the erroneous one, you can correct the line by using the EDIT mode, discussed below.

        If you make a mistake, but don't notice it   until after you depress (EXEC), there   are several easy ways to correct it. To see what some of them are, enter the next line of the inventory program leaving out the   left quotation mark, like this:

        20 PRINT OPENING INVENTORY="; I (EXEC)

        When (EXEC) is depressed, an error message appears which looks like this:

        :20 PRINT OPENING INVENTORY="; I
                ↑ERR 10
        :_

        The error message points to the location at which the system detected a violation of BASIC language syntax. It is a warning that this statement, as it stands, doesn't make sense to your Wang system, and   therefore cannot be executed. A table of error messages and their meanings appears in Appendix A. However, since we simply want to show how to correct the problem, we'll leave a detailed discussion of error reporting for later.

        Line 20 may now be corrected in either of two ways.   It must be corrected because in its present form it will prevent program execution. One way to correct it is simply to reenter the correct statement, in its entirety. Whenever a statement line is entered with the same line number as a statement line already in memory, the new line completely replaces the old. For short program lines, replacement is an easy means of correction. However, here, as is frequently the case, just one character needs to be inserted to correct the line.   An easier means of correction is to use the EDIT mode to recall the line and make the correction to the single incorrect character.

12

2-4   USING THE EDIT MODE
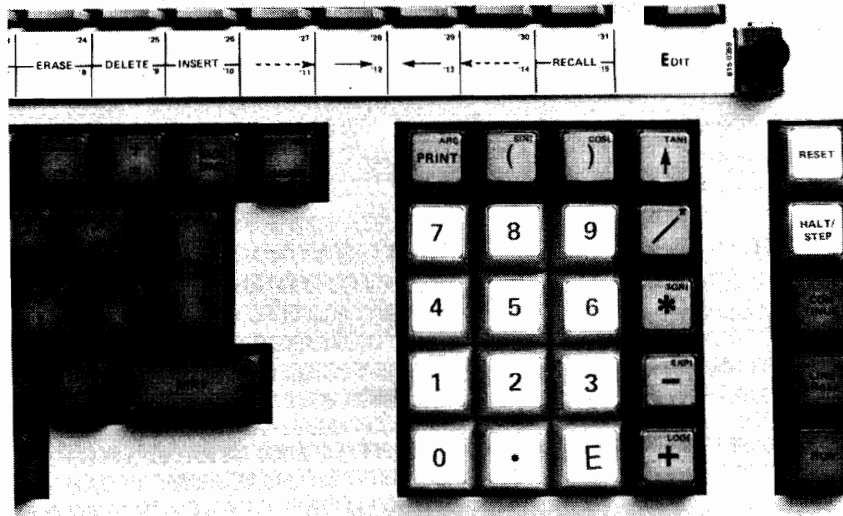
Figure 2.2   The EDIT Special Function Keys

To enter the EDIT mode, depress the EDIT key which lies immediately to the right of the Special Function keys at the top of all Wang keyboards. (See Figure 2.2.)   Notice that after EDIT is depressed, the colon on the CRT is replaced by an asterisk. This symbol indicates that the system is in the EDIT mode.

Continuing with the example from the last section, since the line to be edited is in memory, it must be recalled. Key 20, the number of the line to be   edited,   and   depress   the   RECALL   key located in the EDIT Special Function strip. The line now appears on the CRT.

        *20 PRINT OPENING INVENTORY=";   I_

The cursor appears to the right of the recalled line.

To move the cursor to the position of the missing quotation mark, depress the ◄---- key.   Notice that the cursor moves five positions to the left, and that the characters remain unaffected. The ◄---- and ----► keys cause the cursor to move five character spaces in the indicated direction. The ← and → keys cause the cursor to move one character position.

Depress the cursor movement keys until the cursor is positioned under the 0 of OPENING.   This is the position the " should occupy. Depress the INSERT key. The INSERT key puts a space into the cursor position, and moves all the characters to the right one position to accommodate it. Now, key the quotation mark from the keyboard, just as you normally would.   When in the EDIT mode, the keyboard keys function normally, that is, they cause the entered character to appear in the line, at the cursor position.   If the cursor marks an occupied character position, the old character is replaced by the new. In this example, the quotation mark replaces a space entered by the INSERT key.

13

The corrected line 20 can now be entered into memory by depressing (EXEC). The new line 20 replaces the old one, and with the depression of (EXEC), the system leaves the EDIT mode. The :_ appears on the next CRT line, and program lines can be entered normally.

Briefly, let's look now at the other two EDIT capabilities. The DELETE key does the exact opposite of INSERT. It removes the character at the cursor position and adjusts all the characters left, so that no space appears. For example, given this erroneous line

        *20 PRINT "OPENNING INVENTORY="; I

depressing DELETE produces

        *20 PRINT "OPENING INVENTORY="; I

The ERASE key eliminates all characters from the cursor position to the end of the line. Given

        *20 PRINT "OPENING INVENTORY="; I

depressing ERASE produces

        *20 PRINT "OPEN_

There is one more feature of Wang systems which should be mentioned in connection with EDIT; for it is during editing that it is most apparent. Key EDIT 20 RECALL to bring line 20 back up for editing

        *20 PRINT "OPENING INVENTORY="; I_

First of all, notice that it is correct; the edited line replaced the erroneous line first entered. But now, move the cursor carefully over to the O of OPENING. Slowly depress the ← key, which causes movement one character at a time. When the cursor reaches this position,

        *20 PRINT "OPENING INVENTORY="; I

the next depression of ← causes it to jump under the P of PRINT. This is true regardless of whether PRINT was last entered with a single keystroke on a BASIC keyword keyboard or by keying the individual characters P-R-I-N-T. The reason for this is that your Wang system automatically reduces all BASIC keywords to a special code that uses only as much space as a single character in memory. When the system encounters one of these codes, it automatically displays the entire keyword, though it keeps the code in memory. This system greatly reduces the memory space required for your programs.

With the cursor under the P of PRINT, depress DELETE. The entire word PRINT, and a single space following it, are deleted. This is the result of removing the single special code for the keyword PRINT. The keyword is interpreted as including a space at the end.

14

However, you certainly don't want to eliminate the keyword PRINT from the program. From this position

*20 _ "OPENING INVENTORY="; I

there are several ways of getting the line back the way you had it.

You can use the INSERT key. Depress INSERT once. PRINT is one of the few keywords which can be entered via a single keystroke on any model keyboard. Whenever a keyword can be entered via a single keystroke, a single space is adequate to edit it in. From

*20 _"OPENING INVENTORY="; I

depress the PRINT key. The result is

*20 PRINT_ "OPENING INVENTORY="; I

Alternatively, INSERT can be depressed five times and the letters P-R-I-N-T keyed in character by character.

The other way to get the line back the way it was is to simply use RECALL. Remember that, during editing, memory is not changed until (EXEC) is depressed. This replaces the old line with the new. Therefore, as long as the line number has been left intact and (EXEC) has not been depressed, the line, as it is in memory, can be recalled by simply depressing RECALL. In our example, from

*20 _"OPENING INVENTORY="; I

simply depress RECALL to get

*20 PRINT "OPENING INVENTORY="; I

An exit from the EDIT mode by depressing (EXEC) will now leave the line unaltered.

There is one other type of editing which we have not yet covered. This is the editing of an erroneous line number.

Suppose you had accidentally entered

30 PRINT "OPENING INVENTORY="; I

You want the line number to be 20, not 30. Depress EDIT 30 RECALL to bring up the erroneous line for editing. Move the cursor to the left until it is under the 3 of 30. Key 2, which replaces the 3, and then (EXEC) to enter the new line 20 into memory. The thing that you must remember about editing line numbers is that the old line, line 30 in this case, is still in memory, now accompanied by line 20, which is otherwise identical to it.

15

In this particular case, if you actually made this error, fixed it in this manner, and then went on to key in the correct line 30, no harm would be done. The correct line 30 would wipe out this erroneous line 30 on entry. However, whenever you want to delete a line from a program, you can simply enter the line number and follow it immediately with (EXEC). This deletes the line.

Now, go ahead and enter the rest of the inventory program exactly as it appears in Example 2.1. In the next section you will see how to list the entire program on the CRT. If, while entering the program, you would like to obtain a listing of it, read the next section.

2-5   LISTING A PROGRAM

Frequently, you will want to inspect a program or program segment. A listing of all the program lines in memory can be obtained by keying LIST (EXEC), whenever the :_ display is present. If you wish to clear the screen before listing, depress RESET. LIST does not alter the contents of memory.

For programs larger than 15 lines (the capacity of the standard display), you will want the list to appear in segments. Keying LIST S (EXEC) displays the first 15 lines of program text. Keying (EXEC) again displays the next 15 lines. The process may be repeated until the entire program has been listed.

LIST always displays lines in line number sequence, regardless of the sequence in which the lines were actually entered. You will frequently add a new line to a program with a line number that falls between two lines already in the program. You can then list the program and see the lines in their correct locations.

LIST, LIST S, and CLEAR are called commands because as soon as (EXEC) is depressed, they immediately initiate the action they describe. As commands, they are not entered into memory and cannot be preceded by a line number.

By contrast, a BASIC statement can be preceded by a line number. It can be entered into memory to form part of a program, and is not executed until the program itself is executed. Each line in Examples 2.1 and 2.2 contains a statement.

2-6   EXECUTING A PROGRAM

By now you should have completely keyed in and checked the inventory program shown in Example 2.1. To run the program, first key RESET to clear the screen. This is not strictly necessary, but makes it easier to see the results of this program.

Depress RUN (EXEC) to begin program execution. The display should look like this:

```
READY
:RUN
OPENING INVENTORY = 42500
NUMBER OF TONS RECEIVED (+) OR SOLD (-)?
```

The question mark indicates that the system is awaiting an operator entry. Key 5000 (EXEC) to represent the receipt of 5000 tons of coal. The system displays a blank line followed by:

```
TONS ON HAND = 47500
NUMBER OF TONS RECEIVED (+) OR SOLD (-)?
```

Go on making entries and observe execution of the program. What happens when the screen becomes filled? What happens when TONS ON HAND drops below 100 or below 0?

Depress RESET to stop program execution. To re-run the program, key RUN (EXEC) again. Again, the display shows:

```
OPENING INVENTORY = 42500
NUMBER OF TONS RECEIVED (+) OR SOLD (-)?
```

In this program the message

```
OPENING INVENTORY = 42500
```

appears only once each time the program is executed.

The RUN Command

RUN is used to begin program execution. When you key RUN (EXEC), the system:

1. Scans the entire program for variables, and sets aside space in memory for them. (The inventory program uses two variables, I and T.)

2. Sets the value of all variables to zero.

3. Checks for certain types of errors in the program.

4. If none of these errors are found, begins executing the program at the lowest line number.

Normally, RUN (EXEC) is used to begin program execution; however, it is possible to specify, in the RUN command, the line number at which execution is to begin. For example, RUN 30 (EXEC) causes execution of the inventory program to begin with

```
30 INPUT "NUMBER OF TONS RECEIVED (+) OR SOLD (-)",  T
```

In addition to beginning execution at the specified line, if a line number is specified in the RUN command, variables are not set to zero before execution begins. If the lowest line number of a program is specified in the RUN command, the effect is the same as a simple RUN command, except that the variable values are unaltered.

17

Since there are two forms of the RUN command, one with and one without a line number, we say that the general form of the RUN command is

RUN   [line number]

where the square brackets indicate that the line number is optional.

2-7   CHAPTER REVIEW AND EXERCISES

Chapter Review

1.   Before keying in a new program, memory should be cleared.

2.   When you turn on your Wang system, memory is clear.

3.   Memory can be cleared at any time with the CLEAR command.   Key CLEAR (EXEC) to clear memory.

4.   After keying in a complete program line, key (EXEC) to enter it into memory.

5.   EDIT mode activates the EDIT Special Function keys.   Depress   the   EDIT key to enter EDIT mode.

6.   In EDIT mode, a line can be recalled from memory for editing by depressing the RECALL key in the EDIT Special Function strip.

7.   In EDIT mode, the arrow keys (----→ ,→ , ←, ←----) move the cursor in the indicated direction.

8.   In EDIT mode, the INSERT and DELETE keys insert a space or delete the character at the cursor location.

9.   In EDIT mode, the ERASE key erases all characters right of the current cursor position.

10.   To list a program, key LIST (EXEC).   To list a program in 15-line segments, key LIST S (EXEC).

11.   To execute the program that is in memory, key RUN (EXEC).

Exercises

1.   Using EDIT mode and the inventory program, put extra spaces into some lines; take them out of others.   Verify that this does not affect program execution.

2.   Using EDIT, take out the spaces within the quotation marks of statement 30. What happens to the display during execution?

3.   Delete line 50 from the inventory program, and run the program.   What
      has changed?

4.   Key in and execute the factorial program (Example 2.2). Re-execute the
      program by keying RUN (EXEC) again; use different values for P.

5.   Execute the factorial program entering a value greater than 15.   What
      happens to the way P! is displayed at P=16? How many digits are in P!
      when P=15?

6.   Execute the factorial program entering a value greater than 69. Observe
      the result.

CHAPTER 3
FUNDAMENTAL INSTRUCTIONS

3-1   HOW THE EXAMPLE PROGRAMS WORK

In this section we will reconsider the example programs first introduced in Section 2-2. We will concentrate on what each instruction contributes to making the program run the way it does. The later sections of this chapter take up each instruction as a topic in itself, to see how it can be used in any program.

The question that we first want to take up is, "How do you go from the problem to be solved to the solution in the form of the program?" There is no simple, correct answer to this question. Nevertheless, there is a technique that is frequently used to help further define the task to be accomplished, and to put it into the step-by-step form that is required for a computer program. This technique is called flowcharting.   A flowchart shows the operations that must be carried out to solve a problem, and displays the sequence of these operations by means of connecting arrows.   A flowchart of the inventory problem is shown in Figure 3.1. Appendix B shows standard flowchart shapes and symbols.

Look over the flowchart shown in Figure 3.1. Carefully compare it with the original problem description of Section 2-2. The flowchart of Figure 3.1 can be readily converted to the BASIC program shown in Example 2.1.   Figure 3.2 compares the flowchart with the program by showing the program statements in matching flowchart boxes, opposite the flowchart itself.

```
                    ┌─────────────┐
                   (    BEGIN      )
                    └──────┬──────┘
                           ▼
                    ┌─────────────┐
                    │¹            │
                    │ SET OPENING │
                    │ INVENTORY   │
                    │EQUAL TO 42500│
                    └──────┬──────┘
                           ▼
                    ┌─────────────┐
                    │²  DISPLAY   │
                    │  OPENING    │
                    │  INVENTORY  │
                    └──────┬──────┘
                           ▼
                    ┌─────────────┐
                    │³  REQUEST   │
                    │  ENTRY OF   │
                    │ TRANSACTION │
                    └──────┬──────┘
                           ▼
                    ┌─────────────┐
                    │⁴  OPERATOR  │
                    │   ENTERS    │
                    │ TRANSACTION │
                    └──────┬──────┘
                           ▼
                    ┌─────────────┐
                    │ ADD TRANS-  │
                    │ACTION AMOUNT│
                    │TO INVENTORY │
                    │TO GET NEW   │
                    │ INVENTORY   │
                    └──────┬──────┘
                           ▼
                    ┌─────────────┐
                    │⁵  SKIP A    │
                    │   LINE      │
                    └──────┬──────┘
                           ▼
                    ┌─────────────┐
                    │⁶DISPLAY NEW │
                    │  INVENTORY  │
                    │   AMOUNT    │
                    └──────┬──────┘
                           ▼
                         ◇
                    IS INVENTORY
                    ⁷ GREATER THAN 100 ──── YES ────►
                         ◇
                           │
                          NO
                           ▼
                    ┌─────────────┐
                    │⁸  DISPLAY   │
                    │  REMINDER   │
                    │   MESSAGE   │
                    └──────┬──────┘
                           ▼
```
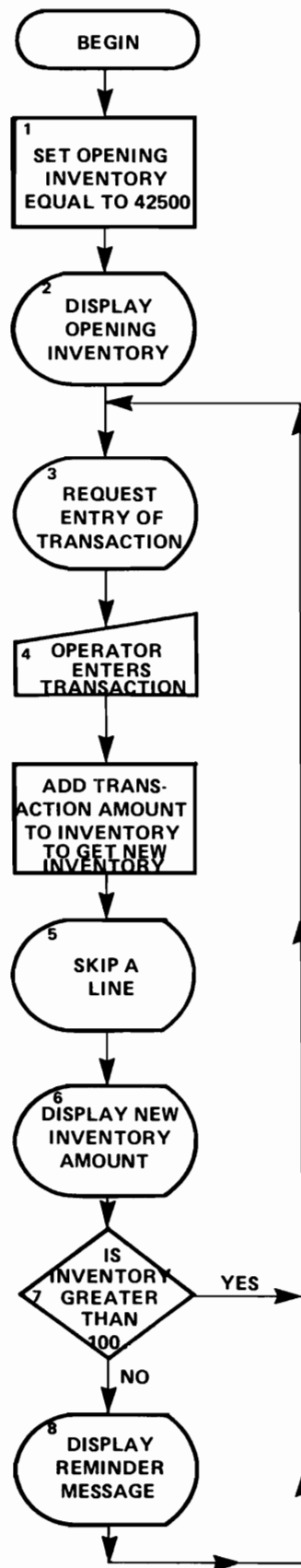
Figure 3.1   Flowchart of the Inventory Problem

21
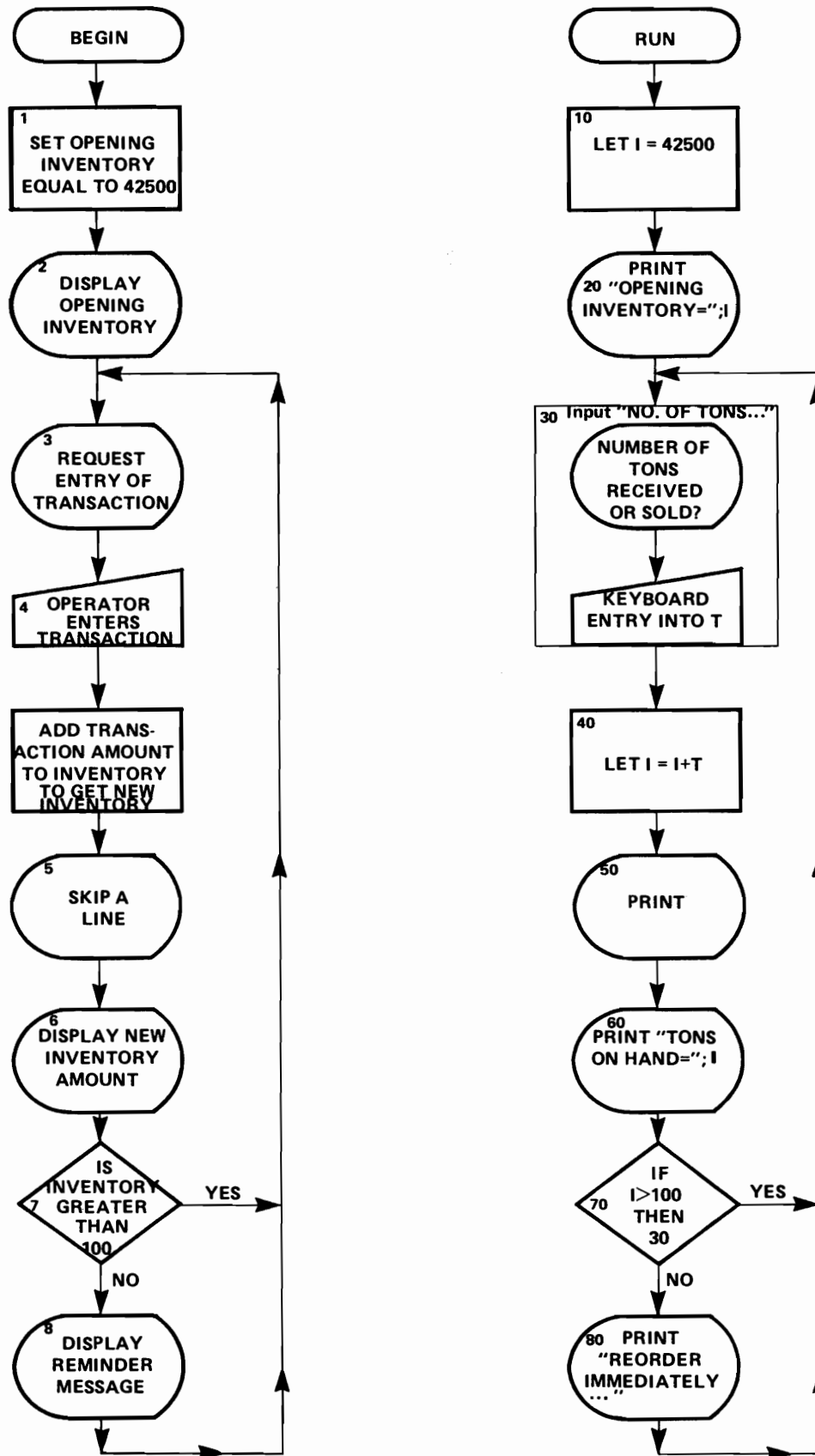
Figure 3.2   Comparison of the Program and the Problem Flowchart

22

Block 1 of Figure 3.1 says that the inventory should be set to 42500 at the beginning. We know that the purpose of this program is to keep track of the changing number of tons in inventory (the changes are effected in block 4); therefore we need a place to keep the inventory quantity. A place to keep a value is called a "variable". Since the value we are concerned with is a number, we will use a numeric variable. In BASIC, numeric variables are named by a letter A, B, C . . .Z, or a letter and a digit together A0, A1, A2 ...B0, B1, B2 ...Z7, Z8, Z9. We use the variable I for the inventory quantity.

Block 1 of Figure 3.1 is accomplished by statement 10 of the program, shown in Figure 3.2. Statement 10

```
10 LET I=42500
```

assigns the valve 42500 to the variable I.

Block 2 of Figure 3.1 says that the opening inventory amount should be displayed. It should be labeled so that the operator knows what the number represents. Block 2 is accomplished by statement 20 of the program (see Figure 3.2).

Statement 20

```
20 PRINT "OPENING INVENTORY="; I
```

displays the characters within the quotation marks, followed by the value of the variable I. We know that statement 10 made I equal to 42500; therefore, statement 20 causes the following display when the program is executed:

```
OPENING INVENTORY= 42500
```

Notice that the quotation marks are not displayed during execution. They are used in the program to indicate that the enclosed characters are to be treated collectively as a unit. A set of characters enclosed by quotation marks is known as an "alphanumeric literal string".

In Figure 3.1 there are two blocks with the number 3. This is to reflect the fact that both operations are accomplished by the single BASIC statement at line 30. Blocks 3 call for display of a message requesting input of the transaction amount and a keyboard entry of the amount.

Statement 30

```
30 INPUT "NUMBER OF TONS RECEIVED (+) OR SOLD (-)", T
```

displays the input message followed by a question mark. The question mark is the BASIC symbol which is displayed whenever the system is awaiting input from the keyboard. The system does not go past line 30 until an acceptable entry has been made from the keyboard.

When a value is to be entered with an INPUT instruction, the instruction must specify a variable to receive the entered value. The entered value in this case is received by the variable T. Whatever value T may have had, before statement 30, is replaced by the new value entered from the keyboard. The old value is lost.

Block 4 stipulates that the transaction amount is to be added to the current inventory amount. The result is the new inventory amount. We know that the current inventory amount is in I, and the transaction amount is now in T. The new inventory is, therefore, equal to I+T. In BASIC, the form I+T is an example of an "expression". The LET statement, which we encountered in Step 10, can be used to assign the value of the expression I+T to a variable. The question now is what variable should receive this new value.

At first you might say the value of I+T should be assigned to a new variable. The variable might be I2, and the statement to accomplish the assignment might be

        40 LET I2 = I+T

Taken by itself, there is nothing wrong with this statement; it will execute successfully. However, at this point, when writing this program, you must look ahead to see how the program as a whole is to function. From that new perspective the form of statement 40, shown above, is defective.

To see this, consider that the next time the inventory assumes a new value we would need a new statement such as

        LET I3 = I2+T

which adds the transaction amount to the current inventory, I2, and assigns it to a new variable I3. For each time inventory assumes a new value, we would have to add to the program new input, assignment, and display statements. To effect 200 inventory transactions would, with this approach, require a program with 1202 statements, using 201 variables. 300 transactions could not possibly be effected; there would not be enough variable names.

The solution to this problem is fundamental to all programming. We must construct this part of the program in such a way that we can reexecute the same instructions whenever the operation must be performed again. This solution is known as a loop, and is made possible by letting new values replace old ones.

We said that LET assigns to a numeric variable the value of an expression. To this we must now add the comment that the receiving variable can itself appear in the expression. Statement 40 (Figure 3.2) illustrates this.

        40 LET I = I+T

In executing line 40, the system first evaluates the expression to the right of the equals sign; then it assigns the result to the variable on the left. In this case, the value of I is added to the value of T, and the result is placed in I, thereby replacing its old value with this new result. After execution, the old value of I is gone. I contains the new value and T is unchanged.

By writing statement 40 in this way, the operation of updating the inventory balance, which in this case comprises the entire program, can be repeated with the same statements, that is, the operation can be performed within a loop.

Block 5 calls for the display of a blank line. This is simply to make the displayed "reports" easier to read. Statement 50

```
50 PRINT
```

accomplishes this. It is a print statement with nothing to print. The result is a blank line.

Block 6 calls for the display of the new on-hand inventory amount. Again, it should be labeled for identification. Statement 60 accomplishes this with a PRINT statement that is similar to statement 20.

```
60 PRINT "TONS ON HAND="; I
```

Block 7 stipulates that the value of the on-hand inventory is to be examined. If it is greater than or equal to 100 tons, then the input message for the next transaction can be immediately displayed. However, if the inventory is less than 100 tons, then, before the transaction input message is displayed, the operator must be told to reorder coal.

Statement 70 does exactly what block 7 calls for, and in doing so, introduces us to another fundamental concept of programming. We said earlier that the system executes instructions in the sequence of the line numbers. We have not shown how this sequence of execution can be changed by the program itself. Statement 70

```
70 IF I >= 100 THEN 30
```

checks the condition expressed between the keywords IF ... THEN. In expressing the condition the symbol $>=$ means "is greater than or equal to". If the condition is true with the current values of the variables, statement 70 alters the normal sequence of execution by telling the system that the next instruction to be executed is at line 30. If the condition is false, in this case if I is less than 100, then the system proceeds to the next instruction in line number sequence.

In this case, if the condition is false, the system executes statement 80, which displays the reorder message. If the condition is true, i.e., the inventory is greater than or equal to 100 tons, then statement 30 is executed next. Once the transfer to line 30 is effected, the normal sequence of execution, line number to next higher line number, takes over.

The IF...THEN statement is known as a conditional branch statement, since it causes the system to branch out of the normal sequence of execution if a stated condition is true. If the condition is false, execution continues with the next statement in line number sequence, as though the IF...THEN statement had not been present in the program.

The IF...THEN statement never changes the values of the variables it tests, regardless of the outcome of the test itself. The only thing it can do is change the sequence of execution of the program statements.

Block 8, which is executed only if the inventory drops below 100 tons, calls for the display of a reminder message, telling the operator that coal should be reordered immediately. The PRINT statement at line 80 of the program causes the message to be displayed.

25

Notice that block 8 is the last block of the flow chart, and statement 80 is the last statement shown in Figure 3.2. However, if statement 80 were the last statement in the program (Example 2.1), then after the reorder message was displayed the program would simply stop. The system, finding no higher line number, would cease execution and display the :_ symbol.

The return to statement 30, which is implied in the flowchart by the returning arrows coming out of block 8, must be made explicit in the program. After statement 80, the system must be told to make 30 the next instruction to be executed. Statement 90

        90 GOTO 30

does this. It tells the system that the next statement to be executed is at line 30.

The Factorial Example

Now that we have seen how the first example works, let's look at the second. The flowchart for the factorial problem and program is shown in Figure 3.3.

```
        BEGIN                              RUN

         │                                  │
    ┌────┴────┐                        ┌────┴──────┐
    │1        │                    ┌ ─10─ ─ ─ ─ ─ ─│─ ┐
    │ DISPLAY │                    │   INPUT "COM- │
    │  INPUT  │                    │   PUTE P! FOR P=O │
    │ MESSAGE │                    │   TO P=",N    │
    └────┬────┘                    │   └────┬──────┘   │
         │                         │        │
    ┌────┴────────┐                │   ┌────┴─────┐    │
    │1     LET    │                │   │          │    │
    │ OPER. ENTER │                │   │          │    │
    │ MAX.FROM KBD│                │   └──────────┘    │
    └────┬────────┘                └ ─ ─ ─│─ ─ ─ ─ ─ ─ ┘
         │                                │
    ┌────┴────┐                      ┌────┴────┐
    │2        │                      │20       │
    │ FOR P=0 │                      │         │
    │ LET P!=1│                      │ LET F=1 │
    └────┬────┘                      └────┬────┘
         │                                │
    ┌────┴────┐                      ┌────┴──────┐
    │3        │                      │30         │
    │ DISPLAY │                      │ PRINT "P=";│
    │ P AND P!│                      │  P, "P!=";F│
    │WITH LABELS│                    └────┬──────┘
    └────┬────┘                           │
    ┌────┴────┐                      ┌────┴──────┐
    │4        │                      │40         │
    │INCREMENT│                      │           │
    │    P    │                      │ LET P = P+1│
    │  BY ONE │                      └────┬──────┘
    └────┬────┘                           │
    ┌────┴────┐                      ┌────┴──────┐
    │5        │                      │50         │
    │MULTIPLY (P-1)!│                │           │
    │   BY P  │                      │ LET F= P∗F│
    │ TO GET P!│                     └────┬──────┘
    └────┬────┘                           │
        ◇                                ◇
      IS P                              IF
    LESS OR   YES                     P<=N   YES
  6 EQUAL TO ──→                   60 THEN  ──→
      MAX.                              30
        │ NO                            │ NO
    ┌───┴────┐                     ┌────┴──────┐
    │7       │                     │70         │
    │INDICATE│                     │  PRINT    │
    │THAT JOB IS│                  │"∗∗∗∗DONE∗∗∗∗"│
    │COMPLETE│                     └────┬──────┘
    └───┬────┘                          │
        │                          ┌────┴──────┐
    ┌───┴────┐                     │    :_     │
    │  END   │                     └───────────┘
    └────────┘
```
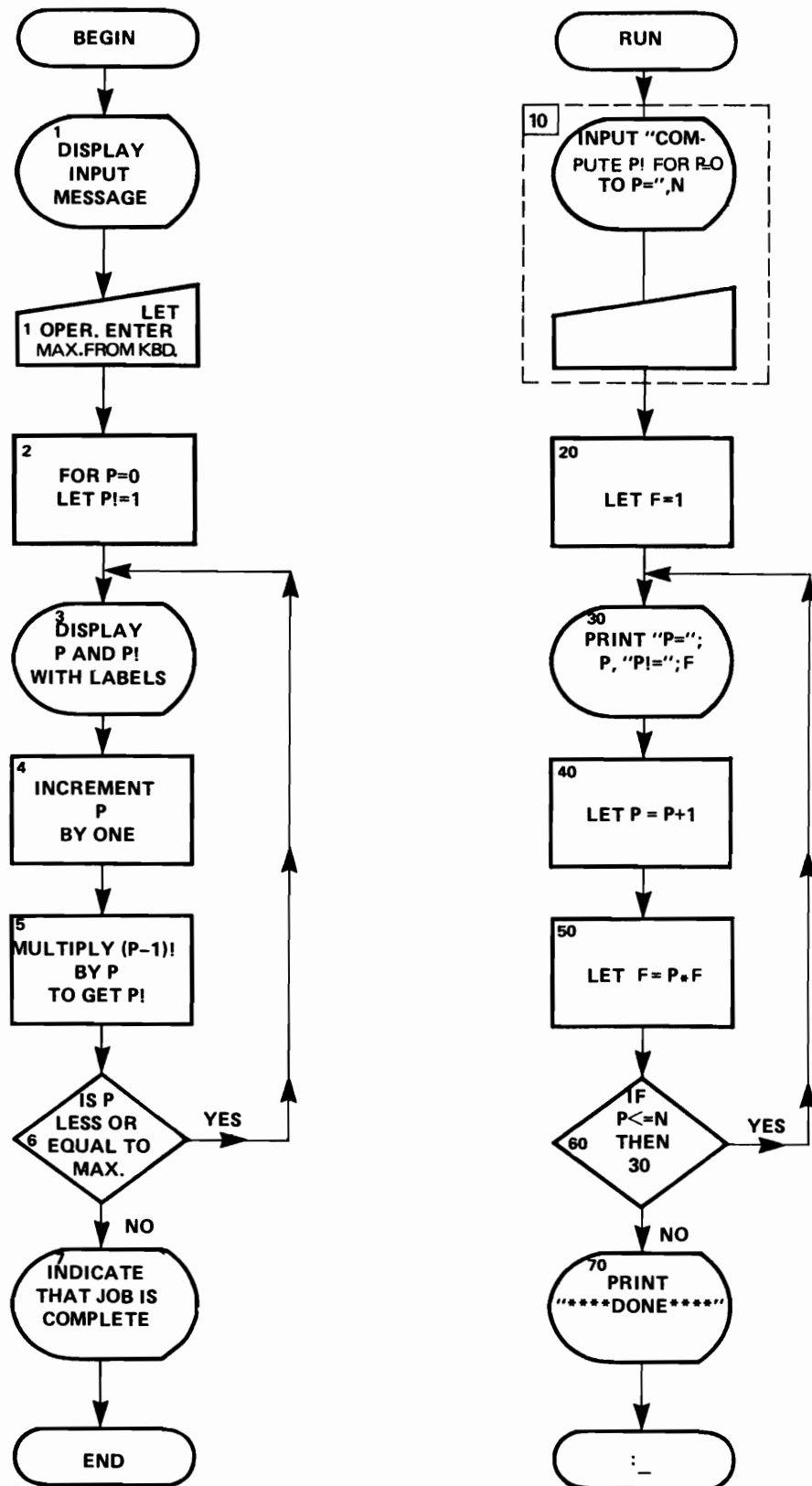
Figure 3.3 Flowchart of Factorial Problem and Program

The two blocks numbered 1 in Figure 3.3 are accomplished by the single INPUT statement at line 10 of the program. The operator enters the maximum value for which P! is to be calculated. The INPUT statement specifies that the variable N is to receive the entered value.

Block 2 notes that for the first value of P, zero, the factorial of P is defined as 1. Since this is true by definition rather than by calculation, the program must simply assign these initial values for P and P!. The form "P!" is not a legal BASIC variable, so the variable F has been chosen to contain the value of the factorial of P. Statement 20 assigns 1 to F.

Statement 20 illustrates one of the essential stages for a loop. This is the initialization stage, in which the variables affected by the loop are set to beginning values which permit the loop operations to be performed successfully.

Block 3 calls for printing the value of P and P! with labels. This block begins the program's processing loop. Statement 30 prints the values of P and F, each preceded by an appropriate label. When we look at the PRINT statement in more detail, we will see how the commas and semicolons in statement 30 produce the desired spacing of the output. For now, though, we know that the values of P and F are correct for the first time through the loop. We know this because the statement which is printed the first time through, P=0 P!=1, is correct.

Block 4 says to add one to P. Statement 40 does this. It evaluates the expression P+1 and assigns the result to P. On the first time through, statement 40 assigns 1 to P.

Now that P has received a new value, the program must calculate a new value for P!. This new value of P! can be calculated by simply multiplying the old value of P! by the new value of P. The old value of P! is in F, and the new value of P is in P, so the expression needed is P*F. (In BASIC the asterisk is the symbol for multiplication.) We want to assign this value to F. To do this the LET statement, shown as statement 50, is used:

50 LET F = F * P

Before the new values of P and F are printed, we want to check to see if P is still within the limit entered at statement 10. Block 6 calls for this check; it is accomplished by statement 60.

60 IF P < = N THEN 30

Remember that N contains the maximum value for which P! is to be calculated. Statement 60 means, "If the value of P is less than or equal to the value of N, then 30 is the next statement line to be executed." If the relationship expressed within the IF...THEN is not true, then the normal sequence of execution prevails, and line 70 is executed next.

Until P is greater than N, statement 60 causes the program to loop through statements 30, 40, and 50. Each time the loop is executed, line 30 prints the values of P and F that were calculated and assigned the last time through. Then statements 40 and 50 produce the next values of P and F. If the value of P hasn't exceeded the limit, the loop is repeated.

28

Since there are no statement lines beyond statement 70, execution ends, automatically, after the "DONE" message is displayed. The :_ symbol reappears in the display, indicating that the system is ready to accept another program or command.

In the remaining sections of this chapter we will look individually at each instruction we have encountered in these two programs.

3-2 THE LET STATEMENT AND NUMERIC EXPRESSIONS

The LET Statement

The LET statement assigns the value of an expression to one or more numeric variables. (It can also be used with alphanumeric variables. This use is discussed in Chapter 8.)

In the example programs, we saw the LET statement used in one of its principal forms: assigning the value of a numeric expression to a single numeric variable. For example,

          10 LET I = 42500

and

          50 LET F = P*F

As a result of each of these statements, the value of the expression to the right of the equals sign is assigned to the variable on the left. The old value of the variable on the left, whatever it may have been, is lost. Variables on the right of the equals sign remain unaltered, unless they also appear on the left.

Though the example programs did not show a LET statement with several receiving variables, any number of variables may appear on the left to receive the value of the expression. Variables must be separated by commas. Enter and execute this program:

          10 LET A, B, B3 = 5
          20 PRINT "A="; A
          30 PRINT "B="; B
          40 PRINT "B3=";B3

Upon execution the system displays

          A= 5
          B= 5
          B3= 5

indicating that the value of the expression to the right of the equals sign in statement 10 was assigned to each of the variables A, B, and B3.

Statement 10 could be written as three separate LET statements:

```
10 LET A=5
20 LET B=5
30 LET B3=5
```

with the exact same effect.

The multi-assignment form of the LET statement is frequently used in what can be called "set-up operations" within programs. For example, it may be necessary to set several variables to 1 or some other constant before entering a loop or routine. This can be done with a single LET statement. Subtotals can be set to zero when necessary with a statement such as:

```
    .
    .
    .
550 LET S1, S2, S3, S4, S5 = 0
    .
    .
    .
```

"LET" Optional

In the "LET" statement the word "LET" is optional. If it is omitted, the statement functions in the exact same way as if it had been present. In the example programs the statements

```
10 LET I = 42500
40 LET I = I+T
```

could have been written

```
10 I=42500
40 I=I+T
```

If you feel that the word "LET" contributes to the intelligibility of the program by highlighting the receiving side of the statement, then feel free to use it. Most experienced BASIC language programmers omit "LET" because it occupies space in memory without having an intrinsic purpose. Since it is seldom used in practice, we do not use it in forthcoming examples.

The LET statement is normally referred to as the assignment statement of the BASIC language since its purpose is to assign a value to variables. The general form of the assignment statement for numeric variables is:

[LET] numeric variable [,numeric variable...] = expression

The square brackets indicate that the enclosed items are optional. The general form of this statement will be expanded in Chapter 8 to accommodate alphanumeric variables.

Now let's look more closely at numeric quantities and numeric expressions.

Numeric Quantities

The inventory and factorial problems we looked at were both concerned with numeric quantities: the number of tons of coal, the integers and their factorials. Numeric quantities appeared as constants in such statements as

        10 LET I = 42500
    and
        50 LET F = 1

Numeric quantities were entered from the keyboard, as in

        30 INPUT "NUMBER OF TONS RECEIVED (+) OR SOLD (-)", T

In every case numeric quantities were stored, and updated, in numeric variables. Since numeric quantities are so important to many programming tasks, we want to look at the range of legal numeric quantities in the Wang 2200 system.

The Wang 2200 can operate on numeric quantities as large as $10^{100}-1$ and as close to zero as 10^{-99}. The quantities may be positive or negative. Quantities within this range can be represented by a maximum of 13 digits, a decimal point and sign, and a two-digit positive or negative integer exponent.

The following are legal numeric quantities in your Wang system, and may be entered as shown. The letter E in a quantity means "times ten raised to the power of".

EXAMPLE	QUANTITY	NOTE:
1)	244	Implied positive value with implied decimal point at right.
2)	+244.	Same quantity as (1) but with explicit sign and decimal.
3)	-4567.12	Explicit sign and decimal.
4)	-4.56712E3	Same quantity as (3), in scientific notation.
5)	-1234567.891234	13 digits, decimal point and sign.
6)	-1234.567891234E3	Same quantity as (5), represented with exponent.
7)	-1.234567891234E6	Same quantity as (5), in scientific notation.
8)	4.5E-12	Explicit sign in exponent.

31

The following are illegal numeric quantities:

EXAMPLE	QUANTITY	NOTE:
1)	103.2E99	Illegal because the value represented is not less than $10^{100}-1$.
2)	.87E-99	Illegal because the value represented is less than 10^{-99}.

The forms of the following quantities are illegal in your Wang 2200 System.

EXAMPLE	QUANTITY	NOTE:
1)	8.7E5.8	Exponent is not an integer.
2)	-1.2E.5	Exponent not an integer.
3)	123456789123.45	Too many digits (14).

It should be noted that the PRINT statement does not necessarily print quantities in the form in which they are entered. This is discussed in more detail in the next section.

Expressions

An expression is a constant or numeric variable, or a series of constants or numeric variables, connected by arithmetic symbols. Functions can also be included within an expression; they will be discussed in Chapter 6.

In the example programs we encountered expressions such as those underlined below.

```
10 I = 42500
50 F = P*F
60 P = P+I
```

Constants within Expressions

The expression in statement 10 of Example 2.1 is a constant. A constant is a valid numeric quantity, represented by digits (and the symbols +-.E), that appears in a BASIC statement without quotation marks. As its name implies, it cannot be changed by statements in the program. Since it is a kind of expression, it can be used whenever an expression can be used. Constants are not the only numbers which appear in programs, but the others, line numbers and such, can easily be identified by context. Examples of constants are shown underlined below.

```
10 K = 1.2E6
10 PRINT 165
10 M = 1234567891234
10 Z9 = -7.66E-28
```

32

Note, however, that "165" as in

        10 PRINT "165"

is neither a constant nor an expression. It is a literal string. Since a literal string cannot be assigned to a numeric variable,

        K = "165"

is not a valid BASIC statement.

Numeric Variables

        A numeric variable in BASIC is designated by a letter of the alphabet A, B, C ...Z or a letter followed by a single digit A0, A1, A2 ... A9, B0, B1 ... B9, C0, C1 ... Z7, Z8, Z9. The letters in variable designations must be uppercase. Numeric variables are also called "scalar" numeric variables. Variables such as A and A0 are distinct. There are 286 numeric variables available for use in BASIC.

        A variable can assume the value of any legal numeric quantity. The value of a variable remains unchanged during program execution, unless it is assigned a new value by a statement in the program. The assignment statement, LET, is one of several statements which can assign values to variables.

Arithmetic Operations

The arithmetic symbols, or operators, of BASIC are:

        +   addition
        -    subtraction or negation
        *   multiplication
        /   division (5/4 reads "5 divided by 4")
        ↑   exponentiation (5↑4 reads "5 raised to the 4th power")

        In an expression any number of variables and constants can be linked together by arithmetic symbols. Some simple expressions, using arithmetic symbols, are underlined below:

        10 F = F*P
        10 Z = 22/7
        10 A3 = A3*11000*Z9*J
        10 G2 = 63/22.5E8
        10 D9 = D8-6
        10 C = 3↑D2

Order of Evaluation

        Expressions are evaluated left to right. For example, in the expression A*B/C*D the product of A and B is divided by C; then this quotient is multiplied by D. However for expressions with mixed arithmetic operators, the following priorities of evaluation are observed:

        First, all exponentiation (↑) is performed (left to right).
        Second, all multiplication and division is performed (left to right).
        Third, all addition, subtraction and negation is performed (left to right).

For example, enter and execute the following program:

```
10 W=4
20 X,Y,Z=3
30 K=W*X↑Y-Z
40 PRINT K
```

Statement 30 first raises X to the power Y. The result, 27, is multiplied by W to yield 108. Finally Z is subtracted from 108 to produce the value 105, which is then assigned to K.

This normal order of evaluation can be altered through the use of parentheses. If parentheses are included in an expression, the portion of the expression within the parentheses is evaluated first.

Change line 30 in the above example to

30 K = (W*X)↑Y-Z

When this program is executed, the result of line 30 is:

First W*X is evaluated, since this portion of the expression is enclosed in parentheses. The result, 12, is raised to the power Y.   From the result of this operation Z is subtracted, yielding 1725 to be assigned to K.

In constructing expressions, parentheses may be nested within parentheses, with no limit to the number of pairs of parentheses used. The innermost parenthetical expressions are evaluated first.

In addition to altering the sequence of evaluation, parentheses have two other closely related uses. They can be used to make the normal sequence of evaluation clearer to someone looking at the program listing, even if they do not alter the sequence that would otherwise take place.   For example, our original line 30 could have been written

30 K = (W*(X↑Y))-Z

This form does not alter the sequence of execution, but it does make it clearer.

In BASIC two arithmetic operators cannot appear next to each other. Parentheses are used to set off terms, so that this rule is not violated. For example,

70 K=7↑-J

violates BASIC language syntax and must be written

70 K=7↑(-J)

Not all combinations of constants and variables connected by arithmetic operators are valid expressions. In order for an expression to be valid, it must be capable of being evaluated in the stipulated sequence.   This means

that at each stage of evaluation the operation to be performed must be defined for the given values, and must yield a valid numeric quantity. For example, the following expressions are invalid for the reason shown.

Expression	Invalid Because
1. (3.4E26↑4)/9.7E17	(3.4E26↑4) yields an invalid numeric quantity$\geq 10^{100}$.
2. 17/((A*B)-20) when A=4 and B=5	After evaluating ((A*B)-20), the system attempts to divide 17 by 0, an undefined operation.
3. (-3)↑3.5	The exponentiation (↑) operation is undefined for non-real results.

Expression 2 is invalid for some, but not all, values of the variables. When an expression of this type appears in a program, it is your responsibility, as the programmer, to ensure that the values assigned to the variables at the time of evaluation yield a valid expression. If evaluated, all of the above examples cause a math error, ERR 03, and interrupt program execution.

If the result of evaluating an expression, or a portion of an expression, yields a quantity Q, in the range

$$-10^{-99} < Q < 10^{-99}$$

the value of Q is zero.

Review of Section 3-2

1. A LET statement causes the system to evaluate the expression to the right of the equals sign, and assign the result to the variable, or variables, on the left.

2. Multiple receiving variables are spearated by commas, e.g.,

    LET A, B, B3 = 5

3. The multiassignment form of the LET statement is often useful in setup or clearing operations within programs, e.g.,

    S1, S2, S3, S4, S5, = 0

4. In the LET statement the word LET is optional, e.g., F=F*P is the same as LET F=F*P.

5. The general form of the assignment statement is:

    [LET] numeric variable [,numeric variable...] = expression

6. Numeric quantities can contain up to 13 digits, a decimal point and sign, and a two digit positive or negative integer exponent, e.g., -7.66E-28.

7. An expression is a constant, or numeric variable, or a combination of constants or numeric variables connected by arithmetic symbols.

8. A constant is a numeric quantity, represented by digits (and the +, -, ., E symbols), that appears in a BASIC statement without quotation marks.

9. A numeric variable is designated by a letter of the alphabet A, B, C...Z or a letter followed by a single digit A0, A1, A2...Z7, Z8, Z9.

10. The arithmetic operations, and their order of evaluation, are given by:

ORDER OF EVALUATION

Operation	Symbol	Order of Evaluation (Priority)
Expressions within Parentheses	( )	Computed 1st
Exponentiation	$\uparrow$	Computed 2nd
Division Multiplication	$\left\{ \begin{array}{c} / \\ * \end{array} \right\}$	Computed 3rd
Subtraction Addition	$\left\{ \begin{array}{c} - \\ + \end{array} \right\}$	Computed 4th

Using the above priorities, all expressions are evaluated left to right.

11. Two arithmetic operators (+, -, *, /,↑) cannot appear next to each other, e.g., use K↑(-J) not K↑-J.

3-3 THE PRINT STATEMENT

Print Elements

In the example programs we saw the PRINT statement used several times, and with slightly differing effects. For example,

80 PRINT "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS"

This PRINT statement specifies just one item to be printed. The item to be printed in this case is enclosed by quotation marks, and is therefore known as

36

an alphanumeric literal string, or literal string, for short. Execution of this statement printed on the display

        REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS

with REORDER beginning at the left-most character position on the line.

    Statement 20

        20 PRINT "OPENING INVENTORY="; I

specified two items to be printed, and the items are different kinds of items. "OPENING INVENTORY=" is a literal string; I is a numeric variable. When statement 20 was executed, the result was

        OPENING INVENTORY= 42500

Statement 20 printed the literal string followed by the value of the variable I. Observe the significance of quotation marks by entering and executing this program

        10 I = 42500
        20 PRINT "OPENING INVENTORY="; "I"

The result of executing this new line 20 is

        OPENING INVENTORY=I

Statement 10 still sets the numeric variable I equal to 42500, but, in the new statement 20, "I" is a literal string of exactly one character, not the numeric variable I.

    The items which are printed in PRINT statements such as

        "OPENING INVENTORY="
        "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS"
        I

are known as the print elements. Any number of print elements can appear in a single PRINT statement, but they must be separated from one another by a comma or semicolon. The significance of the comma or semicolon is discussed later in this section.

    The PRINT statement at line 30 of the factorial example (Example 2.2) has 4 print elements.

        30 PRINT "P="; P, "P!="; F

"P=" and "P!=" are literal strings; P and F are numeric variables.

Literal Strings as Print Elements

    Literal strings are printed exactly as they appear within the quotation marks. No spaces are added, before or after the string. All spaces within the quotation marks are printed.

Execute the following program

```
10 PRINT "ABC"; "DEF"
20 PRINT "ABCDEF"
```

The display shows

```
:RUN
ABCDEF
ABCDEF
:_
```

The first ABCDEF is produced by printing the two literal strings in statement 10, one right after the other. Since spaces are not added to literal strings, the effect of statement 10 is identical to statement 20.

Expressions as Print Elements

Any expression can serve as a print element. When an expression appears as a PRINT element, the expression is evaluated according to the rules of evaluation discussed in the last section. The value of the expression is then printed. The variables in our example programs such as I, P and F are, as we know from the last section, just one kind of expression.

To appreciate the power of using expressions as print elements, enter and execute the following simple program which calculates the area of a circle given the radius R, where area = πR^2.

Example 3.1 Expressions as PRINT Elements

```
10 INPUT "RADIUS", R
20 PRINT "RADIUS="; R, "AREA="; 3.14*(R↑2)
```

Statement 20 first prints "RADIUS=", then the value of R. The next print element, the literal string "AREA=", is printed followed by the value of the expression 3.14*(R↑2). (3.14 is used as an approximation of π.) Statement 20 causes the expression 3.14* (R↑2) to be evaluated. No variables are changed by this evaluation. The result is just printed; it is not saved anywhere.

If a calculated value is only printed, and not used elsewhere in a program, the simplest approach is to let the PRINT statement perform the calculation.

Suppose that in our inventory example we wanted to continuously display the dollar value of the on-hand inventory. Assume that the value of a ton of coal is 20 dollars. Statement 60 (Example 2.1) can be modified as follows to calculate and display this value:

```
60 PRINT "TONS ON HAND="; I, "VALUE=$"; 20*I
```

38

Statement 60 does not change the value of the variable I. A PRINT statement never alters the value of any variable. If this statement 60 is substituted for the old statement 60, the program display appears as:

```
TONS ON HAND = 4500        VALUE=$ 90000
NUMBER OF TONS RECEIVED (+) OR SOLD (-)?
```

Now let's see how expressions are printed with regard to spaces.   Enter the following program:

```
10 I = 42500
20 PRINT "OPENING INVENTORY="; I; "TONS"
```

We have added the literal string "TONS" to statement 20 so that any trailing spaces, printed with the expression I, can be detected. Executing this program produces

```
OPENING INVENTORY= 42500 TONS
```

The value of the expression is printed with a space before and after the digits. Now change line 10 to

```
10 I = -42500
```

Execution produces

```
OPENING INVENTORY=-42500 TONS
```

which reveals that the leading space is actually just an implied plus sign. When the value is negative, the position is occupied by the minus sign.   The space at the end still appears, however.

In summary, the value of an expression is printed preceded by a space, if positive, or a minus sign if negative. The value is always followed by a space.

A seven character value, such as 5600.12 (six digits and a decimal), occupies 9 character locations. Enter and execute

```
10 I = 5600.12
20 PRINT I; -2*I; "&&&"
30 PRINT "ABCDEFGHIJKLMNOPQRSTUV"
```

The result of execution is

```
5600.12 -11200.24 &&&
ABCDEFGHIJKLMNOPQRSTUV
```

The space above the A is the implied + sign and the space above the I is the automatic trailing space.   Similarly, the space above S is the trailing space of the value -2*I.   The ampersands are printed to reveal the final trailing space.

Now that we have seen how expressions are printed, from the point of view of spacing, let's look at the way the value itself appears. The PRINT statement prints the value of an expression in one of two forms depending on the magnitude of the value. If the absolute value of the number being printed (the expression) is greater than or equal to .1 and less than or equal to 9,999,999,999,999 (or it is zero), then the value is printed in normal form. Normal form is

        SZZZZZZ.FFFFFFFΔ

where   S is minus sign (-) if value <0, or blank if value ≥ 0.
         Z is an integer digit.
         F is a fractional digit.
         Δ is a space.

The decimal is in the proper position; it is omitted if the value is an integer. Leading and trailing zeros are omitted.

For example, these numbers are in normal form.

         212
        -4593.17286
        -6.1
         0
         12345678901.23

If the absolute value $|Q|$ of the expression satisfies

        $10^{-99} \leq |Q| < .1$

or
        $Q \geq 10^{13}$

then the PRINT statement outputs Q in scientific notation.

Scientific notation is

    SM.MMMMMMMME±XXΔ

where   S is a minus sign(-) if value < 0, or blank if value ≥ 0.
         M is a mantissa digit.
         E is the symbol indicating the beginning of the exponent.
         ± is the sign of the exponent, always explicit.
         X is an exponent digit.
         Δ is a space.

Nine mantissa digits are printed even if some are zero. Therefore any value printed in scientific notation by the PRINT statement occupies 16 character positions. For example,

        10 PRINT 1E13

produces

        1.00000000E+13

40

Note that the PRINT statement determines the output form, normal or scientific, without regard to the form in which the value is input. For example, enter and execute

```
10 PRINT .01; .05; .09; .1
20 PRINT 5.00/75
30 PRINT 275634*112913534
40 PRINT 2.56E8, 2.56E8 * 1.02E-2
```

Each PRINT statement generates one line of output. The total output is shown below. Compare the output lines with the statements that generate them.

```
1.00000000E-02  5.00000000E-02  9.00000000E-02  .1
6.66666666E-02
3.11228090E+13
256000000       2611200
```

Semicolons and Commas between Print Elements

Semicolons and commas serve a dual function in PRINT statements. They separate one print element from the next, and they help to control the spacing of output.

A comma or semicolon must be used between each PRINT element. This is a requirement of BASIC syntax. For example, this statement violates BASIC syntax, and, upon entry, produces the error message shown:

```
10 PRINT "ABC" "DEF"
              ↑ERR 10
```

A semicolon can be thought of as the "do-nothing" element separator. It fulfills the syntax requirements of BASIC, but causes no additional spacing between print elements. The only spaces that appear between print elements separated by semicolons are those that are output with the print elements themselves.

For example, if we insert a semicolon into the above statement as follows:

```
10 PRINT "ABC"; "DEF"
```

the result, upon execution, is:

```
:RUN
ABCDEF
```

Since there are no spaces within the print elements, no spaces appear in the printed output.

To look again at a previous example,

```
10 I = 5600.12
20 PRINT I; -2*I; "&&&"
30 PRINT "ABCDEFGHIJKLMNOPQRSTUV"

:RUN
 5600.12 -11200.24 &&&
ABCDEFGHIJKLMNOPQRSTUV
```

All the spaces that appear in the first line of output are the implied plus sign and the trailing spaces, output automatically as the result of printing the expressions.

A comma can be used in a PRINT statement to cause the next print element to be printed at the beginning of the next print zone. Each line of the CRT is divided into 16-character zones. The standard 64 character CRT line accommodates four such print zones.

When the system finds a comma in a PRINT statement, it outputs spaces until it reaches the first character position of the next print zone. It then outputs the next print element starting at this position. For example, the following line appeared in the factorial program (Example 2.2).

```
30 PRINT "P="; P, "P!="; F
```

Output from repeated executions of this line looked like this on the CRT:

```
      ZONE 1           ZONE 2
    ┌───────┐    ┌───────────────┐
    P= 0           P!= 1
    P= 1           P!= 1
    P= 2           P!= 2
    P= 3           P!= 6
    P= 4           P!= 24
    P= 5           P!= 120
```

The comma after the second print element causes the system to space to the right, to the beginning of the next zone.

If a comma is encountered when the output from the previous print element has entered the right-most zone of the CRT, output from the next print element begins at the left of zone 1 on the next line. For example,

```
10 PRINT "A", "B", "C", "D", "E"
:RUN
A               B               C               D

E
```

The letters "A" to "D" are printed in the first character position of zones 1 to 4. "E" is printed at the beginning of zone 1 of the next line, since there is room for only four zones on a 64 character line.

Commas can be used anywhere in a print statement, and can be used in succession. For example, if another comma is put into line 30 of the factorial example (Example 2.2) as follows:

```
30 PRINT "P="; P ,, "P!="; F
```

the result appears as:

```
        ZONE 1                 ZONE 2              ZONE 3
   P= 0                                       P!= 1
   P= 1                                       P!= 1
   P= 2                                       P!= 2
   P= 3                                       P!= 6
   P= 4                                       P!= 24
   P= 5                                       P!= 120
   ***** DONE *****
```

The second comma causes the system to output spaces up to the first character position of zone 3.

Commas and Semicolons at the End of Print Statements

After outputting the last print element in a PRINT statement, the Wang 2200 system automatically moves the cursor to the left-most position of the next line. (The cursor defines the CRT screen-location for the next printed character.) The 2200 system moves the cursor by issuing special cursor control characters to the CRT. (These characters are discussed in detail in Chapter 17.) The cursor control characters are the equivalent of a carriage return and line feed on a typewriter, and therefore are known as CR/LF. For example, execution of

```
10 PRINT "ABC"
20 PRINT "DEF"
```

produces

```
:RUN
ABC
DEF
```

"DEF" appears on the line below "ABC" because of the CR/LF issued automatically after "ABC". For the same reason, each PRINT statement in the original example programs (Examples 2.1 and 2.2) causes output to appear on a new line.

Often an automatic CR/LF is exactly what we wish to have occur after a
PRINT statement. Examples 2.1 and 2.2 illustrate this. However, sometimes we
may wish to suppress this automatic CR/LF.   The automatic CR/LF can be
suppressed by putting a semicolon or comma at the end of the PRINT statement.
For example,

```
        10 PRINT "ABC";
        20 PRINT "DEF"

        :RUN
        ABCDEF
```

Here "ABC" and "DEF" are output on the same line because the semicolon at
the end of the PRINT statement suppresses the CR/LF.

Enter and execute this simple program that prints the powers of two from
$2\uparrow1$ through $2\uparrow20$.

Example 3.2 Powers of Two, Illustrating PRINT with Trailing Semicolon

```
        10 N = N+1
        20 PRINT 2↑N;
        30 IF N < 20 THEN 10
        40 PRINT "DONE"
```

The output is:

```
        2  4  8  16  32  64  128  256  512  1024  2048  2096  8192
        16384  32768  65536  131072  262144  524288  1048576 DONE
```

The semicolon at the end of line 20 suppresses the carriage return/line feed.
The system issues a CR/LF only when it senses that a PRINT element will
overflow the line, in this case when $2\uparrow N$ is 16384.   The spaces between the
values are the implied plus sign and trailing space, printed with each
expression.

If a comma appears after the last print element   in   a   PRINT
statement, the cursor is moved to the beginning of the next   zone,
and the CR/LF is suppressed.

Reenter the Powers of Two program as follows:

Example 3.3 Powers of Two, Illustrating PRINT with Trailing Comma

```
        10 N = N + 1
        20 PRINT 2↑N,
        30 IF N < 20 THEN 10
        40 PRINT "DONE"
```

This program produces the following output:

2	4	8	16
32	64	128	256
512	1024	2048	4096
8192	16384	32768	65536
131072	262144	524288	1048576
DONE			

Each time statement 20 is executed, the value of the expression is printed at the current cursor location, and then the cursor is moved to the beginning of the next zone. This causes the output to appear as shown.

The TAB() Print Element

The comma element separator is convenient for organizing output into 16 character columns, but suppose that you wish to use some other column format. For example, suppose you wish to print in three, approximately equal, columns across the CRT. The first of these columns is to begin at CRT column 0, the next at column 22, the next at 43. This column spacing could be accomplished by using commas together with the proper number of spaces enclosed in quotation marks. However, BASIC offers the TAB() print element as an easier means of accomplishing this type of output formatting.

TAB() tells the system to output spaces until the cursor is at a specified CRT column. Example 3.4 calculates the length of the hypotenuse of a right triangle, based on the lengths of the sides, and outputs the three lengths at columns 0, 22, and 43.

Example 3.4   Sides of a Right Triangle, Illustration of TAB()

```
10 INPUT "LENGTH OF SIDE A",  A
20 INPUT "LENGTH OF SIDE B",  B
30 PRINT
40 PRINT "SIDE A"; TAB(22); "SIDE B"; TAB(43); "HYPOTENUSE"
50 PRINT A; TAB(22); B; TAB(43); (A↑2+B↑2)↑.5

:RUN
LENGTH OF SIDE A? 5
LENGTH OF SIDE B? 12

SIDE A                SIDE B                HYPOTENUSE
 5                     12                    13
```

At line 40 of Example 3.4, TAB(22) appears after "SIDE A". TAB(22) is a print element that tells the system to output spaces until the cursor is at column 22 of the CRT. Similarly, after printing "SIDE B", TAB(43) outputs spaces until the cursor reaches column 43. Since TAB() is a print element, an element separator (, or ;) must be used with it. Normally, the semicolon is used, since it is less confusing if the cursor is moved only by the TAB(), rather than by a combination of TAB()'s and commas. For the purpose of TAB(), CRT columns are numbered 0-63, not 1-64.

45

TAB() tells the system to issue spaces until the specified column is reached. Therefore, it is not possible to move the cursor left with a TAB(). If the column specified in the TAB() is to the left of the current cursor location, the TAB() simply does nothing; the cursor is not moved.

If the column specified in the TAB() is greater than the line length of the CRT, the cursor is moved to the leftmost position on the next line. A column specification greater than 255 produces an error.

Any expression can be used in a TAB() to specify the desired column. In Example 3.5, the previous program (Example 3.4) has been modified to allow entry of the desired output column width.

Example 3.5 Using an Expression to Calculate a TAB()

```
10 INPUT "COLUMN WIDTH", T
20 INPUT "LENGTH OF SIDE A",  A
30 INPUT "LENGTH OF SIDE B",  B
40 PRINT
50 PRINT "SIDE A"; TAB(T); "SIDE B"; TAB(2*T); "HYPOTENUSE"
60 PRINT A; TAB(T); B; TAB(2*T); (A↑2+B↑2)↑.5
```

Line 50 prints "SIDE A", and then, in order to execute the TAB() evaluates the expression T. T has the value entered at line 10. The TAB is executed, moving the cursor to the specified column. After "SIDE B" is printed, the expression 2*T is evaluated to determine the second column position. Only the whole number, or integer, portion of the value is used to determine the column. Thus, an entry of 10.6 causes a TAB to column 10, followed by a TAB to column 21 (integer portion of 21.2).

Review of Section 3-3

1.   The things to be printed in PRINT statements are known as print elements. Any number of print elements can appear in a single PRINT statement.

2.   Literal strings are printed exactly as they appear within quotation marks. No spaces are added before or after the string.

3.   When an expression is encountered in a PRINT statement, the expression is evaluated, and the result is printed.

4.   A PRINT statement never alters the value of any variable.

5.   The value of an expression is printed preceded by a space if positive, or a minus sign if negative, and followed by a space.

6.   The PRINT statement prints the value of an expression in normal form or scientific form, depending upon the value.

7.   The cursor, always present on the CRT, occupies the location at which the next character will be displayed.

8.   The semicolon element separator tells the system, "Leave the cursor where it is."

9.   After all the elements in a PRINT statement have been printed, the system automatically issues a cursor control that is the equivalent of a carriage return and line feed on a typewriter. This automatic function can be suppressed by specifying an element separator (, or ;) at the end of a PRINT statement.

10.   The standard CRT is divided into four print zones, each 16 characters wide.

11.   When a comma (,) element separator is encountered in a PRINT statement, it moves the cursor to the beginning of the next zone. If the cursor is in the 4th, or rightmost zone, the comma moves the cursor to the beginning of zone 1 on the next line.

12.   A comma can appear anywhere in a PRINT statement. Commas can appear one after another to move the cursor to the 2nd or 3rd next print zone, etc.

13.   The TAB() print element can be used to move the cursor to the right to a specified column location.

3-4   LINE NUMBERS, LINES, AND THE GOTO STATEMENT

So far our example programs have shown line numbers in the sequence 10, 20, 30... . Line numbers in this sequence are conventionally used in BASIC programs, but any number in the range 0-9999 can serve as a line number. The use of 10, 20, 30... as line numbers is a convention which makes program debugging and modifying easier. It allows up to nine lines to be inserted between two lines of the original program. Non-integer values such as 1.1, 22.5 may not be used for line numbers. Spaces may not precede line numbers. Lines may be keyed in any order. Based on the line number, the system automatically determines the logical location of the line within the program.

In the last two sections, when we said such things as "Any number of print elements can appear in a single print statement," you may have wondered if there is some unexpressed qualification to these statements. There is. The longest statement line that can be entered is one generated by 192 keystrokes. The number of characters which represent this line can exceed 192, since keywords can be entered with a single keystroke. A program line with over 64 characters will, of course, continue onto a second CRT line. This does not affect program execution.

We said earlier that the normal sequence of execution in a program is line number sequence, from the lowest line to the next higher, etc. This sequence of execution is "normal" in the sense that it is the sequence that the system uses, except when an instruction is encountered that tells it to go to some other location for its next instruction. Instructions of this kind are called branching instructions, and the simplest of them is the GOTO statement.

The GOTO statement appeared in our original inventory example at line 90.

        90 GOTO 30

This statement simply causes the system to find its next statement at line 30, rather than at the next higher line number. (In the inventory program there is no higher line number than 90, so without statement 90 the program would end.)

After a branch has been effected, the normal sequence resumes from the new location. Specifically, if statement 90 is executed in the inventory program, the line number sequence of execution is 90, 30, 40, 50, 60... .

The general form of the GOTO statement is

GOTO line number

The words "line number" indicate that only a line number may be used. A variable may not be used. In addition, a BASIC statement must actually be present at the line number specified in a GOTO statement.

Though our inventory example showed the GOTO effecting a branch "back" in a program, a branch "forward," that is, one which branches over higher numbered lines, may also be effected.

A well organized program avoids unnecessary use of the GOTO statement by relying upon the normal sequence of execution, insofar as is possible. An obvious case of poor organization is shown at the top of Example 3.6. Statements 20 and 110 can be eliminated by rearranging the program to rely on the normal sequence of execution. At the bottom, the program is shown rearranged.

Example 3.6 Computing the Sum and Mean of Entered Values, Illustrating Unnecessary GOTO's

Before Reorganization

```
10  PRINT "COMPUTE AVERAGE"
20  GOTO 200
100 PRINT "NUMBER OF ITEMS="; K, "SUM="; S,, "MEAN="; S/K
110 GOTO 300
200 INPUT "ENTER VALUE (OR ZERO TO END ENTRY)", X
210 IF X=0 THEN 100.
220 K=K+1
230 S=S+X
240 GOTO 200
300 PRINT
310 GOTO 10
```

After Reorganization

```
10  PRINT "COMPUTE AVERAGE"
100 INPUT "ENTER VALUE (OR ZERO TO END ENTRY)", X
110 IF X=0 THEN 200
120 K=K+1
130 S=S+X
140 GOTO 100
200 PRINT "NUMBER OF ITEMS="; K, "SUM="; S,, "MEAN="; S/K
210 PRINT
220 GOTO 10
```

In the next section, in which we look at the IF...THEN statement, we will see how the GOTO at line 140 of the lower program can also be eliminated.

Review of Section 3-4

1. A line number can be any number 0-9999. Non-integer values such as 1.1, 22.5,etc. are not allowed.

2. The longest legal statement line is one generated by 192 keystrokes.

3. The normal sequence of execution is line number sequence.

4. Instructions that can tell the system to find its next instruction at some location other than the next higher line number, are known as branching instructions.

5. The simplest of the branching instructions is the unconditional branch to a line number, the GOTO statement. Its general form is:

    GOTO line number

6. Once a branch has been effected, the normal sequence of execution prevails from the new location.

7. A well organized program avoids unnecessary GOTO's.

3-5   THE IF...THEN STATEMENT

The IF...THEN statement causes a branch to a specified line if a stated condition is true. If the stated condition is false, the normal sequence of execution prevails.

The condition, which appears between the keywords "IF" and "THEN", is stated as a relationship between two expressions. We have seen the following IF...THEN statements in example programs; the condition portion is underlined:

    70 IF I> = 100 THEN 30

    60 IF P < = N THEN 30

    30 IF N < 20 THEN 10

The symbols such as <,< =, >=, which separate the expressions are known as relational operators. All the relational operators and their meanings are given below.

RELATIONAL
OPERATOR   MEANING

RELATIONAL OPERATOR	MEANING
=	is equal to
>	is greater than
<	is less than
<>	is not equal to
> =	is greater than or equal to
< =	is less than or equal to

49

The values of variables are never changed by an IF...THEN statement. The statement simply tells the system to evaluate two expressions using the current values of the variables, and compare the results obtained. If the system finds that the results are in the relationship specified by the relational operator, then the system branches to the specified line number to obtain its next instruction.

Uses of the IF...THEN Statement

In the inventory example, IF...THEN was used to decide within the program if a reorder message should be displayed. This simple program decision, based on a quantity calculated in the program, is a typical elementary use of IF...THEN.

In the factorial program the statement

```
60 IF P < = N THEN 30
```

is used to decide whether the loop should continue, or processing is complete.

In the program that prints the first 20 powers of two (Example 3.2) the statement

```
30 IF N < 20 THEN 10
```

performs a task similar to statement 60 in the factorial program.

A common use of the IF...THEN statement, not yet shown, is the testing of keyboard entries. Frequently you want to be sure that an entry is within a certain range. If it is not, you can repeat the request.

For example, in the factorial program a negative entry is meaningless, and an entry of 70 or greater produces an error at P=70 (because the value of P! exceeds 10^{100}). Therefore, it would be good programming practice to restrict operator entries to the valid range 0-69. The program can be modified as follows:

Example 3.7   Testing the Keyboard Entry in the Factorial Program

```
10 INPUT "COMPUTE P! FOR P=0 TO P=", N
12 IF N  < 0 THEN 15
13 IF N < = 69 THEN 20
15 PRINT "INVALID REENTER"
16 GOTO 10
20 LET F = 1
30 PRINT "P="; P, "P!="; F
40 LET P = P+1
50 LET F = P*F
60 IF P < = N THEN 30
70 PRINT "***** DONE *****"
```

An IF...THEN statement can also be used to test a keyboard entry that represents a selection among alternatives. For example, we might combine into a single program the factorial program and the print powers of two program.

50

At the beginning of the combined program, we would want a selection routine in which the operator can choose which of the two operations are to be performed. Such a combined program appears in Example 3.8.

Example 3.8 An Operator Selection Using IF...THEN

```
10 INPUT "ENTER 1 TO CALCULATE FACTORIALS.  ENTER 2 TO PRINT
POWERS OF TWO.", S
20 IF S = 1 THEN 110
30 IF S = 2 THEN 1010
40 PRINT "INVALID. REENTER"
50 GOTO 10
110 INPUT "COMPUTE P! FOR P=0 TO P=", N
120 LET F = 1
130 PRINT "P="; P,   "P!="; F
140 LET P = P + 1
150 LET F = F * P
160 IF P< =N THEN 130
170 GOTO 1040
1010 N = N + 1
1020 PRINT 2↑N;
1030 IF N < 20 THEN 1010
1040 PRINT "DONE"
```

The selection routine occupies lines 10-50, the factorial program 110-120, and the powers of two program 1010-1040. In the selection routine, notice that if neither 20 nor 30 cause a branch, then the entry is invalid, and the operator is prompted to reenter the selection. Also notice that statement 170 avoids executing both programs when the first is selected.

Thus far, in the example programs, the condition portion of the IF...THEN statements has been a relation between simple variables or variables and constants. While in practice many comparisons may be of this form, the capability of the IF...THEN statement to compare complex expressions should not be overlooked.

For example, suppose we wish to determine if the length of one line segment is greater than another. We are given Cartesian coordinates for the end points of the segments, and have assigned them to variables in the following manner.

(X1,Y1) and (X2,Y2) define line segment one.
(X3,Y3) and (X4,Y4) define line segment two.

The following statement effects a branch to line 570 if the length of line segment one is greater than line segment two.

510 IF (X1-X2)↑2+(Y1-Y2)↑2❭ (X3-X4)↑2 +(Y3-Y4)↑2 THEN 570

Another use for complex expressions in IF...THEN statements is to exploit the logic of arithmetic to allow testing of "multiple" conditions in a single IF...THEN statement.

51

Suppose, for example, that somewhere in a program you wish to branch to line 480, if any one of four variables is equal to zero. Assume the variables are W,X,Y,Z. One way to do this would be to write four statements:

```
50 IF W=0 THEN 480
60 IF X=0 THEN 480
70 IF Y=0 THEN 480
80 IF Z=0 THEN 480
```

This sequence of statements would work perfectly, but another way would be to write a single statement such as

```
50 IF W*X*Y*Z=0 THEN 480
```

In the new statement 50 the value of the expression W*X*Y*Z is 0 if any of the variables are zero. The effect is the same as the four statements written above.

The same approach can be used in a slightly more complex situation. Suppose you want to branch to line 115 if W=95 or if X=W*2 or if Y=55. You could write

```
50 IF W=95  THEN 115
60 IF X=W*2 THEN 115
70 IF Y=55  THEN 115
```

or, alternatively, you could write

```
50 IF (W-95) * (X-(W*2)) * (Y-55)=0 THEN 115
```

In the latter statement 50, if W is equal to 95, then the value of the expression (W-95) is zero. It is easy to see that the other expressions in parentheses (X-(W*2)) and (Y-55) are equal to zero when the conditions expressed in lines 60 and 70 above are true. Since the entire expression on the left is equal to zero if any of the terms equal zero, the effect of the latter statement 50 is the same as statements 50, 60 and 70 above it.

Efficient Use of IF...THEN

Now that we have discussed some of the capabilities of the IF...THEN statement itself, let's see how it can be used most efficiently.

Suppose you are writing the inventory program that we introduced in Section 2-2. You have written the first six statements of the program:

```
10 LET I=42500
20 PRINT "OPENING INVENTORY="; I
30 INPUT "NUMBER OF TONS RECEIVED (+)" OR SOLD (-)", T
40 LET I=I+T
50 PRINT
60 PRINT "TONS ON HAND ="; I
```

Now you look back at the description of the problem and read "you want to be reminded to reorder coal if your inventory drops below 100 tons." So, you write

          70 IF I < 100 THEN 90

which says exactly what the statement of the problem said. At 90 you intend to put the PRINT statement which produces the reminder. So, now you add

          80 GOTO 30
          90 PRINT "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS"
          100 GOTO 30

Your program is now complete and appears as follows:

     Example 3.9 The Inventory Program (Example 2.1) Written with an
                  Extra GOTO

          10   LET I=42500
          20   PRINT "OPENING INVENTORY="; I
          30   INPUT "NUMBER OF TONS RECEIVED (+) OR SOLD (-)", T
          40   LET I=I+T
          50   PRINT
          60   PRINT "TONS ON HAND ="; I
          70   IF I < 100 THEN 90
          80   GOTO 30
          90   PRINT "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS"
          100 GOTO 30

When I is less than 100, statement 70 causes a branch around the GOTO statement at line 80.

     This program will produce the exact same results as the example program we originally gave (Example 2.1), but the original example program displays more skillful use of the IF...THEN statement. The two programs are identical in lines 10-60. Statement 70 of the original example program is

          70 IF I > = 100 THEN 30

Instead of testing for the inventory being less than 100 tons, the original program tests for the opposite relation. If $I \geq 100$ then the program need not display the reorder message. The entire original program is:

     Example 3.10 The Original Inventory Program (Example 2.1)

          10 LET I=42500
          20 PRINT "OPENING INVENTORY="; I
          30 INPUT "NUMBER OF TONS RECEIVED (+) OR SOLD (-)", T
          40 LET I=I+T
          50 PRINT
          60 PRINT "TONS ON HAND ="; I
          70 IF I > = 100 THEN 30
          80 PRINT "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS"
          90 GOTO 30

Reversing the relational operator at line 70 let us utilize the normal sequence of execution for the possibility that the reorder message should be displayed. The GOTO statement at line 80 of Example 3.9 has thereby been eliminated.

Sometimes efficient use of IF...THEN requires that the condition be specified the opposite way it might normally be conceived. This condition reversal was seen in the transition from Example 3.9 to Example 3.10 above. A table pairing the opposite relational operators is given below:

```
<> and = are opposites
> = and < are opposites
< = and > are opposites
```

The following program performs identically to the original factorial example. However, it includes a GOTO that was eliminated in the original example through more skillful use of the IF...THEN statement. Compare it to the original example, Example 2.2.

```
10 INPUT "COMPUTE P! FOR P=0 TO P=", N
20 LET F = 1
30 PRINT "P="; P, "P!="; F
40 LET P = P+1
50 IF   P > N THEN 80
60 LET F = P*F
70 GOTO 30
80 PRINT "***** DONE *****"
```

If one of the possible outcomes of a test is to skip forward over intervening statements and the other is to execute the intervening statements, then the IF...THEN should effect the branch to the higher numbered line. For example,

```
   :
   :
50   IF X=Y THEN 70
60   GOTO 200
70   PRINT Y
:  (processing)
   :
200 PRINT X
```

is better written,

```
50   IF X <> Y THEN 200
60   PRINT Y
   :
   :
200 PRINT X
```

Review of Section 3-5

1. The IF...THEN statement tells the system to branch to a specified line if a stated condition is true. If the condition is false, the normal sequence of execution prevails.

2. Values in variables are never changed by an IF...THEN statement.

3. The IF...THEN statement can be used for a decision branch based on a quantity calculated in the program. Its use in the original inventory program is an example of this.

4. IF...THEN can be used to test for continuation of a processing loop. The factorial program uses it in this fashion.

5. IF...THEN can be used to test the validity of an operator entry.

6. IF...THEN can be used to test an entry which represents a selection among alternatives.

7. Complex expressions can be used in the condition portion of the IF...THEN statement.

8. The logic of arithmetic can be exploited to test "multiple" conditions in a single IF...THEN statement.

9. Skillful use of the IF...THEN statement makes use of the normal sequence of execution as much as possible. This eliminates unnecessary GOTO's. The significance of the IF...THEN statement can be reversed by substituting the opposite, or "contradictory", relational operator.

3-6 THE INPUT STATEMENT

The INPUT statement provides an easy means for receiving data from the keyboard during program execution.

Since ordinarily the operator should be told what data is to be entered, the INPUT statement can contain a literal string prompt, which it will display at the current cursor location.

In example programs, we have seen such INPUT statements as

    10 INPUT "NUMBER OF TONS RECEIVED (+) OR SOLD (-)", T

and

    10 INPUT "ENTER 1 TO CALCULATE FACTORIALS. ENTER 2 TO
    PRINT POWERS OF 2", S

Literal string prompts are optional in the INPUT statement. If the above statements had been written,

    10 INPUT T

and

    10 INPUT S

the statements would not display the literal string, but otherwise they would function as the original statements did.

The prompt, if it is used, must be followed by a comma. Commas in the INPUT statement, unlike commas in the PRINT statement, have no effect on the prompt location. They serve only to separate the parts of the INPUT statement.

Regardless of whether a prompt is included in an INPUT statement, a question mark and space are always output. The question mark is a standard signal to the operator that the system is awaiting input.

As the operator keys in characters, they are displayed at the cursor location. Keying backspace eliminates the character at the previous cursor location, and allows for reentry.

When the operator depresses (EXEC), the value entered is assigned to the variable specified in the INPUT statement, and a CR/LF is issued to move the cursor to the beginning of the next line. If anything other than a valid numeric quantity is entered, an ERR is signalled, the question mark is redisplayed, and the operator can again enter the value.

Multiple entries can be made with a single INPUT statement by specifying more than one receiving variable. A statement such as

10 INPUT "ENTER EMPLOYEE NUMBER, THEN HOURS WORKED", E,H

displays the prompt followed by the question mark. The operator can then enter two values, which will be assigned to E and H respectively.

The operator can enter the values separated by a comma and followed by (EXEC) in this fashion:

ENTER EMPLOYEE NUMBER, THEN HOURS WORKED? 1234, 40.00 (EXEC)

Alternatively, the operator can enter the first value, key (EXEC), and then enter the second in the same manner. If this latter form of entry is used, the question mark is redisplayed at the left of the next line, after the first value is entered.

Other than the maximum line length, there is no limit to the number of variables that may appear in a single INPUT statement. All variables must be separated by commas and only one prompt may be specified. However, INPUT statements with many variables tend to be awkward for an operator, and promote entry errors.

If an operator merely depresses (EXEC) in response to an INPUT instruction, the value of the receiving variable is unaltered. However, this response also terminates the INPUT instruction, thereby eliminating the possibility of entering values for any remaining variables.

The general form of the INPUT statement is

INPUT ["character string",] variable [,variable...]

In addition to its primary use as a means of receiving data from the keyboard, the INPUT statement is an excellent way of inserting a processing interruption into a program. If printer paper, a disk, or tape must be mounted in the middle of a program, a simple INPUT statement such as

550 INPUT "MOUNT PAPER. KEY (EXEC) TO RESUME", A9

can be used to interrupt execution. The value of A9 can be ignored, or tested to see that no entry was in fact made. (An entry might suggest that the operator was confused about the operation.)

Review of Section 3-6

1. The INPUT statement allows a value or values to be entered from the keyboard and assigns them to specified variables.

2. A prompt can be specified in the form of a literal string. The prompt is displayed by the INPUT statement at the current cursor location. The prompt is optional, but if it is used, it must be followed by a comma.

3. The INPUT statement causes a question mark and space to be displayed. This occurs after any prompt has been displayed, and is the standard signal that the system is awaiting input from the keyboard.

4. Values entered by the operator are assigned sequentially to the variables specified in the INPUT statement.

5. Multiple receiving variables in the INPUT statement must be separated by commas.

6. If an operator merely depresses (EXEC) in response to an INPUT instruction, the value of the variable which was to receive the entry remains unchanged.

7. The general form of the INPUT statement is

    INPUT ["character string",] variable [,variable...]

3-7   THE REM STATEMENT

The REM statement has not appeared in any of the example programs thus far. REM is an abbreviation of the word "remark"; the purpose of the REM statement is to allow you to insert into a program explanatory comments, or remarks, about the program itself.

In effect, REM says to the system "ignore this statement". REM statements have absolutely no effect on the execution of a program. With REM statements inserted, our inventory program could have appeared as follows:

Example 3.11 Adding Comments (REM's) to Example 2.1

```
10    REM (EG3.11) A SIMPLE INVENTORY PROGRAM
20    REM ********** VARIABLE USAGE **************
30    REM  I  = INVENTORY BALANCE
40    REM  T  = TRANSACTION AMOUNT
50    REM  **************************************
60    REM
80    REM ASSIGN OPENING BALANCE
90        LET  I=42500
100       PRINT  "OPENING INVENTORY=";  I
110   REM ***** MAIN PROCESSING LOOP *****
120       INPUT  "NUMBER OF TONS RECEIVED (+) OR SOLD (-)",  T
130           LET  I=I+T
140           PRINT
150           PRINT  "TONS ON HAND =";  I
160       IF   I> = 100    THEN  120
170   REM DISPLAY REORDER MESSAGE.
180       PRINT  "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS"
190       GOTO  120
```

The inventory program as shown here executes in the exact same manner as the original inventory program did.

REM comments are extremely useful, not only for someone else, who may want to understand your program, but also for yourself, should you later have to go back and revise sections of a program.

CHAPTER 4
SAVING AND LOADING PROGRAMS

4-1   INTRODUCTION

When first created, a program is normally entered into memory via the keyboard.   However, it would be extremely inconvenient to reenter it this way each time it is to be run.   Since memory is automatically cleared each time the CPU power is turned off, some means of saving programs outside of the memory of the system is a practical necessity.   The two principal means in use with Wang systems are storage on tape cassette and magnetic disk.   (Systems which are equipped with both types of device may use either or both for program storage.)

After a program has been keyed-in the first time, it can be saved on tape or disk for later use.   Then, it is a relatively simple and quick operation to load the program when it is needed.

4-2   SAVING PROGRAMS ON CASSETTE TAPE

Cassettes and Cassette Drives

Cassette tape is a convenient medium for saving programs.   It can also be used for saving data.   (Data storage on cassette tape is discussed in Chapter 21.) A single tape cassette can hold many programs.   The exact number depends upon the size of the programs, and whether a long (150 ft) or short (75 ft) tape is being used.   Programs are recorded one after another on the tape.   Each program is treated as a single unit called a "file".

To prevent accidental destruction of tape contents, tape cassettes are equipped with "protect tabs."   These plastic tabs are located in the bottom corners of the cassette.   If these tabs are folded inward to expose the square holes at each end of the cassette, the cassette is "protected"; further recording cannot be performed on it until the tabs are moved back to the non-protect position.   All the information on a protected cassette is available for loading into the system; only recording is prevented.   For protection from dust, cassettes should be kept in their individual plastic boxes when not in use.

To help identify programs saved on tape and to make loading of selected programs easier, you should specify a name for each program saved on a cassette tape.   The name can be up to 8 characters long, including spaces, and should distinguish the program from all others on the cassette.   Any keyboard characters can be used in the name.   You should maintain a list of the programs and program names stored on each cassette.   The list should show the

sequence in which the programs are recorded. Notice that at line 10 of Example 3.11 the name of the program "EG3.11" is given in a REM statement. Putting a program name into a REM at the first program line is a good programming practice.

If a cassette's contents are no longer of any value, a program may be stored at the beginning of the cassette, as if the cassette were blank. However, if any programs on a cassette are to be preserved, new programs should be added beyond the end of the last program to be preserved.

Although up to six tape cassette drives can be used with a Wang system, there is always one tape drive that is said to be the primary tape drive. In this section we will discuss the program saving and loading procedures for operations at this primary (or default) tape drive. For systems equipped with a tape cassette drive in the CRT housing, this tape drive is usually designated as the primary tape drive. Users of systems with stand-alone tape drives only (models 2217 or 2218) will have to determine which drive is the primary drive before proceeding with the cassette mounting operations described below. To determine this, simply key RESET REWIND (EXEC) and observe which drive's tape movement light flashes. The one that flashes is the primary tape drive.

If, after keying a program into memory, you wish to save it on a cassette, open the primary cassette drive door by pressing the white door release button located at the right of the cassette door. Slip a blank, unprotected cassette, with its label side facing you, into place between the brackets on the cassette drive door, as shown in Figure 4.1.
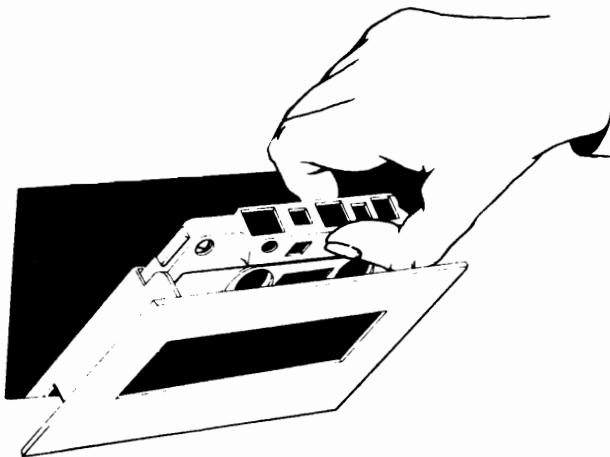
Figure 4.1 Mounting a Cassette

Once the cassette is in the cassette drive door, close the door. Then, rewind the cassette by pressing the REWIND button on the cassette drive housing. Rewinding, at this point, is a precautionary measure to ensure exact positioning of the tape at the beginning of the cassette.

Saving a Program on a Cassette

With a blank cassette mounted at the primary tape device, you can save the entire program currently in memory by keying

:SAVE "name" (EXEC)

where: name is a program identification, up to 8 characters long, enclosed by quotation marks.

For example, to save the inventory program (Example 2.1), you could key

:SAVE "INVT1" (EXEC)

Upon keying (EXEC), the cassette should begin recording. During recording the cassette drive emits audible clicks. When the SAVE operation is complete, the clicks cease, and the colon is restored to the next line of the CRT. The SAVE command does not alter the contents of memory.

After saving the program, the cassette can be rewound by depressing the REWIND button on the tape drive or by keying REWIND (EXEC) on the keyboard. (The cassette must be rewound before it can be removed from the drive.) You should then label the cassette and start a list of the files and file names stored on it.

If a cassette has valuable programs or data files on it, you must ensure that the tape is positioned beyond the end of these files before giving the system the SAVE command. If you do not do this, the system will begin saving the new program file over an old file, thereby destroying the old file.

One way to position a tape is to tell the system to SKIP over preceding files. This must be done in a separate operation, before the SAVE command is executed. For example, if a tape already contains one program, i.e., one program file, to save a second program onto the cassette you should:

1. Mount the unprotected cassette in the device.
2. Depress the rewind button.
3. Key
        :SKIP 1F (EXEC)
   to skip over the one file.
4. Key
   :SAVE "name" (EXEC),
   where "name" is the name of the new program.
5. Rewind and remove the tape cassette.

6. Add the program name and description to the list of files stored on the cassette.

If two programs are already on the tape, be sure to key SKIP 2F at step 3; if three, SKIP 3F, etc.

The cassette should be stored in its plastic case after removal. The program will remain on it unaltered, indefinitely, and may be loaded whenever desired.

Loading a Program from a Cassette

To load a program from a cassette, mount the cassette and key

>   :CLEAR (EXEC)
>   :LOAD "name" (EXEC)

where "name" is the name of the program to be loaded. The CLEAR command ensures that memory is cleared of all other programs prior to loading the new one. When LOAD is executed, the system searches the tape for the beginning of the named program. It then loads the entire program into memory.

For example, if the inventory program has been saved under the name "INVT1", it can be loaded by keying

>   :CLEAR (EXEC)
>   :LOAD "INVT1" (EXEC)

Review of Section 4-2

1. Cassettes are always mounted in the cassette drives with the exposed tape up and the label facing out.

2. A single tape cassette can hold many programs. Programs are stored sequentially one after another on the tape.

3. You should specify a name, up to 8 characters long, for each program saved on a cassette, and you should maintain a list of all the programs saved on the cassette.

4. If any programs on a cassette are to be preserved, new programs should be added beyond the end of the last program to be preserved.

5. To save a program from memory onto a cassette, first skip over the presently saved files by executing

    >   :SKIP xF

    where x is the number of files currently stored on the cassette.

    Next, save the program currently in memory by keying

    >   :SAVE "name"

    where "name" is the name of the new program, up to 8 characters in length.

    After the SAVE command has been executed, depress the rewind button on the cassette drive and remove the cassette. SAVE does not alter the contents of memory.

6. To load a program into memory from a cassette, mount the cassette and key
    ```
    :CLEAR (EXEC)
    :LOAD "name"  (EXEC)
    ```

    where "name" is the name of the program to be loaded.

4-3 SAVING PROGRAMS ON DISK

There are two different kinds of disks used with Wang systems. One kind consists of a large disk platter, about the size of a long playing phonograph record, that is housed within a rigid disk cartridge. This kind of disk is used on the 2230 and 2260 type disk drives. The other kind of disk more closely resembles a 45 rpm record, is flexible, and is contained in a soft envelope. This kind is used with the 2270 drives, and is usually called a "diskette". The two kinds of disks vary principally in their storage capacity and speed of operation. They may be considered to be identical from the point of view of programming.

```
┌─────────────────────────────────────────────────────┐
│                                                       │
│                      NOTE:                            │
│                                                       │
│  While you are learning about disk operations,  you  should │
│  not use a disk that contains important data or programs on │
│  it.   If you must use such a disk, be sure to consult with │
│  an  experienced  programmer,  who  is  familiar  with  the │
│  details of your system before executing anything described │
│  in this section, or in the disk formatting sections of the │
│  Disk Reference Manual.  If your system includes a diskette │
│  drive, a single diskette, devoted to your use, would be  a │
│  good  choice  for storage of programs you write during the │
│  course of reading this manual.                             │
│                                                       │
└─────────────────────────────────────────────────────┘
```

To help prevent accidental destruction of valuable diskette contents, diskettes used with the 2270 type drives are equipped with a mechanical "protect" feature. When a diskette is "protected", it is possible to read (i.e., load) programs or data from it, but it is not possible to record programs or data onto it. A small hole in the diskette envelope, located along the leading inserted edge, controls the protect feature. If this hole is covered on both sides by a small piece of opaque tape, the diskette is unprotected. If the hole is exposed, the diskette is protected.

Nearly every Wang system equipped with a disk drive has one disk drive designated as the primary or "default address" disk drive. This is the disk drive that is selected for use when the system is Master Initialized.

If your system consists of only one physical disk drive unit (which may house more than one disk), then you may assume that this drive unit is designated as the default drive. If your system has more than one disk drive, you will have to determine which drive is the default drive by consulting someone familiar with your system. The operations described in this section take place at the primary disk drive, unless a different drive is intentionally selected by you.

If your system is set up in such a way that it is inconvenient, or unwise for reasons of file safety, for you to use the default disk drive, then prior to performing any of the operations described here, you must select a different drive. To do this, first determine from competent authority what drive is appropriate for your use and what its address is. Then select this drive for use by keying

    :SELECT DISK XXX (EXEC)

where XXX is the address of the disk device to be used.

Once this disk drive has been selected, it will remain selected for the operations described in this section until the power is turned off. Each time the power is turned on, the SELECT command above must be executed, if a drive other than the default drive is to be used.

Preparing a Disk for Cataloged Program Storage

Unlike a tape, a disk device provides direct access to the information stored on it. (In this section the information we are considering consists of programs.) By saying "direct access", we mean to highlight the fact that if several programs have been saved on a disk, each can be located and loaded without having to search through any other programs. This principal of direct access is illustrated in the phonograph. With a phonograph, you can move the tone arm to any location on a record to play a selected piece.   Similarly, with a magnetic disk an arm moves a read/write head to a selected location, rapidly skipping over other locations. This contrasts with a tape system, in which to reach to an item in the middle of the tape, any preceding information must pass by the heads in the sequence in which it was recorded.

When you move a phonograph tone-arm to play a specific piece in the middle of a record, you locate the beginning of the piece by observing a reduced density of grooves at that point.   Reduced groove density is your system for identifying individual items on the phonograph record. In a disk storage system the disk drive positions the read/write head to specific locations; therefore, it too needs a means of identifying these locations. The fundamental system of identification is set up by the disk drive itself, before any information is recorded on the disk.   In this system of identification the recording on the disk takes place in concentric tracks; each track is divided into a number of small chunks, known as sectors, and each sector is assigned an identifying number, called its "address".

The process of setting up this system of addresses on a blank disk is known as formatting the disk. Generally a disk need only be formatted once, when it is first received. The formatting process is a function of the disk drive, so no program is required for it.

If you have a formatted disk that you can use for saving programs, then you can continue now with the remainder of this section. However, if you do not have a formatted disk, read the relevant selections of the Disk Reference Manual for instructions on how to format a disk.

The Disk Catalog

A disk need only be formatted in order to save programs on it. However, in order to load a program from a disk which has merely been formatted, you have to tell the system exactly where the program is located on the disk, in terms of disk sector addresses occupied. Though this approach can be used (it is called Direct Addressing), your Wang 2200 system is prepared to assume this burden by means of its built-in Catalog Mode statements and commands.

Your Wang system can automatically maintain a list of files stored on a disk. The list contains the file names and the locations that the files occupy. To load a particular program, you can refer to it by name, and let the system do the rest.

The "list of files stored on a disk" is known as the disk "catalog"; it is stored and maintained on the beginning sectors of the disk. A cataloged disk thereby carries its own index with it. The area of the disk that follows the catalog, in which the indexed programs are stored, is called the "catalog area".

Since in some cases a disk may contain a great many relatively short files, while in other cases it may be occupied by just a few large files, the proportion of the disk space to be devoted to the catalog itself, versus the catalog area, is open to your specification. Before catalog operations can be undertaken, the catalog and catalog areas must be defined.

CAUTION:

The operation of defining the catalog and catalog area wipes out any previous catalog, thereby denying access to all the information previously stored in the catalog area. Therefore, do not establish a catalog unless you are sure that the disk you use contains nothing important.

The SCRATCH DISK statement is used to establish, a catalog (catalog proper and catalog area) on a disk. For our purposes we can consider the general form of the SCRATCH DISK statement as

$$\text{SCRATCH DISK} \begin{Bmatrix} F \\ R \end{Bmatrix} \quad [\text{LS} = \text{expression 1,}] \quad \text{END} = \text{expression 2}$$

In addition to the fact that a system may have more than one disk drive unit, each unit may itself be capable of handling two disks. (One model, the 2270-3 can handle three disks, but in this section we will not discuss procedures for using the third or rightmost disk of that unit.) The braces in the general form of the SCRATCH DISK statement say that either F or R must be in the statement, but not both F and R. F or R is used to specify which disk at the selected disk drive is to be operated upon by the SCRATCH DISK statement.

If the selected disk drive is of the 2230 or 2260 series, then F in the SCRATCH DISK statement signifies that the fixed, lower disk is to be operated upon. If R is specified, the removable, upper disk is used.

In the Models 2270-2 and 2270-3, F specifies the leftmost diskette port. For the Model 2270-2, R specifies the right port; for the 2270-3, it specifies the middle port. If you are using the 2270-1, you must use F.

The symbol, [LS = expression 1,] is enclosed in brackets to indicate that it is optional. If used it specifies the number of disk sectors to be reserved for the catalog index. For example, in the following statement:

      10 SCRATCH DISK F LS = 30, END = 1023

the LS=30 specifies that 30 sectors are to be reserved for the index. Each sector of the catalog index holds 16 entries, except the first which holds 15. Therefore, a catalog index with 30 sectors can hold 479 file name entries. If the [LS = expression 1,] is omitted, 24 sectors are automatically reserved for the catalog index. The maximum number of sectors which can be reserved for the index is 255.

The "END = expression 2" portion of the statement must always be included. Expression 2 specifies the sector address for the end of the catalog area. The number specified cannot be larger than the highest sector address for the particular disk in use. The diskettes used with the 2270 series disk drives have 1024 sectors on them, addressed as sector 0 to sector 1023. For the rigid disk drives, the sectors and sector addresses are given below.

Disk Model	Sectors per Disk	Lowest Sector Address	Highest Sector Address
2230-1	2400	0	2399
2230-2	4800	0	4799
2230-3	9792	0	9791
2260	19,584	0	19,583

For some operations it may be desirable to have sectors available beyond the end of the catalog area. All operations described in this volume, however, take place within the catalog area.

If you have a formatted but otherwise empty disk, you can establish a catalog on it by entering and executing a one line program such as:

      10 SCRATCH DISK R END = 1023

Saving and Loading Programs on a Cataloged Disk

With a formatted, cataloged disk in hand, you are ready to begin saving programs on the disk. A program to be saved on disk and listed in the disk catalog must have a name. The name can be up to eight characters long and must uniquely identify the program file. For example, you might want to call the 1st inventory example (Example 2.1) "INVT1".

If the inventory program is in memory, it can be saved to a cataloged disk by keying a line such as:

      :SAVE DC F "INVT1" (EXEC)

In this command the DC specifies that it is a disk catalog mode operation. The F specifies the disk location, in the manner discussed above for the SCRATCH DISK statement; R could be used instead. The characters enclosed in quotation marks are the name.

When (EXEC) is depressed, the program is saved in the next available sectors in the catalog area, and the name of the program is entered in the catalog index, together with the program file's starting and ending sector addresses. All of this happens very rapidly; the :_ reappears when the operation is complete.

Programs can be saved without regard to where they are actually recorded on the disk. The automatic cataloging system will ensure that they are indexed, and recorded in previously unoccupied sectors.

To load a program which has been saved on a disk, first clear memory, and then key a line such as:

> :LOAD DC F "INVT1" (EXEC)

The significance of all the parts of this command is the same as in the SAVE statement, except LOAD reverses the operation. The system automatically searches the catalog index for the program name, and then loads the program into memory from the specified sectors.

Listing a Disk Catalog

The contents of a disk catalog index can be listed on the CRT for inspection by executing

$$:LIST\ DC\ \begin{Bmatrix} F \\ R \end{Bmatrix}$$

where F or R specify the disk location.

The result is a list such as this:

```
     FIXED CATALOG
INDEX SECTORS = 00024
END CAT. AREA = 01023
CURRENT END   = 00055

NAME         TYPE     START     END      USED
TEST         P        00024     00028    00005
CHECK6       P        00029     00038    00010
A/P PRINT    P        00039     00042    00004
BINOM        P        00043     00047    00005
FACTRL       P        00048     00051    00004
INVTRY       P        00052     00055    00004
```

Figure 4.2 A Disk Catalog Listing

67

The INDEX SECTORS and END CAT. AREA show the number of sectors allocated to the index and the highest sector address in the catalog area. These are the values established by the SCRATCH DISK instruction. The CURRENT END is the address of the last sector to have been filled with live information. It changes whenever a new file is added. Below this appears the list of cataloged files. The P under TYPE indicates that the named file is a program file. Data files are indicated by a D. The START, END and USED columns give the sector addresses of the first and last sector of the named file and the number of sectors occupied by it.

Reusing Obsolete Files

If you try to save a program onto a disk with the same name as a program which is already on the disk, the system will not save the program, and will report an error (ERR 79 FILE ALREADY CATALOGED). If you wish to save a program in place of a program already saved, either because the program you wish to save is a new version of the old program, or because the old program is no longer of value, you must first mark the old program as obsolete. This is done with a SCRATCH statement. For example, if you key

        :SCRATCH F "INVT1" (EXEC)
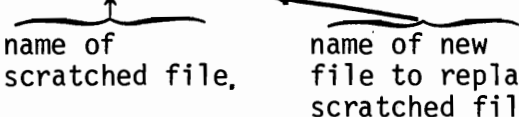
the system will mark the program file called INVT1 as "scratched", which simply means that it is obsolete. If you then LIST the catalog index, the notation "SP" appears in the "TYPE" column, indicating that the file is a scratched program file. A scratched file cannot be loaded.

```
┌─────────────────────────────────────────────────────────────┐
│                           CAUTION:                           │
│                                                              │
│  Do not confuse the SCRATCH statement with the SCRATCH DISK  │
│  statement.  SCRATCH DISK establishes  a  new  catalog  and  │
│  catalog  area,  and  renders   inaccessible  all  files     │
│  previously saved on a disk.                                 │
└─────────────────────────────────────────────────────────────┘
```

Once a file has been scratched, the space it occupies can be reused. For example, after scratching file "INVT1", you might wish to save in its place a new version, which you wish to name "INVT2". You can do this as follows:

        :SAVE DC F ("INVT1") "INVT2"
                      └─┬─┘   └──┬──┐
                    name of   name of new
                scratched file. file to replace
                                scratched file.

If you wish, the name of the new file can be the same as the name of the scratched file it replaces.

In order to replace a scratched file with a new file, there must be enough room in the old file space to save the new file. If you modify a program by adding more statements to it, you may find that it will not fit into the space occupied by the old file. In this case you will have to create a completely new file for it. If, when you first save a program, you anticipate that it may have to be modified several times before it is in a

final form, you can save it on the disk and specify that extra sectors be allocated to the file space to allow room for future modifications. For example, this statement

> :SAVE DCF(4) "INVT1"

saves the program currently in memory onto the disk in the next available sectors, and includes in the file 4 extra sectors. If this file is subsequently scratched, a program could be saved over it which would require as much as four sectors in addition to those required by INVT1. This allows a program to be modified and expanded, and saved back into the same area as was occupied by it prior to modification. Any number of additional sectors may be specified for a file, provided that the resultant file fits into the catalog area.

Review of Section 4-3

1. Though there are two different kinds of disks and disk drives available with Wang systems, they are identical from the point of view of programming.

2. A disk device provides direct access to the information stored on it.

3. In a process called "formatting" the recording area of a blank disk is divided into many small units called sectors. Each sector is assigned an identifying number called an address.

4. Your Wang system will automatically maintain a list of files saved on a disk together with locations occupied by each file. However, the catalog index and catalog area must first be established on a disk with the SCRATCH DISK instruction.

5. To save a program on a cataloged disk, the SAVE DC command is used. It has the following general form:

$$:SAVE\ DC \begin{Bmatrix} F \\ R \end{Bmatrix} "name"$$

where F or R specifies the individual disk at the selected address to be used, and "name" is the name of the program file (up to 8 characters enclosed in quotation marks).

6. To load a program from a cataloged disk, the LOAD DC command is used. It has the following general form:

$$:LOAD\ DC \begin{Bmatrix} F \\ R \end{Bmatrix} "name"$$

where F or R specifies the individual disk from the selected address to be used, and "name" is the name given to the program during the SAVE DC operation.

7. A list of the contents of a disk catalog index can be obtained by executing a LIST DC command, the general form of which is:

$$:LIST\ DC \begin{Bmatrix} F \\ R \end{Bmatrix}$$

where F or R specifies the individual disk from the selected disk address.

8. A program file can be marked as obsolete (scratched) by executing a SCRATCH statement. The SCRATCH statement has the following general form:

$$SCRATCH \begin{Bmatrix} F \\ R \end{Bmatrix} "name"$$

where F or R specifies the individual disk at the selected address, and "name" is the name of the file to be scratched.

9. Additional parameters can be included in the SAVE DC command to specify that the program to be saved is to replace a scratched program already on the disk, or to specify a number of sectors to be reserved in addition to those required to store the program. With these additional parameters SAVE DC is written as follows.

$$SAVE\ DC \begin{Bmatrix} F \\ R \end{Bmatrix} (expression)\ "name"$$

expression stating number of extra sectors.

$$SAVE\ DC \begin{Bmatrix} F \\ F \end{Bmatrix} ("old\ file")\ "new\ file"$$

name of scratched file to be replaced by "new file".

name for program in memory to be saved.

70

CHAPTER 5
SELECT STATEMENTS AND
THE USE OF A PRINTER

5-1   INTRODUCING DEVICE SELECTION

In the examples we have considered thus far, whenever a PRINT statement
has been executed, the output has appeared on the CRT. If you have a printer,
you may be wondering how you can make your results appear on it instead.
Before we can answer this, we must make explicit some things that we have been
able to take for granted.

Consider your system as consisting of a central processor (CPU) and a
collection of input and output (I/O) devices. A CRT and a printer are output
devices; a keyboard an input device. Tapes and disks can act as both; they
input to the CPU, and also receive output from it. Each I/O device in your
system has a three-character address that identifies it for the CPU.   An
address is assigned to a device by your Wang Service Representative at the
time the device is installed. Thereafter, the address may be considered a
permanent characteristic of the device.

Certain BASIC statements and commands cause the CPU to send information
to an output device, or receive it from an input device. These instructions
can be called I/O instructions.   PRINT and INPUT are examples of I/O
statements.   Similarly, LIST is a command which initiates an I/O operation.
All I/O operations are grouped into classes and, whenever the CPU is to
perform an I/O operation, it first checks to see what device address has been
selected for the class of I/O operation involved. It then performs the I/O
operation at the selected device address.

How have we been able to get along so far without selecting I/O devices?
During Master Initialization (power on), the CPU automatically selects a
device address for each class of operation. Unless you select different
device addresses yourself, the CPU will use the device address it has
selected.   These addresses, which are selected by Master Initialization, are
known as the default addresses.

The default address for PRINT operations is 005, the address of the CRT.
Therefore, the PRINT statements in our programs have caused output to go to
the CRT. The LIST default address is also 005.

The SELECT Statement

To cause PRINT output to appear on a printer, you must first execute a
SELECT statement that substitutes the address of your printer for the

currently selected PRINT address. For example, you might put at the beginning of your program

       10 SELECT PRINT 215

Thereafter, when any PRINT statement is encountered, the output will appear at address 215. (Address 215 is the normal address assigned to a 2221, 2221W, 2231W, 2231 or 2261 printer.) If, at some later point in the program, you want PRINT output to again appear on the CRT, you can write a statement such as:

      120 SELECT PRINT 005

This restores the CRT address, as the address for PRINT operations.

The SELECT statement can appear in a program with a line number, as shown above, but it also can be executed as if it were a command, without a line number. For example, if you key

      :SELECT PRINT 215 (EXEC)

the address 215 is substituted for the currently selected address for PRINT operations. This occurs immediately upon depressing (EXEC); the statement itself is not saved in memory.

To produce a printed listing of a program, you must first select the printer for LIST operations. To do this key

      :SELECT LIST 215 (EXEC)

Now, whenever you issue a LIST command, the output will appear at address 215. Note that the printer can be selected for LIST operations and the CRT for PRINT operations, or vice versa. As separate classes of I/O operations, PRINT and LIST are totally independent.

I/O Classes

So far we have seen two different classes of I/O operation, LIST and PRINT. These are referred to as "classes of I/O operations" because more than one BASIC statement or command may fall into a particular class. For example, the LIST DC command, which lists the contents of a disk catalog, causes output at the address selected for LIST operations.

You may be wondering how many different classes of I/O operations there are. In addition to LIST and PRINT, there are Console Input (CI), Console Output (CO), INPUT, DISK, TAPE, and PLOT. DISK and TAPE selection is discussed in the chapters that deal with these operations; PLOT is outside the scope of this manual.

The address specified for INPUT class operations is the address of the device from which the CPU will receive data when an INPUT class statement is executed. The default selection for INPUT operations is address 001, the keyboard. Though other devices, such as a card reader, can be used with INPUT statements, these devices are not discussed in this manual.

The INPUT statement always initiates two I/O operations. It always outputs a question mark in addition to receiving entered data. Furthermore, the received data itself is output; it appears as it is keyed in. The

question mark, prompt, and entered data are output at the address currently selected for Console Output (CO). The CRT, address 005, is the default selection for Console Output. Therefore, the prompt, question mark, and data from each of our example INPUT statements, have appeared on the CRT.

Console Output is a class of I/O operation which defines the output address for all command-generated messages (other than LIST), and for certain BASIC statements which generate operator messages. Included in this latter category, for example, is the question mark of the INPUT statement; in the former the

        READY
        :

of RESET. A complete list of CO class output is given in Appendix D.

The default address for Console Input (CI) is 001, the keyboard. Console Input is a class of input operations that takes place whenever the colon (:) is displayed, such as inputting statement lines and commands.

Line Length

For Console Output, PRINT, and LIST class I/O operations a line length can be specified with the SELECT statement. For example, if your system is equipped with a 2221W printer, which has a line length of 132 characters, you might wish to write a statement such as

        :10 SELECT PRINT 215 (132)

This will select address 215 for PRINT operations, and specify the line length of the device as 132. The effect of specifying 132 is that during a PRINT statement, the system will issue a carriage return when the next print element will cause the total line length to exceed 132 characters. During Master Initialization the line length is set to 64, the width of the CRT. Unless a line length is specified in a SELECT statement, the line length remains at its previous value.

Summary

The general form of the SELECT statement, insofar as we have considered it here, is

        SELECT select parameter [,select parameter...]

where select parameter =
$\begin{cases} \text{CI} & \text{device address} \\ \text{CO} & \text{device address } [(length)] \\ \text{LIST} & \text{device address } [(length)] \\ \text{PRINT} & \text{device address } [(length)] \\ \text{INPUT} & \text{device address} \\ \text{DISK} & \text{device address} \\ \text{TAPE} & \text{device address} \\ \text{PLOT} & \text{device address} \end{cases}$

and

        length = an integer specifying the desired line length $<$ 256.

73

Notice that a single SELECT statement can effect more than one selection. For example, you can write

      10 SELECT PRINT 215 (132), CO 215 (132)

which selects address 215 for PRINT and CO operations, with a line length of 132 characters.

```
+-------------------------------------------------------+
|                        NOTE:                          |
|                                                       |
| If you select a non-existent device for  CI  or  CO  class |
| operations,  or an invalid (non-input) device for CI, your |
| system  becomes  locked  out.  It  must  then  be  Master |
| Initialized,  which  will  clear  all  program  text  and |
| variables.                                            |
+-------------------------------------------------------+
```

Standard device addresses for the peripherals discussed in this manual are given below:

I/O DEVICE CATEGORY	STANDARD ADDRESSES
KEYBOARD	001,002,003,004
CRT	005,006,007,008
TAPE CASSETTE DRIVES	10A,10B,10C,10D,10E,10F
LINE PRINTERS (Models 2221, 2231,2261,2221W)	215,216
OUTPUT WRITER (Model 2201)	211,212
DISK DRIVES	310,320,330

Normally the first address shown is assigned to the first device of that category.   For example, if a system contains only one keyboard and one CRT, they are normally assigned addresses 001 and 005 respectively.

Selected addresses remain selected until replaced in any of three ways:

1) Another SELECT statement is executed for that class of I/O operations.

2) The system is Master Initialized. This reselects all the default addresses.

3) CLEAR (EXEC) is executed. This selects the current Console Input (CI) device for INPUT class operations, and the current Console Output (CO) device for PRINT and LIST class operations.   The other I/O class operations, which have been introduced in this section, remain unchanged.

5-2   USING A PRINTER

In the last section we introduced the I/O addressing scheme of a Wang 2200 system.   In this section we want to briefly consider some of the more important characteristics of printers. If your system is equipped with a Model 2221W, 2221, 2231, 2231W or 2261 printer, read the section below dealing

with the 2221W printer. A separate section is devoted to the 2201 Output Writer.

Using The 2221W Printer

As we discussed in the last section, the 2221W printer must be selected for output before it can be used. Normally, in a system with one printer, the 2221W is assigned address 215; therefore, to select the printer for LIST and PRINT operations, you can execute a statement such as

      :SELECT PRINT 215 (132), LIST 215 (132)

This SELECT operation readies the printer only from the programming standpoint. Physically, paper must be mounted, the printer power must be turned on, and the manual SELECT switch on the printer must be depressed to ready the printer to receive output. Do not confuse the functions of the SELECT switch on the printer with the SELECT instruction. The printer cannot print unless the SELECT switch is on (illuminated), but its being on or off has nothing to do with whether the address of the printer has been selected for any class of I/O operations. For information about physically readying the printer for operation, see your printer reference manual. One programming note, though: you shouldn't select a line length greater than the width of the currently mounted paper.

Once a printer is selected for LIST operations, if you enter a program and key

      :LIST (EXEC)

the program listing is printed on the printer, instead of appearing on the CRT. It appears exactly as it appears on the CRT, except that a line of more than 64 characters can be continued on a wider line, instead of being broken.

There is one major difference between the way a 2221W type printer outputs PRINT statements, and the way the CRT does. To appreciate this, enter and execute the following program.

```
10 SELECT PRINT 005
20 N=N+1
30 PRINT 2↑N;
40 IF N<20 THEN 20
```

the result appears as follows on the CRT.

```
2  4  8  16  32  64  128  256  512  1024  2048  4096  8192
16384  32768  65536  131072  262144  524288  1048576
```

Now change statement 10 to

      10 SELECT PRINT 215 (132)

Execute the program with full-width paper mounted. Nothing is printed.

Nothing is printed because the printer doesn't print each character as it receives it. It waits until the CPU tells it that an entire line is complete, meanwhile keeping the characters it receives in a buffer. The

75

buffer is simply a place for temporarily storing characters, until the signal to print the line is received. The signal it awaits from the CPU is a carriage return code.

Looking back at our example, we see that since line 30 ends with a semicolon (;), and the line length does not exceed 132 characters, a carriage return is never issued by the CPU; hence, the buffer contents are never printed.

To print the buffer, which still contains the output from this program, add this line to your program

50 PRINT

and key RUN 50 (EXEC).

Statement 50 issues a carriage return code. This code says to the printer, "print the contents of the buffer and advance the paper one line". In this case, the buffer contents had been there since the last execution of the program. If you rerun the program with RUN (EXEC), the addition of statement 50 will now let the program function as it should.

As a programmer it is your responsibility to ensure that the output from a PRINT statement is actually printed. Problems are most likely to occur when PRINT statements with trailing element separators are executed within a loop, such as in the above example. By contrast any PRINT statement without a semicolon or comma at the end will always print the buffer.

Now change the example program to print in zone format

```
10 SELECT PRINT 215 (132)
20 N=N+1
30 PRINT 2↑N,
40 IF N<20 THEN 20
50 PRINT
```

The output from this appears in 8 zones across the width of the printer. Just as with the CRT, the zones are 16 characters wide, only now there are more of them. If line 50 is omitted from this program, two lines of output will be printed, because the CPU automatically issues a carriage return when a line is filled; however, without statement 50 to output the carriage return code, the last line is not printed.

The 2201 Output Writer

The 2201 Output Writer must be selected for output before it can be used. Normally in a system with just one 2201, the 2201 is assigned address 211. Therefore to select the 2201 for LIST and PRINT operations, you can execute a statement such as:

:SELECT PRINT 211(157), LIST 211(157)

The carriage width of the 2201 is 157 characters; therefore, 157 is the maximum line length which should be used in a SELECT statement.

76

The select operation shown above readies the output writer only from the programming standpoint. Physically, paper must be mounted, power must be turned on, and the MANUAL/AUTO switch set to AUTO. For information about physically readying the Output Writer to receive output, see the 2201 Reference Manual.

With the Output Writer selected for LIST operations, if you enter a program and key

> :LIST (EXEC)

your program listing is typed on the Output Writer.

The standard typing element on the 2201 is Prestige Elite 72. The character set of this element causes three characters to print differently on the Output Writer than they do on the CRT.

CRT	OUTPUT WRITER
<	[
>	]
↑	!

The 2201 Output Writer has a left margin control above the keyboard. This can be used to set the left margin to any carriage position. However, if it is set to any position other than 0, during AUTO operation the TAB() parameter will not tab to column positions as marked on the front of the 2201. Rather, the position of the left margin barrier will function as column position 0, and TAB() operations will be displaced accordingly.

Review of Chapter 5

1. Every input or output device in your system has a unique "device address". The three character device address is the device's identification for the processor.

2. Device addresses are set by your Wang Service Representative at the time the system is installed. Conventional device addresses are:

CRT	005
Keyboard	001
Tape Cassette Drives	10A, 10B, 10C...10F
Printer	215
Output Writer	211
Disk Drives	310, 320, 330

3. Each input or output operation must take place at a particular device address.

4. For the purpose of determining the device address for an operation, input and output operations are grouped into classes. Each class has a device address associated with it. When the processor must execute an input or output operation, it determines what class the operation falls

77

into and executes the operation at the device address associated with that class. The I/O classes are:

```
Console Input (CI)        LIST
Console Output (CO)       PRINT
TAPE                      PLOT
DISK                      INPUT
```

5. During Master Initialization, the processor automatically associates device addresses with each I/O class. At any time the user may change the device address associated with a particular I/O class. This is done by means of a SELECT statement. For example,

    10 SELECT LIST 215

6. To cause output from the PRINT statement to appear on a printer rather than a CRT, a SELECT statement such as

    SELECT PRINT 215

    must be executed.

7. On the 2221W, 2231W, 2231, 2221, and 2261 model printers, a line is printed only when a carriage return code is received, or the line is filled. Therefore, if PRINT statements end with a comma or semicolon, a blank PRINT statement may be needed to issue the carriage return code.

8. Line length can be changed by including a line length parameter in the SELECT statement. For example,

    10 SELECT PRINT 215 (132)

    132 is the line length.

CHAPTER 6
FUNCTIONS

6-1   INTRODUCTION

Wang 2200 systems have a built-in capability to evaluate a variety of mathematical functions. Each function is evaluated for a single given quantity called its "argument". For example, the function SQR() yields the square root of its argument. If an argument of 25 is supplied, the function

    SQR(25)

is equal to 5.

Functions are not BASIC statements by themselves; rather, they can be used within a BASIC statement wherever an expression can be used. The argument value of the function is supplied by an expression. Therefore, a function can appear within the argument of another function. For example,

    LOG(2+TAN(A))

There is no limit to this "nesting" of functions.

Additional functions, to supplement the built-in functions, can be created in a program by using the "define function" statement, DEFFN.

The built-in functions that find general use in both commercial and technical applications are discussed in Section 6-2. Most of the functions discussed in Sections 6-3 and 6-4 are principally used in programming technical applications. Section 6-5 introduces the DEFFN statement, and should be of general interest.

6-2   THE INTEGER, ABSOLUTE VALUE, AND SIGN FUNCTIONS

The Integer Function

The form of the integer function is:

    INT (expression)

The INT function yields the whole number (integer) with the greatest value less than or equal to the value of the expression. For example, INT(2.5) is equal to 2; INT(14.76) is equal to 14. Carefully examine the results of this program:

```
10 PRINT "LINE 10", INT(3.5), INT(2*3.5), INT(3.5)*2
20 PRINT "LINE 20", INT (-3.6)
30 PRINT "LINE 30", INT(8)
```

yields

```
LINE 10      3     7     6
LINE 20     -4
LINE 30      8
```

Notice in particular that INT(-3.6) is -4 not -3.

The Absolute Value Function

The form of the Absolute Value Function is:

ABS (expression)

The ABS function yields the absolute value of the expression. Absolute value of Q is the value of Q if Q is zero or positive, and -Q if Q is negative.

For example,

```
10 PRINT ABS(-4.92), ABS(4.92)
20 PRINT ABS(3*-4.2*2)
30 PRINT ABS(0)
```

yields

```
4.92        4.92
25.2
0
```

The Sign Function

The form of the sign function is

SGN (expression)

SGN() is defined by

```
If Q > 0, SGN(Q) yields  1
If Q = 0, SGN(Q) yields  0
If Q < 0, SGN(Q) yields -1
```

For example,

```
10 PRINT SGN(-3.1416), SGN(7*8-56)
20 PRINT SGN(11370.2)
```

yields

```
-1       0
 1
```

Some Simple Uses of INT, ABS, and SGN Functions

In your Wang system division is always carried out to the full 13 digits of precision. Sometimes, however, you may want a whole number result (quotient) and a remainder. For example, when dividing 19 by 3, you may want an answer such as "6, remainder 1" rather than 6.333333333333.

The problems that require this type of result are often called problems in "modulo arithmetic". To choose a very simple example, suppose you want to be able to enter some number of inches, and then convert the entered number to feet and inches. If you simply divide the entered number by 12, you will obtain a whole number and decimal fraction, whenever the entry isn't a multiple of 12. What you want is the whole number portion of this quotient and a whole number remainder. The following program uses INT to recover the integer portion of the quotient, and then uses that result to calculate the remainder.

Example 6.1 Using INT() to Obtain Quotient and Remainder

```
10 REM ILLUSTRATION OF USE OF INT FUNCTION
20 REM CONVERT INCHES TO FEET/INCHES
30     INPUT "NO. OF INCHES", D
40 REM FEET =  INT OF (D DIVIDED BY 12)
50     F=INT(D/12)
60 REM EXPRESSION D-(12*F) IS THE REMAINDER IN INCHES
70     PRINT F; "FEET", D-(12*F); "INCHES"
```

The conversion program shown in Example 6.1 requires that a non-negative number of inches be entered.

In general, to obtain a whole number quotient, for quotients that may be either positive or negative, a means of simply truncating the decimal fraction is needed. INT() alone won't work. For example, if you divide 19 by -3, the quotient is -6.333333333333. However, INT (-6.333333333333) equals -7, because INT yields the largest integer less than or equal to the value. A means of simply cutting off the .333... is needed regardless of whether the value is positive or negative.

To do this truncation, take the absolute value of the quotient, Q

ABS(Q)

This yields a positive value. Then take the INT of this positive quantity

INT(ABS(Q))

Since the INT() argument is always positive, we can be sure that INT() simply cuts off the decimal fraction. Now the original sign of Q must be restored to the value INT(ABS(Q)). The SGN() function can be used. The expression INT(ABS(Q)) can simply be multiplied by SGN(Q). SGN(Q) is -1 if Q is minus. Multiplying the positive value INT(ABS(Q)) by -1 simply changes the sign. If Q is positive, SGN(Q) equals +1; multiplying by +1 will leave the value unchanged. Truncated Q, call it Q1, is as follows:

```
70 REM ASSIGN TRUNCATED VALUE OF Q TO Q1
80 Q1 = INT(ABS(Q))*SGN(Q)
```

The following program illustrates our generalized results.

Example 6.2   Integer Quotient and Remainder Using INT(), ABS() and SGN()

```
10 REM ILLUSTRATION OF INT, ABS, AND SGN FUNCTIONS
20 REM
30 REM FOR AN ENTERED DIVIDEND AND DIVISOR
40 REM RETURNS WHOLE NUMBER QUOTIENT AND REMAINDER
50 REM SIGN OF THE DIVIDEND IS THE SIGN OF THE REMAINDER
60 REM ****    D = DIVIDEND
70 REM ****    DO = DIVISOR
80 REM ****    Q  = QUOTIENT
90 REM ****    Q1 = TRUNCATED QUOTIENT
100 REM ****    R  = REMAINDER
110 REM
120 REM
130    INPUT "DIVIDEND, DIVISOR", D,DO
140    Q=D/DO
150 REM TRUNCATE THE QUOTIENT
160    Q1=INT(ABS(Q))*SGN(Q)
170 REM CALCULATE REMAINDER
180    R=D-(DO*Q1)
190 REM OUTPUT RESULTS
200    PRINT
210    PRINT "DIVIDEND", D
220    PRINT "DIVISOR", DO
230    PRINT "QUOTIENT",Q1
240    PRINT "REMAINDER",R
250    PRINT "PROOF",R+Q1*DO
```

ABS() IN IF...THEN

In Section 3-5 we mentioned the possibility of exploiting the logic of arithmetic in an expression to test "multiple" conditions in an IF...THEN statement, and gave as an example:

```
50 IF W*X*Y*Z = 0 THEN 480
```

This is the equivalent of saying "If W or X or Y or Z equals 0 then 480." Suppose, though, we wanted to effect a branch only if all the variables are zero, i.e., "If W and X and Y and Z equal 0." With the absolute value function and addition we can conveniently simulate the "and" connective as follows:

```
50 IF ABS(W)+ABS(X)+ABS(Y)+ABS(Z)=0 THEN 480
```

If we want to effect a branch to 480 on the condition "J=4 and K=6*Q", we can write the single statement

```
IF ABS(J-4)+ABS(K-6*Q)=0 THEN 480
```

Suppose we want to test a variable X, to see if it lies within the range -3 < X < 3, and branch to 200 if it is within this range. We could write

```
40 IF -3>=X THEN 60
50 IF X<3 THEN 200
60 REM OUTSIDE RANGE
:
:
200 REM WITHIN RANGE
```

However, using the ABS function we can perform this test in one statement:

```
40 IF ABS(X)<3 THEN 200
50 REM OUTSIDE RANGE
:
:
200 REM WITHIN RANGE
```

We are able to test in this fashion because the midpoint of the range lies exactly at zero. However, any continuous range can be "moved" so that its midpoint is 0, and then tested in this manner with the ABS function.

For example, suppose the conditions are the same as above except that the range is 1<=X<=6. The midpoint of this range is 3.5. In either direction from this midpoint, 2.5 units away, lies a range boundary (i.e., 3.5 + 2.5 = 6, 3.5 - 2.5 = 1). Therefore, for X-3.5 the boundaries are +2.5 and -2.5 with 0 as midpoint. Our test becomes

```
40 IF ABS (X -3.5)<= 2.5 THEN 200
50 REM OUTSIDE RANGE
:
:
200 REM WITHIN RANGE
```

In general, if the acceptable range of X is L < X < U, then the statement

```
50 IF ABS(X-(U+L)/2)<(U-L)/2 THEN 200
```

effects a branch to 200 if X is within the range.

6-3   π AND THE RANDOM NUMBER FUNCTION

The value π, to 13 significant digits, is permanently stored in the Wang 2200 system, and may be incorporated into any expression by depressing the key marked π or keying #PI. Regardless of how it is entered, it always appears on the screen as #PI. For example,

```
10 PRINT #PI, 4*#PI
```

produces

```
3.14159265359   12.56637061436
```

The Random Number Function

The random number function produces random values between 0 and 1. The form of the function is:

RND(expression)

In the RND function, there are only two significant argument values, zero and non-zero.

The RND() function may be thought of as a means for extracting a random number between 0 and 1 from a fixed "list" of such numbers. Each time RND() is executed, with any non-zero argument, the next number on the "list" is supplied. Thus, the first time RND() is executed after Master Initialization, with a non-zero argument, it yields the first random number in the list; the second time it yields the second, etc. The value of the argument is irrelevant to the value yielded by RND(). As long as the argument is non-zero, RND() gets the next number in its "list".

If the value of the RND() argument is zero, the "zeroth" random number is produced, the "list" pointer is reset, and the next non-zero argument RND() yields the first random number in the list.

The following program illustrates the operation of RND(0). The values produced by lines 10-30 presume that RND() has not been executed after Master Initialization.

```
10 PRINT RND(1)
20 PRINT RND(1)
30 PRINT RND(1)
40 PRINT RND(0)
50 PRINT RND(5)
60 PRINT RND(5)
70 PRINT RND(11)
```

Execution produces:

```
.22762279975
.39869185804
.391328921446
.89459771698
.22762279975
.39869185804
.391328921446
```

Notice that the first three values are identical to the last three, which follow RND(0). RND(0) has reset the "list".

RND(0) is useful in debugging programs that use RND() since it allows the same results to be produced each time the program is run.

Using RND() to Produce Random Integers

To produce a random integer R such that

$X \leq R \leq Y$

a statement of the form

    100 R=INT (RND(1)*(Y+1-X)+X)

can be used. For example, statement 80 generates a random integer, R, between 1 and 50:

    80 R = INT((RND(1)*49)+1)

6-4   THE TRIGONOMETRIC, LOGARITHMIC, AND SQUARE ROOT FUNCTIONS

The trigonometric, logarithmic, and square root functions are shown in the table below.

Function	Meaning
SIN(expression)	Find the sine of the expression.
COS(expression)	Find the cosine of the expression.
TAN(expression)	Find the tangent of the expression.
ARCSIN(expression)	Find the arcsine of the expression.
ARCCOS(expression)	Find the arccosine of the expression.
ARCTAN(expression)	Find the arctangent of the expression.
LOG(expression)	Find the natural logarithm of the expression.
EXP(expression)	Find the value of e raised to the value of the expression.
SQR(expression)	Find the square root of the expression.

For all the trigonometric functions the argument is treated as radians, unless degrees or grads has been selected in a SELECT statement. To SELECT degrees execute:

    :SELECT D

85

prior to calculations, or include SELECT D as a program statement before calculations. Degree measure is then assumed for all trig functions until a SELECT statement selects radians or grads, or the system is Master Initialized.

"R" and "G" in a SELECT statement specify radians and grads, respectively.

In addition to "ARCTAN", the notation ATN(expression) may also be used to specify the arctangent function.

For any value, V, V↑.5 is the equivalent of SQR(V); however, SQR(V) executes in slightly less time, and is more readable.

6-5   THE DEFFN STATEMENT

The "define function", DEFFN, statement allows you to define, within a program, additional functions of one variable beyond those discussed in the preceding sections. Defined functions may be used anywhere in the program, exactly the way the built-in functions are used.

For example, you might want to define the functions, hyperbolic sine, sinh, and hyperbolic cosine, cosh. This can be done as follows:

Example 6.3 Defining SINH and COSH with a DEFFN

```
1010 REM DEFINE HYPERBOLIC SINE (SINH)
1020 DEFFN S(X) = (EXP(X) - EXP(-X))/2
1030 REM DEFINE HYPERBOLIC COSINE (COSH)
1040 DEFFN C(X) =  (EXP(X) + EXP(-X))/2
```

The letter that follows the keyword "DEFFN", "S" and "C" in the example, is the name you give to the function. Any letter or digit can serve as a function name. To use a defined function in the program in which it appears, you refer to it by name. For example, FNC(8) would be the hyperbolic cosine, as defined at line 1040 above, at the argument value 8. FNS(25) is the hyperbolic sine of 25, defined at line 1020. In each case the letter following "FN" is the name of the function.

When the system encounters an FN reference to a defined function, it first evaluates the argument (any expression may be used), then finds the proper DEFFN and uses the argument value as the value of the dummy variable in the DEFFN. In the above example, the dummy variable is X.

The DEFFN statement can appear anywhere in a program, without regard to where the references to the function appear. When encountered in the normal sequence of execution, the DEFFN statement has no effect; it only comes into play when the function it defines appears in another statement.

In effect, the DEFFN statement saves you the trouble of constantly reentering frequently used expressions, and saves the memory space that these expressions would occupy.

The general form of the DEFFN is as follows:

DEFFN   a   (v)   =   expression

  Keyword   Function   Dummy
                Name   Variable

where: a is any letter or digit.

          v is any valid numeric variable form, i.e., A, Z, A0, Z9, etc.

The general form of the reference to a defined function is:

FN   a   (expression)

       Function   Expression whose value is given to the
       Name   dummy variable in the DEFFN statement.

The v in the DEFFN statement form is called a dummy variable because it is simply a place holder. Evaluation of the function has no effect on a true variable of the same name, used elsewhere in the program. Notice that in Example 6.3 the same dummy variable, X, is used in both DEFFN's.

The expression in the DEFFN statement may contain another defined function, provided that the other function does not refer back to it. The system can evaluate up to five levels of defined functions nested within defined functions.

Some of the operations described in Section 6-2 can be incorporated into DEFFN statements. For example, you might wish to define the truncate function (see lines 150 and 160 of Example 6.2).

```
90  REM TRUNCATE FUNCTION
100 DEFFN T(X) = INT(ABS(X))*SGN(X)
```

Many applications require that values be rounded to two decimal places. The following function accomplishes this:

Example 6.4 A DEFFN for Rounding to Two Decimal Places

```
70 REM ROUND TO TWO DECIMAL PLACES
80 DEFFN R(X) = SGN(X)*INT(ABS(X)*100+.5)/100
```

With this DEFFN in a program, to round any value in a variable, V, you can write a statement such as:

```
60 V = FNR(V)
```

or if the value of the variable is simply being printed, you could write

```
470 PRINT FNR(V)
```

Review of Chapter 6

1. Functions can be used within BASIC statements wherever expressions can be used.

2. Functions are evaluated at a single given quantity called the "argument". Any expression may be used to specify an argument.

3. Functions may be nested in the arguments of other functions. There is no limit to this nesting.

4. The value π may be used in an expression by keying π or #PI.

5. After Master Initialization, all trigonometric arguments are considered to be in radians. The unit of measure can be changed at any time with a SELECT statement.

    SELECT D selects degrees.
    SELECT R selects radians.
    SELECT G selects grads.

6. The DEFFN statement is used to define functions of one variable for use in a program. For example, this statement defines a function named "R", which rounds a value to two decimal places.

    80 DEFFN R(X) = SGN(X)*INT(ABS(X)*100+.5)/100

    This defined function can be used anywhere an expression can be used. It is used in this form

    FNR(expression)

    where R is the name of the function.

CHAPTER 7
LOOPS

7-1   THE PARTS OF A LOOP

A block of statements that is executed repeatedly is called a "loop". Loops are one of the most important and widely used program structures. The inventory program, the factorial program, and the powers-of-two program (Examples 2.1, 2.2 and 3.2) all contained loops.

Unlike the loops in factorial and powers-of-two programs, the inventory program loop never ends. It has no built-in "exit"; to end the program, you must key RESET. In this chapter we will be considering the more common type of loop, that has an "exit" built in. Let's look at a powers-of-two program again to see what makes up a loop.

Example 7.1   A Loop to Print Powers of Two

```
110 REM LOOP BEGINS
120 PRINT 2↑N;
130 N=N+1
140 IF N<=20 THEN 120
150 REM LOOP ENDED
160 PRINT "DONE"
```

Keying RUN (EXEC) sets N to zero. Statement 120 does all the "useful work" of the loop, which simply consists of printing the value $2↑N$. Statement 130 changes the value of N, so that the next time through the loop a new value will be printed. 140 asks, "Repeat the loop?" If the answer is "no", it lets the normal sequence of execution prevail, thereby providing an exit from the loop.

In general, four components of a loop can be distinguished, though not all loops exhibit all of them. The functional parts of a loop are:

1. Set-up: operations that take place before the loop actually begins, but which are necessary if the loop is to execute properly. This includes, principally, setting any loop counters to their proper initial values. In Example 7.1, RUN(EXEC) did this job.

2. Body of the Loop: consists of all the processing which is to be repeatedly performed. (Line 120 in Example 7.1.)

3. Modification of a Key Variable: at least one key variable, to be tested at "Test/Exit", is assigned a new value. Frequently this takes the form of adding a quantity to the key variable's old value, in which case the key variable is called a "counter." In Example 7.1 line 130 modifies the key variable N.

4. Test/Exit: the key variable or variables are tested to determine whether the loop should be repeated or exited. (Line 140 of Example 7.1.)

Let's look at a program that uses a loop, and clearly shows all four loop components. Suppose we want to see how varying the interest rate affects monthly mortgage payments for a given mortgage amount and repayment term. We would like the calculations to be performed for 5%, 6%, 7%...12% interest rates. The formula for calculating monthly payment is:

$$M = \frac{P\left(\dfrac{I}{1200}\right)}{1 - \left(1 + \dfrac{I}{1200}\right)^{-12T}}$$

The program shown in Example 7.2 achieves the desired results. The four parts of a loop are identified by REM statements with asterisks. Enter and execute the program. (If your system doesn't have a printer, eliminate statement 160.)

Example 7.2 Monthly Payments as Interest Varies from 5% to 12%

```
110 REM ($MORT1) MONTHLY PAYMENT FOR INTEREST 5% TO 12%
120 REM OPERATOR ENTERS VALUES FOR PRINCIPAL,  TERM
130      INPUT "ENTER PRINCIPAL",P
140      INPUT "ENTER MORTGAGE TERM IN YEARS",  T
150 REM PRINT HEADINGS
160      SELECT PRINT 215 (80)
170      PRINT ,, "INTEREST","MONTHLY"
180      PRINT "PRINCIPAL", "TERM","RATE","PAYMENT"
190      PRINT
200 REM ****  LOOP SET-UP   ****
210      I=5
220 REM ****  BODY OF LOOP   ****
230      M=P*(I/1200)/(1-(1+I/1200)↑(-12*T))
240         REM ROUND M TO 2 DECIMAL PLACES
250            M2 = SGN(M)*INT(ABS(M)*100+.5)/100
260            PRINT "$";P,   T;"YEARS",   I; "%",    "$"; M2
270         REM ****  MODIFY KEY VARIABLE  ****
280            I=I+1
290         REM **** TEST/ REPEAT LOOP?  ****
300      IF I < = 12 THEN 230
310 REM LOOP COMPLETE
320      PRINT
330      PRINT "----- DONE -----"
340      SELECT PRINT 005 (64)
```

In the body of the loop, statement 230 uses the formula given above to calculate the monthly payment, and assigns the calculated amount to M. 250 then rounds M to 2 decimal places using the formula given in Section 6-5. These two operations could have been combined into one; the resultant expression could have been placed as the last print element in 260. However, the combined formulas would have been cumbersome and confusing. The following is an example of the program's output:

PRINCIPAL	TERM	INTEREST RATE	MONTHLY PAYMENT
$ 20000	30 YEARS	5 %	$ 107.36
$ 20000	30 YEARS	6 %	$ 119.91
$ 20000	30 YEARS	7 %	$ 133.06
$ 20000	30 YEARS	8 %	$ 146.75
$ 20000	30 YEARS	9 %	$ 160.92
$ 20000	30 YEARS	10 %	$ 175.51
$ 20000	30 YEARS	11 %	$ 190.46
$ 20000	30 YEARS	12 %	$ 205.72

----- DONE -----

In Examples 7.1 and 7.2 the key variable is a kind of counter. It "counts" the number of times the loop has been processed. In general, if a fixed quantity (positive or negative) is added to a variable each time through a loop, and the value of this variable determines when processing is complete, then the variable is called a "counter". Loops controlled by counters are so frequently used, that BASIC has a pair of statements designed to make it easy to program them. These statements are FOR...TO and NEXT.

7-2   CONTROLLING LOOPS WITH FOR...TO AND NEXT

The FOR...TO and NEXT statements are always used together. They make programming counter controlled loops an easy, straightforward operation. The program shown in Example 7.3 performs identically to the print powers of two program shown in Example 7.1 It uses FOR...TO and NEXT to control the loop.

Example 7.3 Powers of Two Using FOR...TO and NEXT

```
110 REM LOOP BEGINS
120     FOR N = 0 TO 20
130         PRINT 2↑N;
140     NEXT N
150 REM LOOP ENDED
160     PRINT "DONE"
```

FOR...TO and NEXT mark the boundaries of the loop. FOR...TO specifies the counter variable, assigns its initial value, and specifies the range of values over which the loop is to be repeated. NEXT represents the end of the loop, and decides whether or not to repeat the loop. If the loop is to be repeated, NEXT adds 1 to the counter variable, and branches back to the statement following FOR...TO.

91

Let's look at the step-by-step execution of the loop in Example 7.3. The FOR...TO statement at line 120 of the program does this:

1. It designates N as the variable that contains the counter.

2. It assigns zero to N. Zero is N's initial value for the loop.

3. It says this loop is to be repeated as long as N is less than or equal to 20. It saves this information in a special part of memory for use by the NEXT statement.

Statement 130 makes up the body of the loop. Of course, in other programs the processing that takes place inside a loop could require many statements. An unlimited number of statements may appear within a FOR... TO/NEXT loop. (A FOR...TO/NEXT loop may contain branching statements, and may even contain other loops within itself. We'll look at these two possibilities in Section 7-4.)

Despite its simple appearance, the NEXT statement completely controls the repeated execution of the loop. The keyword "NEXT" is always followed by the variable that is being used as the loop counter. The variable serves as a place to keep the counter and as a name pointing to the loop's beginning. Thus, the "N" in NEXT N says, "This loop has its counter in N and begins at the FOR N=... TO... statement."

When the system executes the NEXT statement of Example 7.3, it does this:

It evaluates N+1.

1. If N+1 is less than or equal to 20, the value N+1 is assigned to N and a branch to line 130 is made. Note that the branch is made to the statement which follows the FOR...TO statement.

2. If the sum N+1 is greater than 20, NEXT N decides "The looping is complete". Since the data saved by step 3 of the FOR...TO statement is no longer needed, NEXT clears it from the "special part of memory" where it was being saved. NEXT then exits the loop by letting the normal sequence of execution prevail. (Notice that when the loop ends, the value N+1 is not assigned to N; N retains the last value it had which was less than or equal to 20.)

We can see that the single statement NEXT N in Example 7.3 achieves the same result as statements 30 and 40 of Example 7.1. NEXT N can execute only because the FOR...TO statement has saved the maximum value of the counter, and marked the beginning of the loop. NEXT can never be used alone; the system must always have previously executed a FOR...TO.

You may notice that FOR...TO and NEXT didn't eliminate any statements in this program. This is true because the initial value of N is 0, which let Example 7.1 depend upon RUN (EXEC) to do the set-up. Usually, this is not the case. Example 7.4, which uses FOR...TO and NEXT in the monthly payment problem, illustrates the real convenience of these statements. However, an important feature of FOR...TO is that it says clearly to anyone looking at the program listing, "A loop begins here, and is executed this many times."

Now look at how the problem of Example 7.2 is solved in Example 7.4 using FOR...TO and NEXT. Once again, we've marked off the parts of the loop with asterisked REM statements.

Example 7.4 Monthly Payment Problem with FOR...TO and NEXT

```
100 REM ($MORT2)  MONTHLY PAYMENT FOR INTEREST 5% to 12%
120 REM OPERATOR ENTERS VALUES FOR PRINCIPAL, TERM
130     INPUT "ENTER PRINCIPAL",P
140     INPUT "ENTER MORTGAGE TERM IN YEARS", T
150 REM PRINT HEADINGS
160     SELECT PRINT 215 (80)
170     PRINT ,, "INTEREST", "MONTHLY"
180     PRINT "PRINCIPAL", "TERM", "RATE", "PAYMENT"
190     PRINT
200 REM ****  LOOP SET-UP  ****
210     FOR I = 5 TO 12
220         REM ****  BODY OF LOOP  ****
230             M=P*(I/1200)/(1-(1+I/1200)↑(-12*T))
240             REM ROUND
250             M2 = SGN(M)*INT(ABS(M)*100+.5)/100
260             PRINT "$";P,   T;"YEARS",   I; "%",   "$"; M2
270         REM ****  MODIFY KEY VARIABLE AND TEST / EXIT ****
280     NEXT I
310 REM LOOP COMPLETE
320     PRINT
330     PRINT "-----  DONE  -----"
```

In Example 7.4 the statement FOR I = 5 TO 12 designates the variable I as the counter, and assigns it an initial value of 5. It also saves in a special part of memory the information, "The following loop is to be executed until I is greater than 12." The FOR I = 5 TO 12 statement sets the stage, both for the first time through the loop, and for the successful operation of the NEXT I statement.

The body of the loop hasn't changed at all from Example 7.2 to Example 7.4, but the REM's and spacing have been altered slightly to accord with conventional indentation.

The operation of testing the key variable and modifying it are now performed by the single NEXT I statement. This replaces lines 280 and 300 of Example 7.2.

Reversed Direction in the FOR...TO Statement

Suppose that in Example 7.4 we accidentally reversed the FOR... TO statement so that it read,

    210 FOR I = 12 TO 5

This statement is acceptable to the system, but as it stands, it doesn't make much sense. By repeatedly adding 1 to an initial value of 12, the counter would never get to 5. The normal processing of FOR...TO and NEXT will cause this "loop" to be executed once. The FOR statement assigns 12 to I, and saves the information for NEXT that the loop is to be repeated until I is greater

than 5. The body of the loop will process normally with I=12. When the NEXT I statement is encountered, it tells the system, "If I + 1 is greater than 5, looping is complete; continue with the normal sequence of execution." Of course, since I was set to 12 at the beginning, I + 1 is immediately greater than 5. Thus, the "loop" is executed just once.

On the other hand, suppose that you really want the counter to assume decreasing values from 12 to 5 in steps of -1. This is a plausible operation, and one which is easily performed by adding the STEP specification to the FOR...TO statement. STEP is discussed in Section 7-3.

Exit Values of the Counter Variable

The programs shown in 7.3 and 7.4 do not do exactly what 7.1 and 7.2 do, though in terms of the purpose of 7.1 and 7.2, they are the same. To appreciate the difference, add to Examples 7.1 and 7.3 the line

        170 PRINT N

Now execute the programs. You will find that 7.1 prints "DONE" followed by 21, and 7.3 prints "DONE" followed by 20. The reason for this is that NEXT, used in 7.3, first compares N+1 to 20, and only if the loop is to continue, that is, only if N+1< =20, does it assign N+1 to N. By contrast 7.1 first assigns the new value, then tests if, for the new N, N <=20. To do exactly what the NEXT does in 7.3, 7.1 would have to look like this:

Example 7.5 Exact Duplication of NEXT Operation

```
110 REM LOOP BEGINS
120     PRINT 2 N;
130     IF N+1 > 20 THEN 170
140     N=N+1
150     GOTO 120
160 REM LOOP ENDED
170     PRINT "DONE"
```

Questions For Review

1.   How many times is statement 30 executed in this program?

```
10 REM PRINT SUM OF INTEGERS 6 TO 16
20 FOR K=6 TO 16
30     T=K+T
40 NEXT K
50 PRINT T
```

                                        Answer: 11 times.

2.   How many times is statement 20 executed in the above program?

                                        Answer: once.

3.   If statement 20 in the above program were changed to:

     20 FOR K=-16 TO -6

how many times will statement 30 execute?

                        Answer: 11 times.

4.   If statement 20 were changed to

     20 FOR K=-6 TO -16

how many times will 30 execute?

                        Answer: once.

5.   What would be printed if we added the line

     60 PRINT "VALUE OF K="; K

to this program?   Answer: VALUE OF K = 16.

7-3   STEP and the General Form of the FOR...TO Statement

STEP

In the example programs of the last section, the NEXT statement added 1 to the counter, each time through the loop. Though this is the most common programming requirement, often it is desirable to have some other value added to the counter each time through. This can be accomplished by adding a STEP specification to the FOR...TO statement. When a STEP specification is added, the value that follows the keyword "STEP" is added to the counter each time through the loop.

Example 7.6 is the same as Example 7.4, except that the FOR...TO statement at line 210 has been replaced by:

     210 FOR I=12 TO 5 STEP-1

Example 7.6   Illustration of STEP

```
110 REM ($MORT6) MONTHLY PAYMENT FOR INTEREST 12% TO 5%
120 REM OPERATOR ENTERS VALUES FOR PRINCIPAL, TERM
130      INPUT "ENTER PRINCIPAL",P
140      INPUT "ENTER MORTGAGE TERM IN YEARS", T
150 REM PRINT HEADINGS
160      SELECT PRINT 215 (80)
170      PRINT ,,"INTEREST","MONTHLY"
180      PRINT "PRINCIPAL", "TERM","RATE","PAYMENT"
190      PRINT
200 REM **** LOOP SET-UP ****
210      FOR I = 12 TO 5 STEP -1
220          REM **** BODY OF LOOP ****
230              M=P*(I/1200)/(1-(1-I/1200)↑(-12*T))
240              REM ROUND
250              M2 = SGN(M)*INT(ABS(M)*100+.5)/100
260              PRINT "$";P,   T;"YEARS",    I; "%",    "$"; M2
270          REM ****  MODIFY KEY VARIABLE AND TEST / EXIT ****
280      NEXT I
310 REM LOOP COMPLETE
320      PRINT
330      PRINT "-----  DONE  -----"
```

The FOR...TO statement does the loop set-up. It basically says this: "For the first time through, set I at 12. Then, before each repetition of the loop, NEXT should subtract 1 from I. If I-1 is less than 5, it should discontinue looping." Thus, I, interest, assumes the successive values 12, 11, 10, 9, 8, 7, 6, 5. When executed, this program produces the following results:

PRINCIPAL	TERM	INTEREST RATE	MONTHLY PAYMENT
$ 35000	20 YEARS	12 %	$ 385.38
$ 35000	20 YEARS	11 %	$ 361.27
$ 35000	20 YEARS	10 %	$ 337.76
$ 35000	20 YEARS	9 %	$ 314.9
$ 35000	20 YEARS	8 %	$ 292.75
$ 35000	20 YEARS	7 %	$ 271.35
$ 35000	20 YEARS	6 %	$ 250.75
$ 35000	20 YEARS	5 %	$ 230.98

-----   DONE   -----

What has changed in the operation of FOR...TO/NEXT with the addition of the STEP parameter?

1.   The FOR...TO statement saves the STEP value -1 in the special part of memory for use by NEXT. (Without STEP, NEXT will assume the value to be added is +1.)

2. Since the sign of the STEP value is negative, the value after the keyword "TO" will be approached from the high side, as the loop is executed. That is, I approaches 5 in the following manner: 12, 11, 10 ... 7, 6, 5. Therefore, NEXT I will test if I+(-1) is less than 5 to see if the looping is complete. (Recall that in the examples of the last section, which had an implied step of +1, NEXT I tested if I+1 was greater than the value which followed the keyword "TO".)

Notice that NEXT gets all its signals from the FOR...TO statement; line 280 is the same in Examples 7.4 and 7.6. FOR...TO...STEP does all the setup work for the loop; NEXT does all the processing.

Fractional STEP Values

Now lets look at another use of the STEP function. Suppose, instead of the mortgage table produced by Example 7.4, we would like to calculate the monthly payment for each 1/4 of 1% increase in interest from 7% to 9%. Changing line 210 is again all that's needed. The following substitution does the job:

    210 FOR I=7 TO 9 STEP .25

(If you save this program, don't forget to change the REM statement at line 110.   REM's should always be accurate for the program version in which they appear.)

The results of the modified program look like this:

PRINCIPAL	TERM	INTEREST RATE	MONTHLY PAYMENT
$ 30000	20 YEARS	7 %	$ 232.59
$ 30000	20 YEARS	7.25 %	$ 237.11
$ 30000	20 YEARS	7.5 %	$ 241.68
$ 30000	20 YEARS	7.75 %	$ 246.28
$ 30000	20 YEARS	8 %	$ 250.93
$ 30000	20 YEARS	8.25 %	$ 255.62
$ 30000	20 YEARS	8.5 %	$ 260.35
$ 30000	20 YEARS	8.75 %	$ 265.11
$ 30000	20 YEARS	9 %	$ 269.92

    ----- DONE -----

The General Form of the FOR...TO Statement

The general form of the FOR...TO statement is

    FOR v = expression TO expression [STEP expression]
    where v = a numeric variable.

97

Variables and Complex Expressions in the FOR...TO Statement

Thus far, all of the examples have used constants to specify the range of the loop and the STEP value; however, as can be seen from the above general form, any expression can specify these values. The FOR...TO statement evaluates the expression, and sets up a loop with the resultant values.

For a simple application of this, we can put variables into the FOR...TO statement of our mortgage problem. If we then let the operator enter the values of these variables, the program is considerably more flexible. This program is shown in Example 7.7.

Example 7.7 Using Variables in the FOR...TO Statement

```
110 REM ($MORT7) MONTHLY MORTGAGE PAYMENT - ANY INTEREST RATES
120 REM OPERATOR ENTERS VALUES FOR PRINCIPAL, TERM
130      INPUT "ENTER PRINCIPAL",P
140      INPUT "ENTER MORTGAGE TERM IN YEARS", T
150 REM OPERATOR ENTERS STARTING RATE, INCREMENT, ENDING RATE
160      INPUT "ENTER STARTING INTEREST PERCENTAGE", A
170      INPUT "ENTER THE INTEREST INCREMENT", C
180      INPUT "ENTER ENDING INTEREST PERCENTAGE", B
190 REM PRINT HEADINGS
200      SELECT PRINT 215 (80)
210      PRINT ,, "INTEREST","MONTHLY"
220      PRINT "PRINCIPAL", "TERM","RATE","PAYMENT"
230      PRINT
240 REM ****  LOOP SET-UP  ****
250      FOR I = A TO B STEP C
260          REM ****   BODY OF LOOP  ****
270              M=P*(I/1200)/(1-(1+I/1200)↑(-12*T))
280              REM ROUND
290              M2 = SGN(M)*INT(ABS(M)*100+.5)/100
300              PRINT "$";P,   T;"YEARS",    I;"%",    "$"; M2
310          REM ****  MODIFY KEY VARIABLE AND TEST / EXIT ****
320      NEXT I
330 REM LOOP COMPLETE
340      PRINT
350      PRINT "-----  DONE  -----"
```

The values of A, B, and C are entered at lines 160 to 180 and used in the FOR...TO statement at line 250.

In a similar manner FOR...TO and NEXT can be used in a slightly modified version of the factorial problem. In Example 7.8 the upper value is entered into N, which is then used as the TO expression.

Example 7.8 Printing a Factorial Table Using FOR...TO and NEXT

```
110 REM FACTORIALS FROM 1! TO N! USING "FOR...TO" , "NEXT"
120  INPUT   "COMPUTE P!  FOR P=1 TO P=",  N
130  F=1
140  FOR P=1 TO N
150      LET  F = P*F
160      PRINT   "P=";    P,  "P!=";  F
170  NEXT P
180  PRINT   "***** DONE *****"
```

The following are examples of FOR...TO statements that use more complex expressions. The number of times the loop is executed depends upon the values of the variables at the time the FOR...TO statement is executed.

```
FOR K = LOG(Q/N) TO 2*LOG(Q/N+1) STEP LOG(Q/100)
.
.
.
NEXT K

FOR N = INT(ABS(E1)*10+.5)/10 TO E1+10.5 STEP .5
.
.
.
NEXT N
```

Modifying FOR...TO Values within the Loop

When using variables in FOR...TO statements, it must be remembered that the FOR...TO statement is only executed once, at the beginning of the loop, not each time through the loop. Therefore, the values of the TO and STEP expressions are fixed for the entire loop processing, even if statements within the loop assign new values to the variables in these expressions. This is illustrated in Example 7.9.

Example 7.9 Illustration of the Fact That STEP Value Is Fixed

```
110 T=2
120 PRINT "START, T = ";T
130 FOR K=0 TO 8 STEP T
140      PRINT "K=";K
150      T=T+1
160 NEXT K
170 PRINT "END, T=";T
```

The output from this program is:

```
START, T = 2
K= 0
K= 2
K= 4
K= 6
K= 8
END, T= 7
```

99

T is 2 when the FOR...TO statement is executed. Despite the fact that T changes each time through the loop, the step value is fixed at 2.

If the numeric variable that contains the counter is changed by a statement within the loop, loop processing is affected. The program in Example 7.10 loops endlessly, because statement 130 nullifies the effect of the NEXT statement on the counter.

Example 7.10 An Endless FOR...TO/NEXT Loop

```
110 FOR N=5 TO 15 STEP 3
120     PRINT 2*N,
130     N=N-3
140 NEXT N
```

FOR...TO/NEXT loops in which the counter variable is altered by statements within the loop easily become quite confusing. It is probably best to avoid such techniques.

Branching into the Middle of a FOR...TO/NEXT Loop

Since NEXT depends upon information established by the FOR...TO statement, an attempt to execute NEXT X produces an error if FOR X=...TO has not been executed. Therefore, a branch into the middle of a FOR...TO/NEXT loop is illegal.

7-4 NESTED LOOPS AND BRANCHING WITH LOOPS

Nested Loops

FOR...TO/NEXT loops can be used within FOR...TO/NEXT loops. When loops are used in this manner, they are said to be "nested."

To see how nested loops might be used, let's consider a new version of our monthly mortgage payment problem. Suppose that in addition to calculating the monthly payment for varying interest rates, we would like to calculate it for a varying term as well. We would like the program to start with a 20 year term and calculate the payment for 7%, 7.5%, 8%, 8.5%, 9%. Then it should calculate it at each of these percentages for a 25 year term, then at a 30 year term, then 35, and 40 year terms. The output should look like this if a $35000 principal is entered.

PRINCIPAL	TERM	INTEREST RATE	MONTHLY PAYMENT
$ 35000	20 YEARS	7 %	$ 271.35
$ 35000	20 YEARS	7.5 %	$ 281.96
$ 35000	20 YEARS	8 %	$ 292.75
$ 35000	20 YEARS	8.5 %	$ 303.74
$ 35000	20 YEARS	9 %	$ 314.9
$ 35000	25 YEARS	7 %	$ 247.37
$ 35000	25 YEARS	7.5 %	$ 258.65
$ 35000	25 YEARS	8 %	$ 270.14
$ 35000	25 YEARS	8.5 %	$ 281.83
$ 35000	25 YEARS	9 %	$ 293.72
$ 35000	30 YEARS	7 %	$ 232.86
$ 35000	30 YEARS	7.5 %	$ 244.73
$ 35000	30 YEARS	8 %	$ 256.82
$ 35000	30 YEARS	8.5 %	$ 269.12
$ 35000	30 YEARS	9 %	$ 281.62
$ 35000	35 YEARS	7 %	$ 223.6
$ 35000	35 YEARS	7.5 %	$ 235.98
$ 35000	35 YEARS	8 %	$ 248.59
$ 35000	35 YEARS	8.5 %	$ 261.4
$ 35000	35 YEARS	9 %	$ 274.4
$ 35000	40 YEARS	7 %	$ 217.5
$ 35000	40 YEARS	7.5 %	$ 230.32
$ 35000	40 YEARS	8 %	$ 243.36
$ 35000	40 YEARS	8.5 %	$ 256.58
$ 35000	40 YEARS	9 %	$ 269.98

----- DONE -----

Obviously the interest rate goes through a complete cycle for each value of the term. These results can easily be obtained by placing a FOR...TO/NEXT loop, that varies the interest rate, within another FOR... TO/NEXT loop, that varies the term. Note that in addition to incrementing the term, the outer loop must also output a blank line preceding the next group of calculations. Example 7.11 produces these results.

Example 7.11   Nested Loops in the Mortgage Problem

```
110 REM ($MORT8) ILLUSTRATION OF NESTED LOOPS
120 REM OPERATOR ENTERS VALUE FOR PRINCIPAL
130     INPUT "ENTER PRINCIPAL",P
140 REM PRINT HEADINGS
150     SELECT PRINT 215 (80)
160     PRINT ,,"INTEREST","MONTHLY"
170     PRINT "PRINCIPAL", "TERM","RATE","PAYMENT"
180     PRINT
190 REM ######### OUTER LOOP INCREMENTS THE TERM ############
200     FOR T=20 TO 40 STEP 5
210        REM
220        REM
230        REM **** INNER LOOP UP'S THE INTEREST RATE ****
240        REM ****    AND PERFORMS THE PROCESSING    ****
250           FOR I = 7 TO 9 STEP .5
260              M=P*(I/1200)/(1-(1+I/1200)↑(-12*T))
270              M2 = (SGN(M)*INT(ABS(M)*100+.5)/100
280              PRINT "$";P,    T;"YEARS",  I; "%",    "$"; M2
290           NEXT I
300        REM ********* INNER LOOP ENDED ****************
310        REM
320        REM
330        PRINT
340     NEXT T
350 REM ############### OUTER LOOP ENDED ##################
360     PRINT
370     PRINT "----- DONE -----"
```

In Example 7.11 only the principal is entered by the operator. Statement 200 sets up the outer loop. It sets the term, T, equal to 20 for the first time through the outer loop. It also specifies that the outer loop is to be executed until T+5 is greater than 40, that is, the STEP value is 5 and the upper bound is 40.

Statement 250 now sets up the inner loop. It says I, interest, is to vary from 7% to 9% in steps of .5%. The body of this loop, statements 260-280, hasn't changed from the examples of the last section.

NEXT I causes a branch to 260 if I+.5 is less than or equal to 9. When I+.5 is greater than 9, it lets the "normal sequence of execution" prevail.

330 is the first executable statement after NEXT I. It outputs a blank line that separates different term values. Then, NEXT T is encountered. NEXT T increments the term value, in T, by 5, and effects a branch to line 210. It does this as long as T+5 is less than or equal to 40.

After NEXT T branches to 210, the first statement to be executed is 250 FOR I=7 TO 9 STEP .5. Thus, each time through the outer loop, the inner loop is setup by 250, and executed 5 times by 290 NEXT I.

As a practical matter loops can be nested within loops indefinitely. That is, there is no limit to the number of loops that may be contained within any given loop.

Branching and FOR...TO/NEXT Loops

We have already pointed out that a branch into the middle of a FOR...TO/NEXT loop will cause an error at the NEXT statement. The NEXT statement depends upon information supplied by FOR...TO and without it cannot function. For example, this program segment will report an error when statement 90 is executed, after the branch from 30 to 70.

```
        .
        .
        .
20 PRINT J*4
30 GOTO 70
        .
        .
        .
60 FOR I=1 TO 10 STEP 2
70      Q2=(14/T)*I
80        PRINT Q2;
90 NEXT I
```

Recall that when looping is completed, and the NEXT statement is about to let the normal sequence of execution prevail, it first clears from that "special part of memory" the information that had been saved there by the FOR...TO statement. This is done simply to make room for future FOR...TO information. This implies that a program should not repeatedly branch out of the middle of a FOR...TO/NEXT loop without allowing for a normal NEXT statement loop termination. Repeated execution of FOR...TO statements, that are never terminated by NEXT statements, eventually causes a "table overflow" error (ERR 02).

Example 7.12 shows a program that branches out of a FOR...TO/NEXT loop, without allowing normal NEXT statement termination of the loop.

Example 7.12   A Branch Out of a FOR...TO/NEXT Loop That Causes a Table
              Overflow Error

```
110 REM BRANCH THAT AVOIDS NORMAL "NEXT" STATEMENT TERMINATION
120 REM PROGRAM TO PRINT PRIME NUMBERS 1 TO 1001
130      PRINT 1; 2; 3;
140      N = 3
150 REM TRY DIVIDING N BY EACH ODD INTEGER
160      FOR T = 3 TO SQR(N) STEP 2
170          REM DOES T DIVIDE N EVENLY?
180          IF INT(N/T) = N/T THEN 230
190      NEXT T
200 REM N IS PRIME
210      PRINT N;
220 REM N NOT PRIME.  TRY NEXT ODD NUMBER
230      N = N+2
240      IF N <= 1001 THEN 160
```

This program is supposed to print the prime numbers between 1 and 1001. However, each time a number proves to be non-prime (it's divided evenly by another number), line 180 effects a branch out of the FOR...TO loop; N is incremented (line 230) and the FOR...TO statement is reexecuted (branch from 240 to 160). Reexecuting the FOR...TO statement saves

103

information in memory for another loop. This is in addition to the information for the last loop that was never cleared by a NEXT statement loop termination. Eventually, the memory space allotted for FOR...TO information fills, and a table overflow error interrupts execution. In this program this occurs before all the primes between 1 and 1001 have been found; therefore the program must be corrected.

If a loop must have another exit as well as a "counter" exit, then there are two alternatives. Either forego FOR...TO/NEXT by setting up a counter and counter-test using LET and IF...THEN as in Example 7.2, or use FOR...TO/NEXT and, at the "second way out", add an operation that forces NEXT to terminate the loop. This latter approach is illustrated in Example 7.13.

Example 7.13   Forcing a "NEXT" Termination

```
110 REM FORCING A "NEXT" TERMINATION
120 REM PROGRAM TO PRINT PRIME NUMBERS 1 TO 1001
130     PRINT 1; 2; 3;
140     N = 3
150 REM TRY DIVIDING N BY EACH ODD INTEGER
160     FOR T = 3 TO SQR(N) STEP 2
170         REM DOES T DIVIDE N EVENLY?
180         IF INT(N/T) = N/T THEN 222
190     NEXT T
200 REM N IS PRIME
210     PRINT N;
215     GOTO 230
220 REM N NOT PRIME. TRY NEXT ODD NUMBER
222 REM ****** FORCE A "NEXT" TERMINATION *****************
224     T = SQR(N)
226     NEXT T
228 REM ********************************************************
230     N = N+2
240     IF N <= 1001 THEN 160
```

In this example lines 224 and 226 are executed only if line 180 effects a branch out of the FOR...TO/NEXT loop. 224 assigns the "TO" value SQR(N) to T. Since T is then at the upper limit of its range, NEXT T at line 226 will always "terminate the loop" and clear the FOR...TO information.

If loops are nested, one inside another, a branch to the NEXT statement of an outer loop clears the FOR...TO information of all inner loops. This fact permits a better solution to the problem of Example 7.12, than that shown in Example 7.13.

Example 7.14   Normal Termination of an Inner Loop by an Outer Loop

```
110 REM TERMINATING AN INNER LOOP BY BRANCHING TO AN OUTER LOOP
120 REM PROGRAM TO PRINT PRIME NUMBERS 1 TO 1001
130     PRINT 1; 2;3;
140     FOR N = 3 TO 1001 STEP 2
150         FOR T = 3 TO SQR(N) STEP 2
160             IF INT(N/T) = N/T THEN 190
170         NEXT T
180         PRINT N;
190     NEXT N
```

104

In this example an outer FOR...TO/NEXT loop is used in place of statements 140, 230 and 240 of Example 7.12. Now, the branch at line 160 is not a source of trouble, since execution of the outer loop's NEXT statement (line 190) clears the FOR...TO information for the inner loop.

Review of Chapter 7

1. FOR...TO and NEXT are used to control a loop.

2. The FOR...TO statement marks the beginning of the loop. It specifies the variable that is to serve as the loop counter, and assigns it its initial value. In a special part of memory, it saves the exit value of the loop counter, and the increment (STEP value). These are saved for use by the NEXT statement. For example,

    FOR   K   =   2   TO   12

              ↑   ↑   ↑   ↑
            loop   initial   exit   increment
           counter   value   value   (implied
          variable   +1)

3. The STEP parameter can be included in the FOR...TO statement to specify an increment other than +1. (+1 is implied if the STEP parameter is not used.) For example,

    FOR   N(I)   =   -2   TO   -16   STEP   -2

              ↑   ↑   ↑   ↑
            loop   _   initial   exit   increment
           counter   value   value
          variable

4. Any expression may be used to specify the initial, exit, and STEP values in a FOR...TO statement.

5. FOR...TO merely sets up the loop. NEXT controls the execution of the loop, using the information saved by FOR...TO.

6. NEXT adds the increment to the counter, and effects a branch to the statement following FOR...TO, until the value of the counter passes the exit value. More precisely, if the increment is positive, NEXT tests if adding the increment to the counter would cause the counter to be greater than the exit value; if so, it "exits" the loop by allowing the normal sequence of execution to prevail. If not, it adds the increment to the counter, and branches to the statement after FOR...TO. If the increment is negative, NEXT tests if adding the increment to the counter would cause the counter to be less than the exit value; if so, it "exits" the loop by allowing the normal sequence of execution to prevail. If not, it adds the increment to the counter, and branches to the statement after FOR...TO.

7. When NEXT "exits" the loop, it clears from the special part of memory the information saved by FOR...TO.

105

8. Repeated executions of FOR...TO, which do not lead to a normal NEXT statement loop termination, will eventually cause a table overflow error, as a result of the excess accumulated information saved by FOR...TO.

9. If loops are nested, a branch to an outer loop's NEXT statement clears the FOR...TO information of all inner loops.

10. Since the operation of NEXT depends upon information saved by FOR...TO, a branch into the middle of a loop, which bypasses the FOR...TO, will produce an error at the NEXT statement.

CHAPTER 8
INTRODUCTION TO ALPHANUMERICS

8-1   <u>ALPHANUMERIC VARIABLES</u>

In Section 3-1 we said that a numeric variable is a place to keep a number.   Numeric variables are used to hold numeric quantities, which we may multiply, divide, add, subtract or evaluate in a function.   In addition to numeric quantities, though, there is another kind of information that we may want to process. Typical of this other information are names, addresses, social security numbers, product descriptions, and part numbers. This kind of information is called alphanumeric information, since it may consist of any combination of alphabetic and numeric characters, punctuation, and symbols.

Alphanumeric information never   enters   into   ordinary   arithmetic operations (+, -, /, *, ↑, etcetera). For example, it wouldn't make much sense to say "Take the square root of Jones' address and multiply it times his last name". Nevertheless, we may want to process it. We may want to enter it, update it, delete it, transfer it, save it, sort it, segment it, or print it.

With the statements we have examined thus far, our ability to process alphanumeric information has been very limited. Alphanumeric information has appeared in only one form, the literal string. We've seen examples of literal strings, such as "REORDER COAL IMMEDIATELY: INVENTORY BELOW 100 TONS" and "**** DONE ****".   We know that the quotation marks aren't part of the information per se. They are there only to say, "This is alphanumeric information."

Thus far, all the variables we have been using have been numeric variables.   Numeric variables can only be assigned valid numeric quantities. They can never hold alphanumeric characters, even if the alphanumeric characters outwardly resemble a numeric quantity. For example, as noted in Section 3-2, if we try to enter

10 K = "165"

the result is a syntax error, telling us that a numeric variable cannot be assigned an alphanumeric literal string.

In BASIC, though, there is a kind of variable that can be assigned an alphanumeric literal string. Variables of this kind are called "alphanumeric variables". Just as there are 286 numeric variables in BASIC, there are also 286 alphanumeric variables. They are named in a manner similar to the numeric variables, except that every alphanumeric variable name ends with a dollar sign, $. Thus, the alphanumeric variables are A$, B$, C$...Z$ and A0$, A1$, A2$。..A9$,B0$,B1$...Z7$,Z8$,Z9$.

Alphanumeric values can be assigned to alphanumeric variables in statements that are similar to those that assign numeric quantities to numeric variables. For example, these are all valid assignment statements.

```
 60  D2$ = "154 STATE ST."
170  LET A0$ = "A4078-R"
410  J8$ = "%"
900  L1$,J3$,P9$ = "N/A"
 91  V$ = "CREDIT"
140  Q$ = A9$
```

Notice that the keyword "LET" is optional, and that multiple assignments are possible (line 900). In line 140 the variable Q$ is assigned the value of the variable A9$.

When a literal string is used in an assignment statement, the literal string is always enclosed by quotation marks. The quotation marks do not become part of the value of the alphanumeric variable.

Alphanumeric variables may never be used with arithmetic operators, or where a numeric expression is required. For example, the following are illegal uses of alphanumeric variables:

<u>Illegal</u>

```
10 A$ = C$ + D$
50 P2 = C8$
```

Alphanumeric variables cannot be assigned the value of an expression. The following statements are also illegal:

<u>Illegal</u>

```
20 A$ = 250
40 FOR A$ = 2 TO 20
```

We may now expand the general form of the assignment statement to include alphanumeric operations. This expanded general form is:

[LET] numeric variable [,numeric variable...] = expression

[LET] alphanumeric variable [,alphanumeric variable...] = $\left\{ \begin{array}{l} \text{literal string} \\ \text{in quotes} \\ \\ \text{alphanumeric} \\ \text{variable} \end{array} \right\}$

8-2   A CLOSER LOOK AT ALPHANUMERIC VARIABLES (PRINT AND DIM)

In Section 2-6 we noted that the RUN command sets all numeric variables to zero, whenever it is used without a line number. Analogously, it sets alphanumeric variables to all spaces. An alphanumeric variable which is blank, that is, all spaces, is treated by the system as if it contained just one space. Therefore, we can say that, in effect, the initial value, after RUN, of every alphanumeric variable is one space.

108

To see this, enter and execute the following program.

```
10 PRINT "ABC"; "DEF"
20 PRINT "ABC";A$;"DEF"
```

The program prints

```
ABCDEF
ABC DEF
```

The single blank which appears on the bottom line is the result of printing A$. A$ has a value of one space.

A numeric variable can contain any valid numeric quantity. In Section 3-2 we said that such a quantity can have up to 13 digits, decimal point, sign, and a signed 2 digit exponent. Every numeric variable, then, has a fixed size "big enough" to hold any numeric quantity. By contrast, the size of alphanumeric variables is not fixed; however, it is automatically set to 16 characters, unless you specify a different size. You will see how to specify a different size for alphanumeric variables later in this section.

If you try to assign more characters to an alphanumeric variable than its size will allow, the additional characters at the right are simply ignored. For example, enter and execute:

```
10 D$ = "0123456789ABCDEFGH"
20 PRINT D$
```

The output from line 20,

```
0123456789ABCDEF
```

shows that statement 10 assigned to D$ only the first 16 characters of the literal string. Since the size of D$ is 16, the 17th and 18th characters, "G" and "H", are ignored. Later we will see additional means of assigning values to alphanumeric variables; however, regardless of the method used, extra characters beyond the variable's maximum size are ignored.

Now let's see what happens if you assign to an alphanumeric variable fewer than its maximum number of characters. Enter and execute the following program:

```
20 H$ = "JOHN Q. ADAMS"
30 PRINT H$
40 H$ = "HELP"
50 PRINT H$
55 L$ = "AT"
60 H$ = L$
70 PRINT H$
```

Execution outputs

```
JOHN Q. ADAMS
HELP
AT
```

109

Notice from this example that when an alphanumeric variable is assigned
a new value (lines 20, 40 and 60), the new value completely replaces the old
value, despite the fact that the old value is longer than the new value. That
is, the result of executing line 50 specifically was not "HELP Q. ADAMS". The
"Q. ADAMS" has been replaced in H$ by spaces. The same is true for the
assignment made at line 60 where the "LP" of "HELP" is replaced by spaces.

The example above does not reveal, though, how alphanumeric variables
are printed in regard to spaces. We can add literal strings to the PRINT
statements to give us an answer. Change the program as follows:

```
20 H$ = "JOHN Q. ADAMS"
30 PRINT H$; "&&&"
40 H$ = "HELP"
50 PRINT H$; "&&&"
55 L$ = "AT"
60 H$ = L$
70 PRINT H$; "&&&"
```

The ampersands are added so that we can see any spaces which may be output.

The program now prints

```
JOHN Q. ADAMS&&&
HELP&&&
AT&&&
```

This result may seem paradoxical. We just said that line 40 of these two
programs replaces "Q. ADAMS" in H$ with spaces, but when we add a second print
element to line 50, these spaces don't show up. The same thing is true for
lines 60 and 70.

The answer to this is quite simple: trailing spaces in alphanumeric
variables are not considered to be part of the value of the variable.
Therefore, when an alphanumeric variable is printed, trailing spaces are
omitted. Trailing spaces are adjacent spaces that extend from the last
non-space character to the end of the variable. Since L$ and H$ are both 16
characters long, if you could look into the memory of your Wang system after
step 20, you can imagine seeing something like this:

```
                                              ↓the last non-space character
H$  | J | O | H | N | Δ | Q | . | Δ | A | D | A | M | S | Δ | Δ | Δ |
```

trailing spaces not con-
sidered part of value of H$.

trailing spaces not con-
sidered part of value of L$.

```
L$  | A | T | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
          ↑last non-space character.
```

After step 40 H$ looks like this:

```
H$  | H | E | L | P | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
              ↑
```

trailing spaces not part of value.

last non-space character.

In summary, then, when alphanumeric variables are used as print elements, trailing spaces are never printed. In the program below, note the differing effects, in regard to spaces, of printing a literal string, and printing an alphanumeric variable which has been assigned the value of the literal string.

```
10 PRINT "FIVE "; "DOLLARS"
20 A$ = "FIVE "
30 B$ = "DOLLARS"
40 PRINT A$; B$
```

When executed, this produces:

```
FIVE DOLLARS
FIVEDOLLARS
```

We can see from this example that when the literal string is printed (line 10), the trailing space is included. When "FIVE " is assigned, though, the space included in the literal string becomes merely a trailing space. As the second line of output shows, this space is effectively lost.

In a situation such as this, you can overcome this seeming problem simply by putting a literal space into line 40, as in

    40 PRINT A$; " "; B$

or you can add a space to the front of "DOLLARS" as follows:

    30 B$ = " DOLLARS"

Since the latter is not a trailing space, it will be printed when B$ is printed.

We opened this section by saying that if an alphanumeric variable contains all spaces, it is treated as if it contained just one. We can now see the reason for this. The first space is considered to be a real character, but all the spaces which follow the first are mere "trailing spaces" which, as we have said, are ignored.

Dimensioning Alphanumeric Variables

We have noted that the system automatically sets alphanumeric variable size to 16 characters in the absence of contrary instructions. It is possible to specify a different maximum variable size for any alphanumeric variable by means of a dimension statement. The dimension statement has several other related uses which we will encounter in future chapters. Its use as a means of specifying alphanumeric variable length is simple and straightforward.

In BASIC the word "dimension" has been abbreviated to the keyword "DIM". To specify a size of 30 characters for the variable C$, we include in the program, at a lower line number than any reference to C$, a statement such as:

    20 DIM C$30

This statement fixes the maximum size of C$ at 30 characters for the entire program. Once the size of a variable has been established, whether automatically at 16 characters by the system, or by means of a DIM statement, that size cannot be altered until all variables are cleared.

When you key RUN (EXEC), the system first scans through the entire program, in line number sequence, looking for variables and DIM statements. When it encounters an alphanumeric variable for the first time, it either sets aside 16 characters of memory space for it, or, if it is in a DIM statement, sets aside the amount specified. Once it has established the size of a variable, that size is fixed. If the system later in its scan encounters a DIM statement which attempts to change the size, an error is signaled. This means that DIM statements must precede any program reference to the dimensioned variable, since otherwise the system will have already set the size at 16 characters. The following program violates this rule and will not execute.

    10 PRINT "ABC"; A$; "DEF"
    20 DIM A$10

In the DIM statement the size of the variable must be specified with a number, not an expression. The number must be greater than 0 and less than or equal to 64. Sixty-four characters is, therefore, the absolute maximum size of any alphanumeric variable.

It is possible, and often desirable, to dimension multiple variables in a single DIM statement. Such a statement might look like this:

```
20 DIM A$40, A8$1, C7$25, C9$25
```

In a multiple DIM statement each variable and dimension is separated from the next by a comma. Any number of variables may be dimensioned in a single DIM statement.

Notice that it is possible to specify a variable size less than 16 characters. This helps to conserve memory when less than 16 characters will be assigned to the variable.

Regardless of the size of an alphanumeric variable, it is operated on under the same principles discussed in the opening part of this section. If you attempt to assign more characters than the variable can hold, excess characters are lost. Assigning a new value to a variable replaces the entire old contents of the variable. Trailing spaces are not part of the value of a variable.

8-3   INPUT AND IF...THEN WITH ALPHANUMERIC VARIABLES

INPUT

The INPUT statement can be used with alphanumeric variables to permit operator entry of alphanumeric data. For example, the following program allows a name and address to be entered into four variables, with up to 30 characters for each line. It then prints the entered address.

```
10 DIM A$30, B$30, C$30, D$30
20 INPUT "NAME", A$
30 INPUT "ADDRESS LINE #1", B$
40 INPUT "ADDRESS LINE #2", C$
50 INPUT "ADDRESS LINE #3", D$
60 PRINT A$
70 PRINT B$
80 PRINT C$
90 PRINT D$
```

Use of INPUT with alphanumeric variables is very similar to its use with numeric variables. The literal string prompt is optional, and multiple variables can be included in a single statement. The keyboard entry mandated

by the above lines 10 to 50 could have been accomplished with either of the two methods shown below:

```
10 DIM A$30,B$30,C$30,D$30
20 INPUT A$
30 INPUT B$
40 INPUT C$
50 INPUT D$
60 PRINT A$
   .
   .
   .
```

or

```
10 DIM A$30,B$30,C$30,D$30
20 INPUT A$,B$,C$,D$
60 PRINT A$
   .
   .
   .
```

Alphanumeric and numeric receiving variables may be included in a single INPUT statement. Thus, line 20 of the following program is a legal INPUT statement:

```
10 DIM A$25
20 INPUT "ENTER NAME, HOURLY RATE", A$,R
```

When 20 is executed, the following will appear

ENTER NAME, HOURLY RATE?_

the operator can then make the two entries in either of the following ways

ENTER NAME, HOURLY RATE? JOHN JONES, 6.45 (EXEC)

or

ENTER NAME, HOURLY RATE? JOHN JONES (EXEC)
? 6.45 (EXEC)

Often it is better to avoid multiple-variable INPUT statements, due to the increased probability of operator confusion.

If an operator enters more characters than an alphanumeric variable is capable of holding, the overflowing characters are not assigned to the variable; they are simply lost. No error is signaled. If any character is entered, the entire old value of the receiving alphanumeric variable is replaced.

When entering alphanumeric data on an INPUT instruction, an operator need not enclose the entry in quotation marks. However, if quotation marks are not used, leading spaces, entered by the operator, are not assigned to the variable. Furthermore, without quotation marks, commas act as separators,

that is, an entered comma is taken to mean, "That's all for the first variable." (In the example shown above

        ENTER NAME, HOURLY RATE? JOHN JONES, 6.45 (EXEC)

the comma separates the two values. It is not part of either.) Thus if the operator wishes to enter BOSTON, MASS. 02109 for address line #3 in the first example, it must be entered with quotation marks as follows:

        ADDRESS LINE #3? "BOSTON, MASS. 02109"

Without quotation marks only BOSTON will be assigned to D$; the rest of the entry will be lost.

IF...THEN

The IF...THEN statement can be used to compare alphanumeric values, and branch if a specified condition is true. In the program shown in Example 8.1 the operator can select, by entering the words "YES" or "NO", whether the program results are to appear on the CRT or be printed by the printer.

Example 8.1 Testing Alphanumeric Values with IF...THEN

```
100 REM   TESTING ALPHANUMERIC VALUES WITH "IF...THEN"
120     INPUT "DO YOU WANT THE PROGRAM RESULTS TO BE PRINTED (YES/NO)", A$
130 REM TEST OPERATOR RESPONSE
140     IF A$ = "YES" THEN 210
150     IF A$ = "NO" THEN 230
160 REM ENTRY INVALID
180     PRINT "INVALID. REENTER."
200     GOTO 120
205 REM "YES" ENTERED
210     SELECT PRINT 215 (80)
220     GOTO 250
225 REM "NO" ENTERED, BE SURE CRT IS SELECTED
230     SELECT PRINT 005 (64)
245 REM MAIN PROGRAM BEGINS HERE
250 FOR I = 1 TO 60
260     PRINT 2↑I;
270 NEXT I
290 PRINT
```

With the IF...THEN statement, it is also possible to compare alphanumeric values that are saved in alphanumeric variables. For example, the following are valid BASIC statements:

```
200 IF C$ = D$ THEN 450
240 IF Z9$ = A8$ THEN 710
```

Variables being compared do not have to be the same size. For example, the following is a valid sequence of statements:

```
10 DIM A$6, B$2
   .
   .
   .
40 IF B$ = A$ THEN 70
   .
   .
   .
70 PRINT A$
```

Whenever the alphanumeric terms being compared by an IF...THEN are of different length, the comparison is made as if the shorter one were extended with spaces out to the length of the longer one. To see how this works, assign values in the example above as follows:

```
10 DIM A$6, B$2
20 A$ = "OSCAR"
30 B$ = "OS"
40 IF B$ = A$ THEN 70
50 INPUT R
   .
   .
   .
70 PRINT A$
```

Intuitively we would say that A$, with a value of "OSCAR", and B$, with a value of "OS", should not be considered equal. In fact this is the result of the comparison at line 40; the branch is not taken. The system, noticing that B$ is dimensioned smaller than A$, obtains the value "OS" from B$ and temporarily extends it to the size of A$. The values it then has to compare are as follows:

| O | S | C | A | R | Δ |

| O | S | Δ | Δ | Δ | Δ |

{ temporary
{ extension
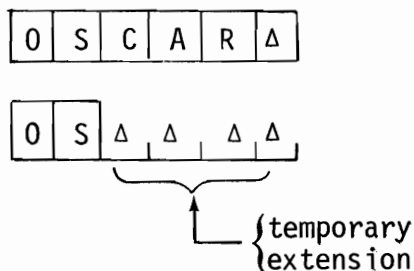
The first two characters of these values are the same. However, when the system looks at the third characters, a C and a space, it finds them not the same, and concludes that the relationship "equals" is not true. The branch is not taken. Despite the "temporary extension" of B$ described here, the real size of B$ hasn't changed, and remains at 2 characters after the IF...THEN statement is complete.

The other IF...THEN relational operators < , > , >=, <=, can also be used with alphanumeric comparisons. For example, suppose that parts are stored in two warehouses. All the parts stored in the first warehouse are given part numbers which begin with the letters A-M. Part numbers for the second

warehouse begin with the letters N-Z. The following program segment tests an
entered part number to determine in which warehouse the part belongs.

```
10      DIM C$6
 .
 .
 .
410     INPUT "PART NUMBER", C$
420 REM SECOND WAREHOUSE?
430     IF C$>="N" THEN 490
440 REM GOES IN FIRST WAREHOUSE
 .
 .
 .
490 REM GOES IN SECOND WAREHOUSE
 .
 .
 .
```

What happens if the operator enters L4906A   at line 410?   At the
IF...THEN statement, the system finds that the literal string "N" is shorter
than the variable C$. Therefore, it takes the value of the literal and
temporarily extends it with spaces to the size of C$. It is then ready to
compare the two values, which now look like this:

temporary
extension

The comparison now takes place character-by-character until an inequality is
found (just as you would proceed if you were alphabetizing these two terms).
It finds an inequality right away: L is less than N. Since this relationship
violates the specified condition for the branch, C$>="N", the branch is not
taken.

Suppose that N2079 were entered instead. The set-up for the comparison
would be

When the first characters are compared they are equal, so the system
compares the next two characters. Space has a lower value than all letters,
numbers, and keyboard special characters.   Therefore, when comparing the
second characters, the system finds the inequality 2 > space.   This
inequality, 2 > space, immediately signifies that C$ > "N"; therefore, the
branch is taken.

117

       Alphanumeric comparisons take place character-by-character until unequal characters are found. The relationship of the first pair of unequal characters determines the relationship of the entire two values. This method is exactly how one proceeds when alphabetizing.

       Other than to say that spaces have a low value, we haven't specified the relative ordering of all the keyboard characters. This relative ordering is given in Table 8.1.

LOWEST

```
SPACE          >          a
 !             ?          b
 "             @          c
 #             A          d
 $             B          e
 %             C          f
 &             D          g
 '             E          h
 (             F          i
 )             G          j
 *             H          k
 +             I          l
 ,             J          m
 -             K          n
 .             L          o
 /             M          p
 0             N          q
 1             O          r
 2             P          s
 3             Q          t
 4             R          u
 5             S          v
 6             T          w
 7             U          x
 8             V          y
 9             W          z
 :             X
 ;             Y
 <             Z
 =
```

HIGHEST
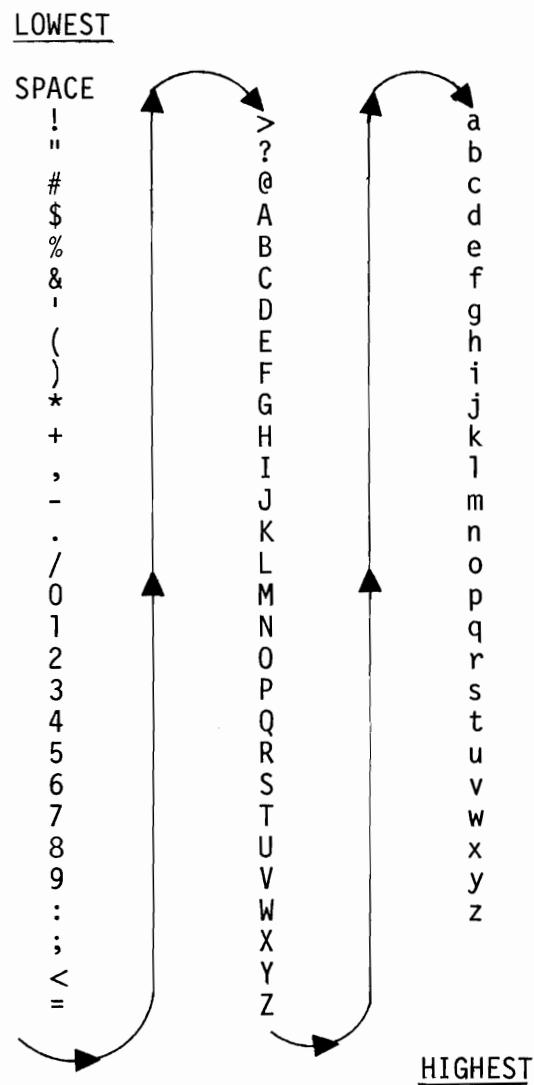
Table 8.1 Relative Ordering of Keyboard Characters

The following program will allow you to experiment with the system's alphanumeric ordering. Try entering different values, and observe the results.

```
110 DIM A$64,B$64
120 PRINT
130 INPUT "ENTER VALUE FOR A$",A$
140 INPUT "ENTER VALUE FOR B$",B$
150 PRINT
160 IF A$ <> B$ THEN 190
170 PRINT A$; " -EQUALS- "; B$
180 GOTO 120
190 IF A$ > B$ THEN 220
200 PRINT A$; " -IS LESS THAN- "; B$
210 GOTO 120
220 PRINT A$; " -IS GREATER THAN- "; B$
230 GOTO 120
```

With the ability to compare alphanumeric values, a simple sort can be performed to rearrange three values into "alphabetical order" (or more precisely, the order specified by Table 8.1). Example 8.3 accomplishes this in a very simple way. Efficient sorting, however, requires subscripted variables, which are introduced in Chapter 11.

Example 8.2 A Simple Alphanumeric Sort

```
110 REM SORT
120     DIM A$20,B$20,C$20,D$20
130     INPUT "ENTER THREE VALUES.  EACH 20 CHAR. MAX.", A$,B$,C$
140 REM TEST #1
150     IF A$ < B$ THEN 210
160 REM SWITCH A$ AND B$
170     D$=A$
180     A$=B$
190     B$=D$
200 REM TEST #2
210     IF B$ < C$ THEN 270.
220 REM SWITCH B$ AND C$
230     D$=B$
240     B$=C$
250     C$=D$
260 REM TEST #3 -(REPEAT OF TEST #1 WITH POSSIBLE NEW VALUES)
270     IF A$ < B$ THEN 320
280 REM SWITCH A$ AND B$
290     D$=A$
300     A$=B$
310     B$=D$
320 PRINT A$,B$,C$
```

Review of Chapter 8

1. Alphanumeric values may consist of any combination of alphabetic and numeric characters, punctuation and symbols.

2. Alphanumeric values cannot enter into arithmetic operations.

3. Alphanumeric variable names are the same as numeric variable names, except that a dollar sign, $, is added, e.g., A$, C$, F2$, X8$, etc.

4. Alphanumeric values may be assigned to alphanumeric variables in many of the same ways that numeric values are assigned to numeric variables. For example,

        10 B$ = "TEST #14.5"
        10 INPUT "EMPLOYEE NAME", N$

5. Alphanumeric variables are 16 characters long, unless set to a different size in a DIM statement.

6. A DIM statement can specify a length of 1 to 64 characters for an alphanumeric variable. For example,

        DIM C2$64, D$2

7. If you attempt to assign more characters to an alphanumeric variable than its length can accommodate, the excess characters at the right are ignored.

8. When a new value is assigned to an alphanumeric variable, any old value is completely replaced by the new, which is padded with spaces at the right.

9. Trailing spaces are not considered to be part of the value of an alphanumeric variable.

10. Alphanumeric values may be compared in an IF...THEN statement. Values are temporarily padded with spaces at the right, so that they are the same length. Then, the comparison takes place character-by-character until unequal characters are found. The first pair of unequal characters determines the relationship of the entire two values. If no unequal characters are found, the values are equal.

CHAPTER 9
DEBUGGING AIDS AND MISCELLANEOUS
SYSTEM FEATURES

In this chapter we are going to look at a number of Wang 2200 System features and a few BASIC statements. Many of these features and statements are used in debugging programs. This use will be highlighted here; however, some of these features have many uses, and the discussion is not meant to imply that program debugging is the only use.

9-1   THE STOP STATEMENT AND THE CONTINUE COMMAND

The STOP statement is a general purpose BASIC statement. When the system encounters a STOP statement, it stops executing the program, and outputs the word "STOP" at the device selected for Console Output (normally the CRT). The colon and cursor (:_) are displayed on the line below the word "STOP". A STOP statement can simply consist of the keyword STOP. However, a character string, in quotation marks, can follow the keyword. If a character string is used, the system outputs the character string on the same line as "STOP".

In general, after a STOP statement has interrupted execution, the operator can continue execution at the next statement by keying CONTINUE (EXEC).

Example 9.1 illustrates a simple use of the STOP statement. The program shown in 9.1 prints, in hardcopy, the factorials from 1 to N. It is similar to Example 7.8, except for the addition of the printer selection statement, and the STOP statement which warns the operator to ready the printer.

Example 9.1   A Simple Use of the STOP Statement

```
110 REM SIMPLE USE OF "STOP" STATEMENT TO PAUSE FOR OPERATOR
120   INPUT   "COMPUTE P!  FOR P=1 TO P=", N
122   STOP "READY PRINTER.(8 1/2  X  11  PAPER)"
125   SELECT PRINT 215 (80)
130   F=1
140   FOR P=1 TO N
150      LET F = P*F
160      PRINT  "P="; P,  "P!="; F
170   NEXT P
180   PRINT   "***** DONE *****"
```

During execution of this program, the following appears on the CRT.

```
:RUN
COMPUTE P!  FOR P=1 TO P=? 8

STOP READY PRINTER.(8 1/2  X  11  PAPER)
:_
```

At this point the operator can ensure that the printer is ready, and then key CONTINUE (EXEC) to resume normal execution.

In some cases the STOP statement is not the best choice for providing a program pause, as shown in Example 9.1. Often the INPUT statement is a better choice. The colon which appears below the word STOP signifies that the system is ready to accept any operator command or action which would be available before execution began. This means, for example, if an operator accidentally keys one or more digits followed by (EXEC) or by CONTINUE (EXEC), the digits are interpreted as a line number. The result could be the destruction of a program line. Furthermore, any such change in a program text prevents the CONTINUE command from resuming execution. By contrast, the system status at an INPUT statement, when the question mark is displayed, is much more restrictive. Accidentally keying a digit or character, at worst, produces an ERR message. Program text cannot be changed.

In some circumstances, however, the broader opportunities offered at a STOP interruption are needed. Program debugging is one such circumstance.

For example, if a particular loop in a program doesn't seem to be doing what you want it to do, you can temporarily insert a STOP statement immediately before the FOR...TO statement that sets up the loop. Then, execute the program with whatever values are causing a problem. When the STOP is reached, the program will output STOP and the colon. You can then use some of the techniques described in the next sections to determine the cause of the failure.

You may want to add a message to the STOP statement such as

```
58 STOP "LINE 58"
```

This identifies the STOP, if you have more than one, and tells you what line to delete when you want to take out the STOP.

The CONTINUE command cannot be used to resume execution after a STOP if any of the following has occurred while execution was stopped.

1. A text or table overflow has occurred (ERR 01, ERR 02).

2. A variable has been added to the program which was not previously part of it.

3. A CLEAR or RENUMBER command has been issued. (Section 9-4 introduces the RENUMBER command.)

4. The RESET key has been depressed.

5. The program has been modified.

9-2   IMMEDIATE MODE OPERATIONS

In Chapter 5 we mentioned that the SELECT statement can be used as a command, without a line number, as well as a BASIC program statement. For example, whenever the colon is present you can key

        :SELECT LIST 215 (EXEC)

to select a printer for program listings. The use of BASIC statements in this fashion, as if they were commands, is known as Immediate Mode operation, so called because the prescribed action takes place immediately upon keying (EXEC), rather than being saved in memory for future execution.

Your Wang system has been designed so that a variety of BASIC statements can be used in the Immediate Mode. For example, key

        :PRINT 4/3.7, SQR(4/3.7) (EXEC)

The results appear immediately in the first two zones of the CRT:

        1.081081081081   1.0397504898

Immediate Mode operations can be useful in debugging programs, and handy whenever a quick and simple calculation needs to be performed. They are legal whenever the colon is displayed. For example, if you insert a STOP statement into a malfunctioning program, as discussed in the last section, you can check the values of any program variables by executing Immediate Mode PRINT statements.

The program in Example 9.2 is supposed to calculate the sum of the first N odd integers by adding them up, but it has a bug. The sum it produces is incorrect.

    Example 9.2   Program with Bug

        110 REM PROGRAM WITH BUG - SUM IS INCORRECT
        120 PRINT "CALCULATE SUM OF FIRST N ODD INTEGERS"
        130 INPUT "ENTER N", N
        140 S = 1
        150 IF N = 1 THEN 200
        160 FOR I = 2 TO N
        170   D = D+2
        180   S = S+D
        190 NEXT I
        200 PRINT "SUM="; S

If we enter a value of 5 for N, this program calculates the sum of the first 5 odd integers as 21. (The answer should be 25, since the sum of the first N odd integers is always N^2.) We can insert a STOP statement, such as 185 STOP "LINE 185", to let us check variable values as program execution progresses.

With the STOP statement inserted, execution produces the following output:

```
CALCULATE SUM OF FIRST N ODD INTEGERS
ENTER N? 5

STOP LINE 185
:_
```

We can now execute an Immediate Mode PRINT statement to check the values of D and S. For example, key

```
PRINT "D="; D, "S="; S (EXEC)
```

The display appears as:

```
STOP LINE 185
:PRINT "D="; D, "S="; S
D= 2            S= 3
:_
```

From the values of D and S we can see that, at this point, the program has already run into trouble. Line 170 is supposed to set D equal to the next odd integer; here, D is even.

Notice that the line :PRINT "D="; D, "S="; S is not entered into memory. Since it has no line number, it is executed immediately.

We can let the program run through the loop once more if we wish by keying CONTINUE (EXEC). Again "STOP LINE 185" appears, and again we can inspect the values of D and S with an Immediate Mode PRINT statement as shown below.

```
STOP LINE 185
:PRINT D,S
 4              7
:_
```

Now we have a good idea of what the problem is: D should have been assigned the value 1 before the loop began; then, adding 2 to it repeatedly will always give the next odd number. We should change line 140 to assign 1 to D, as well as S. If we delete line 185 and list the program, it now looks like this:

Example 9.3 Debugged Version of 9.2

```
110 REM EXAMPLE 9.2 WITHOUT BUG
120 PRINT  "CALCULATE SUM OF FIRST N ODD INTEGERS"
130 INPUT "ENTER N", N
140 S, D =1
150 IF N = 1 THEN 200
160 FOR I = 2 TO N
170     D = D+2
180     S = S+D
190 NEXT I
200 PRINT "SUM="; S
```

.

In order to run the program, we must now use the RUN command. The CONTINUE command cannot be used since the program has been altered.

The PRINT statement functions exactly the same way in the Immediate Mode as it does in a program (or "Program Mode", as it is sometimes called). You can evaluate a large expression, for example, just as you would in a program statement. The only difference is that all Immediate Mode PRINT operations occur at the address selected for Console Output. Therefore, a PRINT statement executed in the Immediate Mode outputs to Console Output (normally the CRT), even if a statement such as SELECT PRINT 215 has been executed.

The assignment statement (LET) can be used in the Immediate Mode. If it is used to change the value of a variable used in the program, then CONTINUE can be used to resume execution after a STOP. If a new variable is established by an Immediate Mode assignment statement, CONTINUE cannot be used. DIM may be used in the Immediate Mode. INPUT and IF...THEN may not be used in the Immediate Mode. Section 9-5 introduces the use of FOR...TO/NEXT in the Immediate Mode.

9-3   THE HALT/STEP KEY, TRACE, SELECT P

The HALT/STEP Key

As its name implies, the HALT/STEP key has a dual purpose. Depressed once, it waits until execution of the current program statement is complete, then halts execution and displays the colon (:_) symbol. This is its "halt" function. The effect is just as if the currently executing statement were followed by a STOP statement, except that the word "STOP" is not displayed.

If you want to temporarily stop program execution, the HALT/STEP key is the ideal choice. After HALT/STEP, you can use CONTINUE to resume execution, just as you would after a STOP statement. The same limitations on the use of CONTINUE apply with interruptions caused by HALT/STEP, as with ones caused by STOP.

In addition to allowing the use of CONTINUE, HALT/STEP is preferable to RESET since, unlike RESET, it allows the current statement to finish execution before interrupting the program. This is of major importance during tape and disk operations, when an instantaneous interruption can easily leave half-written, unintelligible information on the tape or disk.

Keyed once, HALT/STEP stops program execution. Keyed a second time, or keyed after STOP has interrupted execution, HALT/STEP lists and executes the next program statement, and halts again. Thus, it permits you to "step through" the program, executing one statement at a time, and displaying the statement as it is executed. This can be very useful in finding program bugs.

Example 9.4   Program with Bug

```
110 REM PROGRAM WITH BUG - SUM IS INCORRECT
120 PRINT   "CALCULATE SUM OF FIRST N ODD INTEGERS"
130 INPUT "ENTER N", N
140 S = 1
150 IF N = 1 THEN 200
160 FOR I = 2 TO N
170     D = D+2
180       S = S+D
185 STOP "LINE 185"
190 NEXT I
200 PRINT "SUM="; S
```

Example 9.4 reproduces the program with a bug (Example 9.2); however, a STOP statement has been added at line 185. When executed, this program produces output such as this:

```
READY
:RUN
ENTER N? 5

STOP LINE 185
:_
```

Keying HALT/STEP displays and executes the next statement, and then restores the colon. The result looks like this:

```
STOP LINE 185
:

190 NEXT I
:_
```

If HALT/STEP is keyed repeatedly from the position shown above, the results look like this:

```
170     D = D+2

:
180       S = S+D

:
185 STOP "LINE 185"

STOP LINE 185
:
190 NEXT I

:
170     D = D+2
```

```
:
180      S = S+D

:
185 STOP "LINE 185"

STOP LINE 185
:_
```

A program executed in this manner works exactly as it would if executed normally. At any point, the normal execution can be resumed with the CONTINUE command (provided that no action has been taken which would invalidate the use of CONTINUE). HALT/STEP output occurs at the device selected for Console Output.

TRACE

While HALT/STEP is an important debugging feature in that it allows you to follow the sequence of execution, its power can be considerably enhanced through the use of the TRACE mode. When the system is executing a program in TRACE mode, each time a variable is assigned a value, the variable name and its new value are printed. Each time the "normal sequence of execution" is altered, the message "TRANSFER TO line number" is printed showing the line number branched to.

The system is put into TRACE mode by executing the BASIC statement TRACE. TRACE may be executed as an Immediate Mode statement, or it may be included as a line-numbered program statement. The system remains in TRACE mode until a CLEAR command is executed, RESET is keyed, or TRACE OFF is executed.

To see how TRACE works, execute the program of Example 9.4, and when it reaches the STOP, key TRACE (EXEC). Output looks like this:

```
:RUN
CALCULATE SUM OF FIRST N ODD INTEGERS
ENTER N? 5

STOP LINE 185
:TRACE
:_
```

Now, if you key HALT/STEP, TRACE and HALT/STEP together produce the following output:

```
190 NEXT I
I= 3
TRANSFER TO 170
:_
```

In this display you can see that HALT/STEP has listed line 190 and executed it. Since the system is in the TRACE mode, the effect of NEXT I is displayed; I is assigned the value 3 and a branch to 170 is made.

Repeatedly keying HALT/STEP produces this:

```
        170     D = D+2
        D= 4


        :
        180     S = S+D
        S= 7


        :
        185 STOP "LINE 185"

        STOP LINE 185
        :
        190 NEXT I
        I= 4
        TRANSFER TO 170


        :
        170     D = D+2
        D= 6


        :
        180     S = S+D
        S= 13
        :_
```

Trace can be turned off by keying

    :TRACE OFF (EXEC)

If it is turned off in this way, the CONTINUE command allows normal program execution to be continued from the next statement.

TRACE and HALT/STEP together provide the most powerful means of observing program operation. TRACE mode can be used by itself, though, permitting the system to execute instructions at normal speed. This can be especially useful if a printer is available to record the output from the trace. Output generated by TRACE mode always appears on the Console Output device; therefore, to use a printer, you must first execute a statement such as

    :SELECT CO 215

With the STOP statement removed and a printer selected for PRINT and CO output, Example 9.4 produces the following output:

128

```
:RUN
CALCULATE SUM OF FIRST N ODD INTEGERS
ENTER N? 5

S= 1
I= 2
D= 2
S= 3
I= 3
TRANSFER TO 170
D= 4
S= 7
I= 4
TRANSFER TO 170
D= 6
S= 13
I= 5
TRANSFER TO 170
D= 8
S= 21
I= >
SUM= 21
```

Used in this fashion, TRACE allows you to easily review a malfunctioning program. The symbol => is used in trace output when a NEXT statement terminates a loop.

SELECT P

If a printer is not available, you may wish to slow down execution to make the trace output easier to read on the CRT. The system can be instructed to pause for a specified time after each line of output. This is done by executing a statement such as

```
:SELECT P 1
```

In this statement the P stands for "pause", and the 1 says that the system is to pause 1 sixth of a second after each line of output. A statement such as

```
:SELECT P 5
```

causes a 5/6 second pause, and is usually better for observing trace output. The maximum pause value which may be used is 9 for a 9/6, or 1.5, second pause. The digit used always specifies the pause time in sixths of a second.

Once a pause has been selected, it remains selected, until another pause is selected, or the system is Master Initialized. No pause or "pause off" may be selected by executing

```
:SELECT P
```

9-4   THE RENUMBER, CLEAR P, AND CLEAR V COMMANDS

RENUMBER

RENUMBER is a powerful command that rapidly assigns new line numbers to a program, or portion of a program, in memory. It preserves program function by inserting the appropriate new line number in all statements that refer to a specific line. For example, the program at the top of Example 9.5 has been renumbered at the bottom. Notice the changes made to the branch addresses at lines 12 and 14.

Example 9.5   The Effect of RENUMBER

Program before Renumbering
        10 INPUT N
        12 IF INT(N/2) = N/2 THEN 15
        13 PRINT "NUMBER IS ODD"
        14 GOTO 16
        15 PRINT "NUMBER IS EVEN"
        16 STOP

Program after Renumbering
        10 INPUT N
        20 IF INT(N/2) = N/2 THEN 50
        30 PRINT "NUMBER IS ODD"
        40 GOTO 60
        50 PRINT "NUMBER IS EVEN"
        60 STOP

The general form of the RENUMBER command is:

RENUMBER	[line number]	[,line number]	[,integer]
	The first line to be renumbered. All lines with numbers greater than or equal to this number are renumbered. (If omitted, the entire program is renumbered.)	The new line number that the first re-numbered line is to receive. (If omitted, the new number of the first renumbered line equals the increment between the new line numbers.)	The increment between the new line numbers 0 [integer] 100. (If omitted, the increment is 10.)

The renumbering shown in Example 9.5 was effected with the simple command

        :RENUMBER

with no additional parameters. The entire program was renumbered with an increment of 10 and a new starting line number of 10.

130

Now look at some other ways the original program, shown at the top of Example 9.5, can be renumbered. Example 9.6 shows the original program listing and the listing which results from executing

```
:RENUMBER 10, 200
```

Example 9.6   The Effect of RENUMBER 10, 200

```
:LIST
10 INPUT N
12 IF INT(N/2) = N/2 THEN 15
13 PRINT "NUMBER IS ODD"
14 GOTO 16
15 PRINT "NUMBER IS EVEN"
16 STOP
:RENUMBER 10,200
:LIST
200 INPUT N
210 IF INT(N/2) = N/2 THEN 240
220 PRINT "NUMBER IS ODD"
230 GOTO 250
240 PRINT "NUMBER IS EVEN"
250 STOP
:_
```

In Example 9.6, 200 is specified as the new line number for the first line to be renumbered. In this case the same effect could have been achieved by omitting the first line number parameter, in a command such as RENUMBER, 200. The comma preceding 200 indicates that the first parameter is omitted.

In Example 9.7 all three RENUMBER parameters are used. The renumber command RENUMBER 13, 15, 2 says "Renumber the lines starting at line 13, change line 13 to line 15, and from there increment each line by 2."

Example 9.7   Using All the RENUMBER Parameters

```
:LIST
10 INPUT N
12 IF INT(N/2) = N/2 THEN 15
13 PRINT "NUMBER IS ODD"
14 GOTO 16
15 PRINT "NUMBER IS EVEN"
16 STOP
:RENUMBER    13,15,2
:LIST
10 INPUT N
12 IF INT(N/2) = N/2 THEN 19
15 PRINT "NUMBER IS ODD"
17 GOTO 21
19 PRINT "NUMBER IS EVEN"
21 STOP
```

RENUMBER is a command only. It cannot be part of a program.

CLEAR P

The CLEAR P command offers a means of clearing from memory all or part of the program text, without disturbing any variables. Its general form is

CLEAR P [line number [,line number]]

If CLEAR P is executed with no line numbers specified, then the entire program text is cleared. If a single line number is used, as in

:CLEAR P 220

all lines, from the indicated line through the highest numbered line, are cleared. If two line numbers are specified, as in

:CLEAR P 220, 1100

all program lines, from the indicated first line through the indicated second line, inclusive, are deleted. CLEAR P is not programmable.

CLEAR V

The CLEAR V command clears all variables from memory, but leaves the program text intact.   It has no optional parameters, and is used by simply keying

:CLEAR V (EXEC)

It is a command only, that is, it may not be programmed.

9-5   MULTISTATEMENT LINES

Thus far, in all our example programs, each BASIC statement has been given a line number, or, in another way of looking at it, each numbered line has had only one statement on it. However, your Wang system allows any number of statements to appear on a single line, provided that the maximum line length of 192 keystrokes is not exceeded. Statements are separated from one another by colons, and are executed sequentially, left to right, through the line.   The use of multistatement lines can allow a program to occupy less memory space and to execute somewhat faster.

Branch statements such as GOTO and IF...THEN can only cause a branch to a line number, which means to the first statement on a line, not to the second, third or fourth statement on a line. There is no way to branch into the middle of a line with a branch statement that branches to a line number. Therefore, if you want to branch to a statement with a GOTO or IF...THEN, the statement to be branched to must be the first statement on a line.

There are restrictions on the use of multistatement lines with some BASIC statements. None of the statements discussed in previous chapters have any such restrictions. When statements with such restrictions are introduced, the restrictions will be noted.

Using multistatement lines, the original inventory program (Example 2.2) could have been written on two lines. It is shown in Example 9.8 rewritten in this fashion.

Example 9.8 Multistatement Lines Used for the Inventory Program (Example 2.2)

```
10  LET  I=42500:PRINT "OPENING INVENTORY="; I
20  INPUT  "NUMBER OF TONS RECEIVED (+) OR SOLD (-)",  T:LET I=I
+T:PRINT  :PRINT "TONS ON HAND ="; I:IF I > = 100 THEN 20:PRINT "R
EORDER COAL IMMEDIATELY:  INVENTORY BELOW 100 TONS":GOTO 20
```

In Example 9.8, line 10 contains two BASIC statements. The first one

```
LET I=42500
```

is the statement that originally appeared alone on line 10. At the end of this statement a colon appears. This colon was entered as the line was being keyed in, to indicate the beginning of the second statement. The second statement is taken from line 20 of the original program. During execution, the system will execute the two statements on line 10 just as if they were on two separate lines as before.

The INPUT statement, statement 30 of the original program, must be branched to from later statements in the program. Therefore, this statement must appear on a new line, despite the fact that line 10 has not received 192 keystrokes of information. Line 20 now receives all the remaining statements. Each statement is separated from the next by a colon. Notice that the branch statement IF...THEN is embedded in the multistatement line; it is only the statement branched to which must be the first statement on a line.

If execution is stopped, with a STOP statement or the HALT/STEP key, in the middle of a multistatement line, CONTINUE continues execution with the next statement in the line. If HALT/STEP is used to step through a multistatement line, the entire unexecuted portion of the line appears on the CRT, with the statement which is to be executed next in the leftmost position.

The use of multistatement lines can make a program much more difficult to read. This is obvious from a comparison of Examples 2.2 and 9.8. Even when the number of statements per line is reduced to 2 or 3, readability is sometimes impaired. Therefore, in this manual we avoid the use of multistatement lines in examples, and suggest that it may be a good idea for you to do the same. However, one form of multistatement line generally adds to program clarity. This is the use of a REM as the second statement on a two statement line. For example,

```
10 LET I = 42500:REM INITIAL BALANCE
```

Since multistatement lines do offer some advantages, such as reduced memory usage and increased execution speed, a special utility program is available from Wang Laboratories which "compresses" a program by, among other things, building long multistatement lines wherever doing so is consistent with maintaining executability.

Multistatement Lines in Immediate Mode

A multistatement line can be executed in the Immediate Mode, that is, without a line number, by simply separating the statements with a colon. Since Immediate Mode statements are executed and lost when (EXEC) is keyed, they are rarely more convenient than writing a short program. If they fail to execute due to a syntax error, they must be completely reentered. However, when program execution has been stopped, and CONTINUE is to be used, a multistatement Immediate Mode operation can occasionally prove useful. In a multistatement Immediate Mode operation FOR...TO and NEXT can be used.

The following multistatement line can be executed in the Immediate Mode. It calculates and prints the value of 25 factorial.

        :P=1 : FOR I=1 TO 25 :P=P*I :NEXT I :PRINT P

The FOR...TO and NEXT statements execute just the way they would if this were written:

        10 P=1
        20 FOR I=1 TO 25
        30 P=P*I
        40 NEXT I
        50 PRINT P

9-6   THE END STATEMENT

The END statement ends program execution, and displays the message:

    END PROGRAM
    FREE SPACE = XXXXX
    :_

where XXXXX is the approximate amount of memory (in bytes) which remains unallocated to program text or variable space at the time that the END statement is executed.

The message appears on the Console Output device (normally the CRT). After an END statement has been executed, the CONTINUE key cannot be used to resume execution.

The END statement is optional. Your Wang system ends program execution whenever, in the normal sequence of execution, it can find no higher numbered line to execute.

The END statement can be executed in the Immediate Mode. Then, its only function is to report on the amount of memory free space. Section 9-7 discusses in more detail the way memory space is used by your Wang system, and the significance of the END statement's free space report.

9-7   MEMORY USAGE BY PROGRAM TEXT AND VARIABLES

In Chapter 8 we said that an alphanumeric variable would automatically be given a maximum length of 16 characters, unless a DIM statement specifies a different maximum length. When speaking about the length of an alphanumeric variable, it is customary to speak in terms of character capacity, the number of alphanumeric characters which the variable can hold. If an alphanumeric variable is dimensioned to a length of 40 characters, then the system automatically allocates to the variable enough memory capacity to hold 40 alphanumeric characters.

The amount of memory capacity allocated to store a single alphanumeric character exactly corresponds to the fundamental unit of memory of the 2200 system. Outside of the specific context of storing alphanumeric characters, this unit of memory capacity is called a "byte". Thus,

> The memory capacity allocated
> to store one alphanumeric   } = one byte.
> character

At the level of raw capacity, if your 2200 system is equipped with 8K of memory, meaning exactly 8192 bytes of memory, then its character storage capacity would be 8192 characters. However, in fact, memory must be used for many things other than simply storing characters. This means that the effective capacity is somewhat less than the raw capacity.

In every 2200 system approximately 700 bytes are used by the system itself, and are not directly available to the user. The system uses this memory in a variety of ways including the storage of FOR/NEXT data and other information. The rest of memory is available for your program text and variables.

Your Wang 2200 system automatically converts each BASIC keyword and function name to a special code, before storing it as program text in memory. Each of these special codes occupies just one byte of memory. This feature of your Wang system greatly reduces the amount of memory required for program storage, and increases the speed of execution as well.

If you wish to know how much memory is available while you are keying in a new program, you can simply execute an END statement in the Immediate Mode at any time. However, your program will probably need memory capacity for variables as well as for the program text itself. Thus, even if the program text fits into your system, it will not be executable unless there is enough memory capacity for the variables also.

When the system receives a RUN command, it quickly searches through the entire program text, allocating the proper amount of memory space for each of the variables it encounters. Only after it has established space for every variable does it actually begin to execute the program statements. (If there is not enough memory capacity for the program text and the required variable space, the system signals an ERR 02 indicating the overflow.) If you want to know how much actual memory capacity is left over after program text and variables are accounted for, you must first issue a RUN command, and then

execute an END statement. For example, you could temporarily put an END statement at line 1 of your program. When you key RUN (EXEC), the system will set aside memory space for all of the program variables, and then begin execution with the END statement.

Each alphanumeric variable used in a program requires the number of bytes of its dimensioned length (16 bytes if a DIM statement is not used), plus five bytes of special control information. The control information permits the system to find the variable when it needs to do so. Each numeric variable requires eight bytes, plus five bytes, of control information. This is summarized in Table 9.1.

Table 9.1   Memory Space Required per Variable

Type	Space Usable by Value	Space for Control Information	Total Space
Numeric	8 bytes	5 bytes	13 bytes
Alphanumeric	DIM length 1-64 bytes (16 if no DIM statement)	5 bytes	6-69 bytes

Review of Chapter 9

1.   The STOP statement stops program execution, and displays the stop message and the colon.

2.   Depressed once during program execution, the HALT/STEP key stops execution at the end of the currently executing statement. Depressed a second time, it lists and executes the next program statement.

3.   After program execution has been stopped with STOP or HALT/STEP, it may be resumed by keying CONTINUE (EXEC) (provided that certain actions which prohibit continuation have not occurred while the program has been stopped).

4.   Many BASIC statements can be executed immediately by being entered without a line number. These "Immediate Mode" operations can be useful as a debugging aid and as a means of performing quick calculations.

5.   TRACE mode outputs a message each time a branch is effected, and displays the result of each assignment. A pause can be selected by means of SELECT P.

6.   The RENUMBER command renumbers the lines of a program, or portion of a program.

7.   CLEAR P allows all of a program text, or a portion of it, to be cleared without disturbing variables. CLEAR V clears variables without disturbing program text.

8.   Any number of statements can appear on a single program line, up to the maximum line length of 192 keystrokes. Statements must be separated by colons.

136

9. The END statement ends program execution, and displays the amount of memory that is unused by program text and variables.

10. The amount of space allocated to store a single alphanumeric character is called a "byte".

11. BASIC keywords occupy just one byte of memory in your Wang system. Alphanumeric variables require as many bytes as their defined character length (1-64), plus five control bytes. Numeric variables require eight bytes, plus five control bytes.

CHAPTER 10
THE ON STATEMENT WITH GOTO

10-1   <u>SIMPLE USE OF ON...GOTO</u>

A single IF...THEN statement provides two alternative execution paths in a program. Either the normal sequence of execution prevails, or a branch is effected to a specified line number. If more than two alternatives are required, IF...THEN statements can be stacked to provide for all of them. An example of such stacking is shown in Example 10.1. Asterisk REM's have been inserted to highlight the IF...THEN's.

Example 10.1 A Program Segment with Stacked IF...THEN's

```
        .
        .
        .
530 REM CHOOSE JOB CATEGORY
540     PRINT , "1. CARPENTER", "4. ROOFER"
550     PRINT , "2. LABORER",   "5. FOREMAN"
560     PRINT , "3. ROD LAYER", "6. NON-UNION"
570     PRINT
580     INPUT "ENTER NUMBER TO CHOOSE JOB CATEGORY", C
590 REM BRANCH TO UPDATE CATEGORY DATA
600 REM ****************************************
610     IF C = 1 THEN 730
620     IF C = 2 THEN 760
630     IF C = 3 THEN 810
640     IF C = 4 THEN 850
650     IF C = 5 THEN 880
660     IF C = 6 THEN 940
670 REM ****************************************
680 REM SELECTION INVALID
690     PRINT "INVALID. REENTER."
700     PRINT
710     GOTO 540
720 REM UPDATE CARPENTER HOURS
730     H1 = D + H1
740     GOTO 970
750 REM UPDATE LABORER HOURS, INSURANCE, VACATION FUND
760     H2 = D + H2
770     I9 = I9 + (D*.052)
780     V9 = V9 + (D*.1175)
790     GOTO 970
        .
        .
        °
970 REM NEXT OPERATION
```

138

In this example the operator enters a selection number, 1-6, to choose a worker's job category. The entry is made at line 580 and goes into variable C. The six IF...THEN statements then test C for each possible valid entry, and branch to the proper routine. If an invalid entry is made, none of the IF...THEN branches are taken; the normal sequence of execution produces a reenter message, and a branch back to the beginning of the operation.

Example 10.1 is a simple case of a common programming requirement: the value of an expression, in this case C, must determine which of many possible branches is to be taken. The BASIC language provides a single statement to make programming in this situation a simple and concise operation. This statement is the ON statement. The ON statement has two forms, ON...GOTO and ON...GOSUB. Though their operation is similar, we do not take up ON...GOSUB until Chapter 13.

In Example 10.2 a single ON...GOTO statement is substituted for the six IF...THEN's of Example 10.1. If the value of C is 1, the ON...GOTO statement effects a branch to the first listed line number, 730. If C is 2, it branches to the second line number, and so on, up to 6 which effects a branch to line 940. If C is less than 1, or equal to or greater than 7, the ON...GOTO statement lets the normal sequence of execution prevail; this leads to the reenter message.

Example 10.2 ON...GOTO Substituted for the Stack of IF...THEN's in Example 10.1

```
     .
     .
     .
530 REM CHOOSE JOB CATEGORY
540     PRINT , "1. CARPENTER", "4. ROOFER"
550     PRINT , "2. LABORER",   "5. FOREMAN"
560     PRINT , "3. ROD LAYER", "6. NON-UNION"
570     PRINT
580     INPUT "ENTER NUMBER TO CHOOSE JOB CATEGORY", C
590 REM BRANCH TO UPDATE CATEGORY DATA
600 REM *****************************************
610     ON  C  GOTO 730, 760, 810, 850, 880, 940
620 REM *****************************************
680 REM SELECTION INVALID
690     PRINT "INVALID. REENTER."
700     PRINT
710     GOTO 540
720 REM UPDATE CARPENTER HOURS
730     H1 = D + H1
740     GOTO 970
750 REM UPDATE LABORER HOURS, INSURANCE, VACATION FUND
760     H2 = D + H2
770     I9 = I9 + (D*.052)
780     V9 = V9 + (D*.1175)
790     GOTO 970
     .
     .
     .
970 REM NEXT OPERATION
```

139

The general form of the ON statement with GOTO is

ON expression GOTO line number [,line number...]

The square brackets and ellipsis indicate that line numbers beyond the first are optional, and may be added without limit.

The ON...GOTO statement first evaluates the expression following "ON". If the integer value of the expression is 1, a branch is taken to the first line number following GOTO. If the integer value of the expression is 2, a branch to the second line number is effected. In general, if the integer value of the expression is n, a branch is effected to the nth line number following "GOTO". If n is greater than the number of lines specified, or if n is less than 1, no branch is taken; the normal sequence of execution prevails.

Differences between Stacked IF...THEN's and ON...GOTO

ON...GOTO, in Example 10.2, may be a good functional substitute for the six IF...THEN's of Example 10.1. However, there is one difference which you should be aware of, and may want to counteract. In Example 10.1 if a non-integer value, such as 2.5, is entered, no branch is taken, since 2.5 does not equal any of the acceptable values. An entry of 2.5, therefore, yields a "RE-ENTER" message. In Example 10.2 the ON...GOTO statement uses only the integer portion of the expression; it treats an entry of 2.5 as if it were 2. An entry of 2.5 effects a branch to line 750. Thus, in Example 10.2 any entry in the range $1 \leq C < 7$ causes a branch, whereas in Example 10.1 only the values 1, 2, 3, 4, 5 or 6 cause a branch.

In many circumstances this sort of difference may be immaterial. However, if necessary, the difference can easily be eliminated from Example 10.2. Example 10.3 adds a test for a non-integer entry to the program in 10.2. The new test is highlighted by asterisks. If a non-integer is entered, this test effects a branch to the reentry routine. Therefore, Example 10.3 functions identically to Example 10.1.

Example 10.3 Testing for a Non-Integer Expression before ON...GOTO

```
538 REM CHOOSE JOB CATEGORY
540     PRINT , "1. CARPENTER", "4. ROOFER"
550     PRINT, "2. LABORER",    "5. FOREMAN"
560     PRINT, "3. ROD LAYER", "6. NON-UNION"
570     PRINT
580     INPUT "ENTER NUMBER TO CHOOSE JOB CATEGORY", C
581 REM ****************************************
582 REM NON-INTEGER ENTRY?
584     IF INT(C) <> C THEN 690
585 REM ****************************************
590 REM BRANCH TO UPDATE CATEGORY DATA
610     ON C GOTO 730, 760, 810, 850, 880, 940
680 REM SELECTION INVALID
690     PRINT "INVALID. REENTER."
700     PRINT
710     GOTO 540
```

```
720 REM UPDATE CARPENTER HOURS
730     H1 = D + H1
740     GOTO 970
750 REM UPDATE LABORER HOURS, INSURANCE, VACATION FUND
760     H2 = D + H2
770     I9 = I9 + (D*.052)
780     V9 = V9 + (D*.1175)
790     GOTO 970
    .
    .
    .
970 REM NEXT OPERATION
```

One preliminary word of caution is in order about the ON...GOTO statement, though. Later, after you have learned about subroutines and subscripted variables, you will have two powerful tools for using the same program statements to effect similar operations. With these techniques available, when you find yourself using an ON...GOTO, you should ask, "Are these separate operations, which are being branching to, similar enough to be combined into one?" If you ask yourself this question, it should help you to write better, more efficient programs.

10-2 USING MORE COMPLEX EXPRESSIONS IN ON...GOTO

In Example 10.2 the expression used in the ON...GOTO statement is a simple numeric variable. However, since any valid expression is allowed, more complex forms can be used when needed.

Often the SGN function is handy for use with ON...GOTO. Suppose, for example, that you want to do three different operations depending upon whether K (or any expression) is less than, equal to, or greater than zero. SGN(K) returns -1, 0, or 1 respectively for these conditions. SGN(K) + 1, then, always yields 0, 1, or 2. This range is appropriate for use with ON...GOTO, as follows:

Example 10.4 A Simple Use of SGN() with ON...GOTO

```
10 REM USING THE "SGN" FUNCTION WITH "ON...GOTO"
    .
    .
    .
120     ON SGN(K) + 1 GOTO 170,210
130 REM K < 0
140 REM OPERATION FOR K < 0 GOES HERE
    .
    .
    .
170 REM K=0
180 REM OPERATION FOR K=0 GOES HERE
    .
    .
    .
210 REM K > 0
220 REM OPERATION FOR K>0 GOES HERE
    .
    .
    .
```

141

The range of an expression over regular intervals can determine an ON...GOTO branch by simply dividing the expression by the interval size and adding one if necessary. For example, if K is positive, to branch as specified on the intervals

$$0 \leq K < 800 \qquad \text{branch to } 900$$
$$800 \leq K < 1600 \qquad \text{branch to } 800$$
$$1600 \leq K < 2400 \qquad \text{branch to } 700$$
$$2400 \leq K < 3000 \qquad \text{branch to } 600$$
$$3000 \leq K \qquad \text{no branch, execute next statement}$$

This ON statement suffices:

        490 ON (K/800)+1 GOTO 900, 800, 700, 600

Review of Chapter 10

1. The general form of the ON statement with GOTO is

    ON expression GOTO line number [,line number...]

2. The ON...GOTO statement first evaluates the expression following the keyword "ON". If the integer value of the expression is n, a branch is effected to the nth line number following the keyword "GOTO". If n is greater than the number of lines specified, or if n is less than 1, no branch is taken.

3. The SGN() function is often useful for producing an expression with values in the range needed for ON...GOTO.

CHAPTER 11
LISTS

11-1   INTRODUCING LISTS, DIM REVISITED

It is often desirable to arrange information in a list. In the original inventory program, Example 2.2, there was just one product, coal.   Suppose, though, that there are six products in the inventory. We might want to maintain a quantity-on-hand list that looks something like this:

Quantity-On-Hand List

Item	Product Number	Quantity in Units
1	X407	3455
2	D912	1200
3	T612D	120
4	E711	145
5	A816A	192
6	C4121	300

If we receive a shipment of 100 units of product E711, to update the quantity-on-hand list we go down the list to the 4th item, and update the quantity value which we find there.   In this case, we set the quantity to the sum of the present quantity, 145, plus 100.

Suppose that we want to write a program to accomplish this simple task. Make it even simpler for the moment by assuming that the operator enters the item number, 1-6, rather than the product number.   This eliminates the need to search through the list for the right product.

We will need six variables in which to keep the quantities. We can use Q0-Q5. A simple program which updates these variables with entered quantities is shown in Example 11.1.

Example 11.1 A Six-item Inventory Program, without List Variables

```
110 REM * A ROUTINE TO UPDATE ONE OF SIX INVENTORY BALANCES
120 REM * WITHOUT THE USE OF LIST VARIABLES
130 REM    ACCEPT ITEM NUMBER
140        INPUT "ENTER ITEM NUMBER (1 - 6)", A
150 REM    TEST ENTRY
160        IF A < 1 THEN 200
170        IF A > 6 THEN 200
180        IF A = INT(A) THEN 240
190 REM    ENTRY INVALID
200        PRINT
210        PRINT "INVALID.  REENTER"
220        GOTO 140
230 REM    ITEM ENTRY OK.  NOW ACCEPT TRANSACTION
240        INPUT "ENTER INVENTORY TRANSACTION AMOUNT (+ OR -)", B
250 REM    BRANCH ON ITEM NUMBER
260        ON A GOTO 280, 320, 360, 400, 440, 480
270 REM    UPDATE Q0
280        PRINT "OLD BALANCE=", Q0, "NEW BALANCE=", Q0+B
290        Q0 = Q0 + B
300        GOTO 500
310 REM    UPDATE Q1
320        PRINT "OLD BALANCE=", Q1, "NEW BALANCE=", Q1+B
330        Q1 = Q1 + B
340        GOTO 500
350 REM    UPDATE Q2
360        PRINT "OLD BALANCE=", Q2, "NEW BALANCE=", Q2+B
370        Q2 = Q2 + B
380        GOTO 500
390 REM    UPDATE Q3
400        PRINT "OLD BALANCE=", Q3, "NEW BALANCE=", Q3+B
410        Q3 = Q3 + B
420        GOTO 500
430 REM    UPDATE Q4
440        PRINT "OLD BALANCE=", Q4, "NEW BALANCE=", Q4+B
450        Q4 = Q4 + B
460        GOTO 500
470 REM    UPDATE Q5
480        PRINT "OLD BALANCE=", Q5, "NEW BALANCE=", Q5+B
490        Q5 = Q5 + B
500 REM    NEXT OPERATION
510        GOTO 110
```

One of the most conspicuous features of this program is that from line 270 to 490 essentially the same program steps are repeated six times over. The only difference between one of the six update routines and another is the variable name. Whenever you see this kind of repetition in a computer program, you should look for a way to improve the program by using a single set of statements to perform the similar operations.

With the programming features of BASIC which we've covered thus far, though, Example 11.1 is about the best we can do. What is needed is to be able to keep a list of variables to contain the inventory quantities. Such a list should allow us to refer to a specific variable on the list by saying in

effect, "I want the 1st variable (or the 2nd, or the 3rd, etc.) in the inventory quantity list." Furthermore, we should be able to say which variable in the list we want by means of an expression. That is, if J equals 4, we should be able to say "Get me the Jth variable in the list" and we should get the 4th variable, since 4 is the value of the expression J.

BASIC offers just this capability. You can set up a list of variables, and give the entire list a name. If you set up a list of six numeric variables, the individual variables on the list might be known as:

    Q1(1)
    Q1(2)
    Q1(3)
    Q1(4)
    Q1(5)
    Q1(6)

The variables on this list may be referred to via

    Q1(expression)

provided that the value of the expression is greater than or equal to 1 and less than 7. This means that, for example, if I=2, then

    10 Q1 (I) = 50

will assign 50 to the second variable on the list. In $Q1(5) = 70$, the constant 5 is used to specify the 5th variable on the list.

Only the integer portion of the expression is used in specifying the variable. Thus, Q1(2.2) is equivalent to Q1(2); it specifies the second variable on the list. Q1(11/2) is evaluated as Q1(5.5) and specifies the 5th variable on the list.

Dimensioning Lists

In order to use list variables, you must first tell the system to set aside space for a list which contains the desired number of variables. This is done with a dimension or DIM statement. Thus,

    DIM Q1(6)

tells the system, "Set up a list of six numeric variables and call it Q1()." (The symbol Q1() is generally used to refer to the entire list, to avoid confusion with the simple numeric variable Q1, which is completely separate from the list Q1().) In the DIM statement an expression may not be used to specify the number of variables on the list. An integer between 1 and 255 must be used. (255 is the maximum number of variables in any list.)

The names which can be used for lists are the same as the names which can be used for ordinary variables, i.e., A-Z and A0-A9,B0-B9,C0-C9 ... Z0-Z9. However, whenever we refer to an entire list, we will use empty parentheses () to indicate that it is a list we are talking about, and not a single variable. For example, A2() refers to a list of numeric variables

beginning with A2(1) and extending to A2(n), where n is the number of variables in the list. A2 refers to the ordinary numeric variable A2, which is independent and not a part of any list. The ordinary numeric variable A2 as well as the list A2() may be used without conflict in the same program.

Example 11.2 shows how the repetition in lines 280 to 490 of Example 11.1 can be eliminated by using list variables for the quantity-on-hand. The new statements are enclosed in REM asterisks.

Example 11.2   Rewriting Example 11.1 Using List Variables

```
110 REM * THE OPERATION OF FIG 11.2 NOW USING LIST VARIABLES
111 REM * FOR THE QUANTITY BALANCES
112 REM ******* DIMENSION THE LIST *************
115       DIM Q1(6)
117 REM ****************************************
130 REM   ACCEPT ITEM NUMBER
140       INPUT "ENTER ITEM NUMBER (1 - 6)", A
150 REM   TEST ENTRY
160       IF A < 1 THEN 200
170       IF A > 6 THEN 200
180       IF A = INT(A) THEN 240
190 REM   ENTRY INVALID
200       PRINT
210       PRINT "INVALID.  REENTER."
220       GOTO 140
230 REM   ITEM ENTRY OK.  NOW ACCEPT TRANSACTION
240       INPUT "ENTER INVENTORY TRANSACTION AMOUNT (+ OR -)", B
250 REM ****************************************
255 REM   UPDATE SELECTED ITEM
260       PRINT "OLD BALANCE=", Q1(A), "NEW BALANCE=", Q1(A) + B
270       Q1(A) = Q1(A) + B
280 REM ****************************************
290 REM   NEXT OPERATION
300       GOTO 110
```

At line 115 the list is established by means of a DIM statement.   Q1(6) in the DIM statement specifies that there are to be 6 numeric variables in the list Q1(). The same rules apply to using DIM statements for setting up lists as for specifying alphanumeric variable length. That is, the DIM statement must precede any reference to the variable in the program. Furthermore, once the number of variables in the list is set by a DIM statement, any attempt to change it with another DIM statement produces an ERR message.

At line 140 the operator enters the item number into variable A, just as in Example 11.1. The entry is then tested in lines 160 to 180 to ensure that it is an integer within the acceptable range. At line 240 the transaction amount is entered into B.

Line 260 prints "OLD BALANCE=" followed by the value of the variable Q1(A). The expression within the parentheses, A, is evaluated to find which variable, on the list Q1(), is being referred to. Since A received the item number at line 140, Q1(A) is the variable which contains the old balance for the selected item. The new balance is equal to the old balance in Q1(A), plus the transaction in B; therefore the new balance is Q1(A) + B.   Line   270 replaces the old value with the new.

Summary

Using list variables allows you to select a variable from a list by means of the value of an expression. Whenever variables are related by virtue of similar operations which must be performed on them, you should consider whether efficiency might be improved by using list variables instead of individual variables. With list variables you can write one operation, and let the value of an expression specify the variable on which the operation is to be performed.

11-2 ALPHANUMERIC LISTS

Wang 2200 BASIC also permits alphanumeric list variables. The names of such lists are the same as for lists of numeric variables, except that a $ is inserted. For example,

        DIM A$(12)

tells the system to set up a list known as A$(), containing 12 alphanumeric variables. The variables are identified as A$(1) through A$(12). Similarly,

        DIM C8$(42)

sets up a list of 42 alphanumeric variables with the list named C8$(). It is possible to use the same two characters to name different alphanumeric and numeric lists. For example,

        DIM C8$(42), C8(4)

sets up two completely separate lists, the first alphanumeric, the second numeric. Furthermore, the individual variables C8$ and C8 could also be used without any conflict with these two lists; all four are totally distinct.

When using a DIM statement with alphanumeric variables, you must carefully distinguish between a length specification, and the specification of the number of variables in a list. For example,

        DIM A2$4

specifies that the individual alphanumeric variable A2$ is to be long enough to hold four characters. It might look like this in memory:

        A2$ | Δ | Δ | Δ | Δ |

However,

        DIM A2$(4)

specifies that a list, A2$(), containing 4 alphanumeric variables is to be set up. The length of each variable is 16 characters, since that is the length

147

the system always uses unless told otherwise. In memory, the result of DIM A2$(4) would look something like this:

A2$(1) [Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ]

A2$(2) [Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ]

A2$(3) [Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ]

A2$(4) [Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ]

It is possible to set up a list of alphanumeric variables with the maximum lengths of the variables set to other than 16 characters. For example,

DIM A2$(4)6

sets up a list containing 4 alphanumeric variables in which each variable can contain a maximum of 6 characters. In memory it would look something like this:

A2$(1) [Δ | Δ | Δ | Δ | Δ | Δ]

A2$(2) [Δ | Δ | Δ | Δ | Δ | Δ]

A2$(3) [Δ | Δ | Δ | Δ | Δ | Δ]

A2$(4) [Δ | Δ | Δ | Δ | Δ | Δ]

11-3 LISTS AND FOR...TO/NEXT LOOPS

List variables make possible the use of FOR...TO/NEXT loops where separate processing would otherwise be required. For example, suppose we want a very simple program to assign the opening inventory data of Figure 11.1 to two lists. One list is alphanumeric, and contains the product number; the other list is numeric, and contains the on-hand quantity. Values are assigned so that corresponding variables receive corresponding values. That is, the product number in the first variable of the product number list has its associated quantity in the first variable of the quantity list, and so on. A program to set up this list is shown as Example 11.3.

Example 11.3 Setting up the Inventory Lists

```
110 REM SETTING UP THE INVENTORY LISTS
120     DIM Q1(6), N$(6)8
130     FOR I = 1 TO 6
140         PRINT "ITEM #"; I,
150         INPUT "PRODUCT NUMBER ", N$(I)
160         PRINT ,
170         INPUT "OPENING BALANCE", Q1(I)
180         PRINT
190     NEXT I
200 REM LIST COMPLETE
210     PRINT "LIST COMPLETE"
```

Line 120 of Example 11.3 dimensions the two lists. The numeric quantity list is Q1(). The alphanumeric product number list is N$(). Note that the maximum length of a product number is 8 characters, and both lists contain six variables.

The FOR/NEXT loop sets up a counter variable, I, whose value runs from 1 to 6. At lines 150 and 170 the value of the counter variable determines which variable in each list is to receive the entered value. The first time through the loop, N$(I) and Q1(I) specify the first variable on each list, since I is equal to 1. The second time through they specify the second variable on each list, since I equals 2. The entry of information into successive pairs of variables continues until NEXT I terminates loop processing.

We can now append to Example 11.3 a routine that allows posting of inventory transactions. The complete program, with this appendage enclosed in REM asterisks, is shown in Example 11.4.

Example 11.4   Adding Inventory Posting to Example 11.3

```
110 REM SETTING UP THE INVENTORY LISTS
120     DIM Q1(6), N$(6)8
130     FOR I = 1 TO 6
140         PRINT "ITEM #"; I,
150         INPUT "PRODUCT NUMBER ", N$(I)
160         PRINT ,
170         INPUT "OPENING BALANCE", Q1(I)
180         PRINT
190     NEXT I
200 REM LIST COMPLETE
210     PRINT "LIST COMPLETE"
220 REM ********************************************************
230     PRINT
240     PRINT
250     PRINT "POST INVENTORY CHANGES"
260     PRINT
270 REM ACCEPT PRODUCT NUMBER
280     INPUT "ENTER PRODUCT NUMBER", A$
290 REM SEARCH LIST FOR PRODUCT NUMBER
300     FOR I = 1 TO 6
310         IF N$(I) = A$ THEN 380
320     NEXT I
330 REM NUMBER NOT FOUND
340     PRINT
350     PRINT "PRODUCT NUMBER NOT ON LIST.  REENTER."
360     GOTO 280
370 REM PRODUCT NUMBER FOUND.  SAVE I, THEN FORCE "NEXT" EXIT.
380     K = I
390     I = 6
400     NEXT I
```

```
410 REM
420 REM PRINT OLD BALANCE, THEN ACCEPT TRANSACTION AMOUNT
430     PRINT "PRODUCT #  ", A$, "OLD BALANCE  ="; Q1(K)
440     PRINT
450     INPUT "ENTER INVENTORY TRANSACTION AMOUNT (+ OR -)", B
460 REM UPDATE INVENTORY BALANCE
470     Q1(K) = Q1(K) + B
480     PRINT "PRODUCT #  "; A$,"NEW BALANCE ="; Q1(K)
490     PRINT
500 REM
510 REM RESET VARIABLE VALUES AND RETURN TO MAKE NEXT ENTRY
520     A$ = " "
530     B = 0
540     GOTO 280
550 REM **************************************************
```

The key feature of this posting routine, versus Example 11.2, is that the operator enters the actual product number, rather than the list item number. The program then searches down the list of product numbers, N$(), to find the entered one. The search occurs at lines 300-320. I, the counter for the FOR/NEXT loop, is again used to successively specify each variable in the list. As soon as statement 310 finds a listed product number equal to the entered one, it branches out of the FOR/NEXT loop to line 380. Since this branch out of the loop avoids the normal NEXT statement loop termination, the program must force a NEXT termination. This forced termination takes place at lines 390 and 400. However, the program must first save the value of the variable I. This value is the location on the list of the entered product number. Obtaining this value was the purpose of the search. Therefore, line 380 saves the exit value of I in K.

Notice that if the FOR/NEXT search loop terminates without finding the entered value, it leads to a "not on list" message and a reentry at lines 330 to 360.

At line 420 the entered product number has been found; its list location is stored in K. Lines 430-450 print the product number and old balance, and request that the inventory transaction amount be entered. 470 updates the inventory balance, and 480 prints the results of the transaction.

Values of variables A$ and B are cleared at 520 and 530, since otherwise an accidental keying of (EXEC) would result in the previous values being reprocessed.

The simple search technique shown here is acceptable for short lists of unsorted data.

FOR/NEXT loops and list variables are used together in a wide variety of standard programming operations. For example, a loop can be used to set the values of all list variables to a constant. This is illustrated in Example 11.5.

150

Example 11.5 Assigning a Constant to Each Variable in a List

```
110 REM ASSIGNING A CONSTANT TO LIST VARIABLES
120     DIM C2(40)
130 REM ASSIGN VALUES
140     FOR I = 1 TO 40
150         C2(I) = 10000
160     NEXT I
```

Another use of list variables and loops is illustrated in Example 11.6. The program sorts a numeric list A() into ascending order. The number of items on the list is entered at line 200, but the program presumes that the list itself is dimensioned, and has values in it ready to be sorted.

Example 11.6 Sorting the Values in a Numeric List

```
105 REM A NUMERIC SORT USING LIST VARIABLES
110 REM SORTING A LIST INTO ASCENDING ORDER
115 REM ** A DATA LIST IN A() MUST BE SUPPLIED **
200 INPUT "ENTER NUMBER OF ITEMS ON LIST A()", N
210 FOR J = 1 TO N-1
220     FOR K = 1 TO N-J
230     IF A(K) < = A(K+1) THEN 280
240     REM EXCHANGE VALUES OF A(K) AND A(K+1)
250     T = A(K)
260     A(K) = A(K+1)
270     A(K+1) = T
280     NEXT K
290 NEXT J
```

The sort consists of two loops nested within one another, but all the work is done by the inner loop. Assume we have an unsorted list A() of ten items. Therefore, N, entered at line 200, equals 10. The first time into the loops, J equals 1 and K equals 1. Line 230, then, looks at the first two values on the list, in A(1) and A(1+1). The object is to get the greater of these two values into position A(1+1), i.e., A(2). If the condition at line 230, A(K) <= A(K+1) is false, the values of these two variables must be exchanged, so that the greater value is in A(K+1). Thus, if the condition in 230 is false, the normal sequence of execution prevails and lines 250-270 swap the values of A(K) and A(K+1). If the condition is true, then the present order is acceptable, and statement 230 simply branches over the statements that swap the values.

Now suppose that it happens that, when we start the sort, the greatest value of the entire list is in A(1). In other words, it is at the exact opposite position from where we want it after the sort is complete. As a result of the first time through the inner loop, as outlined above, this value will be in A(2). It will have been exchanged with the value of A(2), since it is greater. Now NEXT K sets K to 2, and statement 230 compares A(2) to A(3). We know that now A(2) has the greatest value on the list, so the second time through the inner loop, it will be exchanged with A(3), and ends up in variable A(3). The next time through A(3) and A(4) are compared and it moves to A(4). The inner loop makes a total of nine comparisons (N-J=9):A(1) and

151

A(2), A(2) and A(3), A(3) and A(4), A(4) and A(5), A(5) and A(6), A(6) and A(7), A(7) and A(8), A(8) and A(9), A(9) and A(10). At the end of all these, the greatest value has sunk to its correct position in A(10). The order of the others, however, remains unchanged. At this point NEXT K terminates the inner loop.

NEXT J (line 290) increments J to 2 and starts the inner loop all over again. Only this time we can omit comparing A(9) and A(10) since we know A(10) has the greatest value. Therefore, in the second complete execution of the inner loop, K runs from 1 to N-2, to make 8 comparisons. At the end of these 8 times through the inner loop, we know that the second greatest value must have sunk to its correct position, A(9).

Again the outer loop causes the entire inner loop to reexecute, but this time 7 comparisons are made (N-3=7) and the 3rd greatest value is in its proper position A(8).

This process repeats itself 9 times (FOR J=1 TO N-1). On the 9th time, the 9th greatest value (which is the next-to-the-least value) has "sunk" to A(2). This leaves the least value in A(1) and the sort is complete.

If you wish to test the sort program, the program in Example 11.7 can precede the sort program to generate a list of random numbers to be sorted. The numbers are integers between 1 and 10000.

Example 11.7 Generating a List of Random Integers

```
10 DIM A(25)
20 FOR I=1 TO 25
30    A(I) = INT (RND(1)*10000+1)
40 NEXT I
```

11-4 A NOTE ON TERMINOLOGY

In this chapter we have been talking about variables such as A(2) and N$(6) and have been calling them "list variables." The forms which contain them and refer to them collectively, such as A() and N$(), we have been calling "lists." The term "list" is an unforbidding word in everyday usage that accurately reflects the structure we are discussing. However, a variety of other terms are so commonly used that you should be familiar with them.

In general any ordered arrangement of variable spaces in memory is known as an array. If the location of any variable in the array can be specified by means of a single value, the array is said to be one-dimensional. Thus the lists we have been discussing are also known as one-dimensional arrays. For example, N$() is an ordered arrangement of variables, and we can pick out any variable on the list N$() by specifying a single value such as 5, as in N$(5). In BASIC, two-dimensional arrays can also be used. They are introduced in Chapter 19.

Array variables are also sometimes referred to as "subscripted variables." In the case of a list, the value which specifies a particular variable, for example 4 in A2$(4), is called the "subscript." List variables are then sometimes referred to as "singly subscripted variables", since just

one subscript is required for specification. This terminology is derived from mathematical notation in which successive variables in a sequence are designated with subscripts, for example, a_1, a_2, a_3, a_4....

Matrix algebra has also contributed a terminology of its own. In connection with matrix operations, what we have called a list is called a "vector." The term "matrix", without any qualification, generally refers to a two-dimensional array.

In contrast to list variables, such as C2(4) and A$(8), we have referred to "ordinary" variables to mean individual variables such as A, D$, F4$, or Z8. These "ordinary" variables are often called "scalar variables", when a contrast with "array type variables" is to be drawn.

Review of Chapter 11

1. BASIC allows you to set up a list of variables. You can refer to any variable on a list by giving the list name and an expression that specifies the location of the variable in the list. For example,

    P(7)

    refers to the seventh variable down the list P().

2. Lists can be alphanumeric or numeric.

3. To use a list, a DIM statement must appear on a lower numbered line than any reference to a variable on the list. The DIM statement specifies the number of variables to be in the list and, optionally, the length of the alphanumeric variables in the list. For example,

    10 DIM A2$(16)5, B(50)

4. The maximum number of variables in a list is 255.

5. FOR...TO/NEXT loops can be used to perform a variety of common list-processing tasks.

6. Lists are often called "one-dimensional arrays". In discussions of matrix operations, and elsewhere, they are also sometimes called "vectors." Ordinary variables such as A$, X, F2, etcetera are sometimes called "scalar variables" to contrast them with vectors and matrices (matrices are introduced in Chapter 19).

153

CHAPTER 12
SUPPLYING CONSTANTS WITH DATA,
READ, AND RESTORE

12-1 INTRODUCING DATA AND READ

Some tasks require that a program make use of a relatively large number of constant values. For example, calculating withheld income taxes may require that a state's set of income intervals and associated percentages be available. A wholesale supplier may have 10 different fixed sets of payment terms for customers. Each set may have a low balance finance charge, a high balance finance charge and a cutoff point separating the two. A single digit in the customer's permanent file indicates which terms apply to the customer. Finally, a program performing calculations on the readings of a scientific measuring device may have to use in its calculations the fixed sensitivity characteristics of the machine, over intervals of its readings. All of these examples require that a program have access to a fixed set of numeric data. Analogous situations can exist for alphanumeric data. For example, in a billing application a variety of invoice messages may be used, based upon the past due status of the account.

Thus far, our ability to handle such situations is somewhat limited. Three related statements of Wang BASIC considerably enhance our capabilities in these situations. These statements are DATA, READ, and RESTORE. Let's look at DATA and READ first.

DATA and READ are two statements that depend upon one another for effective operation. In a program, the DATA statement supplies values, but the only way the program can make use of these values is by using the READ statement to assign them to variables. Example 12.1 calculates the mean average of a set of values supplied by a DATA statement.

Example 12.1 A Simple Use of DATA and READ Calculating an Average

```
110 REM CALCULATE THE AVERAGE ITEM VALUE (ARITHMETIC MEAN)
120 REM NUMBER OF ITEMS
130     DATA 10
140 REM ITEMS
150     DATA 26, 22, 28, 29.5, 32, 18, 20, 21.5, 22, 23
155 REM PROGRAM BEGINS
160     READ N
170     FOR I = 1 TO N
180         READ D
190         E = E + D
200     NEXT I
210 REM CALCULATE AND OUTPUT AVERAGE
220     PRINT "AVERAGE="; E/N
```

Notice in Example 12.1 that there are two DATA statements at the beginning of the program. DATA statements do nothing when encountered in the normal sequence of execution. Their only function is to supply values to be referenced by a READ statement. The remark at line 155 calls attention to this fact, that the first executable instruction begins at line 160.

The READ N statement at line 160 reads the first data value in the program, and assigns that value to the variable N. The first data value is always the first value in the lowest line-numbered DATA statement. Therefore, READ N assigns the value 10 to N. READ N also automatically moves an internal pointer to the next data value. The DATA statement at line 130 contains no more data values, so the data pointer is set to the first value of the next DATA statement, which is located at line 150.

At line 170 the newly assigned value of N is used as the "TO" value in the FOR...TO statement. Thus, the loop set up by line 170 executes 10 times. Line 180, within the loop, reads the value at the current data pointer location, and assigns that value to the variable D. It then automatically moves the pointer to the next data value. The first time through the loop, D receives the value 26, since the first time line 180 is executed, the pointer is set to the first value in the DATA statement at line 150.

In order to calculate an average, the program must add up all the values to be averaged. This operation is performed at line 190. The variable E is to contain the sum.

NEXT I causes a branch back to the READ D statement, until this loop has been executed 10 times. Each time through the loop, READ D reads the data value at the current pointer location, assigns the value to D, and moves the pointer to the next data value. In this manner READ successively reads all the values in the DATA statement at line 150. E is updated with each newly read value.

When NEXT I terminates the loop, E contains the sum of all the DATA values given in line 150. Line 220 calculates the average by dividing this sum by N, the number of items.

During the last (10th) execution of the loop, the READ D statement reads the last value, 23, and sets the data pointer beyond this last value. Any attempt to execute another READ statement, with the data pointer set beyond the last value, results in an error (ERR 27 Insufficient Data). Therefore, if you append this statement to the program,

230 READ K

it produces an error. This is because the data pointer lies beyond the end of the data values after the complete execution of the original program.

The RUN command, with or without a line number, always resets the data pointer to the first data value. Therefore, Example 12.1 can be reexecuted successfully by simply rekeying RUN(EXEC).

155

It is not necessary that all data values be read. Since the first data value specifies the number of values to be averaged, if line 130 is changed to

      130 DATA 6

the program will execute successfully, averaging only the first six values of line 150. However, if line 130 is changed to

      130 DATA 11

and no additional data values are provided, then on the 11th time through the loop READ D produces an error (ERR 27 Insufficient Data).

The READ statement proceeds from one value to the next without regard to whether the next value is in the next DATA statement or in the same DATA statement. The fact that two DATA statements were used in this program is not significant.   The data could have been supplied in a single DATA statement such as,

      130 DATA 10, 26, 22, 28, 29.5, 32, 18, 20, 21.5, 22, 23

For that matter it could have been supplied in eleven DATA statements, each with a single value.

The first data value is always the first value in the lowest line numbered DATA statement. The system looks for DATA statements in line number sequence. Thus, if line 130 were numbered 151 instead, the program would not execute properly. DATA statements may appear anywhere in a program, before or after the READ statements that read their data. Lines 130 and 150 could have been numbered, for example, 9010 and 9020 respectively.

Values in DATA statements must be separated by commas. A comma must not appear at the end of a DATA statement. Any number of values, up to the maximum line length of 192 keystrokes, may be included in a single DATA statement.

DATA statements may contain numeric or alphanumeric data or both. Alphanumeric values must appear as character strings in quotes. When the data pointer is pointing to an alphanumeric value, the next READ statement must contain an alphanumeric variable to receive the value of the literal string. An error results from any attempt to read numeric data into an alphanumeric variable, or alphanumeric data into a numeric variable.

The general form of the DATA statement is:

      DATA n [,n...]

Where   n = a numeric constant, or
           a literal string in quotation marks.

Example 12.2 shows a program that reads alphanumeric data into an alphanumeric variable.

Example 12.2   DATA and READ with Alphanumeric Values

```
110 REM READ AND PRINT AN ADDRESS
130     DIM A$25
140     FOR I = 1 TO 3
150         READ A$
160         PRINT A$
170     NEXT I
180     DATA "WANG LABORATORIES, INC.", "836 NORTH STREET", "TEW
KSBURY, MA 01876"
```

Multiple Variables in a READ Statement

A single READ statement can read any number of successive data values into specified variables, that is, a statement sequence such as:

```
40 READ A$
50 READ C
60 READ D
```

can be replaced by

```
40 READ A$,C,D
```

Multiple variables in a READ statement must be separated by commas. The READ statement at line 130 of Example 12.3 assigns an address line to each variable in A$().

Example 12.3 Multiple Variables in a READ Statement

```
110 REM USING READ WITH MULTIPLE VARIABLES
120     DIM A$(3)25
130     READ A$(1), A$(2), A$(3)
140     FOR I = 1 TO 3
150         PRINT A$(I)
160     NEXT I
200     DATA "WANG LABORATORIES, INC.","836 NORTH STREET",
"TEWKSBURY, MA 01876"
```

The general form of the READ statement is:

```
READ variable [,variable...]
```

12-2   THE RESTORE STATEMENT

The RESTORE statement offers a means of moving the DATA pointer to any DATA value.   The statement, RESTORE, with no additional parameters, returns the DATA pointer to the first DATA value.

If we add a line to Example 12.3 as follows,

```
170 GOTO 130
```

the second execution of 130 yields an error, since the data pointer is beyond the last data value. However, if we add

```
170 RESTORE
180 GOTO 130
```

the program will execute indefinitely. The RESTORE statement at line 170 moves the pointer back to its original position, pointing to the first DATA value.

A more practical, though simplified, program which makes use of RESTORE is shown in Example 12.4.

Example 12.4 A Simplified Withholding Tax Calculation

```
110 REM A SIMPLE WITHHOLDING TAX CALCULATION
120     INPUT "ENTER GROSS WAGES FOR WEEK", W
130 REM FIND APPLICABLE TAX BRACKET
140     FOR I = 1 TO 5
150         READ C, P
160         IF W < C THEN 210
170     NEXT I
180 REM W IS IN HIGHEST TAX BRACKET
190     P = .065
200     GOTO 240
210 REM FORCE "NEXT" EXIT
220     I = 5
230     NEXT I
240 REM P NOW CONTAINS CORRECT TAX RATE %
250     T = W * P
260     PRINT "WITHHOLDING TAX = "; T
270 REM RESET DATA POINTER
280     RESTORE
290 REM LET OPERATOR ENTER NEXT WAGE
300     PRINT
310     GOTO 120
320 REM TAX DATA (CUTOFF, RATE)...
330 DATA 80.00, .005, 125.00, .01, 175.00, .025, 255.00, .035,
325.00, .045
```

At line 120 the operator enters the weekly gross wage. Lines 140 to 170 make up a loop that carries out a search for the proper tax bracket. Each time through the loop, line 150 reads an upper limit of a tax bracket and the associated percentage for the bracket. Thus, the first time through the loop, C is assigned the value 80.00 and P the value .005. If the wages for the week are less than 80, then the tax rate is .005. Line 160 tests to see if the wages are less than the tax bracket's upper limit. When 160 effects a branch, P contains the appropriate percentage. If the loop terminates normally, because the entered wage is $325.00 or more, then 190 sets P to the highest tax rate, .065.

If line 160 effects a branch, then the FOR...TO information must first be cleared by forcing a NEXT statement exit. Lines 220 and 230 do this. Then, the tax can be calculated by multiplying the percentage in P times the wage, W.

After line 260 the work of the program has been done, but the DATA pointer is no longer at the beginning of the DATA values. The RESTORE statement at line 280 is used to move it back, so that another wage entry can be successfully processed.

Using an Expression in the RESTORE Statement

An expression can be used with the RESTORE statement to move the DATA pointer to a specific DATA value. If the value of the expression is n, RESTORE n moves the DATA pointer to the nth DATA value. Only the integer portion of the expression's value is used.

Example 12.5   Illustration of RESTORE with Expression

```
110 REM ILLUSTRATION OF "RESTORE" WITH EXPRESSION
120     INPUT "ENTER TWO SINGLE DIGIT POSITIVE NUMBERS", A,B
125 REM OUTPUT FIRST VALUE
130     RESTORE A
140     READ W$
150     PRINT W$; " PLUS ";
155 REM OUTPUT SECOND VALUE
160     RESTORE B
170     READ W$
180     PRINT W$; " EQUALS ";
185 REM OUTPUT SUM
190     RESTORE A+B
200     READ W$
210     PRINT W$
500 DATA   "ONE", "TWO", "THREE", "FOUR", "FIVE", "SIX", "SEVEN",
 "EIGHT", "NINE", "TEN", "ELEVEN", "TWELVE", "THIRTEEN", "FOURTE
EN", "FIFTEEN", "SIXTEEN"
510 DATA "SEVENTEEN", "EIGHTEEN", "NINETEEN"
```

For two entered values, such as 5 and 6, Example 12.5 outputs a line such as:

FIVE PLUS SIX EQUALS ELEVEN.

At line 120 the operator enters two single digit positive values into the variables A and B. At line 130 the variable A is used as the expression in a RESTORE statement. The RESTORE A statement moves the data pointer to the Ath data value. Thus, if A is 5, the data pointer is moved to the fifth data value, "FIVE"; if it were 9, RESTORE A would move the pointer to the 9th data value, "NINE". Line 140 then reads into W$ the value, at the current, newly set, data pointer location. W$ is printed followed by the word " PLUS". Lines 160 to 180 repeat this process for the second word. In line 190 the same process is repeated, but now the RESTORE expression is A+B. RESTORE A+B evaluates the expression A+B and, if we assume the result to be k, moves the data pointer to the kth data value in the program.

The general form of the RESTORE statement is

RESTORE [expression]

Where $1 \leq$ value of expression < 256.

Example 12.6 shows a simple routine that prints one of several invoice messages, depending upon the value of a variable D, where D is the age in days of the oldest outstanding invoice.

Example 12.6 A Program That Prints Invoice Messages

```
110 REM PROGRAM SEGMENT WHICH PRINTS INVOICE MESSAGES
120 REM ** ASSUME THAT D CONTAINS THE AGE IN DAYS OF THE
130 REM ** OLDEST OUTSTANDING INVOICE AND THAT D < 120
140     DIM M$30
520 REM OUTPUT MESSAGE
530     RESTORE D/30 +1
540     READ M$
550     PRINT , M$
560 REM NEXT OPERATION
6000 DATA "THANK YOU FOR PAYMENT", "YOUR ATTENTION IS APPRECIATED",
"PAYMENT OVERDUE", "IMMEDIATE PAYMENT REQUESTED"
```

Example 12.6 assumes that D has been determined previously, and is less than 120. If you want to try out this program segment, a simple INPUT statement such as

150 INPUT "AGE IN DAYS", D

will make it operational.

The RESTORE statement at line 530 is the key to the operation. If there is no outstanding invoice over 30 days old, then the account is current. With a value of D less than 30, the RESTORE expression, D/30+1, has a value greater than or equal to 1 but less than 2. For example, if D is 15, D/30+1 is 1.5. Any value in this range is treated by RESTORE as if it were 1, since RESTORE uses only the integer portion of the value. Thus if D is less than 30, the message "THANK YOU FOR PAYMENT" is read and printed. A value of D in the range $30 \leq D<60$ yields a truncated expression value of 2, which causes the second message to be output. Similarly 60-90 outputs the third message, and 90-120 the fourth.

Review of Chapter 12

1.   The DATA statement contains values that are accessed by the READ
      statement.   When encountered in the sequence of execution, DATA
      statements do nothing.

2.   When the RUN command is executed, the DATA pointer is set to the first
      DATA value in the first DATA statement in the program.

3.   The READ statement assigns to a variable the DATA value at the location
      of the DATA pointer, and then moves the pointer to the next DATA value.
      If more than one variable is specified in the READ statement, it repeats
      this operation until all variables have been assigned DATA values.

4.   The DATA pointer must point to an alphanumeric value at the time that an
      alphanumeric variable is encountered in a READ statement, and to a
      numeric value at the time that a numeric variable is encountered in a
      READ statement. Otherwise, an error results.

5.   RESTORE, without an expression, moves the DATA pointer to the first DATA
      value.   RESTORE with an expression, whose integer value is k, moves the
      DATA pointer to the kth DATA value in the program.

6.   The general forms of DATA, READ and RESTORE are:

           DATA n [,n...]

           READ variable [,variable...]

           RESTORE [expression]

           where: n = a numeric constant, or
                        a character string in quotation marks.

      and,

           $1 \leq$ value of "expression" < 256 .

161

CHAPTER 13
INTRODUCTION TO SUBROUTINES

13-1 GOSUB AND RETURN

Frequently, identical or nearly identical operations must be performed repeatedly within a program. A naive solution in such a situation is to simply append to the program, again and again, the instructions needed to do the job. Such an approach, however, is nearly always impractical, because it rapidly exhausts the limited resources of computer memory and programmer time. For this reason, a variety of programming techniques have evolved that allow a program to use the same instructions over and over, to perform similar operations. A loop is an example of such a technique. The simplest kind of loop allows the same instructions to be used over and over again, when only the values of variables change each time through. The use of list variables can be another such programming technique. This allows the same instructions to be used repeatedly when the variable itself is what must be changed on each repetition. In this chapter we are going to look at another technique for reusing the same instructions to perform similar operations: the subroutine.

When the same operation must be performed at several different locations within a program, it may be a good candidate for being made into a subroutine. For example, a program that prints a report should print the column headings at the top of the first page, and every page thereafter. This requires that the program maintain a count of the number of lines printed on a page, and compare the number printed with the full-page maximum, each time a line is to be printed. If a line would overflow the page, the paper should be advanced above the next perforation and the headings reprinted.

If a program has several print statements that output a line, this entire operation of testing the line count, and possibly printing the headings, must occur before each such PRINT statement. While it would be possible to repeat the required instructions as many times as necessary within the program, this is wasteful. A better approach is to take the instructions required for testing the line count and printing the headings, and locate them outside of the main program, perhaps, at a higher numbered line than the end of the main program. Then, as the program is being written, each time the program must perform the line-count/headings operation, a branch to these instructions is written. While it would be possible to branch to this "subroutine" with a simple GOTO, there would then be the problem of how to get back to the right place in the main program once the subroutine is complete. That is, if the program branches to the line-count/headings operation from several different locations in the main program, how can the line-count/headings operation branch back to the right place in the main program after it has done its work?

The BASIC language offers a simple solution to this problem in the form of two statements specifically designed for programming subroutines. In the main program, when the line-count/headings subroutine is to be executed, a branch is made to it with the statement

GOSUB line number

where "line number" is the starting line number of the subroutine. GOSUB works just like GOTO, except for one thing. GOSUB saves, in a special part of memory, the location of the statement immediately following itself.

At the end of the line-count/headings subroutine, there appears the statement

RETURN

RETURN looks to the part of memory in which GOSUB saves its information, and branches to the statement whose location was saved by the last GOSUB. In the process of doing this, it clears out the location information that was saved by the last GOSUB. If a program branches, with GOSUB statements, to the line-count/headings subroutine from three different locations in the main program, the single RETURN statement will always branch back to the statement following the most recently executed GOSUB.

GOSUB and RETURN may not be used in the Immediate Mode. In addition, GOSUB may not be the last statement in a program.

The diagram shown in Figure 13.1 shows the example situation, programmed without the use of subroutines. Notice that the block "TEST HEADINGS?" is repeated three times, once before each PRINT line.

WITHOUT SUBROUTINES

```
        ┌─────────────────┐
        │ LINE 10         │
        │                 │
        │   PROCESSING    │
        │                 │
        │ LINE 420        │
        └─────────────────┘
                │
        ┌─────────────────┐
        │ LINE 426        │
        │ TEST. HEADINGS? │
        │ LINE 500        │
        └─────────────────┘
                │
        ┌─────────────────┐
        │ 510             │
        │    PRINT        │
        │  REPORT LINE    │
        │                 │
        │  (PROCESSING)   │
        │ 930             │
        └─────────────────┘
                │
        ┌─────────────────┐
        │ 940             │
        │ TEST. HEADINGS? │
        │ 1020            │
        └─────────────────┘
                │
        ┌─────────────────┐
        │ 1030            │
        │    PRINT        │
        │  REPORT LINE    │
        │                 │
        │  (PROCESSING)   │
        │ 1450            │
        └─────────────────┘
                │
        ┌─────────────────┐
        │ 1460            │
        │ TEST. HEADINGS? │
        │ 1540            │
        └─────────────────┘
                │
        ┌─────────────────┐
        │    PRINT        │
        │  REPORT LINE    │
        └─────────────────┘
                │
              ╱   ╲
            ╱  TEST ╲
              ╲   ╱
                │
        ┌─────────────────┐
        │      END        │
        └─────────────────┘
```
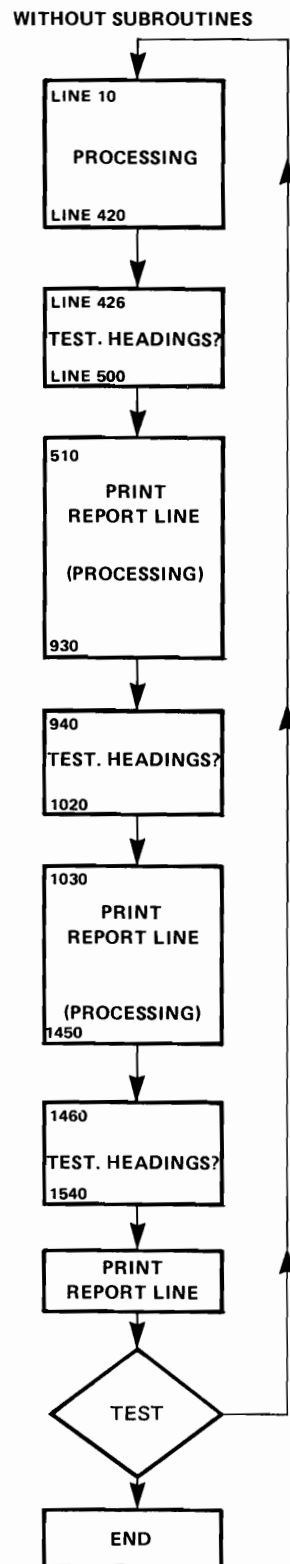
Figure 13.1   Program Flow without Using a Subroutine

164

Figure 13.2 represents the program flow when the "TEST HEADINGS?" routine has been made into a subroutine. The matching pairs of lines show the branches effected by each execution of GOSUB and followed by RETURN. For example, GOSUB, when executed at line 420 causes the next RETURN to branch to line 430, since 430 contains the statement after 420. GOSUB from line 860 causes the next RETURN to branch to 870, again because 870 is the statement following 860. Similarly the GOSUB at line 1280 causes RETURN to branch to 1290.
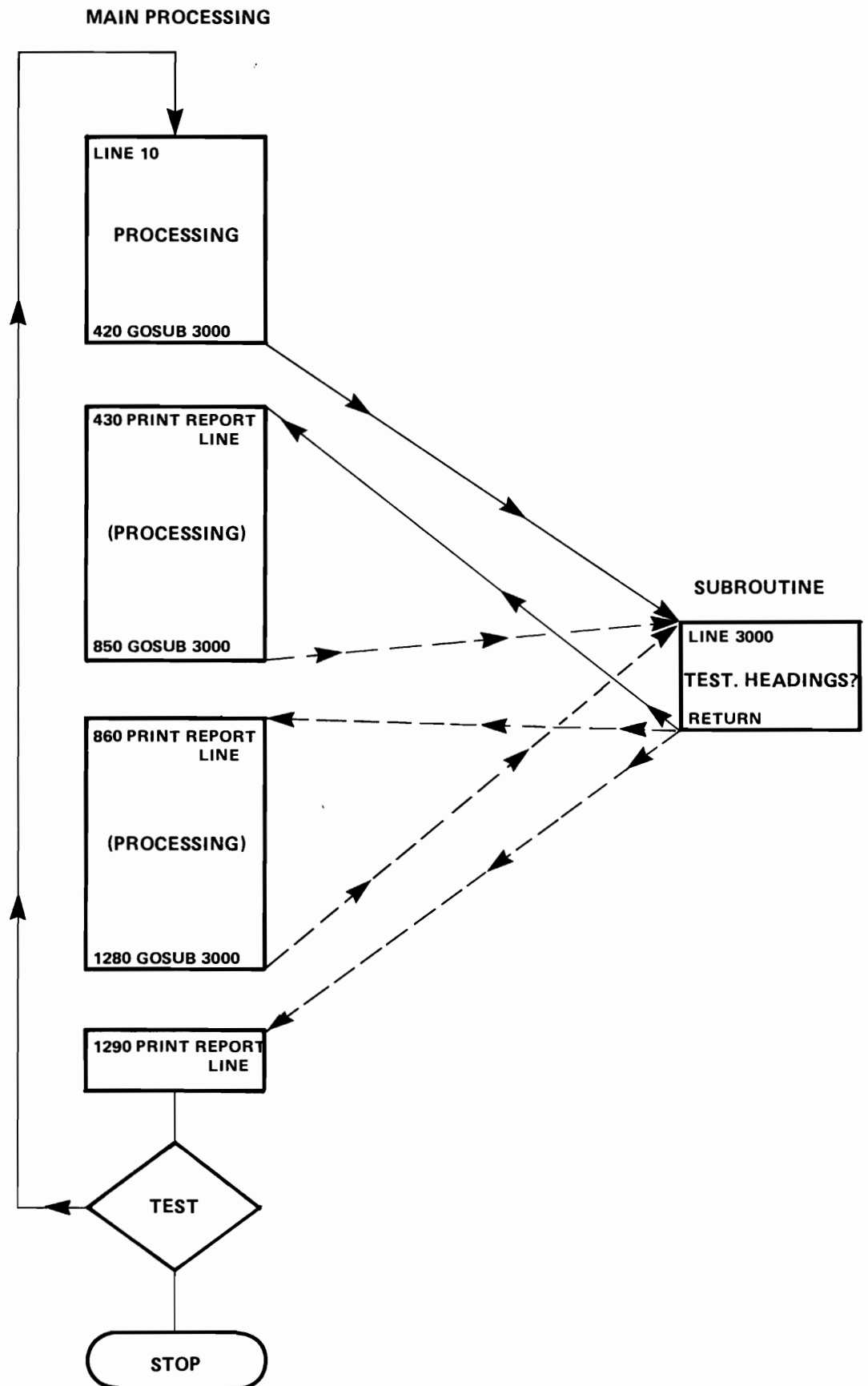
MAIN PROCESSING

Figure 13.2   Program Flow Using a Subroutine

A skeleton of the program we have been discussing is shown in Example 13.1.

Example 13.1 A Skeleton of a Program Using Subroutines

```
10 REM PROCESSING BEGINS
   .
   .
   .
420      GOSUB 3000
430      PRINT A$, C, D$(F), Q(8)
440 REM PROCESSING CONTINUES
   .
   .
   .
850      GOSUB 3000
860      PRINT T$, R, U$(F), R(9)
870 REM PROCESSING CONTINUES
   .
   .
   .
1280      GOSUB 3000
1290      PRINT E$, G, Y$(F), S(9)
1300      IF Q <> T THEN 10
1310      STOP "END OF PROGRAM"


3000 REM SUBROUTINE -- LINE COUNT / HEADINGS
3010      L = L + 1
   .
   .
   .
3090 RETURN
```

Since the RETURN statement branches to the last location saved by a GOSUB, its operation depends upon this information. An error results if the system encounters a RETURN statement when there is no return location information previously saved by a GOSUB. This problem is analogous to that of NEXT in relation to FOR...TO. In general, when using subroutines, you should be certain that every possible route to a RETURN statement will provide the statement with the GOSUB information it needs.

The RETURN statement always branches to the statement following the GOSUB. In particular, if the GOSUB is in a multistatement line, RETURN branches to the next statement in that line, not to the next line.

The RETURN statement clears from memory the return location, saved by the last GOSUB, as it branches to that location. This clearing operation is important because it prevents useless information from accumulating in memory. If a program repeatedly executes GOSUB statements without executing corresponding RETURN statements, for example, by using an IF...THEN to branch back to the main program, then eventually a table overflow error will occur, as a result of the excess accumulated return location information. There are two ways to avoid this difficulty. You can check that each time a GOSUB is executed, a RETURN is executed. If a subroutine has no statements which could

branch out of it, other than RETURN, then there is no danger of avoiding the RETURN. If, under some circumstances, the RETURN must be avoided by means of a branch out of the subroutine, the RETURN CLEAR statement can be used to clear out the information saved by GOSUB.

13-2 RETURN CLEAR

RETURN CLEAR does not branch; it clears from memory the return location information saved by the last executed GOSUB. Example 13.2 illustrates the use of RETURN CLEAR.

Example 13.2 A Simple Use of RETURN CLEAR

```
110 REM EXAMPLE USING RETURN CLEAR
120 REM PROCESSING
 .
 .
 .
170 GOSUB 5000
180 REM NORMAL RETURN POINT FROM SUBROUTINE IS HERE
 .
 .
 .
270 REM RETURN FROM SUBROUTINE TO HERE ONLY IF X=2
 .
 .
 .
4000 STOP "END OF PROGRAM"

5000 REM SUBROUTINE
 .
 .
 .
5060 IF X=2 THEN 5080
5070 RETURN
5080 RETURN CLEAR
5090 GOTO 270
```

In this program, in addition to the normal return point immediately following the GOSUB statement (line 180), if X=2, the subroutine must return to line 270 of the main program. Line 5060, in the subroutine, tests for X equal to 2. If X equals 2, it branches around the normal RETURN statement, at line 5070, to line 5080. 5080 RETURN CLEAR clears out the return location information saved by the last GOSUB, but makes no branch. 5090 then makes an ordinary branch to line 270.

It is possible to branch to a subroutine from within a subroutine. Such a procedure is known as nesting subroutines. Approximately 45 levels of subroutines nested within subroutines are permitted. From within a subroutine, a simple GOSUB statement, giving the line number of another subroutine, is all that is needed to use the other subroutine. A RETURN statement always branches to the location saved by the last executed GOSUB. So, if a subroutine GOSUB's to another subroutine, when that other subroutine is complete, RETURN branches back to the original subroutine. When the original subroutine is complete, RETURN branches back to the main program. A

RETURN CLEAR statement wipes out only the location saved by the last executed GOSUB.   Therefore, if a RETURN CLEAR is executed in a nested subroutine, the RETURN at the bottom of the nested subroutine will branch back to the statement after the second previous GOSUB.

13-3   ON...GOSUB

In Chapter 10 the ON statement with the GOTO parameter was introduced. In addition to ON...GOTO there is another form of this statement, ON...GOSUB. ON...GOSUB works the same as ON...GOTO except that the location of the statement following the ON...GOSUB statement is saved in a special part of memory for use by the RETURN statement.

Review of Chapter 13

1.   The GOSUB statement causes a branch to a specified line number, and saves the location of the statement following itself in a special part of memory for later use by the RETURN statement.

2.   The RETURN statement is generally placed at the end of a subroutine. It causes a branch to the location saved by the last executed GOSUB, and clears the return location information.

3.   The RETURN CLEAR statement simply clears the return location saved by the last executed GOSUB. It does not effect a branch.

4.   The ON statement with GOSUB works the same as ON with GOTO except that the location of the statement following the ON...GOSUB statement is saved in a special part of memory for use by the RETURN statement.

5.   Approximately 45 levels of subroutine nesting are permitted.

CHAPTER 14
THE DEFFN' STATEMENT

14-1 <u>USING DEFFN' TO MARK SUBROUTINES</u>

In the last chapter we looked at subroutines that were branched to with the statement

        GOSUB line number

where "line number" is the number of the first line in the subroutine.   Thus, if you are writing a program and wish to write a subroutine access with GOSUB, you must know the line number of the beginning of the subroutine. This can be somewhat inconvenient, especially if, in the process of writing the program, you wish to renumber it.   Renumbering will   change   the   subroutine's line number, forcing you to look through the program listing to find the subroutine, before writing the next GOSUB statement.

Wang BASIC offers a multipurpose instruction DEFFN' that, in its simplest use, allows you to mark the beginning of a subroutine, and give it an identification number. For example, the statement

        1100 DEFFN' 198

marks the beginning of a subroutine, and identifies it as DEFFN' subroutine 198. To branch to this subroutine the statement

        GOSUB' 198

is used. This statement is like GOSUB except that it causes the system to search through the program for the statement DEFFN'198, and when it finds it, causes a subroutine branch to the DEFFN'198 location.   By using DEFFN' and GOSUB', you don't have to keep track of where a subroutine is in order to use it.

The number 198, chosen as the identification number for this subroutine, has no particular significance, other than to distinguish it from other DEFFN' subroutines. It bears no relationship to the line number of the DEFFN' statement, or to the processing accomplished in the subroutine. The RETURN statement functions exactly the same for subroutines accessed with GOSUB' as for those accessed with GOSUB. It branches to the statement following the last GOSUB' statement. There is no special form of the RETURN statement for use with marked subroutines.

The general form of the DEFFN' statement for simple marking of subroutines is

        DEFFN' integer

where: $0 \leq$ integer $\leq$ 255.

DEFFN' statements in which the specified integer is in the range 0-31 are used to define Special Function keys. This use is discussed in Section 14-3.

    The general form of the GOSUB' statement for simple marked subroutines is

        GOSUB' integer

where $0 \leq$ integer $\leq$ 255, and the integer corresponds to an integer in a DEFFN' statement. GOSUB' may not be the last statement in a program.

    The DEFFN' statement and its subroutine may appear anywhere in a program, either at a lower or higher numbered line than the GOSUB' statements that reference it. However, the DEFFN' statement itself must always be the first statement on a line.   If encountered in the normal sequence of execution, the DEFFN' statement does nothing; its only purpose is to be accessed via a GOSUB' statement or a Special Function key.

14-2 ARGUMENT PASSING

    Often it is necessary to assign values to key variables before branching to a subroutine. For example, if a program requires that several different numeric values be entered from the keyboard, it may be worthwhile to write a simple subroutine to handle all numeric entries.   The subroutine should display a specified prompt, and test the value entered to determine whether it is within an acceptable range.   Only if the value is acceptable should it transfer control back to the main program. Such a subroutine is shown in Example 14.1.

    Example 14.1 A Numeric Entry Subroutine (without DEFFN')

```
        5000 REM A NUMERIC INPUT SUBROUTINE
        5010 REM ** P$ = THE PROMPT (64 CHARACTERS MAX) **
        5020 REM ** L  = THE MINIMUM ACCEPTABLE VALUE
        5030 REM ** U  = THE MAXIMUM ACCEPTABLE VALUE
        5040 REM ** X  = RETURNED VARIABLE
        5050     DIM P$64
        5060     PRINT
        5070     PRINT P$
        5080     INPUT X
        5090     IF X > U THEN 5120
        5100     IF X >= L THEN 5130
        5120     PRINT "INVALID.  REENTER"
        5125     GOTO 5060
        5130     RETURN
```

In order to use this subroutine the main line program must first assign the operator prompt to P$, and the minimum and maximum acceptable values to L and U respectively. Then, GOSUB 5000 can be executed to pass control to the subroutine. (One other thing should be noted, though. The DIM statement in the subroutine must be moved to a lower line number than any in which a reference to P$ occurs.)

Example 14.2 shows a segment of a main program that makes use of this input subroutine.

Example 14.2 Passing Control to the Numeric Entry Subroutine

```
110 DIM P$64
 .
 .
 .
410 P$ = "ENTER HOURLY RATE"
420 U  = 25
430 L  = 2
440 GOSUB 5000
450 PRINT "HOURLY RATE = "; X
 .
 .
 .
```

As can be seen from Example 14.2, three assignment statements must be executed before passing control to the subroutine (lines 410-430). Since one of the primary purposes of a subroutine is to reduce the total number of statements needed for a program, the need to use three assignment statements reduces the advantage of the subroutine. DEFFN' and GOSUB' offer a convenient solution to this problem. GOSUB' can assign values to variables specified in the DEFFN' statement, as it passes control to the subroutine.

The DEFFN' statement must specify all the variables that are to be assigned values when control is passed to the subroutine. For the subroutine of Example 14.1, the DEFFN' statement might look like this:

DEFFN' 100 (P$,U,L)

In this statement, 100 is the number chosen to identify the subroutine. Within parentheses are the three variables that are to receive values when GOSUB' branches to this subroutine.

In Example 14.2 the GOSUB' statement that would be used to make the branch is

GOSUB'100 ("ENTER HOURLY RATE", 25,2)

This GOSUB' statement does the following:

1. Finds DEFFN' 100.

2. Successively assigns the items in parentheses

       "ENTER HOURLY RATE"
       25
       2

   to the variables specified in the DEFFN' statement.

3. Branches to the subroutine.

Example 14.3 shows the main program segment and subroutine, rewritten using DEFFN' and GOSUB'.

Example 14.3 Program and Subroutine with DEFFN' and GOSUB'

```
110    DIM P$64
 •
 •
 •
410    GOSUB' 100 ("ENTER HOURLY RATE", 25, 2)
450    PRINT "HOURLY RATE = "; X
 •
 •
 •
2050 REM END OF MAIN PROCESSING
2060 END
 •
 •
5000 REM A MARKED SUBROUTINE FOR NUMERIC INPUT (DEFFN' 100)
5010 REM ** P$ = THE PROMPT (64 CHARACTERS MAX) **
5020 REM ** L  = THE MINIMUM ACCEPTABLE VALUE
5030 REM ** U  = THE MAXIMUM ACCEPTABLE VALUE
5040 REM ** X  = RETURNED VARIABLE
5060    DEFFN' 100 (P$, U, L)
5070    PRINT
5080    PRINT P$
5090    INPUT X
5100    IF X > U THEN 5120
5110    IF X >= L THEN 5140
5120    PRINT "INVALID. REENTER"
5130    GOTO 5070
5140    RETURN
```

Using DEFFN' and GOSUB' to assign values and branch to the subroutine has eliminated lines 420 through 440 of Example 14.2.

With this subroutine in the program, a single GOSUB' statement can be used to initiate the processing associated with receiving a keyboard numeric entry. For example, in this program there might be other calls to this subroutine such as,

```
920 GOSUB'100 ("ENTER REGULAR HOURS", 40, 0)
1170 GOSUB'100 ("ENTER OVERTIME HOURS", 100, 0)
```

173

Any number of variables may be specified in a DEFFN' statement, but, for each variable specified, the GOSUB' statement must supply an acceptable value. This means not only that there must be an equal number of values and variables, but also that alphanumeric variables must receive alphanumeric values, and numeric variables must receive numeric values. Variables in the DEFFN' statement and values in the GOSUB' statement must be separated by commas.

In the GOSUB' statement the value may be specified by any form that would be legal on the right of the "=" in a "LET" assignment statement. This means that any expression can be used to specify a value to be received by a numeric variable. Alphanumeric values may be specified with a literal string or an alphanumeric variable. For example, if the DEFFN' statement for a subroutine is

        7000 DEFFN' 220 (K, N2(K), T$, F)

a GOSUB' such as this is acceptable :

        850 GOSUB' 220 (SGN(G)+2,3,A$,SQR(G+#PI))

The values supplied in the GOSUB' statement are often called "subroutine arguments."

Example 14.4 shows a subroutine that rounds a value to a specifiable number of decimal places.

Example 14.4   A Subroutine to Round X to N Decimal Places

```
6000 REM A SUBROUTINE TO ROUND X TO N DECIMAL PLACES
6010     DEFFN' 255 (X,N)
6020     X= SGN(X)*INT((ABS(X)*10↑N+.5)/10↑N)
6030     RETURN
```

14-3   DEFINING SPECIAL FUNCTION KEYS WITH DEFFN'

Across the top of your Wang 2200 System keyboard are 16 Special Function keys. These keys may be defined by you, in a program, to perform a variety of different types of tasks. Since each of the 16 keys may be depressed alone or in conjunction with the SHIFT key, an effective total of 32 keyboard Special Function keys is available. The Special Function keys are numbered 0 to 15 and 16 to 31, the latter range obtained by depressing SHIFT together with the appropriate Special Function key. The DEFFN' statement can be used to define the Special Function keys.

In the last two sections we have discussed the use of DEFFN' to mark subroutines, and assign subroutine arguments. We said that the DEFFN' statement can identify the subroutine with any integer 0 to 255. For example,

        5000 DEFFN' 135
        .
        .
        .

assigns the identification number 135 to the subroutine that begins at line 5000. With this statement in the program, the statement

       400 GOSUB' 135

can be used to transfer control to this subroutine. However, if the DEFFN' statement uses a number 0 to 31 to identify a subroutine, the keyboard Special Function keys can also be used to initiate execution of the subroutine. They can be used in this manner whenever the system colon (:) is displayed, or the system is awaiting a keyboard entry on an INPUT instruction.

    If the system colon (:), or the ? of the INPUT statement, is displayed, and a Special Function key is depressed, the system searches through the program text for a DEFFN' statement that has a number corresponding to the number of the depressed key. For example, if Special Function key 5 is depressed, the system looks for a DEFFN' 5 statement; if Special Function key 0 is depressed, it looks for DEFFN'0. When it finds the appropriate DEFFN' statement, it begins executing the subroutine that follows the DEFFN', just as if it had been sent there with a GOSUB'. If a Special Function key is depressed when the system colon is displayed, the RETURN statement at the end of the accessed subroutine simply causes the colon to be redisplayed. If the Special Function key is depressed at an INPUT instruction, the RETURN statement causes the INPUT instruction to be repeated, that is, the prompt is redisplayed with the question mark, and the system awaits an entry.

    The ability to access and execute subroutines upon keyboard selection opens a wide range of programming possibilities. For example, a program may be designed to execute with all angles given in radians. If an observation happens to be recorded in degrees, it can be converted while the main program is stopped at an INPUT instruction, by accessing a conversion subroutine with a Special Function key. Example 14.5 shows a segment of such a program at lines 560 and 570, and a conversion subroutine accessible via Special Function Key 11.

       Example 14.5 A Program with a Special Function Subroutine That
                     Converts Degrees to Radians

```
       .
       .
       .
560       INPUT "ENTER ANGLE T IN RADIANS", T
570       Z = TAN(T+#PI/6)

4000 REM CONVERT DEGREES TO RADIANS SUBROUTINE
4010      DEFFN' 11
4020      PRINT "CONVERT DEGREES TO RADIANS"
4030      INPUT "ENTER DEGREES, MINUTES, SECONDS", D,M,S
4040      A = D +M/60 +S/3600
4050      A = A-INT(A/360)*360
4060      PRINT "ANGLE = ",A*.0174532925; "RADIANS"
4070      RETURN
```

    Whenever the main program requests an entry in radians, as it does at line 560, the operator can enter the value in radians; however, if the requested value happens to be recorded in degrees, the operator can depress Special Function key 11 to access the conversion subroutine. The conversion

175

subroutine requests the degree value, converts it to a radian value, and prints the radian value. The RETURN statement then branches back to the INPUT statement from which the subroutine was accessed, in this case line 560. The operator can then enter the converted value that was printed by the subroutine.

In a similar manner subroutines can be designed which print subtotals or category totals, allow correction of erroneous entries, etcetera. Having access to such subroutines from an INPUT instruction can be very convenient.

Since execution of Special Function subroutines can be initiated wherever the system colon (:) is displayed, up to 32 separate programs to perform related or often-needed calculations can be loaded into memory at once, and accessed by a Special Function key. In such a case there might not be any "main program", really. Each program would be set up as a DEFFN' subroutine designed to perform a specific calculation. When the subroutine's RETURN statement is executed, the system colon is redisplayed. Two calculations, arranged in this fashion, are shown in Example 14.6.

Example 14.6 Special Function Key Access to Independent Calculations

```
110 REM TWO DEFFN' SPECIAL FUNCTION KEY SUBROUTINES
120     STOP "ACCESS SUBROUTINES WITH SPECIAL FUNCTION KEYS."
130 REM INITIAL INVESTMENT
140     DEFFN' 0
150     PRINT
160     PRINT "CALCULATE INVESTMENT AMOUNT NEEDED"
170     PRINT "  TO ENABLE ONE TO WITHDRAW A GIVEN AMOUNT"
180     PRINT "   M TIMES PER YEAR FOR N YEARS."
190     PRINT
200     INPUT "AMOUNT OF WITHDRAWAL", R
210     INPUT "ANNUAL INTEREST RATE (PERCENTAGE)", I
220     INPUT "NO. OF WITHDRAWALS PER YEAR", M
230     INPUT "NO. OF YEARS", N
240     I = I/M/100
250     J = (1+I)↑(N*M)
260     PRINT "INITIAL INVESTMENT = $"; INT((J-1)/(I*J)*R*100+.5)/100
270     RETURN
280 REM WITHDRAWAL FROM INVESTMENT
290     DEFFN'1
300     PRINT "CALCULATE THE AMOUNT THAT CAN BE WITHDRAWN"
310     PRINT "  FROM A GIVEN INITIAL INVESTMENT"
320     PRINT "  M TIMES PER YEAR FOR N YEARS "
330     PRINT "  AT INTEREST I, LEAVING NOTHING AT THE END."
340     PRINT
350     INPUT "INITIAL INVESTMENT", P
360     INPUT "ANNUAL INTEREST RATE (PERCENTAGE)",I
370     INPUT "NUMBER OF WITHDRAWALS PER YEAR", M
380     INPUT "NUMBER OF YEARS", N
390     I = I/M/100
400     R = P*(I/((1+I)↑(N*M)-1)+I)
410     PRINT "AMOUNT OF WITHDRAWAL= $"; INT(100*R+.5)/100
420     RETURN
```

Notice in Example 14.6 that each calculation begins with a DEFFN' statement that defines a Special Function key. Each calculation also ends with a RETURN statement, which simply redisplays the system colon. To facilitate use of Special Function keys, a removable labeling strip can be inserted on the keyboard below the Special Function keys.

The RUN command sets aside space for all variables used in a program. It also resets the DATA pointer, and checks for certain types of program errors. These operations are not performed when execution is begun by depressing a Special Function key. However, variable space must be set aside before any program can be executed. Therefore, whenever Special Function keys are used to initiate execution, the RUN command must be executed at least once after loading the program, so that variable space will be allocated. This is the reason that the program shown in Example 14.6 has a STOP statement at line 120. When the program is first loaded, whether from tape, disk, or keyboard, RUN is used to set up variable space. Immediately, STOP is executed, which restores the colon. Thereafter, Special Function keys can be used to initiate execution of the desired calculations.

When a Special Function key is depressed, a location is saved in memory for use by the RETURN statement, just as if the subroutine were accessed via GOSUB'. This is true even if the Special Function key is depressed while the colon or question mark is displayed. For this reason, some means for clearing this information must be provided. If DEFFN' Special Function key routines all lead to RETURN statements, then the RETURN statement will clear the return location information as it uses it. However, if a DEFFN' Special Function key access does not lead to a RETURN statement, the RETURN CLEAR statement must be used to clear the return location information. Otherwise, repeated Special Function key accesses will pile up return information in memory, eventually producing a table overflow error.

Used with the RETURN CLEAR statement, DEFFN' can provide Special Function key access to a variety of entry points to a program. The program segment shown in Example 14.7 illustrates this use.

Example 14.7 DEFFN' and RETURN CLEAR to Define Program Entry Points

```
110 REM USING SPECIAL FUNCTION KEYS AND "RETURN CLEAR"
120 REM TO DEFINE PROGRAM ENTRY POINTS
130     STOP "CHOOSE OPERATIONS VIA SPECIAL FUNCTION KEYS"
140 REM ENTRY POINT NUMBER 1
150     DEFFN' 1
160     RETURN CLEAR
  .
  .
  .
490     GOTO 530
500 REM ENTRY POINT NUMBER 2
510     DEFFN' 2
520     RETURN CLEAR
530     REM PROGRAM CONTINUES HERE
  .
  .
  .
990     GOTO 1030
```

177

```
1000 REM ENTRY POINT NUMBER 3
1010     DEFFN' 3
1020     RETURN CLEAR
1030     REM PROGRAM CONTINUES HERE
  .
  .
  .
5000 REM EMERGENCY TERMINATION ROUTINE
5010     DEFFN' 31
  .
  .
  .

5190 END
```

The program in Figure 14.7 is designed to be started from lines 150, 510, or 1010. The operator chooses where to begin the program by depressing Special Function key 1, 2, or 3. If Special Function key 1 is depressed, the entire program is executed. If 2 is depressed, lines numbered below 510 are not executed while those numbered above 510 are. Notice that the GOTO statements at lines 490 and 990 branch around the RETURN CLEAR statement. This is done to avoid the execution error that would result if RETURN CLEAR were executed without there being any return information to be cleared.

Line 5000 begins an emergency program termination routine, which is accessible via Special Function key 31. Such a routine can be especially useful when tape or disk data files are being worked on, since these often require special file closing procedures. If the operator always has accessible a routine which closes files and then stops program execution, the likelihood that the system will be turned off without successfully closing the files is greatly reduced.

If a DEFFN' statement that defines a Special Function key has a list of variables to be assigned values at the time of access, values may be assigned by entering them, separated by commas, prior to keying the Special Function key. Such a program is shown in Example 14.8.

Example 14.8 Argument Passing with a Special Function Key Subroutine

```
110 REM ILLUSTRATION OF ARGUMENT PASSING
112 REM  WITH A SPECIAL FUNCTION KEY SUBROUTINE
120 REM CONVERT DEGREES TO RADIANS
130     DEFFN' 11 (D,M,S)
140     A = D +M/60 +S/3600
150     A = A-INT(A/360)*360
160     PRINT "ANGLE = ";A*.0174532925; "RADIANS"
170     RETURN
```

Example 14.8 shows a subroutine to convert degree measure to radian measure. Unlike the conversion subroutine in Example 14.5, the DEFFN' statement of this subroutine requires that the three values for degrees, minutes, and seconds be passed to it at the time of access. If a Special

Function key is used to access this subroutine, values for D, M, and S must be keyed in before the Special Function key is depressed. For example, to access this subroutine, you can key

        :25,15,20 (Special Function key 11)

The values 25, 15 and 20 are successively assigned to the variables D, M and S, and execution of the subroutine begins. The assignment of values is the same regardless of whether the system is at an INPUT instruction or at the system level with the colon displayed. Values to be assigned to alphanumeric variables must be enclosed by quotation marks.

14-4   DEFINING A SPECIAL FUNCTION KEY FOR CHARACTER STRING ENTRY

The DEFFN' can be used to associate a character string with a Special Function key. When used in this fashion, depression of the specified Special Function key causes the characters in the DEFFN' statement to be entered, as if they had been entered one-by-one. For example, if this statement appears in a program,

        100 DEFFN'0 "FREIGHT CHG."

depressing Special Function key 0, at an INPUT instruction or when the colon is displayed, causes the characters "FREIGHT CHG" to become part of the current text line. It must be emphasized that use of the DEFFN' statement to define character strings is unrelated to its use in marking subroutines. Character strings can be used with DEFFN' statements only when the DEFFN' identification number specifies a Special Function key. Depressing the Special Function key associated with a DEFFN' character string merely causes the character string to be entered; it does not initiate execution of any other statements.

If a Special Function key is defined for character string entry, and is depressed while the system is waiting at an INPUT statement, the defined characters appear on the screen as if they had been keyed in character-by-character. If the (EXEC) key is then depressed, they are entered into the receiving variable, and processing proceeds. Thus if a program requires frequent keying of the same characters, a DEFFN' statement can be incorporated which allows the characters to appear with a single stroke of a Special Function key. Such an entry is illustrated below.

        ENTER DESCRIPTION OF ADDITIONAL CHARGES? FREIGHT CHG.
                                                 _____/   ↑
                                                 Key S.F. 0   Key (EXEC)
                                                                TO
                                                                ENTER
                                                                RESPONSE

Special Function keys defined as character strings can also be useful during programming. For example, if a DEFFN' such as

        1 DEFFN'15 "LIST S 100, 9000"

is included in a program, a segmented listing of program lines 100 to 9000 can be obtained by simply keying Special Function key 15 followed by (EXEC). If a

179

program makes frequent use of a particular marked subroutine, a DEFFN'
statement such as

        2 DEFFN' 0 "GOSUB' 243"

will allow you to enter the characters

        GOSUB' 243

into a program line by simply depressing Special Function key 0.

Review of Chapter 14

1. The DEFFN' statement marks the beginning of a subroutine, and gives the
    subroutine an identification number. Optionally a list of variables may
    be specified. Specified variables are assigned values when control is
    passed to the subroutine.

2. The general form of the DEFFN' statement for a marked subroutine is:

        DEFFN' integer [(variable [,variable...])]

        where: integer =   0 to 31 to define Special Function keys for
                             subroutine branching.
                         =   0 to 255 to mark GOSUB' accessable
                             subroutines.

                variable =   any alphanumeric or numeric variable to be
                             assigned a value when a branch to the
                             subroutine is made.

3. The GOSUB' statement initiates a branch to a specified DEFFN' marked
    subroutine. Its general form is:

        GOSUB' integer [(subroutine argument [,subroutine argument...])]

        where: $0 \le \text{integer} \le 255$, and is the integer in a DEFFN' statement.

                "Subroutine argument" is a value to be assigned to the next
                successive variable in the DEFFN' list of variables. Values
                may be specified by a literal string, alphanumeric variable,
                or expression.

4. A DEFFN' subroutine whose integer identification is in the range 0 to 31
    can be accessed by means of the corresponding keyboard Special Function
    key. If a Special Function key is depressed when the system colon or
    question mark is displayed, the system searches through the program text
    for the DEFFN' statement, and initiates a branch to that location. Upon
    execution of the RETURN statement, the system returns to the program
    location from which the Special Function key was depressed; the colon or
    question mark is redisplayed.

5. The DEFFN' statement can be used to associate a character string with a Special Function key. Depression of the Special Function key causes all the specified characters to be entered, as if they had been entered one-by-one.

6. The general form of the DEFFN' statement for character string definition is as follows:

    DEFFN' integer "character string"

    where: $0 \leq$ integer ≤ 31.

PART II
GAINING PROFICIENCY

CHAPTER 15
CONTROLLING OUTPUT FORMAT WITH
IMAGE (%) AND PRINTUSING

15-1 <u>INTRODUCING IMAGE AND PRINTUSING</u>

The Image (%) and PRINTUSING statements are used together to precisely control the format of printed output. To use these statements, you first write an Image statement in which you specify a print format. Then, when you want to print a value according to that format, you use a PRINTUSING statement. The PRINTUSING statement gives the line number on which the Image statement can be found, and it gives the values that are to be printed.

PRINTUSING and Image statements are commonly used in applications dealing with dollar amounts. Dollar amounts should be printed to at least two decimal places, rarely more. The program and output shown in Example 15.1 shows simple numeric output from the PRINT statement compared with the output from PRINTUSING and Image statements.

Example 15.1 Comparison of Output from PRINT and PRINTUSING

```
110 REM FIRST EXAMPLE COMPARING PRINT AND PRINTUSING
120 PRINT  "PRINT OUTPUT",, "PRINTUSING OUTPUT"
130 FOR K = 1 TO 7
140     READ N
150     PRINT   N,,
160     PRINTUSING 180, N
170 NEXT K
180 % ##,###.##
190 DATA 14500.00, 2.00, 2.50, 2.65,  .10,  .01,  1200.456456456

:RUN

PRINT OUTPUT                      PRINTUSING OUTPUT
 14500                              14,500.00
 2                                       2.00
 2.5                                     2.50
 2.65                                    2.65
 .1                                      0.10
 1.00000000E-02                          0.01
 1200.456456456                      1,200.45
```

183

The loop in this program simply reads the next data value and prints it, first with PRINT, then with PRINTUSING. The PRINT statement formats output according to its own fixed rules: trailing fractional zeros are never printed; if the value is an integer, the decimal point is not printed; all significant fractional digits are output; if the value is less than .1, it is output in scientific form; and output is not aligned. By contrast the PRINTUSING statement outputs according to a format specified in the program. In line 160

        160 PRINTUSING 180, N

the "180" is the line number of the Image statement that contains the format to be used. (The % sign is the keyword for the Image statement; the word "Image" is never used.) PRINTUSING must always refer to a line number that contains an Image statement. The variable N in line 160 is the print element. Each time line 160 is executed, PRINTUSING outputs N according to the Image given in line 180.

The Image specified consists of a space followed by a "format specification." The format specification is the

        ##,###.##

part of the Image.

In the format specification the # symbol is used to indicate that a particular digit is to be printed. The decimal point (period) indicates the location of the decimal, and specifies that it is to be printed. Commas merely indicate where a comma is to appear.

In the PRINTUSING output, notice that exactly two digits to the right of the decimal are always output. Digits to the right of these are simply truncated (as in the last DATA value), but, if zeros occupy the first two decimal positions, they are printed. Output is aligned at the decimal point, and scientific notation is not used, regardless of the value.

Probably the easiest way to understand how the PRINTUSING and Image statements work is to imagine that the system takes the image and replaces # symbols with digits. When this process is complete, it outputs the "image" at the current cursor location, not the original image, but the image with the digit substitutions (the original image is unchanged.)

A single Image statement can have several format specifications. Example 15.2 shows an Image statement with four format specifications, and a PRINTUSING statement with four print elements. All the print elements are the same, but the output varies, since it is determined by the format specifications.

Example 15.2 PRINTUSING and an Image with Four Format Specifications

```
110 REM AN IMAGE WITH 4 FORMAT SPECIFICATIONS
120     PRINTUSING 130, 123.45, 123.45, 123.45, 123.45
130     %###    ###.#    ###.##        ###.###

:RUN

123    123.4   123.45          123.450
```

In executing the PRINTUSING statement of line 120, the system takes the first print element (123.45) and starts substituting digits from it into the first format specification (###). Since there is no decimal point in the format, the decimal location is implied immediately to the right of the format. The digit "1" is substituted for the first # symbol, the digit "2" for the second #, the digit "3" for the third. Now the system finds no more # symbols, so the remaining digits in the first print element are ignored. The system proceeds to the second print element, and starts transferring it into the second format specification. It first aligns the decimal, then starts substituting digits for # symbols. After replacing the 4, it runs out of # symbols. It then moves to the third print element and the third format specification. Finally it moves to the fourth print element and format specification. In this last case, after substituting the digits 1, 2, 3, 4 and 5, it finds another # symbol, but no more digits. Since this extra # symbol is to the right of the decimal, the system puts a zero in its place.

Now the system has exhausted the print elements in the PRINTUSING statement, and has an image in which digits have replaced # symbols. This image is output in its entirety beginning at the current cursor location. As a result, the spacing within the Image statement is exactly duplicated in the output. Since no spaces precede the first format specification, the first digit is output directly under the colon. It is important to understand that it is the image which is output, after the digit substitutions have been made.

In the last format of line 130 in Example 15.2, there is an extra # to the right of the decimal. This # symbol is replaced by a zero during PRINTUSING execution. However, if we look again at Example 15.1, we notice that for each DATA value except the first, there are extra # symbols to the left of the decimal. When there are extra number symbols to the left of the decimal, they are replaced by spaces, not zeros. Thus, on the first time through the loop of Example 15.1, this substitution is made:

```
┌─────────────────┐
│  print element  │  →  14 500.00
└─────────────────┘      ↓↓ ↓↓↓ ↓↓
          ┌───────┐
          │ image │ →  % ##,###.##
          └───────┘
```

On the second time through, the system supplies spaces for the #'s (and the comma) left of the first digit; thus,

```
    spaces
    supplied        print
    by system,    element
   ╰──────────╯  ╰───────╯
         ↘          ↙
          ╭─────────╮
          △△△△△2.00
          ↓↓↓↓↓↓ ↓↓
          % ##,###.##
```

Notice that the comma is replaced by a space if there is no digit to its left.

From the 5th and 6th DATA values in Example 15.1, you will notice that if there are no significant digits left of the decimal, the # immediately left of the decimal is replaced with a zero.

185

A single Image statement can be referenced by several PRINTUSING statements. The Image statement has no effect when encountered in the normal sequence of execution, and may be placed anywhere in a program without regard to the location of the PRINTUSING statements that reference it. However, the Image statement must be the only statement on a line.

PRINTUSING and Image are illegal in the Immediate Mode. The keyword "PRINTUSING" may not be entered by keying the keyword "PRINT", and then typing U-S-I-N-G. It may be entered character-by-character, or by means of the PRINTUSING key on a BASIC KEYWORD keyboard.

PRINTUSING output occurs at the address selected for PRINT class I/O operations. Therefore, if a statement such as

SELECT PRINT 215

has been executed, PRINTUSING output occurs at the printer rather than the CRT.

Any expression can be used as a PRINTUSING print element. The expression is evaluated, and its result is substituted digit-by-digit into the format specification. In addition, alphanumeric literal strings and variables can be used as print elements. Their use is discussed in Section 15-4. Each print element in the PRINTUSING statement must be separated from the previous one by a comma or semicolon. The semicolon has a special significance, discussed in Section 15-5. The comma acts as a simple element separator. It does not have the significance it has in the PRINT statement, and causes no cursor movement.

15-2   ALPHANUMERIC LABELS IN THE IMAGE STATEMENT

In addition to format specification, any alphanumeric characters (other than # and colon) can be included in an Image statement. This allows for easy labeling of output. Example 15.3 shows a modification of a program first introduced in Chapter 12. The program is a DEFFN' subroutine that converts degree measure to radian measure. The output is via the PRINTUSING statement. The Image labels the output, and formats it.

Example 15.3 Alphanumeric Labels in the Image Statement

```
110 REM USING ALPHANUMERIC CHARACTERS IN THE IMAGE STATEMENT
120     DEFFN' 11
130     INPUT "ENTER DEGREES, MINUTES, SECONDS", D,M,S
140     A = D +M/60 +5/3600
150     A = A-INT(A/360)*360
160 REM ********************************
170    PRINTUSING 180,  A*.0174533
180    % ANGLE= #.#### RADIANS
190 REM ********************************
200     RETURN
```

Output appears as follows:

```
        ENTER DEGREES, MINUTES, SECONDS? 45,15,15
        ANGLE= 0.7898 RADIANS
```

In Example 15.3 the expression A*.0174533 is the only print element in the PRINTUSING statement. The value of this expression is calculated, and the result is substituted digit-by-digit into the format specification in the Image statement (line 180). The Image contains just one format specification, #.####. The other characters do not constitute a format specification, and are merely output as a part of the Image, after digits have been substituted for # symbols. The format specification calls for just four fractional digits, so just four are output; the additional fractional digits in the result are simply truncated. Notice that the output from the PRINTUSING exactly duplicates the Image, except that print element digits have been substituted for # symbols. The Image begins immediately after the % sign; thus the space between % and "ANGLE" appears in the output.

It is possible to intersperse labels between several format specifications. Example 15.4 shows a slight modification of Example 15.3; it outputs the entered degrees, minutes, and seconds together with the radian value.

> Example 15.4 An Image Statement with Several Labeled Format
>   Specifications

```
110 REM ANOTHER EXAMPLE OF ALPHANUMERICS IN THE IMAGE STATEMENT
120        DEFFN' 11
130        INPUT "ENTER DEGREES, MINUTES, SECONDS", D,M,S
140        A = D +M/60 +S/3600
150        A = A-INT(A/360)*360
160 REM ********************************
170     PRINTUSING 180, D, M, S, A*.0174533
180     %(### DEG, ## MIN, ## SEC) = #.#### RADIANS
190 REM ********************************
200        RETURN
```

Execution produces

```
ENTER DEGREES, MINUTES, SECONDS? 320,15,15
 (320 DEG, 15 MIN, 15 SEC) =  5.5894 RADIANS
```

If there are fewer print elements in the PRINTUSING statement than format specifications in the Image statement, the portion of the Image that lies to the right of an unused format is not output. For example, if the print element S were accidentally omitted from line 170 as follows:

```
170 PRINTUSING 180, D, M, A*.0174533
```

execution would produce:

```
        ENTER DEGREES, MINUTES, SECONDS?  320,15,15
         (320 DEG,  15 MIN,  5 SEC)=
```

187

Here, the first two print elements are output correctly. However, since S is missing, the third print element A*.0174533 replaces the third, but inappropriate, format specification ##. (The 5 which appears before "SEC" is actually the integer portion of 5.5894 radians.) Now there are no more print elements, so the system outputs the image. However, output stops as soon as the system finds a format specification for which digits have not been substituted. No substitution has been made into #.####; therefore, = Δ are the last characters output.

It is possible to use an Image statement in which no format specification occurs, an Image that consists merely of alphanumeric characters. This can be convenient when creating report headings that must be aligned with columns of output. The Image statement for the headings can be directly aligned on the CRT over the data-output Image statement. For example,

Example 15.5 An Image Statement without a Format Specification

```
110 REM USING AN IMAGE STATEMENT WITHOUT FORMAT SPECIFICATIONS
120 %PART NO.     ON HAND     ON ORDER
130 %#######      #####       #####
 .
 .
 .
270 PRINTUSING 120
280 PRINTUSING 130, A, B, C
 .
 .
 .
```

In this example the Image statement at line 120 consists only of alphanumeric characters; there are no format specifications. Line 270 simply tells the system to output the Image found on line 120. Notice in line 270 that the "120" is not followed by a comma, and no print elements are specified. A PRINTUSING statement which references an Image such as this must not have any print elements.

15-3 THE $, +, AND - SYMBOLS

Thus far, in our PRINTUSING examples, we have printed positive numbers only. The output from the following program illustrates the output when a negative value is printed.

Example 15.6 Printing Negative Values without a Sign in the Format

```
110 REM PRINTING NEGATIVE VALUES WITHOUT A SIGN IN THE FORMAT
120 PRINTUSING 140, 25.45, 1615.18
130 PRINTUSING 140, -25.45, -1615.18
140 %####.## ####.##

:RUN
  25.45 1615.18
-  25.45 -1615.18
```

Line 120, which outputs positive values, is included in this example for the purpose of comparison. The negative values output by line 130 cause a minus sign to be output at the left of each format, and increase the length of the format specifications by one character. This increase in length causes the misalignment of the columns, as shown. In general, this effect is undesirable: output is misaligned with previous output, and the minus sign floats at the left of the format, perhaps leaving several spaces between it and the first digit of the value.

Beginning a Format Specification with a Minus Sign

Whenever a format specification may receive a negative value, it should be preceded with a minus (-) sign. A minus sign at the beginning of a format specification has special significance. It tells the system to output a minus sign immediately preceding the leftmost digit of a negative value, or output a space if the value is positive. Example 15.7 shows the result of adding a minus sign to the formats previously shown in Example 15.6.

Example 15.7 A Minus Sign in a Format Specification

```
110 REM USING A MINUS SIGN IN THE FORMAT
120 PRINTUSING 140, 25.45, 1615.18
130 PRINTUSING 140, -25.45, -1615.18
140 %-####.## -####.##

:RUN
   25.45  1615.18
  -25.45 -1615.18
```

Notice in this example that the minus sign always appears immediately to the left of the leftmost digit, regardless of the number of digits output. Since there is room in the format for the minus sign, the format does not have to be expanded to accommodate it. As a result, all columns are aligned.

Beginning a Format Specification with a Plus Sign

A plus sign may be used to begin a format specification. A plus sign has the same general effect as a minus sign, except in one respect: when the value of the print element is non-negative (greater than or equal to zero), the + sign is output immediately preceding the leftmost digit. If the value is negative, the minus sign is output. Example 15.8 illustrates the effect of the + sign when used to begin a format specification.

189

Example 15.8 A Plus Sign in a Format Specification

```
110 REM USING A PLUS SIGN IN THE FORMAT
120 PRINTUSING 140,  25.45,  1615.18
130 PRINTUSING 140, -25.45, -1615.18
140 %+####.## +####.##

:RUN
  +25.45 +1615.18
  -25.45 -1615.18
```

When printing values that frequently alternate between positive and negative, the explicit plus sign can increase clarity.

The Dollar Sign

If a format specification begins with a dollar sign ($), a dollar sign is output immediately preceding the leftmost digit, if the value is positive. If the value is negative, $- precedes the leftmost digit.   The dollar sign cannot be used together with a + or - in the format. It is used instead of these symbols, and causes sign output analogous to the minus symbol.   Example 15.9 illustrates the effect of the dollar sign used to begin a format specification.

Example 15.9 The Dollar Sign in a Format Specification

```
118 REM USING A DOLLAR SIGN IN THE FORMAT
120 PRINTUSING 140,  25.45,  1615.18
130 PRINTUSING 140, -25.45, -1615.18
140 %$####.## $####.##

:RUN
  $25.45 $1615.18
  $-25.45 $-1615.18
```

Notice in Example 15.9 that when the value is positive, the $ always appears immediately left of the most significant digit, and that the format is not expanded. When the value is negative, $- immediately precedes the value. This does not cause expansion of the format unless the entire format specification, including the $, is not large enough to accommodate the value together with the $-. This occurs here for the value -1615.18, and causes a misalignment of the columns. This misalignment can be avoided by simply providing a format with one more #, left of the decimal, than the value will ever occupy. In this case line 140 would be changed to

```
140 %$####.## $#####.##
```

and the output would be aligned as follows:

```
:RUN
  $25.45   $1615.18
  $-25.45 $-1615.18
```

190

A maximum of 16 # symbols can appear in a single format specification used for numeric output.

Rounding

The PRINTUSING and Image statements never round values. Any fractional digits beyond those selected for output are merely truncated. However, if a value is simply to be printed, and does not enter into further calculations, it can be rounded by adding 5 in the decimal position immediately to the right of the rightmost printed digit. Thus, if a format prints to two decimal places, .005 added to the value, prior to printing, will round the output to two decimal places. This assumes the value is positive. The added .005 causes a carry into the second position, if the digit in position three is 5 or greater. For example,

```
110 REM ROUNDING
120 INPUT "NUMBER TO BE ROUNDED", A
130 PRINTUSING 140, A+.005
140 % ####.##
```

If the value to be rounded is negative in the above example, -.005 must be added in order to round. A simple way of rounding, when the sign of the value to be rounded is unknown, is to multiply the rounding factor by the SGN() of the value, before adding. The above example would be modified as follows:

```
110 REM ROUNDING (POSITIVE OR NEGATIVE)
120 INPUT "NUMBER TO BE ROUNDED", A
130 PRINTUSING 140, A + (SGN(A)*.005)
140 % -####.##
```

If a value is to be used in further calculations after it is printed, it is not advisable to round in the manner shown above, since the value of A is not changed by line 130. For example, if A equals 45.779 and .005 is added, truncated A will print as 45.78, but, A still equals 45.779 and could yield apparently erroneous results if it enters into further calculations. In such circumstances, the value itself should first be rounded and truncated, using the procedure given in Example 14.4, or in Chapter 6.

We have noted that if the number of digits left of the decimal is less than the number of # symbols, the # symbols are replaced by spaces. However, a different problem arises if there aren't enough # symbols left of the decimal to accommodate the digits. For example, if the format specification is -###.## and the value of the print element is 7500.50, there is no place for the 7 in the format. When this happens, the system doesn't substitute any digits into the format specification. The result is that the format specification itself is printed. In this case -###.## would show up in the output. This, then, is an indication of a programming error: a format specification is too small for a value it is to receive.

Example 15.10 shows a modification of Example 7.10, a mortgage payment program.

191

Example 15.10   PRINTUSING in the Mortgage Payment Problem of Chapter 7

```
110 REM PRINTUSING IN A MORTGAGE PAYMENT PROGRAM
120 REM OPERATOR ENTERS VALUE FOR PRINCIPAL
130     INPUT "ENTER PRINCIPAL",P
140 REM *******************
150     PRINTUSING 280
160     PRINTUSING 290
170 REM *******************
180     PRINT
190     FOR T=20 TO 40 STEP 5
200          FOR I = 7 TO 9 STEP .5
210               M=P*(I/1200)/(1-(1+I/1200)↑(-12*T))
220          REM *****************************
230               PRINTUSING 300, P, T, I, M+.005
240          REM *****************************
250          NEXT I
260        PRINT
270     NEXT T
280 %                         INTEREST      MONTHLY
290 %PRINCIPAL        TERM        RATE      PAYMENT
300 %$##,###.##    ## YEARS     ##.##%     $###.##
```

In this example, notice that the report layout stands out in the program much more clearly than it did in Example 7.10. The complicated round and truncate operation of line 270 of Example 7.10 has been replaced by a simple rounding effected by M+.005, and a truncation accomplished by the format specification. Column output presents an even right edge, as shown below.

PRINCIPAL	TERM	INTEREST RATE	MONTHLY PAYMENT
$30,000.00	20 YEARS	7.00%	$232.59
$30,000.00	20 YEARS	7.50%	$241.68
$30,000.00	20 YEARS	8.00%	$250.93
$30,000.00	20 YEARS	8.50%	$260.35
$30,000.00	20 YEARS	9.00%	$269.92

15-4   ALPHANUMERIC PRINT ELEMENTS

The PRINTUSING statement can be used to output alphanumeric values. Alphanumeric print elements may be in the form of literal strings or alphanumeric variables. In the Image statement, the format specification for alphanumeric values usually simply consists of # symbols. Alphanumeric values are substituted character-by-character into the format specification. The leftmost # receives the leftmost character of the value, producing a left alignment of the value in the format specification. If there are fewer characters in the value than #'s in the format, the extra #'s at the right are filled with spaces. If there are more characters in the value, the extra characters at the right are simply truncated. A format specification used for alphanumeric output may contain any number of # symbols.

A single PRINTUSING statement may contain both numeric and alphanumeric print elements.   Example 15.11 shows a program segment that outputs a report line of mixed numeric and alphanumeric values.

Example 15.11   Alphanumeric Print Elements

```
120 DIM N$20
130 %    EMPLOYEE          JOB     REG.      O/T       GROSS
140 %    NAME              NUM.    HOURS     HOURS     PAY
150 %##################    ####    ##.##     ##.##     ###.##
 .
 .
 .
890 PRINTUSING 130
900 PRINTUSING 140
910 PRINTUSING 150, N$, J$, R, O, G
 .
 .
 .
```

Output from these lines appears as:

```
EMPLOYEE                  JOB     REG.      O/T       GROSS
   NAME                   NUM.    HOURS     HOURS     PAY
R. J. THOMAS               155    40.00      5.50     258.60
```

A "numeric" format specification, one that contains $,.-+, may also be used for alphanumeric output. However, none of these special characters are edited into the output, as they are with numeric print elements. Instead, if such a format specification receives an alphanumeric value, these characters act as if they are # symbols. The alphanumeric characters of the print element are substituted for characters in a format specification such as $##,###.## as if this specification were #########. Occasionally this can be useful. Example 15.12 shows Example 15.11 modified so that the column-output Image is used for the output of the headings. This yields some reduction in total memory occupied by the program.

Example 15.12   Printing Alphanumerics with "Numeric" Format
              Specifications

```
110 REM ALPHANUMERIC PRINT ELEMENTS INTO "NUMERIC" FORMATS
120 DIM N$20
150 %##################    ####    ##.##     ##.##     ###.##
 .
 .
 .
890 PRINTUSING 150, "EMPLOYEE","JOB","REG ","O/T","GROSS"
900 PRINTUSING 150, "NAME","NUM ","HOURS","HOURS","PAY"
910 PRINTUSING 150, N$, J$, %, O, G
```

Output from these lines appears as

```
EMPLOYEE                  JOB     REG       O/T       GROSS
NAME                      NUM.    HOURS     HOURS     PAY
R. J. THOMAS              A155    40.00      5.50     248.60
```

193

15-5   SUPPRESSING THE CR/LF

Reusing an Image

If the number of print elements in a PRINTUSING is greater   than   the
number of format specifications in the Image statement, a carriage return/line
feed is issued when the formats are exhausted. The Image is then reused, from
the beginning, for the remaining print elements. In Example 15.13, a single
PRINTUSING uses an Image three times to complete its output.

Example 15.13 Using an Image Repeatedly with a Single PRINTUSING

```
110 REM USING AN IMAGE REPEATEDLY WITH ONE PRINTUSING
120 DIM A(3), B(3)
130 % ACCT. NO.            AMOUNT
140 %    ######        -#,###,###.##
.
. (values entered into A() and B())
.
380 PRINTUSING 130
390 PRINTUSING 140, A(1), B(1),  A(2), B(2),  A(3), B(3)
```

Output from this program segment appears as:

```
ACCT.  NO.                AMOUNT
        101           14,512.01
        105           44,500.00
        112           16,357.95
```

In this example, line 390 outputs the three lines of data by using the
Image (line 140) three times. In executing 390 the system first substitutes
the digits of A(1) into the first format specification, ######.   Then, the
digits of B(1) are substituted into the second format specification. The
system now notices that there are more print elements but no more format
specifications.   It   therefore issues a CR/LF, which moves the cursor to the
leftmost position on the next line.   Now it reuses the Image from the
beginning, substituting A(2) into the first format specification and B(2) into
the second.   A CR/LF is issued and the process repeats itself a third time
outputting A(3) and B(3). After the third line of output, PRINTUSING issues a
final CR/LF, and passes control to the next instruction.

The Semicolon

The CR/LF that is issued before the system reuses an Image, may be
suppressed by placing a semicolon in the PRINTUSING statement. The semicolon
is used instead of the comma, and must follow the print element associated
with the last format specification in the Image. Suppressing the CR/LF in
this fashion causes the output from the Image to be repeated on the same line.
Example 15.14 shows a PRINTUSING statement that uses an Image twice, and
suppresses the CR/LF with a semicolon.

Example 15.14   Suppressing a CR/LF with a Semicolon

```
110 REM SUPPRESSING THE CR/LF WITH A SEMICOLON
120 % ###      #### PPM     #### CU.MM
130 DIM I(15), C(15), V(15)
140 PRINT "ITEM     CONCEN-      TOTAL      ITEM     CONCEN-     TOT
AL"
150 PRINT " NO.    TRATION      VOLUME      NO.     TRATION     VOL
UME"
   .
   .
   .
270 PRINTUSING 120, I(K), C(K), V(K);  I(K+1), C(K+1), V(K+1)
```

Output from this program segment appears as follows:

```
ITEM    CONCEN-     TOTAL        ITEM    CONCEN-     TOTAL
NO.     TRATION     VOLUME       NO.     TRATION     VOLUME
 13      455 PPM    0150 CU.MM    14      315 PPM    0170 CU.MM
```

Notice in this example that the Image at line 120 contains three format specifications. The PRINTUSING at 270 contains six print elements. Execution of line 270 causes print elements I(K), C(K), and V(K) to be substituted into the three formats of line 120. Normally a CR/LF would be issued at this point, and the output of the next three elements would appear on the next line. However, the print element V(K) is followed by a semicolon.   This suppresses the CR/LF. The Image is reused for print elements I(K+1), C(K+1), and V(K+1). Output appears on the same line.

Regardless of the number of format specifications actually used by a PRINTUSING statement, PRINTUSING normally outputs a CR/LF immediately prior to passing control to the next instruction. This final CR/LF can be suppressed by placing a semicolon at the end of the PRINTUSING statement. Example 15.15 shows a program which prints a table of random numbers using RND and PRINTUSING. In the PRINTUSING statement the normal CR/LF at the end is suppressed by a trailing semicolon.

Example 15.15 A Semicolon at the End of the PRINTUSING Statement

```
110 REM SUPPRESSING THE CR/LF WHICH NORMALLY FOLLOWS PRINTUSING
120 %#####
130 FOR I = 1 TO 24
140 PRINTUSING 120, RND(1)*1E5;
150 NEXT I
```

Output from this program appears as follows:

```
85710   91609   24725    5294   76934   75577   39969   34105
48561   44686   10999   14629   44239   31110   64724   70047
88009   55461   21020   80680   29352   51562   75265   31696
```

In this program a single print element, RND(1)*1E5, is substituted into a single format specification. Normally, after each execution of line 140, a CR/LF would be issued.   However, the semicolon at the end of line 140 suppresses the CR/LF. A CR/LF is issued only when the line becomes completely full.

Looking at Example 15.15 you may wonder what is causing the spacing between the columns; no spaces are visible in the Image. Despite the fact that none are visible, three spaces are in the Image following the format. They do not appear in the listing, but if line 120 were recalled in EDIT mode, it would appear as

     *120 %#####   _

with the cursor positioned three spaces to the right.

15-6   EXPONENTIAL FORMAT

Exponential format may be specified by ending a format specification with ↑↑↑↑.   (Four up-arrows (↑) are always used.) The four up-arrows are replaced in the output with the standard exponent form E+XX.   The   mantissa is scaled so that its most significant digit occupies the leftmost # symbol, and the value of exponent is adjusted to offset this scaling.   Any numeric value may be output in exponential format. Example 15.16 shows the results of output in exponential format.

Example 15.16 Using Exponential Format Specifications

```
110 REM EXPONENTIAL FORMAT SPECIFICATIONS
120% COEFF = +.#### ↑↑↑↑ERROR = -## ↑↑↑↑
130 PRINTUSING 120,2.13E-5, 2.3E-9
:RUN
 COEFF = +2130E-04 ERROR = 23E-10
```

Notice that the mantissa has been scaled to fit the number of # symbols left of the decimal in the format, and the exponent adjusted accordingly.

CHAPTER 16
MORE ABOUT ALPHANUMERICS

16-1 <u>HEX CODES</u>

In Chapter 8 we introduced the idea of alphanumeric variables. We depicted an alphanumeric value in the memory of a Wang 2200 system without saying very much about how the characters are actually recorded. For example, if A$ = "BOSTON, MA", and A$ can hold a maximum of 16 characters, we would have depicted memory as

A$ | B | O | S | T | O | N | , | Δ | M | A | Δ | Δ | Δ | Δ | Δ | Δ |

where Δ means "1 space"

In fact, each character in A$ is not recorded in memory the way we see it here. In memory, each character is recorded as a binary code. The value of A$ in memory consists of the binary code for B, followed by the binary code for O, followed by the binary code for S, etcetera. Since there is a binary code for a space, A$ is recorded in memory as 16 binary codes, one for each character and each space (a space just being a special kind of character).

These codes are called "binary" codes since the code for each character is made up entirely of ones and zeros. For example, the character B is represented by the binary code

01000010

The letter O, the second character in A$, is represented by

01001111

With each code consisting of eight binary digits, there are a total of 256 possible codes.

Each keyboard character is represented by one of these codes. When a key is depressed, the keyboard sends the proper code to the CPU. However, as Table 8.1 shows, there are only 86 characters that can be entered from the keyboard. This leaves 170 codes unused by the keyboard characters. These "extra" codes are used in a variety of ways.

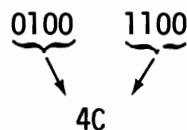For example, if the CRT receives the code

00000011

it interprets this to mean, "Clear the screen and move the cursor to the top
left corner." There are other codes that are given special interpretations by
the CRT.   In addition, printers execute form control and other operations
based upon receipt of certain special codes. However, before we discuss how
these special codes can be used, we must look at another, more convenient, way
of writing them.

Binary codes are inconvenient for human use. In order to write one code
for a single character, such as L, you must write eight binary digits

01001100

For this reason, a kind of shorthand for binary, called "hexadecimal" or
"hex", is used instead.   Hex uses one hex digit to stand for each group of
four binary digits. By breaking the eight binary digits for "L" into two
groups of four, and using Table 16.1 below, you can see that, in hex, this
code would be written as 4C, as follows:

0100   1100
\   /
4C

So, we say that for the Wang 2200 system the hex code for "L" is 4C.

BINARY	HEX EQUIVALENT	BINARY	HEX EQUIVALENT
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Table 16.1 Binary and Hex Equivalents

Since binary character codes always contain eight binary digits, the hex
representation always contains two hex digits. The hex digits 0-9 and A-F
must be thought of as just 16 symbols, each arbitrarily chosen to stand for
four binary digits. By contrast, the character "A" is represented by the hex
code 41 (its binary code is 01000001).

For the remainder of this volume we can ignore the cumbersome binary
representation and refer   only to hex codes, aware that when we say
something like "This puts hex 4C into A$," we are just using hex 4C as a
kind of shorthand for the actual binary code.

It should be noted that the hex codes for the lower case characters are
different from the hex codes for the upper case characters. To the processor,
"A" and "a" are as different as "A" and "Z". A complete table of characters
and hex codes is given in Appendix C.

16-2 THE HEX() FUNCTION

Suppose we want to use the control codes for the CRT that we mentioned earlier. We want to clear the CRT screen and put the cursor in the top left position. (This cursor position (line 0, column 0) is sometimes called "home".) The hex code 03, received by the CRT, tells it to clear the screen and home the cursor. How can we get the processor to send this code to the CRT?

When we want the CRT to print a character, we just write a statement such as

        10 PRINT "A"

or two statements such as

        10 A2$ = "A"
        20 PRINT A2$

In each case, execution of these statements causes the CPU to send the hex code for "A", hex 41, to the CRT. However, there isn't any key on the keyboard that we could use in place of the "A" key which would tell the CPU to send hex 03 to the CRT. (The keyboard numeral "3" is represented by hex 33.) Wang BASIC, therefore, provides the alphanumeric function HEX() as a means of directly specifying any desired hex code or series of hex codes. HEX() says to the system, "Interpret the characters in parentheses to be a directly specified hex code, or codes." For example, to clear the CRT screen, you can simply execute:

        :PRINT HEX(03)

This may be executed as shown, in the Immediate Mode, or as a program statement, provided that the CRT is selected for PRINT output. Alternatively, this could be accomplished with the statements

        10 A2$ = HEX(03)
        20 PRINT A2$

Statement 10 assigns the hex code 03 to the variable A2$. The code hex 03 then occupies the first, leftmost, character position in A2$. The remaining character positions in A2$ are occupied by spaces, hex 20. Therefore, A2$ looks like this in memory:

A2$ | 03 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |

At statement 20 the CPU looks at A2$, determines that it contains one non-space character, the hex 03, outputs that character and ignores the trailing spaces, the fifteen hex 20 codes. The result is that the screen is cleared and the cursor is put in the home position.

The HEX function may be thought of as a special kind of literal string, used when a non-keyboard character code is needed. An alphanumeric value may be specified with a HEX function whenever a literal string in quotation marks may be used, with a few exceptions. The exceptions are:

A HEX function may not be used in

1) An INPUT statement prompt.
2) A keyboard response to an INPUT statement.
3) A STOP statement message.
4) A PRINTUSING statement.
5) A DATA statement.
6) (On the 2200S only) a DEFFN' statement used to define a character string for Special Function key entry.

There is no limit to the number of hex codes which may be used in a single HEX function; however, a hex code must always consist of 2 hex digits. An odd number of hex digits may never be used. The following statements are examples of legal uses of HEX functions.

```
 10 A$ = HEX(03)
 40 C$ = HEX(030A)
 85 PRINT HEX(030A0A); "PROCESSING FILE"; N
100 IF A$ = HEX(09) THEN 10
200 GOSUB' 50 (4,"NAME",HEX(09))
310 DEFFN'15 "LISTS 100, 9000"; HEX(0D)*
        *NOT LEGAL ON 2200S
```

Chapter 17 discusses the use of hex codes in controlling the CRT. Chapter 18 discusses their use in controlling a printer.

The Hex Function in the DEFFN' Statement

On Wang 2200T systems (or 2200S with OP-24), the HEX function may be used in DEFFN' statements, when these are used to associate a character string with a Special Function key. The general form of the DEFFN' statement for the 2200T is

DEFFN's $\left\{ \begin{matrix} \text{HEX()} \\ \text{"character string"} \end{matrix} \right\}$ $\left[\begin{matrix} ; \\ ; \end{matrix} \left\{ \begin{matrix} \text{HEX()} \\ \text{"character string"} \end{matrix} \right\} \quad .. \right]$

or

DEFFN'i   [(variable [,variable])]

Where s is an integer 0-31 defining a Special Function key.
       i is an integer 0-255 (if i $\leq$ 31, it defines a Special Function key).

The lower form shown is for marked subroutine definition; the top form is for character string definition of Special Function keys.

Wang CRT's and printers can print certain characters which do not appear on the keyboard. If you wish to use these characters you can define a Special Function key to enter the required hex code for the character. For example, hex 5B and 5D print left and right brackets [] on any Wang CRT. However, these characters do not appear on the keyboard. If you wish an operator to be able to enter these characters during program execution, you could define Special Function keys 0 and 1 to the proper hex codes as follows:

```
6010 DEFFN'0 HEX(5B)
6020 DEFFN'1 HEX(5D)
```

With these two statements in a program, depressing Special Function key 0 causes a left bracket to be entered and appear on the CRT. Special Function key 1 enters a right bracket. (Note, however, that these characters may not be used in BASIC statements as substitutes for parentheses.)

The (EXEC) key causes the hex code OD to be entered. If a character string in a DEFFN' statement is followed by HEX(OD), the string will appear on the CRT, and be entered with one stroke of the defined Special Function key. For example, if this statement appears in a program

        10 DEFFN'1 "LISTS 500, 1000"

then keying Special Function key 1 when the colon is displayed causes the following to appear on the screen:

        :LISTS 500, 1000

To execute this LISTS command, (EXEC) must be keyed. However, if the statement is changed to

        10 DEFFN' 1 "LISTS 500, 1000"; HEX(OD)

then merely depressing Special Function key 1 will initiate execution of the LISTS command. The HEX(OD) at the end is the equivalent of keying (EXEC). A semicolon must be used to separate HEX() functions from literal strings in the DEFFN' statement.

Complete tables of hex codes and characters for the CRT's and printers are given in Appendix C.

16-3 THE STRING FUNCTION

The string function, STR(), allows you to use and operate on any specified section of an alphanumeric variable. For example,

        STR(A$,2,4)

specifies the section of A$ beginning with the 2nd character position, taking a total of 4 consecutive character positions. Thus, if

A$ = | N | 4 | 0 | 7 | 5 | - | A | 2 | 1 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |

then

STR(A$,2,4) = | 4 | 0 | 7 | 5 |

The string function

        STR(B$,9)

201

specifies the portion of B$ beginning at the 9th character position and extending through the end of the variable.

Thus, if

B$ = | Δ | Δ | Δ | Δ | Δ | Δ | A | B | C | D | E | F | G | Δ | Δ | Δ |

then

STR(B$,9) = | C | D | E | F | G | Δ | Δ | Δ |

A string function can be used whenever an alphanumeric variable can be used. The string function is an extremely versatile feature of Wang BASIC.

If,

S$ = | K | 1 | 0 | 4 | 3 | - | R | B | A | Δ | Δ | Δ | Δ | Δ | Δ | Δ |

and

R$ = | 5 | 2 | 6 | 7 | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ |

then after executing

200 R$ = STR(S$,2,4)

S$ is unchanged.

R$ = | 1 | 0 | 4 | 3 | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ |

The following are examples of valid uses of STR() functions:

10 INPUT "CREDIT RATING", STR(C9$,17,2)

(assigns the entered credit rating to the 17th and 18th characters of C9$.)

200 IF STR(C9$,17,2) = "A1" THEN 280

(branches to 280 if the 17th and 18th characters of C9$ equal "A1".)

750 PRINT "SUPPLIER CODE="; STR(P$,1,5)

(prints "SUPPLIER CODE=" followed by the first five characters of P$.)

900 READ STR(A2$(5),6,1)

(assigns the next DATA value to the sixth character in the list variable A2$(5).)

200 DEFFN'40 (STR(A$,5,15),B$,K)

(specifies that the first value passed to the DEFFN' by the GOSUB' is to be assigned to the 5th - 19th characters of A$, that is, to the 15 consecutive characters beginning with character 5.)

    405 STR(B$,2,7) = STR(N$,10,7)

(beginning at the 10th character of N$, assigns the next seven characters to the portion of B$ beginning at the second character and extending through the next seven consecutive characters.) Thus, if

N$ =

| G | 5 | 9 | 5 | - | 6 | 2 | 3 | - | A | 4 | 0 | 1 | 1 | B | Δ |

                                      STR(N$,10,7)

B$ =

| N | Q | 5 | 2 | 7 | 2 | A | 2 | 2 | - | F | A | L | 2 | 3 | 1 |

       STR(B$,2,7)

then after executing statement 405

    N$ is unchanged

B$ =

| N | A | 4 | 0 | 1 | 1 | B | Δ | 2 | - | F | A | L | 2 | 3 | 1 |

       STR(B$,2,7)

The general form of the STR(function is:

STR(alphanumeric variable, expression 1, [expression 2])

where: alphanumeric variable = any alphanumeric variable (subscripted or scalar).

        expression 1 = an expression specifying the starting character in the string. Its integer value must be 1 or greater and less than or equal to the maximum size of the variable.

        expression 2 = an optional expression specifying the number of consecutive characters desired. If this expression is omitted, the entire remaining portion of the variable is specified. (If a variable, A$, is dimensioned to hold a maximum of 20 characters, then

        STR(A$,5) is equivalent to STR(A$,5,16).

        Both specify the entire remaining portion of the variable from character 5 on.)

Any numeric expression can be used in the string function to specify the starting character and the number of consecutive characters to be used. Thus, string functions of the following forms are legal (provided the expressions yield values that keep the string function within the defined size of the variable).

STR(B2$,Z9,4)

(here the value of the numeric variable Z9 specifies the starting character of the string function.)

STR(A2$,(N+2)/2, K)

(here the value of the expression (N+2)/2 determines the starting character; the value of K specifies the number of consecutive characters to be used.

NOTE: Only the integer portion of a value is used.)

When a variable is specified with a string function, all the characters in the string are included in the value of the variable, even trailing spaces. For example, enter and execute:

```
10 A$ = "T"
20 PRINT A$; "&&&"
30 PRINT STR(A$,1); "&&&"
```

The result is:

```
T&&&
T                &&&
```

The variable A$ after statement 10 consists of the character T followed by 15 trailing spaces. When A$ is printed at line 20, the trailing spaces are ignored; they are not printed. In line 30 STR(A$,1) specifies the entire variable A$, that is, "starting at the first character, take all remaining characters." Characters specified within a string function are taken to be part of the value of the variable, even if they are trailing spaces. Therefore, when STR(A$,1) is printed, all 15 spaces are printed with it.

Frequently, it is advantageous to store different data items in a single variable and access individual items by means of a string function of that variable. This is particularly true when data storage on tape or disk is considered, topics we take up in Chapters 20 and 21. However, the string function has a great many uses and we will see some of them in the coming chapters.

16-4 INITIALIZING AN ALPHANUMERIC VARIABLE WITH A SPECIFIC CHARACTER (INIT)

When RUN (EXEC) is keyed, the system sets aside space in memory for all the variables used in a program. Each numeric variable is assigned an initial value of zero, and each alphanumeric variable is filled with spaces, hex 20's.

Sometimes, though, we may wish to fill an entire alphanumeric variable, or even an entire alphanumeric array, with some other character, or hex code. For example, suppose we want to assign the hex code OA to each character in the variable A$. We could do it in this manner:

    A$ = HEX(OAOAOAOAOAOAOAOAOAOAOAOAOAOAOAOA)

This approach, however, becomes inefficient and inconvenient as the dimensioned length of A$ increases. If an entire array is to be filled in this manner, a loop must be used to assign the hex codes to each variable in the array.

On the 2200T a BASIC statement is available that assigns a specific character to an entire alphanumeric variable or to an entire alphanumeric array. This is the INIT statement. For example, the statement

    20 INIT (OA) A$

assigns the hex code OA to each character in A$. The statement

    20 INIT (OC) R$()

assigns the hex code OC to each character in each variable in the array R$().

With the INIT statement, any single hex code (two hex digits) may be specified within the parentheses as the value to be assigned. Alternatively, the value may be specified as a character in quotation marks. For example,

    30 INIT ("X") A$

assigns "X" to each character of A$.

Finally, the value to be assigned may be specified by putting an alphanumeric variable within the parentheses. If this method is used, the first character in the variable is the character which is assigned. For example,

    20 A$ = "DEF"
    30 INIT(A$) B$()

This sequence assigns to each character in each variable in B$(), the value "D", since "D" is the first character in A$.

In an INIT statement several variables may be initialized with the same character, by separating them with commas.

For example,

    40 INIT (OA) A$,B$,R$

205

The general form of the INIT statement is:

```
          (hh)              alpha variable          ,alpha variable
INIT      ("character")                                                    ...
          (alpha variable)  alpha array designator  ,alpha array designator
```

The INIT statement is legal in the Immediate Mode.

The INIT statement is not part of the 2200S instruction set. It is available to users of the 2200S as part of Option 22. For 2200S owners, without OP-22, the following technique is suggested for use whenever a long alphanumeric variable must be initialized with a specific character.

The statement sequence

```
10 A$ = HEX(0A)
20 STR(A$,2) = STR(A$,1)
```

assigns the hex code 0A to each character in the dimensioned length of A$.

This programming technique exploits the fact that the processor actually makes the assignment at line 20 on a character-by-character basis, that is, it first fetches a character from the source (STR(A$,1)), then assigns it to the receiver (STR(A$,2)), then returns to the source for the next character, assigns it, and so on. Statement 10 makes the first character in A$ equal to hex 0A. The remaining characters we can assume to be spaces. Statement 20 takes the first character in A$, hex 0A, as specified by STR(A$,1) and assigns it to the second character position in A$, as specified by STR(A$,2). However, STR(A$,1) specifies the entire length of A$, so the processor continues the assignment by taking the next character out of STR(A$,1) and assigning it to the next character position in STR(A$,2). The character it takes out of the second character position in STR(A$,1) is the hex 0A that just a moment ago it put there. It takes this 0A and assigns it to the second character position in STR(A$,2). Now, A$ looks like this:

A$	0A	0A	0A	20	20	20	20	20	20	20	20	20	20	20	20	20

The processor continues taking a character out of one character position and assigning it to the next until STR(A$,2) is full.

16-5 THE LEN() FUNCTION

The LEN() function is used to determine the number of characters in an alphanumeric variable, excluding trailing spaces. For example, if

A$ =	A	B	C	D	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ

then LEN(A$) returns a value of 4, since there are 4 characters preceding the first trailing space. Though LEN() operates on an alphanumeric variable, that is, its argument is alphanumeric, it yields a numeric value and may be used anywhere that a numeric expression may be used. The following are examples of legal uses of the LEN() function:

```
100 IF LEN(A$) = 20 THEN 400
100 R = LEN(K2$(3))*INT(V)
100 PRINT TAB(32-LEN(A$)); A$
100 STR(B$,1,LEN(A$)) = A$
```

If the alphanumeric variable in the LEN() function is all spaces, LEN() returns a value of 1, not 0. This is in keeping with the discussion of Chapter 8 which pointed out that the value of an alphanumeric variable, which is all spaces, is 1 space.

In Section 16-3 it was pointed out that the STR() function causes the system to treat all the characters within the specified string as significant, even trailing spaces. Therefore, LEN() with a STR() as an argument, such as

```
L = LEN(STR(A$,1))
```

assigns to L the dimensioned length of A$, even if A$ is all spaces.

The LEN() function can be useful in a variety of programming situations. For example, it can be used in the TAB() parameter to produce right-aligned output. This use is shown in Example 16.1.

Example 16.1   Right-Aligning PRINT Output

```
110 REM RIGHT-ALIGNING PRINT OUTPUT
120     READ N
130     FOR J = 1 TO N
140         READ P$
150         PRINT TAB(20-LEN(P$)); P$
160         PRINT
170     NEXT J
990 DATA 5, "EMPLOYEE NAME", "JOB NO.", "JOB CATEGORY", "REGULAR
 HOURS", "OVERTIME HOURS"
:RUN
```

```
           EMPLOYEE NAME

                 JOB NO.

            JOB CATEGORY

           REGULAR HOURS

          OVERTIME HOURS
```

Notice that the output presents an even right edge with each line ending at column 20 (last character in column 19). It is easy to see why this works. When P$ is printed, PRINT outputs all characters up to a trailing space, in other words, as many as the LEN() of P$. Therefore, for each line, the total number of spaces from the TAB(), and characters from P$, is 20 - LEN(P$) + LEN(P$), or 20.

On an INPUT statement with an alphanumeric receiving variable, an operator can enter any number of characters, but if the number of characters entered exceeds the dimensioned size of the variable, the extra characters are lost, without the operator being alerted. This problem can be largely overcome by using a single 64 character alphanumeric variable to receive all alphanumeric input, and then testing the LEN() of the entry before assigning it to a shorter variable. A DEFFN' subroutine that takes this approach is shown in Example 16.2.

Example 16.2 Using LEN() to Test the Number of Characters in an Entry

```
110 REM AN ALPHANUMERIC ENTRY SUBROUTINE
115     DIM F$8, R$64, P$64
    .
    .
    .
320     GOSUB '186 ("ENTER FILE NAME (MAX. 8 CHARACTERS)",2,8)
330     F$ = R$ :REM ASSIGN RESPONSE
    .
    .
    .
990     DEFFN' 186 (P$, L1, L2)
910         PRINT P$;
920         R$ = " "
930         INPUT STR(R$,2)
940         IF LEN(R$)-1 < L1 THEN 960   :REM LESS THAN MIN.?
950         IF LEN(R$)-1 < = L2 THEN 980 :REM LESS THAN MAX.?
960         PRINT "INVALID. REENTER"
970         GOTO 910
980         R$ = STR(R$,2)
990     RETURN
```

The GOSUB' at line 320 passes a prompt and the minimum and maximum number of characters acceptable as a response. In this case the maximum, 8, is the dimensioned size of the variable F$ that ultimately receives the value. Line 920 is used to ensure that a previous entry will not be accepted as a new one if the operator merely keys (EXEC). The value is received by STR(R$,2) rather than R$, so that LEN(R$)=1 indicates unambiguously that no entry was made (otherwise an entry of one character would be indistinguishable from no entry, since LEN() returns a minimum value of 1). Line 980 eliminates this space before the subroutine returns.

16-6 CONVERTING ALPHANUMERIC VALUES TO NUMERIC VALUES, AND VICE VERSA

To facilitate the evaluation of numeric expressions, numeric quantities are contained in numeric variables in a unique format. This format is completely different from the simple hex codes used to represent alphanumeric characters. This is why alphanumeric values cannot be included in a numeric expression, and why numeric values cannot be directly assigned to an alphanumeric variable.

However, the BASIC statement CONVERT can be used to convert alphanumeric values to numeric values, and vice versa. For example,

```
10 A$ = "1200.50"
20 CONVERT A$ TO N
```

Line 10 assigns the literal string "1200.50" to A$. In A$ this literal string is represented simply as a series of characters followed by trailing spaces. Line 20 takes the characters in A$, converts them to a numeric quantity (in numeric format), and assigns this numeric quantity to the numeric variable N. A$ is unchanged. N contains the numeric quantity 1200.50, and can be used anywhere an expression is allowed.

The characters to be converted in a CONVERT statement must be an alphanumeric representation of a valid numeric quantity. This means that up to 13 digits, decimal point, sign, and a signed two digit exponent may be included in the alphanumeric value to be converted. In the above example, if A$ were assigned "12#87" instead of "1200.50", an error would result when CONVERT attempted to convert "12#87" to a numeric quantity, since "12#87" is not a valid representation of a number.

In many programming situations it is desirable to check whether an alphanumeric variable contains a valid representation of a number before attempting to convert it. This helps to avoid error interruptions during execution of CONVERT. The NUM() function can be used to facilitate such a test. NUM() is similar to LEN() in that it operates on an alphanumeric variable as its argument, but returns a numeric quantity. NUM() examines an alphanumeric variable, and counts characters until it finds one that would be illegal in a valid representation of a number, or it reaches the last character. It includes all spaces, even trailing spaces, in its count.

If A$ = | 1 | 2 | 0 | 0 | . | 5 | 6 | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ |

then

NUM (A$) returns 16, since 1200.56 is a valid representation of a number, and all the trailing spaces are counted.

If A$ = | 1 | 2 | 0 | 0 | X | A | B | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ | Δ |

then NUM(A$) returns 4, since the sequence of characters fails to conform to standard BASIC number format when the X is encountered.

209

To test whether an alphanumeric variable can be converted to numeric, a program can simply test if the NUM() of the variable is equal to the dimensioned length of the variable. A simple INPUT routine might look like this:

```
10 INPUT A$
20 IF NUM(A$) = 16 THEN 50 : REM NUMERIC FORM?
30 PRINT "NON-NUMERIC.  REENTER."
40 GOTO 10
50 CONVERT A$ TO X
```

With the ability to convert alphanumeric values to numeric ones, it is possible to write a single DEFFN' subroutine for alphanumeric and numeric entry. Receiving numeric entries as alphanumeric and then converting permits the program to prescribe the error procedure when a numeric entry fails to conform to numeric form. Example 16.3 shows such a subroutine.

Example 16.3 A General Purpose Input Subroutine

```
110 REM GENERAL PURPOSE INPUT SUBROUTINE
120     DIM P$64, R$64, T$1
130 DEFFN' 187 (P$, T$, L1, L2, L3)
140     PRINT P$
150     R$=" "
160     INPUT STR(R$,2)
170     IF T$ = "A" THEN 250 :REM ALPHANUMERIC ENTRY?
180 REM NUMERIC TESTS
190     IF NUM(R$)    64 THEN 280 :REM NON-NUMERIC?
200     CONVERT R$ TO R
210     IF R < L1 THEN 280 :REM TOO LOW?
220     IF R > L2 THEN 280 :REM TOO HIGH?
230     IF INT(R*10↑L3) = R*10↑L3 THEN 330: REM DECIMALS OK?
240 REM ALPHANUMERIC TESTS
250     IF LEN(R$)-1 < L1 THEN 280 :REM ENTRY TOO SHORT?
260     IF LEN(R$)-1 <= L2 THEN 320 :REM ENTRY SHORT ENOUGH?
270 REM DISPLAY ERROR MESSAGE
280     PRINT "INVALID. REENTER"
290     PRINT
300     GOTO 140
310 REM EXIT
320     R$ = STR(R$,2)
330     RETURN
```

The values passed to this subroutine are:

P$ = prompt, 64 characters maximum.

T$ = type of entry code: "A" = alphanumeric.
                          Any other value = numeric.

Numeric entry:

L1 = minimum acceptable value.
L2 = maximum acceptable value.
L3 = maximum number of digits right of decimal point.

Alphanumeric entry:

    L1 = minimum number of characters.
    L2 = maximum number of characters.
    L3 = not used.

Numeric values are returned in R, alphanumeric in R$. Maximum response = 63 characters.

Converting Numeric to Alphanumeric

The CONVERT statement can also be used to convert numeric values to alphanumeric values. However, when converting from numeric to alphanumeric, a question arises as to the form in which the numeric value is to be represented. For example,

    3207.4500
    3207.45
    3.20745E+03

all represent the same numeric quantity. Therefore, when converting numeric to alphanumeric, the programmer must specify an image for the converted value. The image is written directly into the CONVERT statement. For numeric to alphanumeric conversion the general form of CONVERT is

    CONVERT expression TO alphanumeric variable, (image)

where: (image) = ([+] [#...] [.] [#...] [↑↑↑↑]

    1 ≤ number of #'s ≤ 13

For example,

    10 N = 3.1416
    20 CONVERT N TO A$, (#.##)

This sequence assigns "3.14" to A$. The image in the CONVERT statement is similar, though not identical to, a format specification in an Image (%) statement. In executing the CONVERT statement, the system first evaluates the expression, then starts substituting digits from the result for the # signs in the image. Once this substitution is complete, the image, now with digits in place of # signs, is assigned to the specified alphanumeric variable.

The rules for construction of the image are given below. The principal differences between the image in the CONVERT statement and the PRINTUSING format specification are noted by asterisks.

In general there are two formats:

    Normal Format - e.g., ##.##
    Exponential Format - e.g., #.## ↑↑↑↑

1.   If the image starts with a plus sign (+), the sign of the value (+ or -) is substituted for the plus sign in the image.

2. If the image starts with a minus sign (-), a blank for positive values and a minus (-) for negative values is substituted for the minus sign in the image.

*3. If no sign is specified in the image, no sign is included in the character string.

4. If the image is Normal Format:

    a) The digits of the value are substituted for the # signs with the decimal point in the proper position.

    *b) If there are more # signs left of the decimal in the image than there are digits left of the decimal in the value, leading zeroes are substituted for the extra # signs. The sign, if present, appears to the left of the leading zeroes.

    *c) If there are fewer # signs left of the decimal than digits left of the decimal in the value, an error results.

    d) Extra # signs right of the decimal receive zeroes. Extra digits right of the decimal are truncated.

5. If the format is Exponential:

    The value is scaled as specified by the image, so there are no leading zeroes. The exponent is always substituted into the image in the form: E+XX.

CHAPTER 17
CONTROLLING A CRT

17-1 <u>CRT HEX CONTROL CODES</u>

If the cursor is positioned on the bottom line of the CRT when a CR/LF code is received, all the lines of the CRT are shifted up one line, and the top line is removed. This clears a new bottom line for output. This is called "rolling", and is an automatic function of the CRT. It is not under CPU control. The effect of rolling is that once a program fills the screen, all new lines appear at the bottom of the screen.

For some applications rolling is fine, and no programming steps need be taken to circumvent it. However, frequently it is better to control the line location of CRT output, and maintain a "steady-screen" display. This is more aesthetically pleasing, and results in superior operator/system interaction. For example, you might plan to have all input requests appear in the upper left corner of the CRT, and use mid-screen for display of recently entered items. To do this you must be able to control the cursor position (the location at which characters will appear), move it up, down, right and left, and be able to clear the screen when necessary. All of these things can be done with special hex control codes. These codes are sent to the CRT by the processor just as if they were characters, but instead of causing the CRT to display "A", for example, they cause it to clear the screen or move the cursor. The CRT cursor control codes are given in Table 17.1.

Table 17.1 The CRT Cursor Control Codes.

<u>HEX CODE</u>	<u>ACTION</u>
01	Moves cursor to top left corner of the CRT (home).
03	Clears screen and homes the cursor.
08	Backspaces cursor.
09	Non-destructive space right.
0A	Moves cursor down one line (line feed).
0C	Moves cursor up one line (reverse index).
0D	Moves cursor to leftmost position of the current line.

Hex 01

Hex 01 moves the cursor to the top left corner of the CRT. No characters are cleared from the screen. (This top left position is referred to as "home" or column 0 row 0.) To observe the effect of HEX(01), execute these two programs:

```
10 PRINT HEX(01);
20 GOTO 20
```

and

```
10 PRINT HEX(01); "OUTPUT ON LINE 0"
```

The first of these programs goes into an endless loop at line 20 so that the cursor can be observed; the second illustrates how a message might be printed on line 0 by using the HEX(01) code.

Hex 03

Hex 03 also homes the cursor, but first clears the entire screen. Clearing the screen has no effect on memory. Execute the statements shown above but with HEX(03) rather than HEX(01).

Hex 08

HEX(08) backspaces the cursor one character position. No character is erased. If the cursor is already in the leftmost position, column 0, it moves to the right end of the line. (The keyboard backspace key, when depressed during an INPUT instruction or when the colon is displayed, does not simply tell the CPU to output a HEX(08) because, when depressed, it erases the last character in addition to backspacing the cursor.)

Hex 09

HEX(09) moves the cursor right one position but does not erase any character. (The space character, HEX(20), which is the padding character for alphanumeric variables and the character input by the keyboard space bar, erases the character at the current cursor location before moving the cursor right one position.) HEX(09) is usually called the "non-destructive space".

To see the relationship between HEX(08), HEX(09) and HEX(20), execute the following program:

```
10 PRINT "ABCD"; HEX(080809);
20 GOTO 20
```

The display appears as

ABC<u>D</u>

The two 08 codes move the cursor back two spaces and then 09 moves it right one space for a net left movement of 1. No characters are erased. Now change the program to:

```
10 PRINT "ABCD"; HEX(080820);
20 GOTO 20
```

The display appears as

AB <u>D</u>

The cursor is in the same position but HEX(20), which was output when the display was

AB<u>C</u>D

erases the character at the current location before moving the cursor right one.

Hex 0A

HEX(0A) moves the cursor down one line from its current position. It remains in its previous column; no character is erased.

```
:PRINT "AB"; HEX(0A); "CD"
```

produces

```
AB
  CD
```

Hex 0C

HEX(0C) moves the cursor up one line from its current location. No characters are erased.

```
:PRINT "AB"; HEX(0C); "CD"
```

produces

```
    CD
  AB
```

Hex 0D

HEX(0D), the so-called "carriage return", moves the cursor to the leftmost position (column 0) on the current line. No characters are erased.

When the PRINT or PRINTUSING statements issue a CR/LF, the codes that are received by the CRT are HEX(0D0A). Also, the system automatically issues this HEX(0D0A) when a PRINT print element is about to overflow the current line, and under a variety of other conditions described in previous chapters.

17-2   THE LINE LENGTH CHARACTER COUNT

The 2200 System counts the number of characters it has output to a single line of the CRT. It uses this count to issue a carriage return/line feed when the maximum line length would be exceeded by the next character or next print element. The TAB() parameter also depends upon this count for its operation.

Whenever the system outputs a space (hex 20) or a character to the CRT, it updates its count by one. Whenever a hex 0D (carriage return) is output, the count is set back to zero. All of the other hex control codes have no effect upon the character count; they neither update it, nor reset it. This can occasionally be a source of bewilderment for the unwary programmer.

For example, if you execute the following statement,

    :PRINT TAB(62); HEX(03); "A"; "XYZ"

the result is:

        On line 0:   A
        On line 1:   XYZ

The 62 spaces output by TAB(62) update the internal character count to 62. HEX(03) clears the CRT and homes the cursor, but does not reset the character count. Therefore when "A" is output the system "thinks" its outputting it at column 63. Were this the case, "XYZ" would overflow the 64 character line length; therefore the system issues a CR/LF prior to outputting "XYZ". To correct a problem such as this, the statement can be changed to

        PRINT TAB(62); HEX(030D); "A"; "XYZ"

Here the added hex 0D has no effect on the cursor position. It simply resets the character count to zero.

The TAB() print element can be affected by the use of hex control codes. TAB() issues spaces until the character count equals the value within parentheses. Therefore, it will move the cursor to the correct column only if the character count accurately reflects the cursor position when the TAB() is executed.

17-3   USING THE CRT HEX CONTROL CODES

How can the cursor control codes be used to implement steady-display CRT usage? An example to consider is a rewritten version of Example 2.2, the first inventory program. Suppose we wish the inventory status to always be displayed on line 0, the input message to appear on line 1, and a reorder message, if any, to appear on line 3. The modification to Example 2.2 shown in Example 17.1 uses the CRT control codes to obtain this form of display.

Example 17.1   The Inventory Program (Example 2.2) Rewritten for a
               Steady Display

```
10   LET I=42500
20   PRINT HEX(03); "OPENING INVENTORY="; I
30   INPUT  "NUMBER OF TONS RECEIVED (+) OR SOLD (-)",  T
40   LET  I=I+T
50   PRINT  HEX(03); "TONS ON HAND =";  I
60   IF   I > = 100    THEN   30
70   PRINT  HEX(0A0A);"REORDER COAL IMMEDIATELY: INVENTORY BELOW
100 TONS"; HEX(01)
80   GOTO  30
```

At line 20 HEX(03) clears the screen and homes the cursor, before
displaying the opening inventory status report. After executing line 20, the
system issues a CR/LF. This moves the cursor to column 0 line 1. On line 1
the input prompt is displayed. Now, since the inventory status is always to
appear on line 0, the cursor must be homed before a new inventory status is
output. Furthermore, the old inventory status must be erased from the screen,
as well as the old operator entry. A HEX(03) at the beginning of line 50 does
this.   Since it clears the entire screen, it clears the prompt as well. This
is not a problem, however, since the prompt will be redisplayed when the INPUT
statement is executed.

Assuming there is no reorder message, the program simply loops from line
60 to 30. In a sense, we can think of this loop as "beginning" at line 50,
where the screen is cleared and the inventory status displayed. Then line 30
outputs the prompt on CRT line 1. When the operator makes an entry, the
process is repeated.

Now, suppose the inventory drops below 100 tons. This occurs at line
40.   Line 50 displays the new status on CRT line 0, and then drops the cursor
to line 1. If we inserted the statement 55 GOTO 55 to see the cursor location
and display, the display would look like this:

TONS ON HAND = 40

—

Line 60 does not effect a branch; line 70 is executed.   HEX(0A0A) drops the
cursor down two lines to line 3. Then, the reorder message is printed. Now
the cursor must be returned to line 1, column 0, so that the INPUT prompt will
be in its correct location. The easiest way to do this is by homing the
cursor HEX(01) and letting the normal CR/LF be issued to move the cursor to
line 1. Try running Example 17.1, and also try changing the various HEX codes
to see the results.

Status-Reports

When lengthy internal computations are taking place, or when external
files are being operated on without operator intervention, it is often a good
idea to maintain a system status message on the CRT. This message indicates
that the system is busy, and perhaps gives some idea of how long it will be
before the present operation is complete. Example 17.2 maintains a steady
display of a loop counter, during loop processing, and indicates the upper
bound of the loop as well.

217

Example 17.2 Maintaining a Steady Processing Message on the CRT

```
550 PRINT HEX(030A0A0A0A0A0A0A)
560 FOR I = 1 TO N
570     PRINT HEX(0C);TAB(18); "PROCESSING LOOP";I;"OF"; N
580 REM LOOP PROCESSING BEGINS HERE
    .
    .
    .
1050 NEXT I
```

In this figure, line 550 clears the screen and moves the cursor to line 8 (seven HEX(0A)'s plus the CR/LF at the end). Line 8 is one line below the line on which the display is to appear. Line 560 sets up the loop.   In 570, HEX(0C) moves the cursor up to the output line, TAB(18) centers the message, the message is output, and a CR/LF at the end moves the cursor to the leftmost position on the next line. This program can be executed, as shown, without any loop content, if a value for N is supplied.

Example 17.2 may seem unnecessarily elaborate. Why not simply backspace the cursor, and only output the print elements I; "OF"; N each time through? This solution is impossible because backspace, hex 08, does not decrease the character count, despite the fact that it moves the cursor left. Each time I; "OF"; N is output, the characters update the count. When the count is about to reach 64, the line length, the system issues a CR/LF that drops the output to the next line.

If a program calls for frequent cursor repositioning, it is convenient and efficient to include in the program a general cursor-positioning subroutine. Such a subroutine is shown in Example 17.3.   To use it, one simply writes a statement such as GOSUB'185 (7,45) which tells the subroutine to position the cursor at line 7, column 45.

Example 17.3 A Cursor-Positioning Subroutine

```
110 REM POSITION CURSOR SUBROUTINE
120 DIM C$64
130 DEFFN' 185 (R,C)
140     C$= HEX(01)            / 2/
150     INIT(09) STR(C$,2)  / 2 2
160     R$ = HEX(00)            / 2 3
170     INIT(0A) STR(R$,2)  / 2 4
180     PRINT HEX(0D); STR(C$,1,C+1); STR(R$,1,R+1);
190     RETURN
```

In this subroutine, C$ is dimensioned to a length of 64 characters. Line 140 assigns hex 01 to C$. After 140, therefore, C$ has hex 01 in the leftmost character position, followed by 63 trailing spaces.   Statement 150, changes all the trailing spaces to hex 09's. After 150, C$ looks like this:

C$ | 01 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | 09 | ... ... | 09 | 09 | 09 |

one hex 01

63 hex 09's
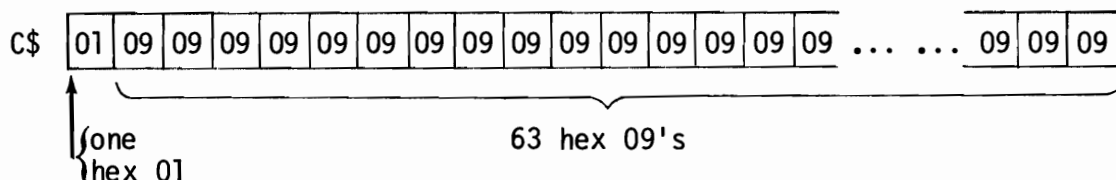
Lines 160 and 170 go through a similar procedure with R$, leaving it as follows:

```
R$ | 00 | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A
```
one hex 00

15 hex 0A's
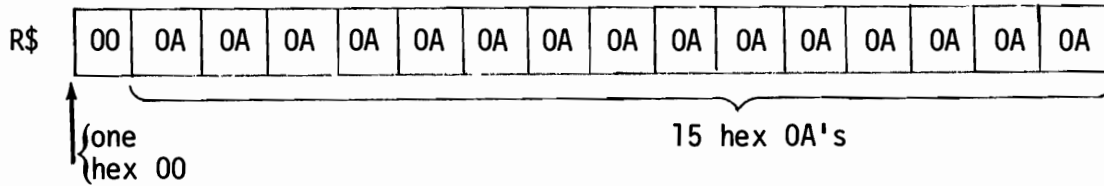
Lines 140 to 170 are simply a preparation for moving the cursor. Their only function is to put the needed hex codes into C$ and R$. The subroutine could be rewritten so that this operation was not repeated on each execution.

Line 180 positions the cursor to the selected column and row. The first print element HEX(0D) moves the cursor to the leftmost position of whatever line it happens to be on. However, its main function here is to reset the internal character count so that we can be sure an automatic CR/LF will not be issued during the cursor positioning. The second print element STR(C$,1,C+1) starts outputting characters from C$ beginning with the first character. The first character is a hex 01, which moves the cursor to home position (row 0, column 0). If the value of C, which was passed to the subroutine, was 0 for column 0, STR(C$,1,C+1) is equivalent to STR(C$,1,1), which specifies just one character, beginning in the first character position. If the value of C is 45, the first 46 characters of C$ are specified for output. These 46 characters consist of one hex 01 followed by 45 hex 09's. The 45 hex 09's space the cursor to column 45. Now the cursor is in the correct column but on row 0. STR(R$,1,R+1) moves it to the correct row. The first character in R$, which is the only character output if R is 0, is a hex 00. Hex 00 does absolutely nothing; the cursor isn't moved and no characters are erased. This "do nothing" is exactly what we want if the desired row is row 0. However, if the desired row is 7, STR(R$,1,R+1) outputs 8 characters: a "do nothing" followed by 7 line feeds, hex 0A's, to move the cursor to its desired location.

Clearing Selected CRT Lines

We have seen that the hex 03 code clears the entire CRT screen. Sometimes, though, you may wish to clear just one line of the CRT. The easiest way to clear a single line of the CRT is to move the cursor to column 0, and then TAB(64), assuming that the line length is 64. The TAB() outputs spaces, thereby clearing the line.

219

CHAPTER 18
CONTROLLING A PRINTER

There are two fundamentally different kinds of printers available with Wang 2200 systems. There are the matrix printers, Models 2221W, 2231W, 2221, 2231, and 2261, and the character printer, the 2201 Output Writer. Users of matrix printers should read Section 18-1 on the Model 2221W, in which the minor differences of the other models have been noted. Section 18-2 is devoted to the 2201 Output Writer.

18-1 HEX CONTROL CODES FOR THE 2221W PRINTER

When the 2221W printer receives a hex code for a printable character, it simply puts the code into its print buffer. Unless the buffer becomes full, no immediate action is taken. However, certain special hex codes are not entered into the buffer, but rather cause immediate action by the printer. These special codes are the printer control codes.

The printer control codes for the 2221W are:

HEX(0D)   PRINT BUFFER CONTENTS: The buffer is printed with line feed and carriage return generated automatically. The buffer is cleared after printing, and the processor's internal character count is reset.

HEX(0A)   LINE FEED: Advances paper one line.

HEX(0E)   EXPANDED PRINT: Causes the printer to print the first 66 characters in the print buffer in an expanded format, when the next HEX(0D) is received.

HEX(07)   BELL: Generates a two second audible tone.

HEX(0C)   FORM FEED: Advances paper until the next hole in channel 7 of the forms tape is reached.

HEX(0B)   VERTICAL TAB: Advances paper until the next hole in channel 5 of the forms tape is reached.

HEX(7F)   Clears current buffer contents.*

*Models 2221W and 2231W only.

Hex 0D

The hex 0D code directs the printer to print the characters in its buffer. After printing the buffer contents, the printer automatically feeds the paper up one line, and returns the print head to the left of the print well.

The processor issues a hex 0D code whenever a PRINT or PRINTUSING statement ends without a comma or semicolon. Therefore, it is never necessary to use the HEX function to issue a hex 0D code. In fact, because your Wang system reduces all keywords to a one character code, a statement such as

        100 PRINT "ABC CO."; HEX(0D);"15 BEACON ST."; HEX(0D); "BOSTON,MA"

occupies less memory if written as a multistatement line with the HEX functions omitted:

        100 PRINT "ABC CO.":PRINT "15 BEACON ST.":PRINT "BOSTON,MA"

Hex 0A

The HEX(0A) code causes the printer to feed the paper up one line. Buffer contents are not affected. It is important to realize that for all control codes the printer acts immediately upon receipt of the control code, and does not await a hex 0D. Example 18.1 shows a program which uses hex 0A's and the resultant output.

        Example 18.1   Illustration of the Fact That Control Codes Are Output
                       Immediately

        110 SELECT PRINT 215 (132)
        120 PRINT "LINE ONE"
        130 PRINT "ABC CO."; HEX(0A0A); "BOSTON,MA"
        :RUN
        LINE ONE

        ABC CO. BOSTON, MA

Despite the fact that two hex 0A's are embedded in the PRINT statement at line 130, they are not acted upon by the printer after "ABC CO." is printed and before "BOSTON, MA" is printed. In this program the system first executes line 120 which prints LINE ONE for reference. At line 130 the processor outputs the characters ABC CO. to the printer. The printer senses that these are printable characters, not control codes, and puts them into the buffer. The processor now sends the two control codes 0A0A to the printer. The printer checks the first 0A, finds it's a control code, and, therefore, immediately executes it, feeding the paper up one line. The process is repeated for the second 0A. The 0A's are not put into the buffer. Now the processor sends the characters BOSTON, MA to the printer. The printer puts them into the buffer. Finally the processor issues a hex 0D code, since the PRINT statement ends without a comma or semicolon. The printer, on receipt of the hex 0D code, prints the buffer contents. The results, as shown, is "LINE ONE", followed by two line feeds, followed by "ABC CO. BOSTON, MA".

Hex OE

A hex OE code tells the printer that the next time it prints the buffer contents, it should print it in expanded form. Expanded characters take twice the space of normal characters; therefore, only the first 66 characters in the buffer can be printed. Any additional ones are simply lost. A hex OE code does not itself initiate printing. It simply specifies that, when a hex OD is received, characters should be printed in their expanded form. A hex OE can be sent to the printer before or after the characters to be printed are put into the buffer. Once the buffer is printed, after a hex OE, the printer automatically reverts to normal print form, unless another hex OE arrives prior to the next printing of the buffer.

Expanded print can be used for for producing striking report titles, total lines, etcetera. Example 18.2 shows the use of expanded print.

Example 18.2 Expanded Print

```
110 REM USING EXPANDED PRINT
260 REM PRINT P&L
270     SELECT PRINT 215(80)
280     PRINT HEX(OE);"PROFIT AND LOSS STATEMENT"
```

Output from line 280 appears as:

PROFIT AND LOSS STATEMENT

Figure 18.1 Sample of Expanded Print

When using expanded print, remember that each printed character takes up twice the normal paper space, and that, therefore, a line length specification in the SELECT statement does not offer the usual protection against printing beyond the right end of the paper.

Hex 07

The hex 07 code, when received by the printer, generates a two second audible tone. The tone is output immediately upon receipt of the hex 07 code; the code does not become part of the buffer. A continuous tone can be generated by putting PRINT HEX(07) within a loop. For example,

```
110 SELECT PRINT 215
120 FOR I=1 TO 6
130 PRINT HEX(07)
140 NEXT I
```

This generates a tone of approximately 12 seconds duration. A pulsing tone can be produced by putting a "do-nothing" FOR...TO/NEXT loop within the loop shown above. For example, you could add the lines

```
133 FOR K=1 TO 500
134 NEXT K
```

to the above example. This loop simply consumes time, to make the tone pulsate.

Forms Control with Hex 0B and Hex 0C

The vertical format tape loop, located under the printer's left cover, can be used to control paper advance. Each time the paper is advanced one line, the vertical format tape moves the distance of one sprocket hole past the tape-reading photo-cells. This tape movement occurs as a result of a mechanical linkage between the paper advance mechanism and the tape sprocket wheel.

The vertical format tape can have holes punched in any of three "channels" as shown below:
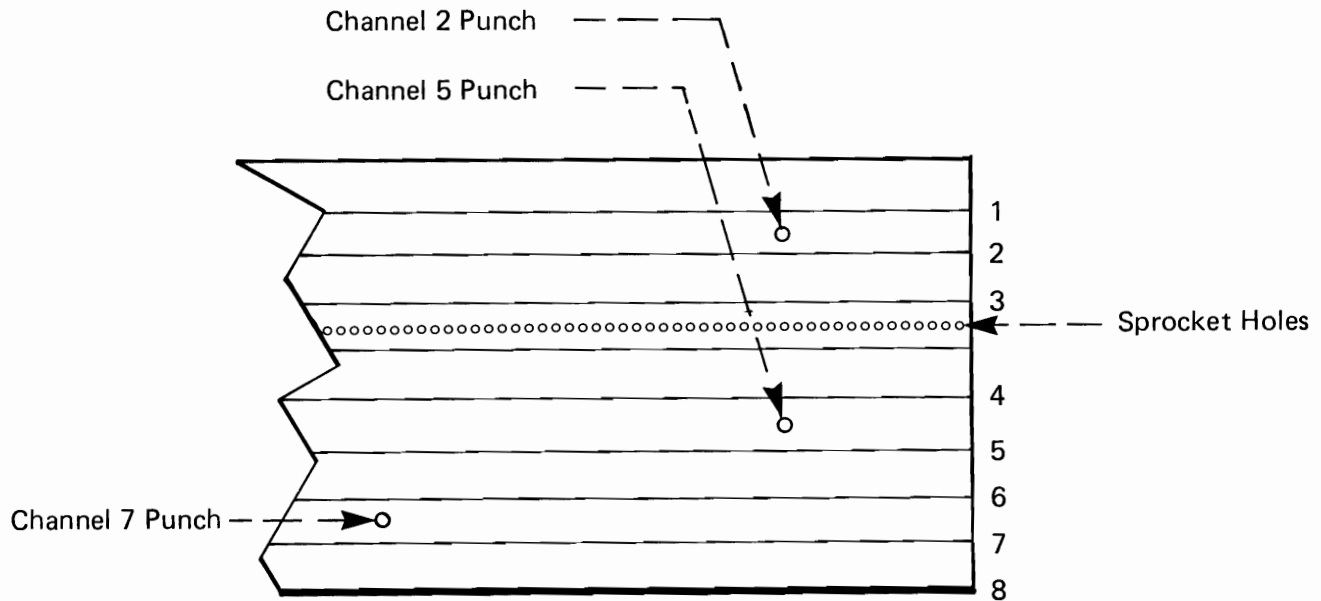
Figure 18.2   A Section of a Vertical Format Tape (Enlarged)

Generally a tape is prepared so that the distance between Channel 7 punches, measured by the number of sprocket holes, is the same as the overall length of the paper form being used, measured in the number of lines that can be output on it. Standard 11" report-paper is 66 lines long; therefore the distance between Channel 7 punches on the supplied standard vertical format tape is 66 sprocket holes. When the printer receives a hex 0C code, it rapidly advances the paper until the vertical format tape reader senses a hole in Channel 7 of the tape. Provided that the paper was properly aligned in the first place, this advances the paper to the top of the next form. The hex 0C code is called a "form feed", by virtue of this function.

On special forms such as invoices, checks, account statements, etcetera, a vertical format tape is prepared that has punches in Channel 5, spaced to align with the major divisions on the form. For example, an invoice tape might have Channel 5 punches that align with the first line of the "ship to" address, the "sold to" address, the body of the invoice and the "total" line. If the "ship to" address is the top of the form, it might be punched in Channel 2 as well as Channel 5. A hex 0B, when received by the printer,

223

advances the paper until the vertical format tape reader senses a hole in Channel 5 of the tape.

Form advance using the vertical format tape is considerably faster than that which can be achieved by issuing line feeds (hex OA's). In addition, it causes much less wear on printer mechanisms than repeated hex OA's, and, therefore, should be used for all repetitive multi-line form-up. Example 18.3 is a skeleton of a simple invoicing program designed to show the use of hex OC and OB codes for paper advance. It presumes that a properly prepared tape has been mounted in the vertical format tape reader.

Example 18.3 Using Vertical Tabs in an Invoice Program

```
110 REM USING VERTICAL TABS, HEX(OB), TO FORM-UP AN INVOICE
120 SELECT PRINT 215
130 PRINT HEX(OC)
140 INPUT "ALIGN FORM BELOW PERF. KEY (EXEC) TO RESUME",A9$
 .
 .
 .
200 REM PRINT "SOLD TO" ADDRESS
210 PRINT HEX(OB)   :REM FORM UP TO FIRST "SOLD TO" LINE
 .
 .
 .
260 REM PRINT "SHIP TO" ADDRESS, IF DIFFERENT
270 PRINT HEX(OB)   :REM FORM UP TO FIRST "SHIP TO" LINE 1
 .
 .
 .
320 REM BODY OF INVOICE BEGINS
330 PRINT HEX(OB)   :REM FORM UP TO FIRST ITEM LINE
335 L=0             :REM RESET LINE COUNT
340 L=L+1           :REM INCREMENT LINE COUNT
350 IF L = 17 THEN 1400   :REM INVOICE OVERFLOW?
 .
 .
 .
900 INPUT "MORE INVOICE LINES? (Y OR N)", A9$
910 IF A9$="Y" THEN 340
920 IF A9$<>"N" THEN 900
930 REM NO MORE INVOICE LINES
940 PRINT HEX(OB)  :REM FORM UP TO "TOTAL" LINE
950 REM CALCULATE DISCOUNTS AND PRINT TOTAL LINE
 .
 .
 .
1100 INPUT "MORE INVOICES? (Y OR N)", A9$
1110 IF A9$= "Y" THEN 200
1120 IF A9$<> "N" THEN 1100
1130 REM PRINT COMPANY TOTALS
 .
 .
 .
1380 END
1400 REM INVOICE OVERFLOW ROUTINE
```

At line 130 a HEX(0C) is issued to advance the form and the vertical format tape to the top-of-form position (Channel 7 hole). The operator should then check that the form is properly positioned, and adjust it if necessary. Often it is a good idea to have the program print a blank form, with all fields filled with # symbols, to let the operator exactly position the form for vertical as well as horizontal alignment. (Assuming that the PRINTUSING statement is employed for printing on the form, this can be accomplished by attempting to print 1E99 to all the format specifications. 1E99 is too large a number for any format specification, other than exponential, and will therefore cause the Image statement # signs to be output.)

The invoices used by Example 18.3 have four major divisions: sold to address, ship to address, body of the invoice, and total line. At the beginning of the program routines that operate on each section, a HEX(0B) is printed to advance the form to the proper line. The number of lines used by each of the sections, except the body of the invoice, is fairly standard, so line counts are unnecessary. However, as the number of invoice items is indeterminate, a line count of the body of the invoice must be maintained. This line count takes place at lines 340 and 350. If a 17th line is to be added to the invoice, it must appear on the next form; a branch to an invoice overflow routine is provided.

On the 2221W printer, a punch in Channel 2 of the vertical format tape carries special significance. Unlike Channel 5 and 7 punches, which are used only to terminate a form advance, Channel 2 punches initiate a form advance. When the printer senses a hole in Channel 2 of the tape, it automatically initiates a form-up that ends when a Channel 7 hole is sensed. This feature is known as "automatic page eject", and, with a properly punched tape, can eliminate the need for maintaining a line count within a program. For example, standard 11" report paper is 66 lines long. Allowing 3 lines for top and bottom borders leaves 60 print lines. The standard vertical format tape supplied with the printer contains a Channel 2 hole sixty sprocket holes (60 lines) after an "initial" Channel 7 hole. Therefore, if the paper is aligned after form-up to 3 lines below the paper perforation, then 60 lines later an automatic form up will take place, triggered by the Channel 2 hole. Six lines after this Channel 2 hole is the next Channel 7 hole which stops the form-up at the top of the next sheet. This system automatically prevents printing on a perforation.

If a program is being run that contains an internal line count to trigger form-up, then channel 2 holes in the Vertical Format Tape should be covered with an opaque material to prevent undesired automatic form-ups.

Hex 7F

A hex 7F code clears the print buffer on model 2221W and 2231W printers. Nothing is printed, and no forms movement occurs. Whatever characters are in the buffer, at the time the 7F is received, are simply lost. The hex 7F code has no effect on the 2221, 2231, or 2261 series printers.

Outputting a hex 7F at the beginning of a program is often a worthwhile precaution. The previous program may have left characters in the print buffer which would otherwise spoil the first print output.

18-2   HEX CONTROL CODES FOR THE 2201 OUTPUT WRITER

The hex control codes for the 2201 Output Writer are:

HEX (0A)   LINE FEED: Advances paper one line.   Print assembly
                does not move.

HEX (0D)   CARRIAGE RETURN/LINE FEED: Moves the print assembly to
                the left margin, advances paper one line, and resets the
                processor's internal character count.

HEX (09)   TAB: Moves the print assembly to the right until a
                mechanical tab-stop is reached.

HEX (1A)   SET TAB: Sets a mechanical tab-stop at the current
                print position.

HEX (19)   CLEAR TAB: Clears a mechanical tab-stop at the current
                print position.

HEX (08)   BACKSPACE: Backspaces the print assembly one character
                position.

HEX (5F)   UNDERSCORE: Prints an underscore mark at the current
                print position.

Hex 0A

The hex 0A code causes the platen to advance one line.   (The platen
will advance two lines if the manual single-space/double-space lever on the
right side of the Output Writer is set to double-space.) The hex 0A code
does not affect the print assembly and, therefore, should be used instead
of hex 0D whenever multi-line form-up is desired. For printing that requires
frequent form feeding, it may be desirable to initialize a form-feed
alphanumeric variable as follows:

        10 DIM L9$64
        20 INIT (0A)L9$:REM L9$ IS FILLED WITH 0A's

With this variable, L9$, initialized as above, a carriage return and six line
feeds can be accomplished with:

        PRINT STR(L9$,1,5)

Here five 0A's are output followed by the automatic carriage return/line feed.
Twenty line feeds without a carriage return can be accomplished with:

        PRINT STR(L9$,1,20);

The semicolon suppresses the automatic line feed/carriage return.

Example 18.4 shows a segment of an invoice preparation program that uses
the 2201 Output Writer.

Example 18.4 Invoice Forms Control on the 2201

```
110 REM PROGRAM SEGMENT SHOWING FORM CONTROL ON 2201
115 REM INITIALIZE FORM-UP VARIABLE
120     DIM L9$29
140     INIT(0A) L9$
 .
 .
 .
560 REM FORM UP TO BODY OF INVOICE AND START LINE COUNT
565     PRINT STR(L9$,1,3)  :REM 4 LINES UP
570     L=0
580     L=L+1     :REM NEXT LINE, BRANCH BACK TO HERE
590     IF L = 25 THEN 2010    :REM LINE OVERFLOW?
600 REM OUTPUT ONE LINE OF INVOICE
 .
 .
 .
710     INPUT "MORE INVOICE LINES (Y OR N)", R$
720     IF R$ = "Y" THEN 580
730 REM FORM UP TO "TOTAL" LINE FROM LINE L
740     PRINT STR(L9$,1,25-L)
750 REM OUTPUT INVOICE TOTAL LINE

2000 REM LINE OVERFLOW ON BODY OF INVOICE
2010     PRINT L9$   :REM 30 LINES TO BODY OF NEXT FORM
2020     GOTO 570
```

Line 130 initializes L9$ to 29 hex 0A's. This is the maximum number of 0A's which ever need to be output in this program. Between lines 140 and 560 the "Sold To", "Ship To" and "Salesman" lines are printed. Line 565 advances the form to the first invoice line. Line 570 resets the line count variable, L. Line 580 increments the line count by 1. There is room for 24 lines in the body of this invoice. Line 590 tests if the line about to be output is the 25th line. If it is, 590 effects a branch to a routine which advances the form to the body of the next invoice for continuation of output. Lines 600-710 are used to construct and output one line of the invoice. At line 720, if there are more lines to be entered, R$ equals "Y" and a branch is effected back to line 580. If there are no more lines to be entered, the "total" line must be printed. Counting the 1st line in the body of the invoice as 1, the "total" line is the 26th line, and, therefore, can be reached by outputting 25 line feeds from line 1. If L is 1, line 740 outputs the first 24 characters of L9$, followed by the automatic CR/LF for a total of 25 line feeds. Similarly if L is any number in its range of 1-24, line 740 outputs the correct number of lines to reach the "total" line.

Hex 0D

In general it should not be necessary to explicitly include a HEX(0D) in a program for the 2201, since this code can always be generated by simply ending a PRINT statement without a comma or semicolon.

Hex 09 and Hex 1A

The hex 09 code causes the 2201 to rapidly move the print assembly to the right until a mechanical tab stop is reached. This is the equivalent of depressing the tab key on the 2201 when using it in manual mode. In order for the hex 09 code to be useful, mechanical tab stops must previously have been set at the desired positions. They may be set manually, or under program control by issuing the hex 1A code.

The TAB() print element is not related to this mechanical tab capability of the 2201. TAB() print elements cause spaces to be output until a desired column location is reached. Outputting spaces in this fashion does not cause the print assembly to move as rapidly as it moves when executing a mechanical tab, but, on the other hand, it does not require that a mechanical tab be preset. If a program requires a substantial amount of tabbing by the 2201, it may be faster to have the program set mechanical tabs, and then move from tab stop to tab stop by issuing hex 09's.

It should be noted that when a mechanical tab is executed, the system "loses track of" the position of the print assembly. That is, the internal character count, which is used by TAB() and causes the end-of-line automatic CR/LF, is not properly updated, and therefore, TAB() cannot be used together with the mechanical tab feature of the 2201.

For more information on setting and clearing mechanical tabs, see the 2201 Output Writer Reference Manual (#700-3113A).

Hex 08 and Hex 5F

The hex 08 code backspaces the print assembly one character position. It does not update the internal character count, and, therefore, effectively disables the TAB() print element. It can be used with the underscore mark, hex 5F, to underline characters. The program shown in Example 18.5 illustrates a simple underlining subroutine that first backspaces past a specified number of characters and then underlines them, leaving the print assembly in its original position.

Example 18.5 An Underlining Subroutine for the 2201

```
110 REM ILLUSTRATION OF UNDERLINE SUBROUTINE FOR 2201
120 SELECT PRINT 211
130 PRINT "UNDERLINE THIS";
140 GOSUB '202 (14)
150 STOP
2000 REM UNDERLINE SUBROUTINE FOR 2201
2010 DEFFN' 202 (N) :REM N = NUMBER OF CHARACTERS TO UNDERLINE
2020 FOR I = 1 TO N
2030     PRINT HEX(08);
2040 NEXT I
2050 FOR I = 1 TO N
2060     PRINT HEX(5F);
2070 NEXT I
2080 RETURN
```

CHAPTER 19
TABLES (TWO-DIMENSIONAL ARRAYS)

19-1 <u>INTRODUCING TWO-DIMENSIONAL ARRAYS</u>

Thus far we have seen two ways of referring to variables. Ordinary (or scalar) variables each have a fixed name assigned to them, for example, A, C2, G8$, N$, X. In Chapter 11 we introduced list variables. With list variables, a name, such as D3(), is assigned to an entire list of variables. Each variable on the list is referred to by giving the list name and a single expression in parentheses. The expression gives the position of the desired variable in the list. A reference to a list variable might be A2(3), which specifies the third variable down the list A2(). Assuming that it has four elements, the variables on the list A2() can be named by:

       A2(1)
       A2(2)
       A2(3)
       A2(4)

In many instances list variables make simple and compact operations out of ones that would be cumbersome using ordinary scalar variables. However, sometimes a programming problem suggests that variables be arranged in a kind of table. In a table arrangement of variables, a particular variable is specified by means of two expressions that together give the variable's position in the table. For example, a table of variables called N() is shown below.

A Table of Variables Called N()

row 1	N(1,1)	N(1,2)	N(1,3)
row 2	N(2,1)	N(2,2)	N(2,3)
row 3	N(3,1)	N(3,2)	N(3,3)
row 4	N(4,1)	N(4,2)	N(4,3)
	column 1	column 2	column 3

The table N() consists of 12 variables, or elements, each named by giving the table name together with the variable's row and column position. The table N() has four rows and three columns. Thus, N(1,1) names the variable at row 1 column 1; N(3,2) names the variable at row 3, column 2; N(4,3) names the variable at row 4, column 3. Numbering of the rows and columns of a table always starts with one, never zero.

229

In Section 11-3 we mentioned that list variables are often called one-dimensional arrays, since any variable on the list can be referred to by means of a single value in parentheses. To specify a variable within a table requires two expressions; therefore, variables arranged in table form are said to be in a two-dimensional array.

Just as for a list, any expressions can be used to give the position of a variable in a two-dimensional array. For example, in the array N(), above, the variable in row 3, column 2 can be referred to not only by

        N(3,2)

but also by

        N(#PI,(6↑2/18))

since the integer value of the expressions #PI, and 6↑ 2/18 is 3 and 2 respectively.

Establishing a Two-Dimensional Array

In order to use a two-dimensional array of variables, you must specify in a DIM statement a name for the array and the number of rows and columns it is to have. In BASIC the array names that are used for one-dimensional arrays are also used for two-dimensional arrays. Thus, the names we gave in Chapter 11 as being legal for naming lists, A()...Z() and A0(), A1(), A2()...A9(), B0(), B1()...Z7(), Z8(), Z9(), are also legal for naming two-dimensional arrays. The dimension statement for the two-dimensional array N(), shown above, would be

        DIM N(4,3)

where N() gives the name of the array, and the integers 4 and 3 give the number of rows and columns the array is to have. The total number of variables in this array is 12 (or 4 times 3). In the DIM statement the row and column size of the array must be given by integers; an expression may not be used. The maximum number of rows or columns in any array is 255; however, the total number of elements may not exceed 4096. Therefore, a DIM statement such as

        DIM K(8,255)

is legal whereas

        DIM(65,65)

is not, since a 65 by 65 array would contain more than 4096 elements.

A single DIM statement may be used to specify any number of one and two-dimensional arrays, as well as giving the size of alphanumeric scalar variables. For example,

        40 DIM A$30, B(12), R(5,5), B$(4)2

dimensions the scalar alphanumeric variable A$ to a length of 30 characters, establishes the list of 12 numeric variables B(), the numeric two-dimensional array R() with 25 variables arranged in 5 rows and 5 columns, and finally the alphanumeric list B$() with 4 variables each 2 characters long. The DIM statement for an array must appear at a lower line number than any reference to a variable in the array.

In addition to two-dimensional arrays of numeric variables, BASIC also permits two-dimensional arrays of alphanumeric variables. For example,

DIM R5$(10,12)

sets up an array of alphanumeric variables with 10 rows and 12 columns, a total of 120 variables. Since a character length for the variables is not specified, each variable in this array would be given a length of 16 characters. It is possible, though, to specify a different variable length by simply adding a length specification, 1-64, outside the parentheses. For example,

DIM K$(10,12)20

sets up an array of alphanumeric variables, each of which has a maximum length of 20 characters.

It is not possible to use the same array name for a one-dimensional array and for a two-dimensional array in the same program. Therefore, this DIM statement produces an error:

DIM K(4,7), K(9)

However, a numeric array and an alphanumeric array may have the same name, except for the $. For example, this DIM statement is legal

20 DIM R2(10,10), R2$(10,10)

Memory Usage

The total memory space required for an array, plus the memory space occupied by the program statements and other variables, cannot exceed the amount of memory available. The total amount of memory available can be determined by executing an END statement in the Immediate Mode, after the system memory has been cleared. (It is approximately 700 bytes less than the the absolute amount, i.e., 4K, 8K, 16K, etc.)

In Chapter 9 it was mentioned that numeric variables occupy 8 bytes of memory, and that, in addition, 5 bytes are used for the control information that enables the processor to find the variable. Alphanumeric variables may occupy from 1 to 64 bytes, and also require 5 bytes of control information. The memory space, M, in bytes required for an array is given by

$$M = Nl + 7$$

where   n = the total number of variables or elements.
                 L = 8 if numeric array,
                    or
               = dimensioned length of each alphanumeric array element (1-64).

231

and 7 is the seven bytes of control information required for an array.

For example, a 20 by 20 numeric array dimensioned as follows

      DIM E2(20,20)

has 20*20 = 400 elements. Thus,

      M = 400*8+7 = 3207 bytes.

An alphanumeric array dimensioned as

      DIM (13,13)30

has 169 elements; thus,

      M = 169*30+7 = 5072 bytes.

     Remember that only if the RUN command has been executed does END take into account the space occupied by variables and arrays.

19-2 USING TWO-DIMENSIONAL ARRAYS

     A simple example of the use of a table, or two-dimensional array, in commercial data processing is a withholding tax routine in a payroll program. For example, a state supplied withholding tax table might look something like this:

		Number of Dependents							
		1	2	3	4	5	6	7	8 or more
Lowest Income	1	.005	.003	.001	0	0	0	0	0
	2	.008	.005	.0025	.001	0	0	0	0
	3	.015	.009	.004	.003	.0025	0	0	0
	4	.015	.010	.009	.007	.005	.0035	.0025	.001
Income Category	5	.0175	.011	.010	.009	.009	.007	.004	.003
	6	.02	.018	.0105	.0105	.010	.010	.009	.008
	7	.028	.021	.019	.017	.014	.013	.012	.011
	8	.03	.029	.02	.019	.018	.016	.015	.015
Highest Income	9	.04	.04	.034	.030	.029	.025	.025	.022

The proper withholding tax percentage can be found at the location specified by a given number of dependents and a given income category. The routine to calculate withholding tax simply uses a table of variables dimensioned to accommodate this tax table supplied by the state. For example, the program might include a dimension statement such as

        120 DIM T(9,8)

which defines T() as a table with 9 rows and 8 columns. Each variable in the table must then be assigned the proper percentage as specified by the state. To "look up" the percentage in the table, the program simply specifies the correct variable with

        T(I,D)

where I and D give the row and column (income and dependents).

We must assume that the state supplies an annual gross wage cut-off point for each income category. For example,

PROJECTED ANNUAL GROSS WAGE IS UNDER	CATEGORY IS
5100	1
6200	2
7600	3
8400	4
12300	5
14900	6
17100	7
21600	8
21600 and over	9

Prior to accessing the tax table, the program must determine a person's income category. Here, a list containing the cut-off points can be searched, comparing the cut-off point with the person's projected annual wage. Example 19.1 does this, and then calculates the tax to be withheld.

Example 19.1 Determining Tax Bracket and Tax Using a List and a Table

```
110 REM CALCULATING TAXES BY ACCESSING A TABLE
120 DIM T(9,8), C(8)
 .
 .
 .
450 REM D CONTAINS NUMBER OF DEDUCTIONS
460 REM CALCULATE STATE TAX
470 REM FIND TAX BRACKET NUMBER BY SEARCHING CUT-OFF LIST C(),
480     K = 0
490     K = K+1
500         IF K = 9 THEN 520      :REM TOP BRACKET?
510     IF A >= C(K) THEN 490      :REM HIGHER BRACKET THAN K?
520 REM K NOW HAS CORRECT TAX BRACKET
530 REM T(K,D) IS WITHHOLDING PERCENTAGE
540     S = W * T(K,D)
550 REM S NOW HAS STATE TAX AMOUNT
560 REM
```

233

At line 120 a tax table T() and a cut-off list C() are dimensioned. Between 120 and 450 we assume that the variables in the cut-off list and the tax table have received the fixed values supplied by the state and that variable A is assigned the projected annual wage for the individual being processed, D is assigned a number of dependents (1-8), and W is the week's gross wages.

Lines 490 to 510 form a loop in which the annual wage is compared to each of the values in the cut-off list until the cut-off value C(K) is greater than the annual wage or the highest bracket, 9, is reached. After 510, the search is complete, and K contains the tax bracket (income category). Now the state tax to be withheld can be calculated by simply multiplying the variable W times the variable T(K,D), since K and D specify by row and column the variable containing the proper percentage. This calculation is performed at line 540.

We have passed over one major point in this discussion, how the values are to be assigned to the variables in the tax table and cut-off list. Normally this would be accomplished by loading the values into these arrays from data files saved on tape or disk. Data files are discussed in Chapters 20 and 21.

In many situations, processing the elements of a two-dimensional array can be accomplished with nested FOR...TO/NEXT loops. For example, to print the value of each of the elements of a 5 by 5 array, D(), the following routine can be used.

Example 19.2 Nested Loops Used to Process a Two-Dimensional Array

```
110 DIM D(5,5)
    .
    .
    .
220     FOR R = 1 TO 5
230         FOR C = 1 TO 5
240             PRINT D(R,C);
250         NEXT C
260         PRINT
270     NEXT R
```

In this program all the elements in a single row are printed by the inner loop 130-150; then a carriage return line feed is issued (line 160). Then, NEXT R, in the outer loop, increments the row counter R, so that the next row will be output when the inner loop executes again.

Instead of printing in such a loop, a READ, INPUT or assignment statement could be used to assign values to each element. For example,

```
240     D(R,C) = D(R,C)*5
```

would multiply the value of each variable by 5.

Technical Applications

Outside of strictly commercial processing, two-dimensional arrays of variables introduce particularly powerful programming simplifications for computational problems in linear algebra. In programming applications of this variety, the terminology of linear algebra supplants that which we have been using. A table or two-dimensional array is called a "matrix", and a list or one-dimensional array is called a "column vector".

Historically, the first problem of linear algebra is the solution of a set of n linear equations in n unknowns. Because computational problems from many fields of mathematics can often be reduced to problems of this and related types, research into efficient computational techniques for these problems is ongoing, and a substantial technical literature has been generated. However, for the most common problems of matrix algebra, a set of special BASIC statements is available that permits operations on entire matrices to be performed with a single statement. These statements are discussed briefly in the next section, and in more detail in the reference manual Matrix Statements (#700-3332B).

The special matrix statements of BASIC are not part of the standard instruction set of the 2200S processor. They are, however, available as an optional addition to the 2200S.

19-3 THE MATRIX STATEMENTS

BASIC includes a special set of 14 instructions which are designed to facilitate matrix operations. Each of the instructions in this set operates on an entire array of variables, rather than on individual variables within an array. These statements are collectively known as the Matrix Statements, and all of them begin with the keyword "MAT". Their operations are summarized in Table 19.1.

TABLE 19.1				MATRIX STATEMENT OPERATIONS	
Operation	**E**	**I**	**A/N**	**Description**	**Example**
Matrix Addition		√		array = array + array	MAT X = Y + Z
Matrix Subtraction		√		array = array – array	MAT X = Y – Z
Matrix Multiplication		√		array = array * array	MAT X = Y * Z
Scalar Multiplication		√		array = scalar expression * array	MAT X = (3) * Y
Matrix Inversion and Determinant		√		matrix = inverse matrix and scalar variable = determinant of matrix	MAT X = INV(Y), D
Matrix Transposition		√		array = transpose of array	MAT X = TRN(Y)
Matrix Assignment		√		array = array	MAT X = Y
Identity Matrix	√			array = identity matrix	MAT X = IDN
Zero Matrix	√			each array element = 0	MAT X = ZER
Matrix Constant	√			each array element = 1	MAT X = CON
READ Matrix	√		√	array elements = successive DATA values	MAT READ X
PRINT Matrix	√		√	print all array elements	MAT PRINT X
INPUT Matrix	√		√	array elements = values from keyboard	MAT INPUT X
Redimension Array	√		√	array shape changed as specified	MAT REDIM X(R, C)

√ E   = Array can be redimensioned explicitly

√ I   = Resultant array redimensioned implicitly

√ A/N = Can be performed on alphanumeric as well as numeric arrays

---

Note:

In addition to Matrix Statements, Wang BASIC includes another group of statements that begin with the word "MAT". These are the Sort Statements. Their syntax, operation, and purposes are not related to the linear algebra operations performed by Matrix Statements, and the two groups should not be associated, nor implications drawn from one to the other.

Matrix Statement Syntax

In several important and unique ways the Matrix Statements depart from standard BASIC syntax and operation. Matrix Statements operate on entire arrays not just on individual variables within an array. To facilitate use of these statements, and to make their notation more similar to that commonly used in linear algebra, an array of variables is referred to in a MAT statement without appending the empty parentheses, (), to the name. Thus, when it appears in a Matrix Statement, X can refer not to a scalar variable,

as it does in all other BASIC statements, but to a numeric array X, which in all previous chapters we have named X(). In the Matrix Statements, an array name never includes the () symbols. For the Matrix Statements, the symbols at the right below replace the symbols at the left.

Standard Basic Syntax	Matrix Statement Syntax
C2()	C2
A()	A
D$()	D$
K4$()	K4$

Arrays that are referred to in Matrix Statements are automatically dimensioned to 100 element 10 x 10 arrays, unless a DIM statement at a lower line number has already established them with different dimensions. Dimensioning of arrays occurs immediately after RUN (EXEC) is keyed, before execution actually begins, and takes place in line number sequence. The automatic or default dimensioning of arrays means that if a 10 x 10 array is adequate for the operations being performed, a DIM statement need not be used. Thus,

        20 MAT A = B + C

is equivalent to

        10 DIM A(10,10), B(10,10), C(10,10)
        20 MAT A = B + C

However, notice that

        10 A(5,4) = 75
        20 MAT A = B + C

produces an error, since a reference to an array element, A(5,4), occurs at a lower line number than the statement that dimensions the array (statement 20). Alphanumeric arrays, referred to in MAT statements, are also dimensioned to 10 x 10 size, with each variable 16 characters long.

Several MAT statements can change the shape of an array during execution. Here we must distinguish between the total size of an array (the amount of space it occupies in memory) and the shape of an array. For example, an 8 x 8 numeric array is square; it has 8 rows and 8 columns. A 2 x 32 numeric array is not square; it has 2 rows and 32 columns. Despite the fact that the shapes of these arrays are different, the memory space required for them is the same, since each contains 64 numeric variables.

Memory space for variables and arrays is allocated at only one time, before execution begins, immediately after RUN (EXEC) is keyed. Therefore, it is impossible to increase the total size of an array, once execution has begun. However, some MAT statements can change the shape of an array. This is called "redimensioning" and occurs when the MAT statement is executed. For example, the statement MAT REDIM has redimensioning as its sole purpose.

237

```
10 DIM K2(20,20)
   .
   .
   .
730 MAT REDIM K2(12,N)
```

In the program segment shown above, the following occurs: After RUN is keyed, prior to execution, the DIM statement at line 10 allocates space for a 20 by 20 array, and calls it K2(). Later, sometime during program execution, line 730 is encountered. It changes the shape of array K2 so that it now has 12 rows and as many columns as the current value of the ordinary scalar variable N (provided of course that 12*N is less than or equal to 20*20).

In the MAT REDIM statement a variable, or any expression, can be used to give the new dimensions. This is not possible in the DIM statement.

If the redimensioned array is alphanumeric, a length specification may be added to the new dimensions. For example,

```
10 DIM A$(20,20)20
   .
   .
   .
430 MAT REDIM A$(25,30)10
```

In line 430, the 10 outside the parentheses specifies 10 characters as the new maximum length for each variable. Notice that line 430 also increases the total number of variables, or elements, from 400 to 750. This is made possible by the length decrease of each variable from 20 characters to 10, which keeps the total array size less than the original array size. The total number of characters in the redimensioned array is 7500 versus 8000 in the original. Redimensioning, however, does not actually reduce the total amount of memory used; it merely changes the shape of the array, and in this case, leaves 500 bytes of memory inaccessible. It would be possible, however, after statement 430, above, to again redimension array A$ to any size or shape up to the maximum size to which it was originally dimensioned in line 10. Thus,

```
760 MAT REDIM A$(10,20)40
```

would be a legal addition to the program above. The total number of bytes in the array it specifies is again 8000.

Five of the MAT statements permit explicit redimensioning in the same manner as MAT REDIM, but then perform specific operations on the redimensioned array.   In addition, six of the MAT statements may redimension an array automatically, the new dimensions being implicit from the operation and the dimensions of the arrays operated upon. For the detailed syntax of each Matrix Statement, see the publication Matrix Statements (#700-3332B).

A Programming Example

Presented below is a program that solves systems of up to 20 linear equations in 20 unknowns, provided such a solution exists.

Before we look at the program, a brief discussion of the mathematics employed is warranted. In general, a system of N linear equations in N unknowns can be represented by

$$c_{11}x_1 + c_{12}x_2 + \ldots + c_{1n}x_n = b_1$$

$$c_{21}x_1 + c_{22}x_2 + \ldots + c_{2n}x_n = b_2$$

$$\vdots$$

$$c_{m1}x_1 + c_{m2}x_2 + \ldots + c_{mn}x_n = b_m$$

$$m = n$$

where: c_{11} to c_{mn} are the coefficients

x_1 to x_n are the unknowns

b_1 to b_n are the absolute terms, or "right hand sides"

This system can be written in matrix notation as

$$CX = B$$

where C is the matrix of coefficients, order n.
X is the vector of unknowns.
B is the vector of the right-hand sides.

The solution to this system can be written as

$$X = \frac{B}{C}$$

However, since direct matrix division is, in general, undefined, the solution is always written

$$X = C^{-1}B$$

where C^{-1} represents the inverse of the matrix C, a defined matrix operation on C.

With the BASIC Matrix Statements, a solution can be obtained in two steps. The matrix of coefficients, A, is inverted, and then multiplied by the vector of the right-hand sides. The resulting vector X, of dimension m, is the solution.

The program shown in Example 19.3 implements this solution at lines 280 and 290. The MAT INPUT and MAT PRINT statements are used to facilitate the required I/O operations. For systems of order less than 20, MAT REDIM (line 180) redimensions arrays to the proper size. It should also be noted that the matrix inversion, performed at line 280, is done in place; a second array to receive the inversion of C is not required. This reduces the amount of memory needed by the program.

239

Example 19.3   Solving a System of n Linear Equations in n Unknowns

```
110 REM SOLVING A SYSTEM OF N LINEAR EQUATIONS IN N UNKNOWNS
120     DIM C(20,20) :REM MATRIX OF COEFFICIENTS
130     DIM X(20)    :REM VECTOR OF THE UNKNOWNS
140     DIM B(20)    :REM VECTOR OF THE RIGHT HAND SIDES
150     PRINT HEX(03)
160 REM REDIMENSION ARRAYS TO SIZE OF LINEAR SYSTEM
170     INPUT "NUMBER OF VARIABLES (<= 20)", K
180     MAT REDIM C(K,K), X(K), B(K)
190 REM ASSIGN COEFFICIENTS TO MATRIX C
200     PRINT HEX(OA);"ENTER MATRIX OF COEFFICIENTS."
210     PRINT "ENTER ELEMENTS ROW BY ROW, SEPARATING ELEMENTS WI
TH COMMAS."
220     PRINT "KEY EXEC AFTER EACH ROW."
230     MAT INPUT C
240 REM ASSIGN RIGHT HAND SIDES TO VECTOR B
250     PRINT HEX(OA);"ENTER VALUES OF THE RIGHT HAND SIDES"
260     MAT INPUT B
270 REM SOLVE
280     MAT C = INV(C)
290     MAT X = C*B
300 REM OUTPUT SOLUTION VECTOR
310     PRINT HEX(OA);"SOLUTION VECTOR"
320     MAT PRINT X
```

The user of this program should be alerted to two kinds of possible problems. The first problem is that of inaccuracy in the result due to the accumulation of round-off errors during execution of MAT Inversion and MAT Multiplication.   The other problem is of a fundamentally different character. Some matrices, called "ill-conditioned" matrices, can yield solutions which vary enormously with only small changes in the values of the entered coefficients. As a practical matter, then, small errors in the process of determining the values of the coefficients can produce wildly inaccurate results.

The problem of round-off errors is most likely to be serious when values along the main diagonal are not in the same range as other values in the matrix, in particular when those in the main diagonal have large negative exponents. Rows can be rearranged, and values close to zero zeroed to help overcome this problem.   If you suspect significant round-off errors in the results of Example 19.3, the simplest test is to reinvert the matrix C, in the Immediate Mode:

```
:MAT C = INV(C)
```

and follow this with MAT PRINT

```
:MAT PRINT C
```

The discrepancies between this result and the original matrix of coefficients will be at least as great as any round-off errors in the solution vector, provided the matrix is not ill-conditioned.

240

To test for an ill-conditioned matrix, the normalized determinant of C should be calculated. The normalized determinant of C is defined as

$$\text{Norm } |C| = \frac{|C|}{\alpha_1 \alpha_2 \alpha_3 \widetilde{\cdots\cdots} \alpha_n}$$

where: $|C|$ = determinant of C

$$\alpha_k = \sqrt{c_{k1}^2 + c_{k2}^2 + \ldots + c_{kn}^2} \qquad k = 1, 2, 3 \ldots n$$

If this value, norm$|C|$, is small relative to 1, then an ill-conditioned matrix should be suspected. It should be noted that the determinant $|C|$ is not a reliable indicator of an ill-conditioned matrix.

Example 19.3 can be modified to calculate and display the normalized determinant by changing line 280 to GOSUB 1000 and adding the subroutine shown in Example 19.4.

Example 19.4 A Subroutine That Calculates the Normalized Determinant

```
999  END
1000 REM SUBROUTINE CALCULATES NORMALIZED DETERMINANT OF C()
1010     A=1
1020     FOR Y = 1 TO K
1030         A1 = 0
1040         FOR X = 1 TO K
1050             A1 = C(Y,X)↑2 +A1
1060         NEXT X
1070         A = A * SQR(A1)
1080     NEXT Y
1090     MAT C = INV(C),D1 :REM INVERT AND GET DETERMINANT D1
1100 REM NORMALIZED DETERMINANT
1110     PRINT HEX(0A);  "NORMALIZED DETERMINANT ="; D1/A
1120 RETURN
```

The Matrix

```
7    8    9
8    9   10
9   10    8
```

is ill-conditioned, yet its determinant is 3. Its normalized determinant is 8.29E-04. Try executing Example 19.3 with this matrix of coefficients and with the right-hand sides close to the values:

```
24
27
27
```

You will notice large changes in the result with only small changes in the values of the right-hand sides.

CHAPTER 20
AN INTRODUCTION TO DISK DATA FILES

20-1 <u>OVERVIEW OF CHAPTER 20</u>

This chapter covers the fundamentals of the use of disk memory for data storage and retrieval. Sections 20-2 through 20-7 provide an overview of the basic catalog mode operations on data files: how to create a file, save data in it, and read data sequentially and randomly from it. Sections 20-8 through 20-11 discuss in more detail some elementary topics introduced in the preceding sections.

This chapter is not an exhaustive treatment of all the capabilities of the catalog mode statements. After reading this chapter, the beginner is urged to read the chapter of the Disk Memory Reference Manual which presents the general forms of the Automatic File Cataloging Statements and Commands. This provides an excellent review, and will introduce some auxiliary statement capabilities not discussed here.

20-2 <u>FILES AND THE DISK CATALOG</u>

In Section 4-2 we introduced the use of disks for program storage. We said that the recording area of a disk is divided into a large number of small chunks called sectors. Each of these sectors has an identifying number called its address. Disk devices are known as direct access devices because the disk read/write mechanism can move directly to any sector on the disk, when it is given the address of that sector.

We said that it would be possible to save and load programs by manually keeping an accurate list of the sectors occupied by each program, and then supplying the proper beginning sector address to the disk when saving or loading is desired. However, this is extremely inconvenient. Therefore, a group of instructions have been built into your Wang 2200 system that create and maintain, on the beginning sectors of a disk, a complete list of the names and associated sector addresses of all the files on the disk. These statements are known as the "Catalog Mode" statements. They include SCRATCH DISK which sets aside disk space for the catalog index and catalog area, SAVE DC which saves a program and enters its names and sector addresses into the catalog index, and LOAD DC which searches the catalog index for a specific program name and then loads the program into memory from the sector addresses it finds listed in the index. Catalog Mode statements also include a group of seven statements for establishing and operating on files of data. This chapter is an introduction to programming with these statements. All of these statements may be executed in the Immediate Mode.

In addition to the Catalog Mode statements, your Wang 2200 system includes a group of statements called the "Absolute Sector Addressing" statements. In order to use these statements, the program must supply the absolute disk sector addresses upon which operations are to take place. Though these statements are essential for certain types of operations, the Catalog Mode statements are easier to use and safer, in most cases.

A "file" is a collection of information about a topic. Generally, within a file this collection consists of "records." For example, an employee file might be a collection of the pay records for employees, one record per employee. A test results file might be a collection of the records from each repetition of a particular test, one record per repetition. When data files such as these are maintained on a disk using the Catalog Mode statements, the catalog index contains the starting and ending sector addresses of the file, together with the file name. A single disk may contain many such files, or perhaps only one. In any case, the catalog index contains the information needed to find the file, and to prevent accidental destruction of an old file with an incoming new one.

Programs saved on a disk are called "files" because they are treated as files by the catalog system, i.e., their names are entered in the catalog index with starting and ending sector addresses. The catalog index also contains a notation indicating for each file whether it is a program file or a data file.

The catalog index contains only the names of files and file boundary addresses, not the sector addresses of the individual records within each file. A variety of methods are employed to find individual records within a file; these methods are discussed in Section 20-9.

An overview of the Catalog mode statements and their functions is given below:

STATEMENT	FUNCTION
1. DATA SAVE DC OPEN	Establishes a new data file on a disk, and readies the processor for operations on the file.
2. DATA LOAD DC OPEN	Readies the processor for operations on a previously established data file.
3. a) DATA SAVE DC	Takes values from memory, and saves them as one "record" in a data file.
b) DATA SAVE DC END	Saves a special record that marks the end of live data in a file.
4. DATA LOAD DC	Reads values from a record (or records), and assigns the values to variables in memory.
5. DSKIP 6. DBACKSPACE	These statements change the sector address at which the next DATA SAVE DC or DATA LOAD DC will occur. They permit rapid nonsequential accessing of records.

7. DATA SAVE DC CLOSE   Protects against accidental file damage, by clearing information needed for operations on the file.

20-3 ESTABLISHING AND OPENING DATA FILES

The SAVE DC statement, introduced in Section 4-2, can be seen as performing two distinct operations. First, it writes the program name into the catalog index, specifying the new file's starting and ending sector addresses.   Secondly, it saves the program into these sectors. For disk data files, these two kinds of operations are performed by different statements.

Establishing a New File

The statement DATA SAVE DC OPEN is used to write a new file name into the catalog index, and to set aside a specific number of unused sectors for the file. The addresses of the file's starting and ending sectors are entered into the index. The user need only indicate a name for the file and the number of sectors needed. DATA SAVE DC OPEN automatically allocates the next available sectors to the file. This statement also writes a sector of control information in the last sector allocated to the file (this sector serves as a marker of the end of the file space).   DATA SAVE DC OPEN does not save any data into the file.

The statement

        DATA SAVE DC OPEN F 200, "INVTORY"

establishes a new file called INVTORY, and assigns to the file the next 200 sectors in the disk's catalog area.   The name INVTORY, together with the starting and ending sector addresses, is entered into the catalog index, and one sector of control information is written into the last sector in the new file.

In the DATA SAVE DC OPEN statement shown above, the F indicates that the file is to be opened on the F disk, of the device selected for DISK class I/O operations.   The F disk is the fixed lower disk of 2230 series or 2260 type drives, or the diskette mounted in the leftmost diskette port of 2270 type drives.   R could be used instead of F to specify the removable disk of 2230/2260 drives, or the right port of the 2270-2 drive (the middle port 2270-3).

(During Master Initialization, address 310 is automatically selected for DISK class I/O operations.   The DISK class address may be changed by executing a statement such as

        SELECT DISK 320

A system of file numbers is available to add greater flexibility to the addressing procedures of disk operations.   This system is discussed in Sections 20-10 and 20-11. Until then, we shall rely upon the SELECT DISK statement to provide the device address of the disk unit, and the F and R parameters to specify the disk.)

Just as with program files, the name of a data file can contain up to eight characters. It may be expressed as a character string in quotes or as an alphanumeric variable. Therefore,

        100 DATA SAVE DC OPEN F 200, "INVTORY"

is functionally equivalent to

        100 A$ = "INVTORY"
        110 DATA SAVE DC OPEN F 200, A$

In all Catalog Mode statements that require a file name, the name can be specified by means of alphanumeric variable.

Before the DATA SAVE DC OPEN statement is executed, the programmer must give some thought to how much space is needed in the file. Generally the series of questions to be asked is as follows:

1. What information will be stored in each record?

2. How much disk space, in sectors, is required for one record of information?

3. How many records are now waiting to be stored in the file?

4. How many additional records will be generated in the lifetime of the file? Can these new records replace old records deleted from the file?

These questions are discussed in detail in Section 20-9. For now, though, we should note that in the DATA SAVE DC OPEN statement the exact number of sectors desired for the file must be specified. If the number of sectors proves to be too small, it is not possible to simply expand the file. Generally, a new, larger file must be created, and all the data in the old file transferred to the new.

We have seen that DATA SAVE DC OPEN establishes a new disk file by putting its name and its starting and ending sector address into the catalog index. However, it does something else as well. It readies the processor for Catalog Mode operations on the new file.

In order to operate on a file in Catalog Mode, the processor must have three sector addresses in a special part of its memory called the Device Table. These sector addresses are the starting sector address of the file, the ending sector address of the file, and a special third address called the Current Sector address. When the processor has these addresses for a file in its Device Table, the file is said to be "open," because data can then be saved in or loaded from the file.

DATA SAVE DC OPEN establishes a new file and "opens" it, that is, it puts the name and file boundaries into the catalog index, and puts the starting, ending and Current Sector addresses into the Device Table (in the processor). The starting and ending addresses in the Device Table are the

same as those in the catalog index. They are put into the Device Table for quick reference by the processor, and represent the fixed boundaries of the file. The Current Sector address is set to the starting sector address by DATA SAVE DC OPEN.

Unlike the starting and ending sector addresses, which are fixed, the Current Sector address is continuously updated by Catalog Mode operations. As we shall see in the next section, this Current Sector address is the address supplied to the disk drive when data is saved or loaded.

For now, we can consider the Device Table as follows:

Starting Address	Ending Address	Current Sector Address
0	0	0

After DATA SAVE DC OPEN F 200, "INVTORY", the Device Table might look like this:

Starting Address	Ending Address	Current Sector Address
28	228	28

Re-Opening a File

The DATA SAVE DC OPEN statement is executed only when a new file is being established for the first time. Thereafter, the file's name and sector boundaries are permanently stored in the catalog index. They need only be read from the disk into the Device Table in order to "open" the file. The statement that must be used to open an already existent file is DATA LOAD DC OPEN. For example,

DATA LOAD DC OPEN F "INVTORY"

searches the catalog index of the F disk for a data file named "INVTORY". When it finds the entry, it reads the associated file boundaries into the Device Table and makes the Device Table's Current Sector address equal to the starting sector address of the file.

DATA SAVE DC OPEN and DATA LOAD DC OPEN are often confusing to the beginner because they are not good descriptions of the operations they perform. The following summary is therefore provided:

DATA SAVE DC OPEN

1. Establishes a new data file of specified sector length by entering its name and sector boundaries into the catalog index.

2. Opens the new file for Catalog Mode operations by putting its addresses into the Device Table, and setting the Device Table's Current Sector address equal to the starting address of the file.

DATA LOAD DC OPEN

2. (ONLY) opens a file by getting its sector addresses from the catalog index and putting them into the Device Table. Sets the Current Sector address equal to the starting sector address of the file.

Once a file is open, data can be saved in it or read from it, regardless of whether it was opened for the first time with DATA SAVE DC OPEN or later opened with DATA LOAD DC OPEN.

20-4 SAVING DATA IN A FILE

Once a new file has been established and opened with DATA SAVE DC OPEN, the next step is to save data in it. The statement used to save data in a cataloged file is DATA SAVE DC. For example,

DATA SAVE DC N$, D$, S$, Q, R

causes the disk drive to save the current values of the variables N$, D$, S$, Q, and R. The Current Sector address is supplied to the disk drive as the address at which it is to begin saving the data. After the data has been saved, the Current Sector address in the Device Table is updated so that it contains the address of the sector following the last one in which data was saved. Saving values specified by a DATA SAVE DC statement may require just one sector or many sectors. However, even if the final sector is not completely filled, the Current Sector address is updated so that it has the address of the next sector following the partially filled one.

In addition to saving the specified values, the DATA SAVE DC statement surrounds the saved values with certain control information. This control information marks off the values, collectively, as a "record." Thus, the stored result of a DATA SAVE DC statement is, by definition, a record, and it is the task of the program to specify all the values needed for a single record, prior to executing the DATA SAVE DC statement. For example, suppose that the above DATA SAVE DC statement saves an inventory record for one product. Prior to executing this statement, N$ might receive the product number, D$ the product description, S$ the supplier code, Q the on-hand

quantity, and R the reorder level. These values are the complete record for one product in this simple inventory file. A very simple program that creates a new inventory file, and allows records to be saved in the new file is shown in Example 20.1.

Example 20.1 Creating a New File and Saving Records in It

```
110 REM A SIMPLISTIC PROGRAM FOR CREATING A NEW FILE
112 REM   AND SAVING RECORDS INTO IT
120     DIM N$10, D$40, S$6
130 REM ESTABLISH A FILE AND OPEN IT.
140      DATA SAVE DC OPEN F 200, "INVTORY"
150 REM ENTER DATA FOR ONE INVENTORY RECORD
160     INPUT "PRODUCT NUMBER", N$
170     INPUT "PRODUCT DESCRIPTION", D$
180     INPUT "SUPPLIER CODE", S$
190     INPUT "ON HAND QUANTITY", Q
200     INPUT "INDICATED REORDER LEVEL", R
210 REM SAVE RECORD ON DISK
220      DATA SAVE DC N$, D$, S$, Q, R
230 REM MORE RECORDS?
240     INPUT "MORE PRODUCTS (Y OR N)", R$
250     IF R$ = "Y" THEN 160
```

Line 140 establishes a new file called "INVTORY", with 200 sectors allocated to it, and opens the file. Lines 160 to 180 enter values into each of the variables used to define the record. Line 220 writes the record into the file, at the Current Sector address. Each time through the loop a record is saved, and the Current Sector address is updated to the next available sector. Since DATA SAVE DC OPEN sets the Current Sector address to the first sector in the file, this program saves the first product record starting at the first sector of the file. Each subsequent record is saved into the next available sector. Thus, the records are saved into sequential sectors on the disk in the same order that they are entered.

This program will execute successfully; however, it does not display good programming practices for disk data files, for reasons that are discussed below. It also should be noted that it cannot be executed twice with the same disk, since on a second attempt, DATA SAVE DC OPEN would be trying to open a second file with the name "INVTORY", an illegal operation. To reopen the file created by this program, the DATA LOAD DC OPEN statement must be used.

Values to be saved by the DATA SAVE DC statement can be specified by giving the name of the alphanumeric or numeric variable that contains the value, as shown in line 220 of Example 20.1. If an entire array of values is to be saved, an array designator such as A() or K2$(), may be used in the DATASAVE DC statement. Finally, a value may be specified by means of a literal string or an expression. In the latter case, the expression is evaluated, and the result is saved.

Within a record, values are saved in the sequence in which they are specified in the DATA SAVE DC statement. Arrays are saved row by row. Each value is preceded in the record by a one byte Start of Value code, which separates the value from the preceding value, and marks it as numeric or alphanumeric. Only the value is saved in the record, not the name of the variable or the quotation marks around a literal string.

20-5   MARKING THE END OF DATA IN A FILE AND CLOSING THE FILE

Usually, when a new file is created, it is given extra sectors to allow for gradual growth in the number of records. Therefore, most files, at any given time, will contain some unused sectors, between the last record saved and the end of the file space. In many programming operations it is useful to have this end of the live data clearly marked. This permits the end of the live data to be found very quickly when a new record needs to be added, and it is useful in other operations as well.

The statement DATASAVE DC END is used to write a special one-sector trailer record that marks the end of live data in a file. DATA SAVE DC END puts the special trailer record into the sector specified as the Current Sector address.

In the program of Example 20.1, the Current Sector address is always set to the first empty sector at the end of the live data; therefore, the DATA SAVE DC END statement can simply be appended to the program as follows:

Example 20.2   Adding a DATA SAVE DC END to Example 20.1

```
110 REM MARKING THE END OF DATA
120     DIM N$10, D$40, S$6
130 REM ESTABLISH FILE AND OPEN IT.
140     DATA SAVE  DC OPEN F 200, "INVTORY"
150 REM ENTER DATA FOR ONE INVENTORY RECORD
160     INPUT "PRODUCT NUMBER", N$
170     INPUT "PRODUCT DESCRIPTION", D$
180     INPUT "SUPPLIER CODE", S$
190     INPUT "ON HAND QUANTITY", Q
200     INPUT "INDICATED REORDER LEVEL", R
210 REM SAVE RECORD ON DISK
220     DATA SAVE  DC N$, D$, S$, Q, R
230 REM MORE RECORDS?
240     INPUT "MORE PRODUCTS (Y OR N)", R$
250     IF R$ = "Y" THEN 160
260 REM MARK END OF DATA
270     DATA SAVE  DC END
280     DATA SAVE  DC CLOSE
```

Marking the end of live data with a DATA SAVE DC END statement is not required for the use of disk catalog operations; however, it is so often useful to have the end of the data marked that it is a part of all good programming.

Closing a File

Earlier we said that a file is "open" when starting, ending, and Current Sector addresses are present in the Device Table. An open file can be operated on with the Catalog Mode statements. A particular open file can be "closed" by opening another file. This replaces the old file's sector addresses with those of the newly opened file. (In Section 20-10 we will see a means by which several files can be open simultaneously.) A file can also be closed by executing CLEAR, or Master Initializing, which puts zeros into the sector addresses in the Device Table. Since a closed file cannot be operated upon with DATA SAVE DC, closing a file protects it from accidental destruction by another program, or by an Immediate Mode operation. For this reason, the statement

        DATA SAVE DC CLOSE

is available to close a file. It simply replaces the sector addresses in the Device Table with zeros. It does nothing to the disk file itself or to the disk catalog index. Files closed with DATA SAVE DC CLOSE may, of course, be reopened with DATA LOAD DC OPEN just as they could be if they had been closed by any of the other means discussed above. Mere termination of a program does not close a file. For this reason, DATA SAVE DC CLOSE should be used. Therefore, this statement should be added to Example 20.2.

        280 DATA SAVE DC CLOSE

20-6 LOADING DATA FROM A FILE

Once a data file has been set up, and records saved in it, the records can be read. In order to read a record in the Catalog Mode, the file must be "open", that is, a starting, ending, and Current Sector address must be entered into the Device Table. Except when first established, a data file is always opened with DATA LOAD DC OPEN. This gets the starting and ending addresses of the file, puts them in the Device Table, and sets the Current Sector address equal to the starting sector address of the file.

The DATA LOAD DC statement is used to load data from a file in Catalog Mode. For example, the statement

        DATA LOAD DC N$, D$, S$, Q, R

causes the disk drive to begin reading, starting at the sector specified as the Current Sector Address. The values that are read are assigned successively to the variables specified in the DATA LOAD DC statement. After values have been assigned to each of the specified variables, the Current Sector address, in the Device Table, is updated to the address of the first sector of the next record.

Assignment of values is according to the conventional procedure. For example, if an alphanumeric variable is too short to contain an entire alphanumeric value, the assignment is made with the extra characters on the right truncated. It is especially important to note that an error results if a numeric value, encountered in the record, is matched with an alphanumeric variable, in the DATA LOAD DC statement, or vice versa. For this and other reasons, it is important that precise documentation be maintained of data file record layouts.

Values may be loaded into an entire array by specifying the standard array designator in the DATA LOAD DC statement. For example, in the statement

        400 DATA LOAD DC A$(), X()

values from the data file are assigned row by row to the array A$(), until each variable in the array has been assigned a value; then the next values are assigned in the same way to X().

It is not necessary that the same variables be used to receive data in the DATA LOAD DC statement as were used originally to save the data. The only requirement is that numeric values be loaded into numeric variables, and alphanumeric values into alphanumeric variables.

It is good programming practice to read exactly one record with a DATA LOAD DC statement, though this is not required. For example, the DATA SAVE DC statement of Example 20.1 creates a record with three alphanumeric values followed by two numeric ones. Since the statement

        DATA LOAD DC N$, D$, S$, Q, R

shown above specifies three alphanumeric variables followed by two numeric ones, it loads exactly one record, as created by the DATA SAVE DC statement of Example 20.1.

If fewer variables are specified in the DATA LOAD DC statement than there are values in the record, the extra values are ignored. If more variables are specified than there are values in the record, successive values from the next record(s) will be assigned to the variables. It must be emphasized, though, that after a DATA LOAD DC statement, the Current Sector address is set to the beginning of the next record, even if only some of the values from the previous record have been assigned to variables. In Catalog Mode it is impossible to begin reading values in the middle of a record.

A simple program that reads and prints the inventory data file created by Example 20.1 is shown in Example 20.3.

Example 20.3 Printing the Inventory File of Example 20.1

```
110 REM A SIMPLISTIC PROGRAM FOR PRINTING THE INVENTORY FILE
120     DIM N$10, D$40, S$6
130     SELECT PRINT 215 (100)
140 REM OPEN FILE
150     DATA LOAD DC OPEN F "INVTORY"
160 REM LOOP TO READ AND PRINT EACH RECORD
170     DATA LOAD DC N$, D$, S$, Q, R
180         PRINTUSING 200, N$, D$, S$, Q, R
190     GOTO 170
200     %  #########      ######################################
#   ######      ##,###    ##,###
```

Notice at line 120 of Example 20.3 that the alphanumeric variables are dimensioned to the same size as those which were used to save the record. Variables N$ and S$ could be allowed to be dimensioned to the default length of 16 characters; however, if D$ were not dimensioned to at least 40 characters, part of the second value in the record might be lost, since it can contain 40 characters.

Statements 170 to 190 form a loop. Each time through the loop a record is read, the Current Sector address is automatically updated, and then the record is printed. However, this loop has no exit, so, eventually, either the end of live data will be reached, or the control sector at the very end of the file space will be reached, or both. When either happens, an error is signaled as the DATA LOAD DC statement attempts to load nonexistent data. Remember, the file which this program reads was created by Example 20.1, so no end-of-data trailer record is present.

Testing For End-of-Data

Now we can see a value in using the end-of-data trailer record. If the Current Sector address is the address of an end-of-data trailer record in the file, and a DATA LOAD DC statement is executed, several things happen. First of all, the values of the variables in the DATA LOAD DC statement are left unchanged. The Current Sector address is not updated, and a notation is made in a special part of memory that a DATA LOAD DC statement has read an end-of-data trailer record.

A special BASIC statement is available to test if an end-of-data trailer has been read. The form of this statement is

IF END THEN line number

The IF END THEN statement checks the special part of memory to see if an end-of-data trailer has been read during the last DATA LOAD DC statement. If it has been read, IF END THEN effects a branch to the line number following "THEN". Thus, the IF END THEN statement can be used to exit from a record reading loop, when the end-of-data is reached (provided that the end-of-data is marked with the special end-of-data trailer record). IF END THEN may not be used in the Immediate Mode.

Example 20.4 shows a modification of Example 20.3. It uses IF END THEN to exit from the loop. It will read and print a file created by Example 20.2.

Example 20.4 Printing the Inventory File Created by Example 20.2

```
110 REM A BETTER PROGRAM FOR PRINTING THE INVENTORY FILE
120     DIM N$10, D$40, S$6
130     SELECT PRINT 215 (100)
140 REM OPEN FILE
150     DATA LOAD DC OPEN F "INVTORY"
160 REM LOOP TO READ AND PRINT EACH RECORD
170     DATA LOAD DC N$, D$, S$, Q, R
175        IF END THEN 210
180        PRINTUSING 200, N$, D$, S$, Q, R
190     GOTO 170
200     %  #########      #######################################
#    ######      ##,###     ##,###
210     DATA SAVE  DC CLOSE
```

The IF END THEN statement, added at line 175, effects a branch out of the loop when DATA LOAD DC reads an end-of-data trailer record. Thus, this program does not terminate in an error message, and were it necessary, processing could continue uninterrupted.

It should be noted that only the special end-of-data trailer record created by DATA SAVE DC END has the effect described above. An error results if a DATA LOAD DC statement is executed when the Current Sector address contains the address of the control sector at the end of the file.

20-7 NON-SEQUENTIAL ACCESS WITH DSKIP AND DBACKSPACE

We have seen that whenever a DATA SAVE DC statement is executed, the Current Sector address is updated to the sector following the last one in which data was saved. When a DATA LOAD DC statement is executed, the Current Sector address is updated to the sector address of the next record. Thus, at the most fundamental level, the Catalog Mode write and read statements are designed for accessing records sequentially. Many applications require that records be accessed sequentially, and, therefore, having this type of accessing built into the Catalog Mode statements can be extremely convenient. This is particularly true if the records can be kept in a particular sequence, either by sorting the records or adding records that logically come at the end.

For example, suppose that in our inventory example the product number is assigned to new products by the user of the system. Further suppose that the product number is a composite. The first character in the product number is a letter of the alphabet that indicates the warehouse in which the product is stored. The remaining characters in the product number provide a consecutive numbering of the products. Thus the product numbers for the first five products might be B1, A2, B3, F4, M5. If there are now 217 such products in the inventory file and a new product, stored in warehouse A, must be added, it would be assigned product number A218 and placed at the end of the file.

In such a situation, the file may be initially created sequentially, as shown in Example 20.2 and also may be printed sequentially as shown in Example 20.4. But, what happens when a new record must be added to the end of the file, or the on-hand quantity reduced for product number B185? One way to find the end of the file would be:

```
200      DATA LOAD DC N$, D$, S$, Q, R
210      IF END THEN 230
220      GOTO 200
230 REM FOUND END OF FILE
```

This reads each record until the end-of-data trailer is read. A similar approach to updating prouct B185 is:

```
200 DATA LOAD DC OPEN F "INVTORY"
210 FOR K=1 TO 184
220 DATA LOAD DC N$, D$, S$, Q, R
230 NEXT K
240 REM NOW READY TO UPDATE B185
```

However, both of these solutions fail to make use of the disk's feature of offering direct access to every sector. These solutions merely represent roundabout ways of getting the Current Sector address equal to the address of the end-of-data trailer (first problem), or equal to the sector address of product B185 (second problem). Once the Current Sector address in the Device Table is correct, the disk device is used to directly save new data at the proper location. If the Current Sector address could be changed in a more direct fashion, the time consuming sequential loading of the above examples could be eliminated.

Wang BASIC offers two statements whose purpose is to change the Current Sector address. These two statements are DSKIP and DBACKSPACE. Each of these statements can be seen as having three forms, summarized in Table 20.1.

Table 20.1 Forms of the DSKIP and DBACKSPACE Statements

Forms of the DSKIP Statement	Operation
DSKIP expression S	Evaluates the expression and adds the truncated result to the Current Sector address. Thus, if the value of the expression is n, n sectors are "skipped" over.
DSKIP END	Sets the Current Sector address equal to the address of the end-of-data trailer record.
DSKIP expression	Evaluates the expression and skips over a number of records equal to the truncated value of the result.

254

<u>Forms of the DBACKSPACE Statement</u>   <u>Operation</u>

DBACKSPACE expression S   Evaluates the expression and subtracts the truncated result from the Current Sector address. Thus, if the value of the expression is n, n sectors are "backspaced" over.

DBACKSPACE BEG   Sets the Current Sector address equal to the starting sector address of the file.

DBACKSPACE expression   Evaluates the expression and backspaces over a number of records equal to the truncated value of the expression.

The statement DSKIP END is used to set the Current Sector address to the address of the end-of-data trailer record. Of course, the trailer must be present for DSKIP END to execute correctly. Execution of DSKIP END is much faster than the technique of reading each record shown above.

The DSKIP...S and DBACKSPACE...S statements simply add and subtract, respectively, from the Current Sector address, the truncated (integer) value of the expression. The value of the expression must be positive. If the resultant Current Sector address would be less than the first sector of the file, the address of the first sector of the file becomes the Current Sector address. Similarly, if the resultant Current Sector address would be above the end of the file space, the end of file address becomes the Current Sector address. Thus, it is impossible to accidentally set the Current Sector address to a sector outside of the file. However, it is possible to accidentally set it above the address of the end-of-data trailer record. The programmer must be careful to avoid this.

DSKIP...S and DBACKSPACE...S are the fastest ways of changing the Current Sector address, since they require no action by the disk drive itself. Their use, however, presupposes that the length of each file record, in sectors, is known. For example, if the length of each record is known to be 2 sectors, and the program should skip forward "over" 20 records, then DSKIP 40 S could be used. Most often, record length is known and, therefore, DSKIP...S and DBACKSPACE...S should be used. (Record length is discussed in detail in Section 20-8.)

In some technical applications in which a few large arrays may constitute an entire file, record length may not be known, or may be highly irregular. In these circumstances DSKIP... and DBACKSPACE... may be the easiest way to skip or backspace over a specific number of records. These instructions require the disk to read all intervening records, though, and are, therefore, much slower than DSKIP...S and DBACKSPACE...S.

Example 20.5 shows a program for adding new product records of the type discussed above to the end of the inventory file. Each inventory record in this system occupies one sector.

Example 20.5 Adding Records to the End of the Inventory File

```
110 REM ADDING RECORDS TO THE END OF THE INVENTORY FILE
120     DIM  N$10, N2$10, D$40, S$6
130 REM OPEN FILE
140     DATA LOAD DC OPEN F "INVTORY"
150 REM
160 REM READ LAST PRODUCT RECORD
170     DSKIP END
180     DBACKSPACE 1 S
190     DATA LOAD DC N$, D$, S$, Q, R
200     CONVERT STR(N$,2) TO N
210 REM GET NEW PRODUCT NUMBER AND CHECK.
220     PRINT HEX(03); "TO END PROGRAM KEY S.F. 31 AT ANY TIME"
230     INPUT "NEW PRODUCT NUMBER", N2$
240     IF NUM(STR(N2$,2)) <> 9 THEN 400 :REM NON-NUMERIC?
250     CONVERT STR(N2$,2) TO N2
260     IF N+1 <> N2 THEN 400 :REM OUT OF FILE SEQUENCE?
270 REM PRODUCT NUMBER OK. ENTER RECORD VALUES.
280     INPUT "PRODUCT DESCRIPTION", D$
290     INPUT "SUPPLIER CODE", S$
300     INPUT "ON HAND QUANTITY", Q
310     INPUT "REORDER LEVEL", R
320 REM SAVE NEW RECORD AND GO BACK FOR NEXT
330     DATA SAVE  DC N2$, D$, S$, Q, R
340     N = N2
360     PRINT
370     GOTO 220
380 REM
390 REM BAD PRODUCT NUMBER ENTERED
400     PRINT "INVALID PRODUCT NUMBER"
410     PRINT
420     GOTO 230
430 REM END PROGRAM ROUTINE
440     DEFFN' 31
450     DATA SAVE  DC END  :REM MARK END OF DATA
460     DATA SAVE  DC CLOSE
```

Line 170 changes the Current Sector address to the address of the end-of-data trailer. Line 180 then backspaces one sector so that the last record in the file can be read. The last record is read so that when the new product number is entered, it can be tested to see that is actually the next consecutive number. This protects the integrity of the file organization. Before this test can be made, the consecutive portion of the product number, STR(N$,2), must be converted to numeric form (line 200).

At line 230 the new product number is entered into N2$. Line 240 checks that the number portion of this can be converted to numeric. Without this test, an accidental entry of "XB125", for example, would cause an error at line 250, since B125 cannot be converted to numeric form. Assuming the form is acceptable, 250 makes the conversion, and 260 tests to see if the product number is the next consecutive product number. If the entered product number fails the validity tests at lines 240 or 260, a branch to an "invalid" message occurs, and the number must be reentered.

Lines 280 through 330 allow the remaining record values to be entered, and save the new record. The new record is saved over the end-of-data record, which is thereby eliminated.

Line 340 sets N equal to the converted consecutive portion of the new product number. If another record is to be added, the test for consecutive product numbers can be performed without reading the last record.

Line 370 branches back to the product number entry routine, making this program into a closed loop. The program can be ended any time the "?" is displayed, by keying Special Function key 31. This causes a branch to the end program routine, which writes the end-of-data trailer, and ends the program. The end-of-data trailer must be written before the program is ended, since operations on this file depend on its presence. Providing the loop exit in this fashion gives the operator an emergency termination procedure, as well as a routine one, both of which preserve the file's integrity.

Example 20.6 shows a program designed to allow updating of the quantity-on-hand value in any selected record.

Example 20.6 A Program to Update Product Records

```
110 REM UPDATING A PRODUCT RECORD
120     DIM N$10, D$40, S$6
130 REM OPEN FILE
140     DATA LOAD DC OPEN F "INVTORY"
150 REM GET HIGHEST PRODUCT NUMBER
160     DSKIP END
170     DBACKSPACE 1 S
180     DATA LOAD DC N$, D$, S$, Q, R
190     CONVERT STR(N$,2) TO M :REM M IS MAX PRODUCT NUMBER
200 REM GET PRODUCT NUMBER OF RECORD TO BE UPDATED
210     PRINT HEX(03) :REM CLEAR CRT
220     INPUT "NUMBER OF PRODUCT TO BE UPDATED", N$
230     IF NUM(STR(N$,2)) <> 9 THEN 460 :REM INVALID?
240     CONVERT STR(N$,2) TO N
250     IF N > M THEN 460 : REM TOO HIGH?
260 REM PRODUCT NUMBER OK. FIND PRODUCT RECORD.
270     DBACKSPACE BEG
280     DSKIP N-1 S
290 REM LOAD AND PRINT PRODUCT RECORD
300     DATA LOAD DC N$, D$, S$, Q, R
310     PRINT
320     PRINT D$
330     PRINT "QUANTITY ON HAND ="; Q
340     PRINT
350     DBACKSPACE 1 S
```

```
360 REM ENTER TRANSACTION AND UPDATE RECORD
370     INPUT "ENTER AMOUNT RECEIVED (+) OR SOLD (-)", T
380     DATA SAVE  DC N$, D$, S$, Q+T, R
390 REM MORE RECORDS TO UPDATE?
400     R$ = " " :REM NO DEFAULT ENTRY
410     INPUT "MORE RECORDS TO UPDATE (Y/N)", R$
420     IF R$ = "Y" THEN 210
430     IF R$ <> "N" THEN 410 :REM OPERATOR ERROR?
440     DATA SAVE  DC CLOSE
450 END
460 REM ERROR ROUTINE
470     PRINT "INVALID. REENTER"
480     PRINT
490     GOTO 220
```

Lines 160-190 load the last record in the file, and convert the number portion of its product number to a numeric value. This value is saved in M (line 190) so that when product numbers are entered, they may be checked to see that they are not greater than the highest product number.

Lines 210 to 250 allow the product number to be entered, convert its number portion to numeric, and test that it is within the file.

Line 270 sets the Current Sector address equal to the beginning sector address of the file. This permits DSKIP (line 280) to set the Current Sector address equal to the address of the desired record. After line 270, the Current Sector address points to the first record in the file. If the first record is the record sought, then no skip is needed. For each product, the number of sectors to be skipped to locate its record is one less than the product number; therefore, N-1 sectors are skipped at line 280. After line 280, the Current Sector address is set to the address of the desired record.

Since DATA SAVE DC always saves an entire record, in order to update one value in the record, the program must first read the whole record, update the proper value, and write the entire record. Lines 300 to 350 read the record, print the product description and quantity-on-hand, and backspace one sector so that the updated record will be recorded in the same sector from which it was read.

At line 370 the transaction is entered. The new value of the on hand quantity appears as an expression in the DATA SAVE DC statement (line 380). The DATA SAVE DC statement evaluates the expression, and writes the result into the record.

20-8 DATA RECORDS AND PLANNING OF DATA FILES

The DATA SAVE DC OPEN statement requires that a file size in sectors be specified. Thus, before a new file can be established, some planning must be done to determine the size of each record to be saved in the file, and the maximum number of such records which may be saved in the file at any one time during the expected lifetime of the file. It is not possible to simply expand a file which proves to be too short. In general, a new, larger file must be opened on the same disk or another disk, and the contents of the old file copied to the new.

When planning records and file space, careful attention must be paid not only to the space occupied by actual data values, but also to the space used by the various kinds of control information that the Catalog Mode statements automatically supply. For example, the last sector of every data file is occupied by a special control sector, which contains certain information about the file in addition to that maintained in the catalog index. Also, all data files should have an end-of-data trailer record. This also occupies one sector. Thus, when estimating file size, two sectors should always be added to the total required for the actual data records.

Each sector on a disk has a physical storage capacity of 256 bytes. (Remember that a byte is the amount of space required to contain a single alphanumeric character. We could as well say that a sector has a physical storage capacity of 256 characters, though the term "byte" is customarily used when referring to capacity.) However, control information is automatically written into each sector by the DATA SAVE DC statement. This control information reduces the amount of space available for data. Control information is of two types:

1. Sector control bytes.
2. Start of value (SOV) control bytes.

[handwritten: $256 (-3) = 253$ (-1) for an value / # Values + # Total Bytes in Value $\frac{253}{\text{# Values}}$]

The DATA SAVE DC statement automatically writes three sector control bytes into each sector of a record. Two of these bytes occupy the first two byte locations in each sector. These are used to indicate whether the sector is the first, last, or a middle sector in the record. They serve to separate one record from the next. The third control byte follows the last byte of the last data-value in the sector, and marks the end of valid data within that sector. Thus, after subtracting the three sector control bytes, a total of 253 bytes are available for data and start-of-value (SOV) bytes.

In addition to the sector control bytes, a single start-of-value (SOV) byte precedes each value saved in a record. This byte indicates to the system whether the value is numeric or alphanumeric and its length in bytes. Consider, for example, the inventory record of the preceding sections. The storage space for the record is defined by:

    120 DIM N$10, D$40, S$6
    .
    .
    .
    220 DATA SAVE DC N$, D$, S$, Q, R

259

The record looks like this in a sector:
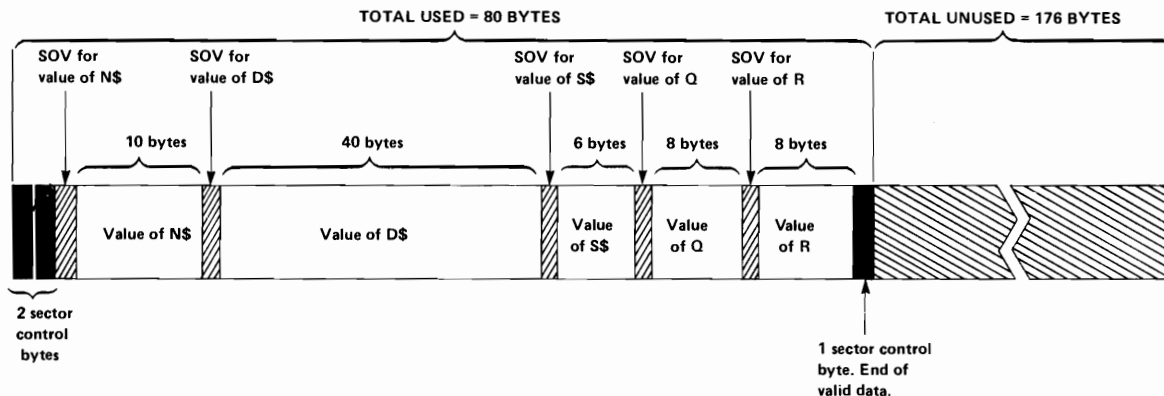
Disk storage space requirements can be summarized as follows:

1. There are 253 bytes available for storage in each sector, after allowing for the sector control bytes.

2. Each numeric value requires 9 bytes, 8 bytes for the value plus 1 byte for the SOV. (Numeric values may be specified in the DATA SAVE DC statement by numeric variables, expressions, or numeric array elements.)

3. Each alphanumeric value requires a number of bytes equal to the dimensioned size of the specifying alphanumeric variable or array element plus one byte for the SOV, or the number of characters in the specifying literal string plus one byte for the SOV. Note that in the case of an alphanumeric variable or array element, it is the dimensioned size, including all trailing spaces, which must be counted.

4. If a value does not completely fit into the space remaining in a sector, it is automatically written in the next sector. Values do not overlap from one sector to the next, though a single record may require many sectors.

When entire arrays are saved in a record by using the array designator, the values are saved row by row into the record. For example, in the record saved by

        10 DIM K(3,3)
        .
        .
        .
        50 DATA SAVE DC K()

the values are saved from the variables in this order: K(1,1), K(1,2), K(1,3), K(2,1), K(2,2), K(2,3), K(3,1), K(3,2), K(3,3). The value of an individual array element can always be specified in a DATA SAVE DC, if the entire array is not to be saved. Within the record there is no indication of whether a value was saved from an array, or any other source, and, therefore, values may be loaded into array or scalar variables regardless of their origin.

When planning a disk file, it is wise to design files that do not waste disk space. For the beginning programmer this means designing records so that the total unused space at the end of each sector is held to a minimum. There are several techniques for doing this.

When a record extends over several sectors, there can be more efficient and less efficient ways of saving the record, depending only upon the order in which the values are specified in the DATA SAVE DC statement. For example,

        10 DIM A$(5)50, B$(3)64, C$48
        .
        .
        .
        100 DATA SAVE DC A$(), B$(), C$

This record requires three sectors which, after subtracting the three sector control bytes and consolidating the SOV bytes into the values, look like this:

1 RECORD

261
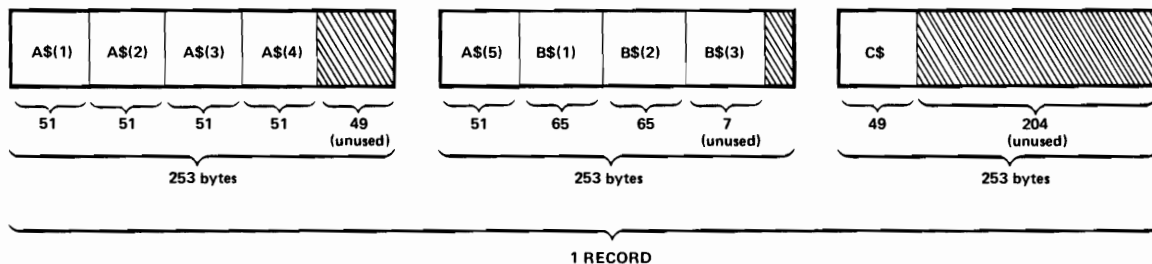
A total of 49+7+204 or 260 bytes are unused in the three sectors occupied by this record. However, merely rearranging the order of specification of the values in the DATA SAVE DC statement can result in a savings of one entire sector as follows:

100 DATA SAVE DC C$, A$(), B$()

C$	A$(1)	A$(2)	A$(3)	A$(4)
49	51	51	51	51

253 bytes

A$(5)	B$(1)	B$(2)	B$(3)	
51	65	65	65	7 (unused)

253 bytes

1 RECORD

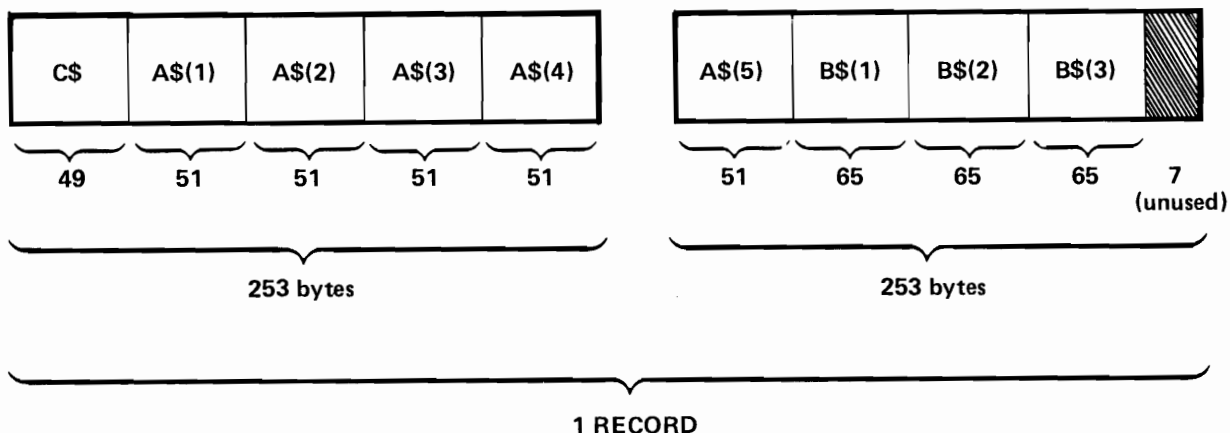A different approach is required when a record is so short that it uses only one sector, and leaves most of the sector empty. An example of this is the inventory record discussed in the previous sections. It uses only 80 bytes out of 256, leaving 176 bytes wasted. Beginning programmers are often tempted to ignore this waste of disk space, especially when available space exceeds present needs. This temptation should be overcome. Files tend to grow more rapidly than may be anticipated, and operations on more compact files are generally faster.

The programming technique used to reduce this waste is known as "blocking" records. In the inventory file discussed in the preceding sections, one disk record, that is, the result of a single DATA SAVE DC statement, contains one data record, the data on a single product. DATA SAVE DC always creates one disk record and always uses at least one sector. It is, therefore, impossible to put more than one disk record into a single sector. However, it is possible to let one disk record contain several data records. From the point of view of the disk and the control information written with a DATA SAVE DC statement, the several data records will "look like" a single record, since they are written collectively with a single DATA SAVE DC statement. The programs that create and use the file must internally distinguish one data record from another, within a single disk record.

For example, since each inventory data record requires 77 bytes (after subtracting the three sector control bytes), three such data records would occupy 231 bytes and can fit into a one-sector disk record. An obvious way of saving three such data records in a single disk record would be by simply using three sets of variables, and specifying one set after the other in the DATA SAVE DC statement:

```
120 DIM N1$10, D1$40, S1$6, N2$10, D2$40,
S2$6, N3$10, D3$40, S3$6
  .
  .
  .
400 DATA SAVE DC N1$, D1$, S1$, Q1, R1, N2$,
D2$, S2$, Q2, R2, N3$, D3$, S3$, Q3, R3
```

Though intuitively obvious, this approach is extremely awkward from the point of view of the programming required to manipulate the separate data records. In most cases a far superior approach is to associate the corresponding values in each of the data records, as follows:

	Product Number	Description	Supplier Code	Quantity on Hand	Reorder Level
record 1	N1$	D1$	S1$	Q1	R1
record 2	N2$	D2$	S2$	Q2	R2
record 3	N3$	D3$	S3$	Q3	R3

and, instead of using scalar variables to specify the values, use one-dimensional array variables in which the subscript identifies the data record. Thus, each of the values in the three records can be identified as:

	Product Number	Description	Supplier Code	Quantity on Hand	Reorder Level
record 1	N$(1)	D$(1)	S$(1)	Q(1)	R(1)
record 2	N$(2)	D$(2)	S$(2)	Q(2)	R(2)
record 3	N$(3)	D$(3)	S$(3)	Q(3)	R(3)

All three data records can be written into a single disk record as follows:

```
120 DIM N$(3)10, D$(3)40, S$(3)6, Q(3), R(3)
  .
  .
  .
400 DATA SAVE DC N$(), D$(), S$(), Q(), R()
```

This is known as "array type blocking". Notice that as a result of the DATA SAVE DC statement, the corresponding values of each of the data records are saved one after the other. Thus, at the beginning of the disk record are the three product numbers. These are followed by the three product descriptions, followed by the three supplier codes, etcetera.

263

Records saved in this fashion are loaded as follows:

```
120 DIM N$(3)10, D$(3)40, S$(3)6, Q(3), R(3)
    .
    .
    .
190 DATA LOAD DC N$(), D$(), S$(), Q(), R()
```

Example 20.7 shows a modification of Example 20.2. It creates the inventory file with three product records per sector (block).

Example 20.7 Creating a Blocked-Record Inventory File

```
110 REM EXAMPLE 20.2 MODIFIED FOR ARRAY TYPE BLOCKED RECORDS
120     DIM N$(3)10, D$(3)40, S$(3)6, Q(3), R(3)
130 REM ESTABLISH FILE AND OPEN IT.
140     DATA SAVE  DC OPEN F 100, "INVTORY"
150 REM LOOP TO ENTER DATA FOR THREE RECORDS
160     FOR K = 1 TO 3
170         PRINT HEX(03); "KEY S.F.31 TO END PROGRAM"
180         INPUT "PRODUCT NUMBER", N$(K)
190         INPUT "PRODUCT DESCRIPTION", D$(K)
200         INPUT "SUPPLIER CODE", S$(K)
210         INPUT "ON HAND QUANTITY", Q(K)
220         INPUT "INDICATED REORDER LEVEL", R(K)
230     NEXT K
240 REM SAVE THREE DATA RECORDS IN ONE DISK RECORD
250     DATA SAVE  DC N$(), D$(), S$(), Q(), R()
260     GOTO 160 :REM LOOP BACK FOR MORE
270 REM END OF DATA ENTRY
280     DEFFN' 31
290     IF K = 1 THEN 370 :REM NO LEFT-OVER DATA RECORDS?
300 REM FILL UNUSED DATA RECORDS
310     FOR J = K TO 3
320         INIT("↑") N$(J), D$(J), S$(J)
330     NEXT J
340 REM SAVE LEFT OVER DATA RECORD(S) AND FILLED RECORD(S)
350     DATA SAVE DC N$(), D$(), S$(), Q(), R()
360 REM MARK END OF DATA
370     DATA SAVE  DC END
380     DATA SAVE  DC CLOSE
```

Notice that the data entry loop (lines 160-230) is executed three times before a disk record is actually written. The loop counter, K, acts as a subscript, specifying the variables to receive the values for a single record. The main part of the program (lines 160-260) forms a closed loop. Special Function key 31 is used to end the program.

Since the number of product records may not be evenly divisible by 3, it is possible that an entered data record may not have been written to the disk when S.F. 31 is depressed. If all records have been written, then K will have a value of 1, ready to specify the first product record in a new block, and the DATA SAVE DC END trailer can be immediately written (lines 290 and 370). If, however, K has a value of 2 or 3, then 1 or 2 records have been entered but not saved on the disk. They must be saved and, in addition, the unused records in this last block should be filled with padding characters. The padding characters have several functions: they mark the end-of-data within the block, and they ensure that, if the file should be sorted, garbage in the last unused record position will not be confused with live data. The values which should be padded are those which could serve as possible identifiers (or so-called "keys") of the record. The loop at lines 310-330 fills the alphanumeric values in the unused records with up-arrow characters. (Up arrows are often used for this purpose since their character value is greater than all the uppercase keyboard characters; they, therefore, sort high in an ascending sort of the file.)

Example 20.8 is a modification of Example 20.4. It prints the blocked inventory file.

Example 20.8 Printing a Blocked File

```
110 REM PRINTING A BLOCKED INVENTORY FILE
120     DIM N$(3)10, D$(3)40, S$(3)6, Q(3), R(3)
130     SELECT PRINT 215 (100)
140 REM OPEN FILE
150     DATA LOAD DC OPEN F "INVTORY"
160 REM LOOP TO READ EACH DISK RECORD
170     DATA LOAD DC N$(), D$(), S$(), Q(), R()
175         IF END THEN 210
176         REM LOOP TO PRINT EACH DATA RECORD
177         FOR K = 1 TO 3
178             IF N$(K) = "↑↑↑↑↑↑↑↑↑↑" THEN 210 :REM PADDING?
180             PRINTUSING 200, N$(K), D$(K), S$(K), Q(K), R(K)
185         NEXT K
190     GOTO 170
200     % ##########    #####################################
#    ######    ##,###    ##,###
210     DATA SAVE  DC CLOSE
```

In Example 20.8 notice that within the larger loop that reads the disk records (lines 170-190) another loop has been nested. For each DATA LOAD DC (line 170), this inner loop prints the three records.

There are two ways for this program to end. If the number of product records divides evenly by 3, then IF END THEN (line 175) will be the loop exit. Otherwise, line 178 will detect padding characters and cause an exit.

To add records to the end of a file, when using blocked records, the last block can be found by skipping to the end of the data and then backspacing one sector (as shown in Example 20.5). However, the program must then test to see if the block is filled with product records, or has some padded records. If records are padded, then the new records must replace the padded records in the last block. If the block is filled, then the new records must begin a new block.

To access a specified product record with blocked files such as these, the number of sectors to skip must be calculated, as must the record's subscript within the block. The relationship between these three values for the first 8 products is as follows:

Consecutive Portion of Product Number	Number of Sectors to Skip	Subscript of Product Record
1	0	1
2	0	2
3	0	3
4	1	1
5	1	2
6	1	3
7	2	1
8	2	2

The number of sectors to skip, B, is given by

$$B = INT((N-1)/3)$$

where: N is the consecutive portion of the product number.

The subscript is given by

$$K = N - (B*3)$$

where: B and N are defined as above.

Programs that add records to the end of this file and update the quantity on hand balance of selected records are given in Appendix E.

An additional consideration when planning file size is whether the system of file organization will permit new records to replace old records deleted from the file.

In the simple consecutive file organization which we have been discussing here, it would not be too difficult to let new products replace old ones in the file, provided that the reassignment of an old product number would not cause any problems external to the system. Unless a single program was used to mark "deleted" records (by filling the description with asterisks or some such technique) and to add new records, a list of available deleted records would have to be maintained. In the next section we discuss the general problem of accessing records, and will return to the topic of replacing deleted records with new ones from a broader perspective.

266

20-9 RECORD ACCESS TECHNIQUES

In Section 20-2 we mentioned that the catalog index contains the starting and ending sector addresses of each file saved on the disk using Catalog Mode statements. However, it does not contain the sector address of each individual data record in a file. Therefore, some additional means of determining the sector address of a desired record must be employed, whenever data is to be saved in or loaded from a file.

When records in a file can be processed in their physical sequence, then the automatic updating of the Current Sector address, by DATA SAVE DC and DATA LOAD DC, can be used to supply the sector address of each record. A simple example of this was shown in Examples 20.2 and 20.4.

The situation becomes more complicated when sequential accessing of records is impractical. Example 20.6, which allows selected records to be updated, is an example of nonsequential accessing. In this example a portion of the product number directly identifies the record's location in the file. The record is accessed by extracting this portion of the product number, and "skipping" the specified number of sectors from the beginning of the file.

In general when accessing a file, as in Example 20.6, the known value that is used to identify the desired record is called the "key" or "key field." (Sometimes a "key" may be only a portion of a field, as it is in Example 20.6.) Example 20.6 assumed that the key field could be specified to fit the needs of the system, that is, each new product is assigned the next consecutive number. Often it is impossible or impractical to specify keys in this fashion; the keys are determined by considerations external to the system. For example, it may be necessary to access an employee record, given the social security number as a key, or an accounts payable record given the invoice number as a key. In situations such as these, there are a great many different ways of approaching the problem, but they may be divided into two classes: (1) solutions that involve the use of another, specially structured data file that acts as an index to the data file containing the records to be accessed, and (2) solutions that do not use a separate index file. These generally depend upon the data file having been sorted on the key fields, or upon the discovery of a formula for converting the key to a sector address.

When solutions of the first type are employed, the specially structured index file contains some or all of the keys from the file to be accessed (the object file). In the index file each key is associated with the sector address of its record in the object file. The index file is structured so that, given a key, the index entry for that key can be rapidly found. The address in the index is then used to access the object record, or one sufficiently close to the object record so that a minimal amount of searching is needed.

Writing programs to create and maintain an index file can be a highly complex programming task. The associated techniques are outside the scope of this volume. However, a general purpose keyed file accessing system is available from Wang Laboratories. Called KFAM, it creates and maintains an index file for the records in a user's data file. The index file contains an entry for each record in the user's file. The entry contains the record key,

its sector address, and, for blocked records, its position within the block. Included in the KFAM system are DEFFN' subroutines which, when passed a key, automatically search the index file and set the Current Sector address to the address of the record in the user's data file. Thus, after passing the key of the desired record to the KFAM subroutine, the user's program can simply execute a DATA LOAD DC to obtain the record.

The structure of the KFAM index is such that it can be efficiently used for files that frequently have new records added or old ones deleted. In addition to random accessing of records, it permits rapid processing of the user's file in key sequence. (In a user file accessed with KFAM, the physical sequence of user data records will generally not be key sequence.)

The techniques employed for finding records when an index file is not used are beyond the scope of this volume, except for simple cases such as the inventory system shown above in which a portion of the key is directly related to the sector address of the record. The beginning programmer who needs to have random access to file records should use the KFAM system.

When using KFAM, it is possible to reuse the space occupied by deleted records in the user's data file. However, it may be necessary to maintain a separate file that contains a list of the sector addresses of deleted records. For more information about this, see the KFAM manual.

20-10 MULTIPLE OPEN FILES

Many data processing tasks require that data from several files be accessed to complete a single operation. For example, a program to process customer orders for merchandise might require that, for each order, the inventory file be updated, the customer (accounts receivable) file be updated, and perhaps the salesperson's file as well.

With the disk catalog capabilities we have outlined thus far, such an operation would require that a DATA LOAD DC OPEN statement be executed each time the program goes from working on one file to working on the next file. Each time this is done, the new starting, ending, and Current Sector addresses replace the old ones in the Device Table. This gets the job done, but can be quite awkward and inefficient. For example, it means that repeated searches of the catalog index must be made, since this is what DATA LOAD DC OPEN does. Furthermore, if one of the files is being accessed sequentially, the convenience of having the Current Sector address automatically updated by DATA SAVE DC and DATA LOAD DC is lost, since opening another file replaces the Current Sector address with the starting address of the new file.

Thus far we have considered the Device Table as, for example,

Starting Sector Address	Ending Sector Address	Current Sector Address
28	228	28

Implicitly, the device address and platter parameter has been associated as well, so that a more explicit picture of the Device Table might have looked like this:

Device Address	Platter Parameter F or R	Starting Sector Address	Ending Sector Address	Current Sector Address
310	F	28	228	28

The device address was provided by Master Initialization or SELECT DISK, and the platter parameter was specified by the F or R in DATA LOAD DC OPEN or DATA SAVE DC OPEN. These two statements also provided the starting, ending and Current Sector addresses for the table.

The Device Table as shown above contains all the information needed by the processor in order for it to access a disk sector, that is, the device address, the platter (F or R), and the sector address on that platter.

If the Device Table consisted of just one row, or "slot" as it is sometimes called, then in order to access different files, DATA LOAD DC OPEN would have to be executed repeatedly, as discussed above. In fact, however, the Device Table contains seven rows exactly like the one shown above, and permits seven files to be "open" simultaneously. The rows are numbered 0-6. Thus the actual entire Device Table looks like this:

Row or "File" Number	Device Address	Platter Parameter (F or R)	Starting Sector Address	Ending Sector Address	Current Sector Address
#0	310	undefined	0	0	0
#1	000	undefined	0	0	0
#2	000	undefined	0	0	0
#3	000	undefined	0	0	0
#4	000	undefined	0	0	0
#5	000	undefined	0	0	0
#6	000	undefined	0	0	0

Figure 20.1   The Device Table after Master Initialization

Notice in Figure 20.1 that address 310 is saved by Master Initialization in Device Table row #0. Row #0 is the row we have been using in all the preceding examples. We were able to do this because row #0 is used by default if a Device Table row is not explicitly specified in a disk statement. In the general form of each of the statements we have considered, a specific device table row can be specified. Thus expanded, the general forms look like this:

269

DATA SAVE DC OPEN $\left\{ \begin{matrix} F \\ R \end{matrix} \right\}$ [#n,] expression, name

DATA LOAD DC OPEN $\left\{ \begin{matrix} F \\ R \end{matrix} \right\}$ [#n,] name

DATA SAVE DC CLOSE $\left[\begin{matrix} \#n \\ ALL \end{matrix} \right]$

DATA SAVE DC [#n,] $\left\{ \begin{matrix} END \\ \text{argument list} \end{matrix} \right\}$

DATA LOAD DC [#n,] argument list

DSKIP [#n,] $\left\{ \begin{matrix} END \\ \text{expression [S]} \end{matrix} \right\}$

DBACKSPACE [#n,] $\left\{ \begin{matrix} BEG \\ \text{expression [S]} \end{matrix} \right\}$

For example, the statement

        DATA LOAD DC OPEN F "INVTORY"

is equivalent to

        DATA LOAD DC OPEN F #0, "INVTORY"

Each of these two statements specifies that the address of the disk device is
to be   found in   row #0 in the Device Table, and that, when the data file
"INVTORY" is opened, row #0 is to receive the platter parameter, F, and the
file's sector addresses.

270

Now suppose that we had opened the inventory file in this manner and wish to open, in addition, a customer (accounts receivable) file. At this point, with just the inventory file open, the Device Table looks like this:

Row or "File" Number	Device Address	Platter Parameter (F or R)	Starting Sector Address	Ending Sector Address	Current Sector Address
#0	310	F	028	228	028
#1	000	undefined	0	0	0
#2	000	undefined	0	0	0
#3	000	undefined	0	0	0
#4	000	undefined	0	0	0
#5	000	undefined	0	0	0
#6	000	undefined	0	0	0

Before we can open a file in one of the other rows of the Device Table, we must first put a device address into the row. The statement used to put an address into one of the rows #1 through #6 is

SELECT 'file symbol' 'device address' ['file symbol' 'device address'...]

where: 'file symbol' is one of the following: #1,#2, #3, #4, #5, #6.
and 'device address' is the device address to be put into the row.

For example, if the customer file were on a disk mounted in the same disk unit as the inventory file, the program would execute

190 SELECT #1 310

This statement would put the address 310 into row #1 of the Device Table.

Now a statement such as

200 DATA LOAD DC OPEN R #1, "CUSTOMER"

would be used to open the file called "CUSTOMER" on the R disk, and put the R parameter, starting sector address, ending sector address, and Current Sector address into row #1 of the Device Table (Current Sector set to starting sector address). After this statement the Device Table might look like this:

Row or "File" Number	Device Address	Platter Parameter (F or R)	Starting Sector Address	Ending Sector Address	Current Sector Address
#0	310	F	28	228	28
#1	310	R	485	960	485
#2	000	undefined	0	0	0
#3	000	undefined	0	0	0
#4	000	undefined	0	0	0
#5	000	undefined	0	0	0
#6	000	undefined	0	0	0

With both of these files open, the customer file, specified in row #1, can be
operated on with statements such as:

```
210 DATA LOAD DC #1, A$, R, K, Q$(), N
    .
    .
    .
270 DBACKSPACE #1, 2S
280 DATA SAVE  DC #1, A$, R, K, Q$(), N
    .
    .
    .
330 DSKIP #1, END
    .
    .
    .
420 DBACKSPACE #1, BEG
```

In each case the #1 in the statement says that the specification of the device
address, platter, and sector addresses is to be found in row #1 of the Device
Table. To operate on the inventory file, the symbol #0 could be included in a
statement, or the specification may be omitted, since #0 is used automatically
if no row is specified. Up to seven files may be "open" simultaneously by
using all seven rows of the Device Table. The files may be located on the
same disk, at different disks mounted in the same drive (F or R), or at disks
mounted in different drives.

In the SELECT statement a 'file symbol' must be specified by the #
symbol and a digit 1-6. A variable may not be used, nor may a variable be
used to contain the device address. However, in disk statement references to
a Device Table row, the # symbol may be followed by a numeric variable. The
value of the variable (0, 1, 2, 3, 4, 5, 6) then determines which Device Table
row is used.

Example 20.9 shows how several open files can be accessed from the same
program.

Example 20.9 Accessing Records in Several Open Files

```
110 REM ACCESSING SEVERAL OPEN FILES

130     SELECT DISK 310, #1 320, #2 320
140     DATA LOAD DC OPEN F "INVTORY"
150     DATA LOAD DC OPEN R #1, "CUSTOMER"
160     DATA LOAD DC OPEN F #2, "SALES"

310 REM UPDATE INVENTORY FILE
320     DSKIP K S
330     DATA LOAD DC N$(), D$(), S$(), Q(), R()

350     DATA SAVE DC N$(), D$(), S$(), Q(), R()

480 REM UPDATE CUSTOMER FILE
490     DBACKSPACE #1, BEG
500.    DSKIP #1, J S
510     DATA LOAD DC #1, A$, R, K2, A$(), N

530     DATA SAVE  DC #1, A$, R, K2, A$(), N

600 REM UPDATE SALESPERSON FILE
610     DATA LOAD DC #2, E4$(), N2$(), G(), C()

630     DATA SAVE  DC #2, E$(), N2$(), G(), C()

940     DATA SAVE  DC CLOSE ALL
```

In Example 20.9 notice that a single SELECT statement (line 130) is used to select addresses for several rows in the Device Table. The first part of the SELECT statement, DISK 310, is used to ensure that address 310 is in row #0 of the Device Table. Master Initialization, of course, puts 310 in row #0, but a different address could have been selected by an intervening program. It is not possible to assign a device address to row #0 with the form

SELECT #0 310

The word DISK must be used.

Lines 140 to 160 open the files at the selected addresses, thereby assigning the platter parameter (F or R) and the sector addresses to the proper rows in the table. Notice that #0 is not specified, but is used by default for statements 140 and 320-350. The other statements contain explicit references to the device table row so that the proper file is accessed.

The statement shown at line 940

940 DATA SAVE DC CLOSE ALL

273

closes all the open files by filling the Device Table sector addresses with zeros. It does not affect the device addresses, which remain as selected by line 130. To close a particular file, a statement of this form can be used:

DATA SAVE DC CLOSE #n

where: n is a constant, or numeric variable with a value of 0, 1, 2, 3, 4, 5, or 6.

On the 2270-3 disk drive unit, the leftmost and middle diskette ports are identified by means of the device address and the F or R platter parameter (F referring to the left port, R the middle). However, the third or rightmost diskette port is treated as if it were a separate disk drive unit. It has its own disk address and is referred to by means of this address and the F platter parameter. The device address of this third port can be determined by adding 40 to the device address of the other two ports. Thus, if the two main ports are identified by 310, F and R, the rightmost port will be identified by 350, F only. If the left and middle ports are 320, F and R, the rightmost will be 360, F only, etcetera.

20-11 THE "T" PLATTER PARAMETER

It is possible to let an operator choose the device address at which a disk file is mounted. For example,

```
130 PRINT "1. 310", "2. 320"
140 INPUT "ENTER 1 OR 2 TO CHOOSE DEVICE ADDRESS",R
150 ON R GOTO 180, 190
160 PRINT "INVALID. REENTER"
170 GOTO 140
180 SELECT DISK 310
190 GOTO 210
200 SELECT DISK 320
210 DATA LOAD DC OPEN F "INVTORY"
```

With the techniques we have considered thus far, it is not possible to let an operator choose whether a disk is to be mounted at the F or R disk locations, at any given disk unit. This is because statements such as 210, above, have contained a fixed F or R platter parameter. However, there exists a third alternative to F and R, which may be used in place of these platter parameters. This is the T parameter; it allows either the F or R disk to be accessed with the same disk statement.

In order to execute a statement such as

210 DATA LOAD DC OPEN T "INVTORY"

the system must have some way of determining whether "INVTORY" is to be found at the F or R disks of the selected disk drive. It does this by looking at the first character of the device address.

All the disk device addresses we have mentioned thus far have had a 3 as the first character (310, 320, 350, etcetera). However, the same physical disk drive units can also be addressed if the 3 is replaced with a B. A complete device address, such as 310, is actually a composite, in which the last two characters are the address of the physical disk drive unit, and the first character is a special code, often called the "Device Type" code. For example, with the addresses 310 and B10

```
┌─────────────────────────────┐   ┌─────────────────────────────┐
│        ╱ 310 ╲              │   │        B10 ╲                │
│       ╱       ╲             │   │       ╱      ╲              │
│      3         10           │   │      B        10            │
│    ‿‿‿       ‿‿‿           │   │    ‿‿‿       ‿‿‿           │
│      │         │            │   │      │         │            │
│      ↓         ↓            │   │      ↓         ↓            │
│   Special    Unit          │   │   Special    Unit           │
│   code       Address       │   │   Code       Address        │
│   called                   │   │   called                    │
│   "Device                  │   │   "Device                   │
│   Type"                    │   │   Type"                     │
└─────────────────────────────┘   └─────────────────────────────┘
```
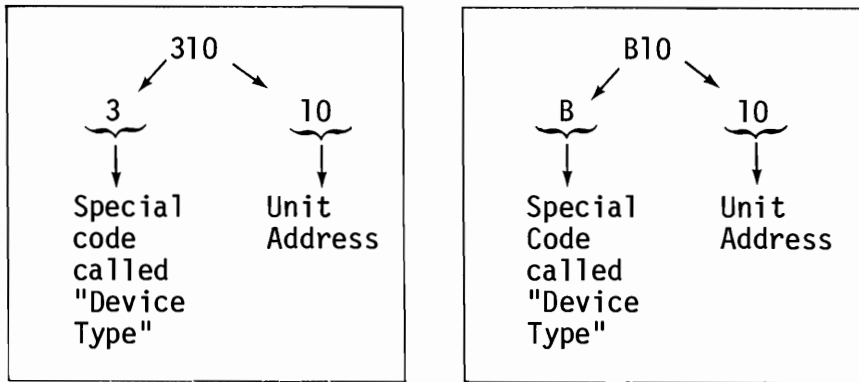
Since 310 and B10 have the same unit address, either may be used to address the same disk unit. If F or R is specified in a disk statement, such as

DATA LOAD DC F "INVTORY"

the device address may be specified in the Device Table as 310 or B10, with exactly the same results. However, when the T parameter is used instead of F or R, then the 3 in the device address 310 indicates that the F disk at unit address 10 is to be used; alternatively the B in the device address B10 indicates that the R disk at the same unit, unit 10, is to be used. Thus, by using the T parameter, a disk statement such as

DATA LOAD DC OPEN T "INVTORY"

is completely generalized as to the location where INVTORY is to be found. The location is specified entirely by the device address, whose last two characters indicate the unit, and whose first character indicates the platter.

The sequence

110 SELECT #1 B10
120 DATA LOAD DC OPEN T#1, "INVTORY"

causes a search of the catalog index of the R disk at unit 10 for a file called "INVTORY". When it finds the index entry, it puts the starting and ending sector addresses of the file into row #1 of the Device Table, sets the Current Sector address equal to the starting sector address, and puts an R into the platter parameter column (since the T in the statement and the B in the address together indicate the R platter).

275

In common parlance, the F platter of the disk unit 10 is referred to as address 310, while the R platter is referred to as address B10. When using this terminology, it is important to be aware that these device addresses, 310 and B10, carry this platter distinction only for disk statements in which the T parameter is used; otherwise 310 and B10 are functionally identical.

Example 20.10 shows a portion of a program that allows an operator to choose the location of a disk file. The T parameter is used so that 310 and B10, 320 and B20, specify different disk platter locations.

Example 20.10   Operator Selection of Disk File Location

```
110 REM OPERATOR SELECTION OF A SINGLE DISK FILE LOCATION
120     DIM P$64
130     SELECT DISK 310, #1 320, #2 350, #3 B10, #4 B20
140     PRINT HEX(030A0A0A0A0A) :REM CLEAR CRT POSITION CURSOR
150     PRINT , "1. 310", "4. B10"
160     PRINT , "2. 320", "5. B20"
170     PRINT , "3. 350"
180     GOSUB '100 ("ENTER 1 - 5 FOR ADDRESS OF 'INVTORY' FILE",
    5, 1, 0) :REM NUMERIC ENTRY
190     F1 = X - 1 :REM ASSIGN THE FILE NUMBER TO F1
200     DATA LOAD DC OPEN T#F1, "INVTORY"
    .
    .
    .
3000 REM NUMERIC ENTRY SUBROUTINE
3010     DEFFN' 100 (P$, U, L, D)
3020     PRINT HEX(010A); P$
3030     INPUT X
3040     IF X > U THEN 3070 :REM TOO HIGH?
3050     IF X < L THEN 3070 :REM TOO LOW?
3060     IF INT(X*10↑D) = X*10↑D THEN 3090 :REM # DECIMALS OK?
3070     PRINT HEX(0C); TAB(64); "INVALID. REENTER."
3080     GOTO 3020
3090     RETURN
```

Line 130 assigns each of the available disk addresses to a row of the Device Table. This changes the programming problem from one of choosing an address to choosing the desired row of the Device Table. Lines 140 to 170 display the available addresses. Line 180 passes control to a numeric entry subroutine that displays a prompt and validates the operator entry. The returned variable, X, contains a number (1 to 5) indicating the selection number of the desired address (see lines 150-170). For each address, the selection number is 1 greater than the row in the Device Table which was assigned the address. Line 190, therefore, assigns to variable F1 the appropriate row number. The numeric variable F1 is then used to specify the Device Table row.

Since, with this form of selection, the Device Table row which is used is unknown until execution, a different technique must be used if multiple files are to be open simultaneously. Otherwise, if both files were on the same disk, opening one would close the other. A program that allows addresses for two files to be selected is shown in Example 20.11.

Example 20.11 Selecting Locations for Multiple Files

```
110 REM OPERATOR SELECTION OF DISK FILE LOCATIONS
120     DIM P$64
130 REM KEYBOARD SELECT OF 'INVTORY' DISK ADDRESS
140     GOSUB ' 182 ("ENTER 1 - 5 FOR 'INVTORY' ADDRESS")
150     ON X-1 GOTO 170, 180, 190, 200
160     SELECT DISK 310: GOTO 210
170     SELECT DISK 320: GOTO 210
180     SELECT DISK 350: GOTO 210
190     SELECT DISK B10: GOTO 210
200     SELECT DISK B20:
210 REM OPEN FILE
220     DATA LOAD DC OPEN T#0, "INVTORY"
230 REM KEYBOARD SELECT OF "CUSTOMER" DISK ADDRESS
240     GOSUB ' 182 ("ENTER 1-5 FOR 'CUSTOMER' DISK ADDRESS")
250     ON X-1 GOTO 270, 280, 290, 300
260     SELECT #1 310: GOTO 310
270     SELECT #1 320: GOTO 310
280     SELECT #1 350: GOTO 310
290     SELECT #1 B10: GOTO 310
300     SELECT #1 B20:
310 REM OPEN FILE
320     DATA LOAD DC OPEN T#1, "CUSTOMER"
   .
   .
   .
2000 REM SUBROUTINE FOR KEYBOARD SELECTION OF DISK ADDRESS
2010    DEFFN' 182 (P$)
2020      PRINT HEX(030A0A0A0A0A) :REM CLEAR CRT POSITION CURSOR
2030      PRINT , "1. 310", "4. B10"
2040      PRINT , "2. 320", "5. B20"
2050      PRINT , "3. 350"
2060      GOSUB '100 (P$, 5, 1, 0) :REM NUMERIC ENTRY
2070      RETURN
3000 REM NUMERIC ENTRY SUBROUTINE
3010    DEFFN' 100 (P$, U, L, D)
3020      PRINT HEX(010A); P$
3030      INPUT X
3040      IF X > U THEN 3070 :REM TOO HIGH?
3050      IF X < L THEN 3070 :REM TOO LOW?
3060      IF INT(X*10↑D) = X*10↑D THEN 3090 :REM # DECIMALS OK?
3070      PRINT HEX(0C); TAB(64); "INVALID. REENTER."
3080      GOTO 3020
3090      RETURN
```

The T parameter can be useful even when operator selection of addresses is not needed. For example, even though a particular data file may seemingly always be mounted at the R disk, there may come a time when, due to system expansion or other factors, it would be convenient to be able to mount it at some other location. If the programs which operate on the file are written using the T parameter, then only the address in one SELECT statement need be changed to modify the program for operation at a different location. By contrast, if F or R is used in the statements, then all F or R references must be changed.

CHAPTER 21
DATA STORAGE ON TAPE CASSETTES

21-1 <u>OVERVIEW OF CASSETTE DATA FILE OPERATIONS</u>

In Section 4-1 we introduced the use of tape cassettes for program storage. We said that a single program saved on cassette constitutes a "file", and that a cassette may be used to save many such program files, the exact number depending upon the length of the cassette tape and the sizes of the programs. In addition to saving program files, cassettes can be used to save data files.

A data file is a collection of information about a topic. Within a file this collection consists of one or more "records". For example, a test results file might be a collection of the records from a particular test run, one record per test. A record for statistical analysis might consist of an X value and a Y value, the coordinates of a point in the plane; a file of such records might define a statistical population.

Regardless of the content of the data file, Wang BASIC allows you to easily create cassette data files, save data records, read data records, skip forward and back over records and files, and update individual data records. Table 21.1 provides an overview of the functions performed by the cassette statements for standard-format data storage and retrieval operations.

Table 21.1 The Tape Data File Statements

STATEMENT	FUNCTION
1. a) DATA SAVE OPEN "file name"	Saves onto the cassette tape a special "header" record which marks the beginning of a data file. This special record contains the name of the file.
b) DATA SAVE	Takes values from memory and saves them as one record on a cassette.
c) DATA SAVE END	Saves a special "trailer" record which marks the end of a data file.

2. a) DATA LOAD "file name"   Searches forward through a cassette for the "header" record of a specific file.

   b) DATA LOAD   Reads values from a tape cassette record, and assigns the values to variables in memory.

3. a) SKIP END   Searches forward for the next special "trailer" record.

   b) SKIP n   Skips forward over n records on tape.

   c) SKIP nF   Skips forward over n files on tape.

4. a) BACKSPACE BEG   Searches backwards for a header record.

   b) BACKSPACE n   Backspaces over n records.

   c) BACKSPACE nF   Backspaces over n files.

5. a) DATA RESAVE OPEN   Saves a new special header record which replaces (updates) the header record of an existing data file.

   b) DATA RESAVE   Takes values in memory and saves them as one record on cassette, replacing (updating) an existing data record.

A single cassette can be used for just one file or for many files. Program files and data files can be saved on a single cassette; however, many programmers prefer to store program files and data files on separate cassettes.

Unless otherwise specified, all cassette operations occur at the device whose address is selected for TAPE class I/O operations. This TAPE class address is set to 10A by Master Initialization, and may be changed at any time by executing a SELECT statement with a TAPE address specified. For example,

      :SELECT TAPE 10B

sets the TAPE class address to 10B.

In addition to the TAPE class parameter, there are two other ways of specifying the address at which a cassette operation is to take place. These are discussed in Section 21-8. However, when these other techniques are not used, the system defaults to the TAPE address.

All the tape cassette statements discussed in this chapter can be executed in the Immediate Mode .

21-2 MARKING THE BEGINNING OF A FILE WITH DATA SAVE OPEN

The DATA SAVE OPEN statement is used to record a special header record that marks the beginning of a data file. This special header record contains the name of the file. The maximum length of the name is eight characters. For example, the statement

120 DATA SAVE OPEN "TESTA114"

records a special header record at the current tape location, with the file name "TESTA114".

It is not strictly necessary to save a header record at the beginning of a data file. However, the presence of a header record makes it easier to carry out certain operations. For example, it enables the system to search forward through a cassette tape for the beginning of a named file, or to backspace to the beginning of a file from any location within the file. In general, if you wish to save several data files on one cassette, you will want to mark the beginning of each of the files with the special header record created by DATA SAVE OPEN. If you plan to put just one file on a cassette, the header record may still be useful for cassette handling and control purposes, since it lets you easily record the name of the file onto the cassette. Finally, some of the utility programs supplied by Wang Laboratories require that data files have header records.

21-3 SAVING DATA RECORDS

The DATA SAVE statement records data on a cassette tape, and collectively marks off the recorded data as one "record." It starts recording at the current position of the tape. For example,

100 DATA SAVE X,Y

causes the values of the variables X and Y to be saved on the cassette tape. In addition to saving the specified values, DATA SAVE surrounds the saved values with certain control information used by the system. This control information collectively marks off the values as a record.

Values to be saved by a DATA SAVE statement can be specified in any of the following ways:

a) A numeric variable, e.g., A, B1, C5, D.

b) An alphanumeric variable, e.g., A$, B2$, M4$.

c) A specific element of a one-dimensional numeric or alphanumeric array, e.g., A(2), F1(3), X(10), B$(4), G3$(5).

d) A specific element of a two-dimensional numeric or alphanumeric array, e.g., A(3,4), Y5(2,9), A(1,15), C$(5,7), D1$(4,9).

e) An array designator (an array name followed by a left And a right parenthesis), e.g., A(), P$(), M4$(). The entire array of values is saved.

f) A mathematical expression, e.g., X*Y-Z, SQR(A↑2 + B↑2), 2*D, X. The expression is evaluated and the result is saved.

g) A string function or hexadecimal function, e.g., STR(A$,3,8), HEX(22), HEX(0D0A).

h) A literal string, e.g., "WANG LABORATORIES".

Any number of values may be specified in the DATA SAVE statement. Each value specification, or "argument", must be separated from the next by a comma. Values are saved in the sequence in which they appear in the DATA SAVE statement. If an entire array is specified using an array designator, for example,

    100 DIM R(2,3)
    140 DATA SAVE R()

the values in the array are saved row by row. Thus, in the above example, the values are saved in this sequence: R(1,1), R(1,2), R(1,3), R(2,1), R(2,2), R(2,3).

Regardless of how a value is specified in a DATA SAVE statement, only the value is saved, not the name of the variable or array, nor the quotation marks.

Example 21.1 allows the operator to enter the X and Y coordinates of points in a plane. The values for each point are saved as a record. The record also contains, as an alphanumeric value, the consecutive number of the record.

Example 21.1 A Program That Creates a Tape Data File

```
110 REM CREATING A SIMPLE DATA FILE
120     DIM R$1, N$4
130 REM NAME FILE IN HEADER RECORD
140     DATA SAVE OPEN "POINTS1"
150 REM SET UP FOR RECORD
160     N = N + 1 :REM RECORD COUNTER
170     PRINT HEX(03); "KEY S.F. 31 TO END PROGRAM"
180     PRINT "RECORD NUMBER ="; N
190     PRINT
200 REM RECEIVE RECORD VALUES
210     INPUT "X VALUE", X
220     INPUT "Y VALUE", Y
230     PRINT
```

```
240 REM OPERATOR CHECK OF VALUES
250     R$ = " " :REM NO DEFAULT ENTRY
260     PRINT "CHECK VALUES"
270     INPUT "ENTER + TO ACCEPT, - TO REJECT", R$
280     IF R$ = "-" THEN 170 :REM REJECTED?
290     IF R$ = "+" THEN 330 :REM ACCEPTED?
300     PRINT HEX(OC); TAB(64); HEX(OCOC): REM OPERATOR ERROR
310     GOTO 260
320 REM SAVE RECORD
330     CONVERT N TO N$,(####)
340     DATA SAVE N$, X, Y
350     GOTO 160
360 REM END PROGRAM
370     DEFFN' 31
380     PRINT HEX(03)
390     END
```

Line 140 writes a header record which contains the file name "POINTS1". The display shows the consecutive record number and requires operator verification of each set of values (170-310). Line 340 saves the record number and the X and Y values of the point.

Lines 160-350 form a loop. The operator ends this loop by keying Special Function key 31, which effects a branch out of the loop to line 370, and ends the program.

There are several deficiencies in this simple program. One of these deficiencies is that the end of the file is not marked. This problem is discussed in the next section.

21-4 MARKING THE END OF A DATA FILE

The statement DATA SAVE END is used to mark the end of a cassette data file. It writes a record that has special significance to certain other BASIC statements.

The use of DATA SAVE END is not mandatory; however, as we shall see in the next several sections, it is often very useful to have the end of the file marked with DATA SAVE END.

In Example 21.1, the DATA SAVE END statement can simply be added to the "end program" routine as follows:

```
375 DATA SAVE END
```

In future references to Example 21.1, we shall assume that this line has been added to the program.

21-5   LOADING DATA FROM A FILE

DATA LOAD "file name"

Before data can be loaded from a data file that has a header record marking its beginning, the header record must be read. The statement,

        DATA LOAD "file name"

searches forward through a cassette tape for the header record of the specified file. After it reads the specified header record, it leaves the tape positioned to read the first actual data record.

For example, the statement

        150 DATA LOAD "POINTS1"

searches a cassette tape until it reads the header record of the file "POINTS1". It leaves the tape positioned so that the first record can be read.

The "file name" parameter in this statement must be a literal string containing the file name, as saved by DATA SAVE OPEN.

DATA LOAD

The DATA LOAD statement reads values from a tape cassette record (or records), and assigns them to specified variables.

For example, the statement

        DATA LOAD N$, X, Y

causes the cassette drive to start reading values from the next record. The values are assigned successively to the variables N$, X, and Y. The system reads values until all the specified variables have been assigned a value. After the last variable is assigned a value, the tape is positioned so that it is ready to read the first value of the next record.

The assignment of values in DATA LOAD takes place just as if it were performed in an assignment (LET) statement. For example, if an alphanumeric variable is too short to contain an entire alphanumeric value, the assignment is made, but the extra characters on the right are lost. Conversely, if the variable is longer than the value, it is padded with spaces on the right. It is especially important to be aware that an error results if a numeric value, encountered in the record, is matched with an alphanumeric variable in the DATA LOAD statement, or vice versa. For this, and other reasons, it is important that precise documentation be maintained of data file record layouts.

Values may be assigned to an entire array by specifying the standard form array name (array designator) in the DATA LOAD statement. For example, the statement

DATA LOAD A$(), P()

reads values from a record or records and assigns them element by element, row by row to the array A$(), until each element in the array has been assigned a value. Then, the next values are assigned in the same way to P().

The same variables need not be used to receive the values as were originally used to save them. The only requirement is that numeric values be loaded into numeric variables or numeric array elements, and alphanumeric values be loaded into alphanumeric variables or array elements.

Usually, it is good programming practice to read exactly one record with one DATA LOAD statement. For example, the statement

DATA LOAD N$, X, Y

specifies that three values be read and assigned successively to one alphanumeric variable, and two numeric ones. It therefore reads exactly one record as written by statement 340 of Example 21.1.

A DATA LOAD statement need not read exactly one record. If fewer variables are specified in the DATA LOAD statement than there are values in the record, the extra values are ignored. If more variables are specified than there are values in the record, successive values from the next record(s) will be assigned to the variables. It must be emphasized, though, that after a DATA LOAD statement, the cassette tape is positioned to read the next record, even if only a portion of the previous record has been assigned to variables. It is impossible to begin reading values in the middle of a record with the DATA LOAD statement.

Example 21.2 shows a simple program for reading and displaying the data file created by Example 21.1 (with the addition of the DATA SAVE END record discussed in Section 21-4).

Example 21.2 Reading and Displaying the Data File Records

```
110 REM PRINTING THE DATA FILE
120     DIM N$4
130     SELECT PRINT 005
140 REM FIND FILE
150     DATA LOAD "POINTS1"
160 REM SET UP FOR OUTPUT
170     PRINT HEX(03)     :REM CLEAR CRT
180     PRINTUSING 300:    REM HEADINGS
```

```
190 REM READ AND PRINT RECORDS
200      FOR L = 1 TO 14
210          DATA LOAD N$, X, Y
220          IF END THEN 280  :REM NO MORE RECORDS?
230          PRINTUSING 310, N$, X, Y
240      NEXT L
250      INPUT "KEY EXEC FOR MORE LINES", R$
260      GOTO 170
270 REM END
280      PRINT "END OF DATA FILE"
290      REWIND
300 %    RECORD NO.        X VALUE          Y VALUE
310 %       XXXX         #,###.###        #,###.###
```

Statement 150 searches the cassette tape for the header record of "POINTS1", and positions the tape so that it is set to read the first data record. This program is designed to operate on a 16 line CRT. The headings are output on line zero by statement 180. The FOR/NEXT loop reads and outputs 14 records, filling all but the bottom line of the CRT. On the bottom line a prompt appears (statement 250) informing the operator that the next 14 records can be displayed by keying (EXEC).

Example 21.2 reveals the importance of having an end-of-file trailer record. When a DATA LOAD statement is executed, if the system reads the special record saved by DATA SAVE END, several things happen. First of all, the variables specified in the DATA LOAD statement, which would have been assigned values had a normal data record been encountered, instead retain their current values. Secondly, the DATA LOAD statement is terminated, and the tape is repositioned so that the end-of-file trailer would be immediately re-read on a subsequent DATA LOAD execution. Finally, a notation is made in a special part of memory that an end-of-file trailer record has been read.

The IF END THEN Statement

A special BASIC statement is available to test if an end-of-file trailer has been read. The form of this statement is

IF END THEN line number

The IF END THEN statement checks the special part of memory to see if an end-of-file trailer has been read during the last DATA LOAD statement. If it has been read, IF END THEN effects a branch to the line number following "THEN". Thus, the IF END THEN statement can be used to exit from a record reading loop when the end of a file is reached (provided that the end of the file is marked with the DATA SAVE END trailer record). IF END THEN may not be used in the Immediate Mode.

In Example 21.2 the IF END THEN statement is used to exit from the record reading loop (line 220) when the tape has reached the end-of-file trailer.

It should be noted that the REWIND statement, line 290, is included at the end of this program as an operator convenience. Tape cassettes should not be removed from the drive unless they are rewound.

21-6   THE SKIP AND BACKSPACE STATEMENTS

The SKIP and BACKSPACE statements are used to move the cassette tape a specific distance forward or back, without reading the intervening values. Each of these statements has three forms whose functions are summarized below.

The SKIP Statement:

a)   SKIP END

Searches forward for the next DATA SAVE END trailer record. Positions the tape so that the tape drive head is in front of the trailer record.

b)   SKIP n
     where n = expression

Skips forward over the number of records specified by the truncated value of n. If a trailer record is encountered, the tape is positioned so that the tape drive head is in front of the trailer record.

c)   SKIP nF
     where: n = expression

Skips forward over the number of trailer records specified by the truncated value of n. The tape is positioned so that the tape drive head is immediately beyond the nth trailer record. (Note: Program file trailer records as well as data file trailer records are counted by SKIP.)

The BACKSPACE Statement:

a)   BACKSPACE BEG

Searches backwards for a DATA SAVE OPEN header record. Positions the tape so that the tape drive head is at the end of the header record, that is, in front of the first data record in the file.

b)   BACKSPACE n
     where n = expression

Backspaces over the number of records specified by the truncated value of n. If a header record is encountered, the tape is positioned so that the tape drive head is at the end of the header record, in front of the first data record in the file.

c)   BACKSPACE nF
     where n = expression

Backspaces over the number of header records specified by the truncated value of n. The tape is positioned so that the tape drive head is set to read the nth header record.

The SKIP END statement is particularly useful when records must be added to the end of a data file (provided, of course, that another data file does not follow the end-of-file trailer). To add records, a program can simply execute a SKIP END statement and begin saving the new records with DATA SAVE. The program should write a new trailer record, when all the new records have been added.

286

Example 21.3 illustrates a possible use for skipping and backspacing over individual records in a file. It operates on the file created by Example 21.1, with the DATA SAVE END trailer present. The operator enters a record number, the program then SKIPs or BACKSPACEs the required number of records to find the desired record, and prints the record, when it is found.

Example 21.3 Printing Selected Records from a File

```
110 REM PRINTING SELECTED RECORDS FROM THE FILE
120     DIM N$4
130     SELECT PRINT 005
140     M = 300 :REM INITIAL MAXIMUM RECORD NUMBER ENTRY
150 REM FIND THE FILE
160     DATA LOAD "POINTS1"
170     P = 1   :REM TAPE POSITIONED IN FRONT OF RECORD 1
180 REM OPERATOR ENTERS RECORD NUMBER SOUGHT
190     PRINT HEX(030A) :REM CLEAR SCREEN; LINE 1
200     INPUT "ENTER THE NUMBER OF THE RECORD TO BE PRINTED", R
210     IF R < 1 THEN 230 :REM TOO LOW?
220     IF R <= M THEN 260 :REM LOW ENOUGH?
230     PRINT "INVALID. REENTER"; HEX(0D0C); TAB(64); HEX(0C)
240     GOTO 200
250 REM BRANCH ON POSITION OF TAPE RELATIVE TO RECORD SOUGHT
260     ON SGN(R-P)+1 GOTO 330, 310
270 REM TAPE POSITIONED BEYOND DESIRED RECORD
280     BACKSPACE P-R
290     GOTO 330
300 REM DESIRED RECORD IS BEYOND THE CURRENT TAPE POSITION
310     SKIP R-P
320 REM TAPE SET TO READ DESIRED RECORD
330     DATA LOAD N$, X, Y
340     IF END THEN 490 :REM ENTERED RECORD NUMBER TOO HIGH
350     CONVERT N$ TO N
360     IF N < > R THEN 570 :REM FILE OUT OF SEQUENCE
370     P = R + 1:REM UPDATE TAPE POSITION COUNTER
380 REM OUTPUT RECORD
390     PRINT HEX(0C); TAB(64); TAB(64); HEX(0A0A0A)
400     PRINTUSING 620
410     PRINTUSING 630, N$, X, Y
420     PRINT HEX(010A)
430     INPUT "MORE RECORDS (Y/N)", E$
440     IF E$ = "N" THEN 590 :REM DONE?
450     E$ = " "
460     R = 0
470     GOTO 190
480 REM ERROR ROUTINES
490     PRINT "ERROR: RECORD NUMBER"; R; "IS TOO HIGH"
500     BACKSPACE 1
510     DATA LOAD N$, X, Y :REM READ LAST RECORD
520     CONVERT N$ TO M :REM PUT ACTUAL LAST RECORD NO. IN M
530     PRINT "THE HIGHEST RECORD NUMBER IS"; M
540     P = M + 1 :REM UPDATE TAPE POSITION COUNTER
550     PRINT HEX(010A);TAB(64);HEX(0C)
560     GOTO 430
570     STOP "FILE OUT OF SEQUENCE. CANNOT BE PROCESSED"
```

287

```
580 REM END PROGRAM
590     REWIND
600     PRINT HEX(03)
610     END
620 %       RECORD NO.    X VALUE        Y VALUE
            ####      #,###.##        #,###.##
```

This program maintains the current tape position in variable P.   It   is maintained as the record number which the tape head is set to read. When the file is found (line 160), P is assigned the value 1 since at this point a DATA LOAD would read record number 1.

At first the maximum record number is not known.   However, the assumption is made that it is not larger than 300 (lines 140 and 220).   The grounds for this assumption, which is based on the record format and maximum tape capacity, are discussed in Section 21-7.

A record number is entered and validated at lines 200-240. The tape can be in any of three positions relative to the record sought: it can be beyond the record, it can be set to read the record, or it may have to skip ahead to find the record. Line 260 effects no branch under the first condition, a branch to line 330 under the second, and one to 310 under the third. Line 280 BACKSPACEs, or line 310 SKIPs, the required number of records, to reach the desired record.

Line 330 attempts to read the record. If an end trailer is read, then the entered record number is greater than the number of records actually in the file. If a data record has been read, it is checked to ensure that it is the record sought (line 360), and the tape position, P, is updated. The record is displayed, and the operator may then continue or end the program.

If an end-of-file trailer is read (line 340), the error routine (lines 490-560) indicates the problem, and reads the preceding record to determine the exact maximum record number. This maximum is then substituted for the earlier estimate in M. Thus, the error cannot occur twice.

It would have been possible to SKIP END at the beginning of this program, to determine the last record number in the manner used by the error routine. However, this is quite time consuming, and offers no   real advantages.

The operator of this program will quickly notice that backspacing over records is much slower than skipping over them.   For this reason it is generally advantageous to process tape records sequentially.

21-7   EFFICIENT DATA STORAGE

How Data Is Recorded

In Section 21-3 we said that each DATA SAVE statement saves a single record, that is, it collectively marks off the values it saves as a single record. A single record can contain any number of values, and, therefore, it is not strictly necessary to consider how data is recorded on cassette tape in order to use the tape data file statements. However, knowing how data is recorded can produce dramatically more efficient use of tape data files.

All recording on cassette tape is done in physical blocks. Each block has an absolute capacity of 256 bytes (remember a byte is the amount of space needed to store one alphanumeric character). The cassette drive always records at least one block each time a DATA LOAD is executed.

If a record requires less than 256 bytes for storage on tape, the DATA SAVE statement simply leaves empty space (garbage) between the end of the record and the end of the 256 byte block. Reducing the amount of this wasted space is an important objective for the programmer. It results in more efficient tape utilization and faster program execution.

In order to reduce this waste, you must first know exactly how much space is occupied by recorded values. Although each tape block has a physical capacity of 256 bytes, the system automatically records certain control information in addition to the actual data values. There are two types of control information:

1. Block Control bytes.
2. Start of Value (SOV) control bytes.

The DATA SAVE statement always writes three Block Control bytes into each block. Two of these occupy the first two byte locations in each block. These are used to indicate whether the block is the first, last, or a middle block in the record. They distinguish one record from the next. The third Block Control byte follows the last byte of the last data value in the block, and marks the end of valid data within the block. After subtracting the three Block Control bytes, a total of 253 bytes is available for data values and Start of Value (SOV) bytes.

A single Start of Value (SOV) byte precedes each value saved in a record. This byte indicates to the system whether the value is numeric or alphanumeric and the length of the value in bytes. Consider, for example, the X,Y coordinate record in the preceding sections. The storage space for the record is defined by:

```
120 DIM N$4
    .
    .
    .
220 DATA SAVE DC N$, X, Y
```
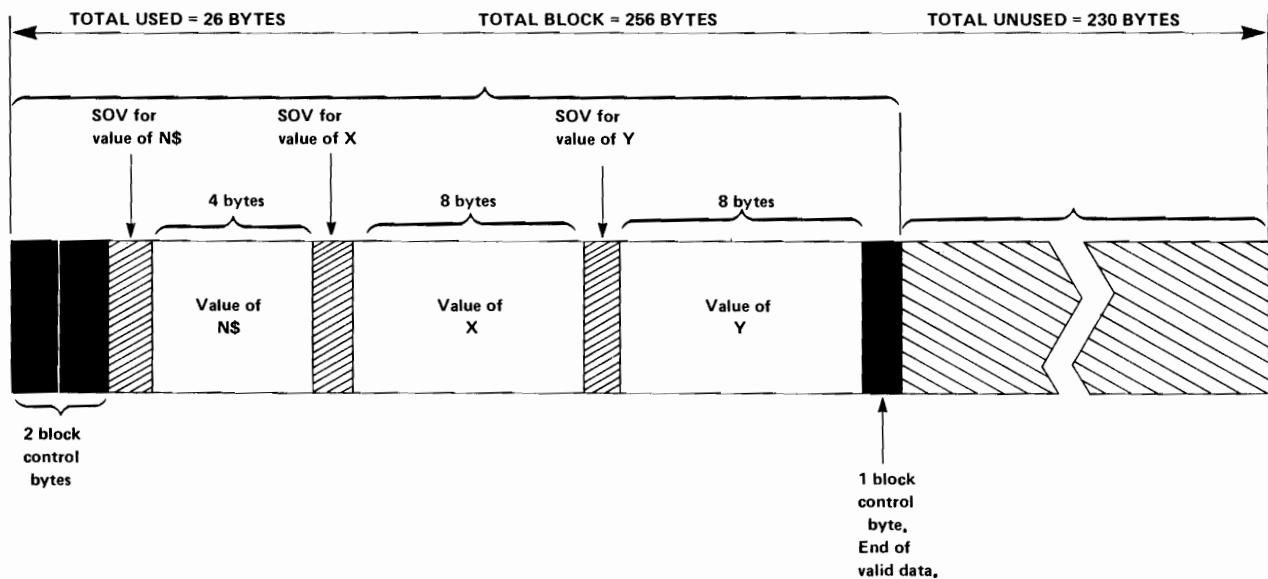
and the record looks like this in a block:

Figure 21.1   A Record in a Tape Block

Space requirements can be summarized as follows:

1. There are 253 bytes available for storage in each block after allowing for the Block Control bytes.

2. Each numeric value requires 9 bytes, 8 bytes for the value plus 1 byte for the SOV. (Numeric values may be specified in the DATA SAVE statement by numeric variables, expressions, or numeric array elements.)

3. Each alphanumeric value requires a number of bytes equal to the dimensioned size of the specifying alphanumeric variable or array element plus one byte for the SOV, or the number of characters in the specifying literal string plus one byte for the SOV. Note that in the case of an alphanumeric variable or array element, it is the dimensioned size, including all trailing spaces, which must be counted.

4. If a value does not completely fit into the space remaining in a block, it is automatically written in the next block. Values do not overlap from one block to the next, though a single record may require many blocks.

When entire arrays are saved in a record by using the standard array designator, the values are saved element by element, row by row into the record. Within the record there is no indication of whether a value was saved from an array or any other specific source, and, therefore, values may be loaded into array or scalar variables regardless of their source specification.

290

Programming Techniques That Improve Storage Efficiency

As can be seen from Figure 21.1, the records saved by the program of Example 21.1 waste a tremendous amount of tape space. Each record uses 26 bytes of a block, and wastes 230. As a result of the program of Example 21.1, one tape record (the result of a single DATA SAVE statement) contains one data record, that is, the data needed to specify a particular point in the X,Y coordinate plane. DATA SAVE always creates one tape record and always uses at least one complete block. There is no way to put more than one tape record into a single block. However, since the concept of what constitutes a data record is defined by the program, it is possible to let one tape record contain several data records (in this case the data for several points). From the point of view of the tape record and the control information written with a DATA SAVE statement, the several data records will "look like" a single record since they must be written collectively with a single DATA SAVE statement. The programs that create and use the file must themselves distinguish one data record from another within a single tape record.

Each point in our simple data file requires that two numeric values be saved. The two values occupy a total of 18 bytes including the SOV bytes. Since a total of 253 bytes is available per block for SOV's and data, we could save 14 point specifications in a block. These would occupy 252 bytes, leaving just one byte unused. However, as in Example 21.1, we may wish to maintain some additional information in each block. For a blocked file it might be a good idea to save in each block a block number and a notation indicating the number of points actually saved in the block. To accommodate this information, we reduce the number of points saved in each block to 13.

One way to save 13 point specifications is to simply use 13 sets of variables in the DATA SAVE statement. For example, if the additional block information (block number, number of records) is in B$, and the 13 points are given by

        (X,Y), (X0,Y0), (X1,Y1), (X2,Y2)...(X9,Y9)
        (A,B), (A0,B0)

then the DATA SAVE statement would be

        DATA SAVE B$,X,Y,X0,Y0,X1,Y1,X2,Y2,X3,Y3
        X4,Y4,X5,Y5,X6,Y6,X7,Y7,X8,Y8,X9,Y9,A,B,A0,B0

This approach to saving the records is extremely awkward. A far better approach is to put all the X values into a one-dimensional array, X(), and all the Y values into a one-dimensional array, Y(). In this way X(1) and Y(1) can define a point, as can X(2) and Y(2), X(3) and Y(3)...X(13) and Y(13). The 13 points are saved as follows:

        120 DIM X(13) Y(13) B$6
        .
        .
        .
        400 DATA SAVE B$, X(), Y()

Saving multiple data records with a single DATA SAVE statement in this fashion is often called "array-type blocking".

Example 21.4 shows a program that creates a data file in this manner, saving 13 points per block.

Example 21.4 Array Type Blocking of a Data File

```
110 REM ARRAY TYPE BLOCKING OF A DATA FILE
120     DIM X(13), Y(13), R$1, B$6
130 REM NAME FILE IN HEADER RECORD
140     DATA SAVE OPEN "POINTS2"
150 REM SET UP PER BLOCK
160     B = B + 1 :REM BLOCK COUNTER
170 REM LOOP TO ENTER VALUES FOR ONE BLOCK
180     FOR R = 1 TO 13  :REM 13 RECDS. PER BLOCK
190         PRINT HEX(03); "KEY S.F. 31 TO END PROGRAM"
200         PRINT "RECORD NUMBER="; (B-1)*13 + R
210         PRINT
220     REM RECEIVE RECORD VALUES
230         INPUT "X VALUE", X(R)
240         INPUT "Y VALUE", Y(R)
250         PRINT
260     REM OPERATOR CHECK OF VALUES
270         R$ =" ":REM NO DEFAULT ENTRY
280         PRINT "CHECK VALUES"
290         INPUT "ENTER + TO ACCEPT, - TO REJECT", R$
300         IF R$ = "-" THEN 190 :REM REJECTED?
310         IF R$ = "+" THEN 340 :REM ACCEPTED?
320         PRINT HEX(OC); TAB(64); HEX(OCOC): REM ENTRY ERROR
330         GOTO 280
340     NEXT R
350     GOSUB 390 :REM SAVE BLOCK OF RECORDS
360     GOTO 160
370 REM
380 REM SAVE BLOCK
390     CONVERT B TO B$,(####)   :REM BLOCK NUMBER
400     CONVERT R TO STR(B$,5), (##) :REM # OF RECDS. IN BLOCK
410     DATA SAVE B$, X(), Y()
420     RETURN
430 REM
440 REM END PROGRAM
450     DEFFN' 31
460     IF R = 1 THEN 490 :REM ALL ENTERED RECORDS SAVED?
470     R = R - 1           :REM R = NO. OF COMPLETED RECORDS
480     GOSUB 390           :REM SAVE LAST GROUP OF RECDS.
490     DATA SAVE END       :REM MARK END OF DATA FILE
500     PRINT HEX(03)
510     END
```

In this program a FOR/NEXT loop (180-340) allows the 13 sets of point values to be entered. The counter variable, R, is used as the specifying subscript for the receiving variables (lines 230 and 240).

After 13 points have been entered, the loop terminates, and a record is saved by the "save block" subroutine (lines 340, 350). The program then loops back for more point entries.

The "save block" subroutine (lines 390-420) converts the block number and record counter to alphanumeric characters.   Both of these values are assigned to B$, which has a length of 6 characters. Converting these numeric values to alphanumeric form reduces their storage space requirement from 18 bytes (2 numeric values and SOV's) to 7 bytes. In this program this isn't strictly necessary, since the block could have accommodated two more numeric values; however, it serves to illustrate a commonly used technique.

The operator ends this program by keying Special Function key 31 which causes a branch to line 450. However, before program execution terminates,a check is made to see if all entered points have been saved. Unless the number of points entered is a multiple of 13 when S.F. 31 is keyed, there are left-over points not yet saved, that must be saved. If the record counter, R, equals 1 at line 460, then all entered and accepted points have been saved. In general, R-1 is equal to the total number of entered and accepted points that have not yet been saved. Line 480 saves the last block of records, with the last two characters of B$ now containing a number less than 13.   Finally, line 490 saves the special end-of-file record.

Example 21.5 shows a program that reads and displays the data file created by Example 21.4. It is functionally similar to Example 21.2.

Example 21.5 Printing the Array-Blocked Data File

```
110 REM PRINTING THE ARRAY BLOCKED DATA FILE
120     DIM X(13), Y(13), B$6
130     SELECT PRINT 005
140 REM FIND FILE
150     DATA LOAD "POINTS2"
160 REM READ AND PRINT
170     DATA LOAD B$, X(), Y()
180     PRINT HEX(03) :REM CLEAR CRT
190     IF END THEN 290
200     PRINTUSING 310:   REM HEADINGS
210     CONVERT STR(B$,1,4) TO B  :REM BLOCK NUMBER
220     CONVERT STR(B$,5) TO N    :REM NUMBER OF RECDS IN BLOCK
230     FOR J = 1 TO N
240         PRINTUSING 320, J+(B-1)*13, X(J), Y(J)
250     NEXT J
260     IF N < 13 THEN 290 :REM NO MORE BLOCKS?
270     INPUT "KEY (EXEC) FOR MORE RECORDS", Z$
280     GOTO 170
290     REWIND
300     STOP "END OF DATA FILE"
310 %   RECORD NO.      X VALUE      Y VALUE
320 %        ####      #,###.###    #,###.###
```

293

In this program, line 170 reads the entire block of records. Lines 210 and 220 convert the block number and number of records in the block to numeric form. Every block except the last contains 13 significant records. The loop at lines 240-250 displays the 13 records, each on a separate line of the CRT. At line 270, the operator keys (EXEC) to get the next block of records displayed.

After all records have been displayed, the program ends in either of two ways. If the total number of records in the file is a multiple of 13, then every block, including the last, is full. In this case, after the last block of records has been displayed, line 170 reads an END trailer, and line 190 branches to end the program. Otherwise, if the number of records is not an exact multiple of 13, then the last block contains less than 13 records. (The FOR/NEXT loop only displays as many records as are actually present, since its upper bound is N (line 230).) Line 260 tests if N is less than 13, and ends the program if it is.

21-8   SPECIFYING TAPE DEVICE ADDRESSES

All of the cassette operations in the preceding example programs have occurred at the address selected for TAPE class I/O operations. Master Initialization selects address 10A for TAPE operations. Generally any system that contains at least one cassette drive will have a cassette drive with address 10A. If a system includes additional cassette drives, they usually are addressed as 10B, 10C, 10D, 10E, 10F, in that order.

If your system consists of just one cassette drive and that drive is address 10A, then you never have to specify any cassette address, since Master Initialization automatically selects 10A for all TAPE operations. However, if your system includes more than one cassette drive, then in order to use any drive that has an address of 10B...10F, you must in some way specify its address.

There are three ways of specifying an address for a cassette operation:

1. An address may be directly specified in the tape cassette statement. A directly specified address is always written in the form

    /xyy,

    where xyy is the device address.

    For example,

        100 DATA SAVE /10C, OPEN "POINTS1"

    This statement writes a header record on the cassette tape mounted at device 10C. Additional examples, for the statements we have considered, are:

        100 DATA LOAD /10B, N$, X, Y
        100 DATA SAVE /10A, B$, X(), Y()
        100 BACKSPACE /10B, BEG
        100 SKIP /10C, K2 F
        100 DATA SAVE /10A, END

294

Each of these operations takes place at the specified address. When an address is specified directly in this fashion, the address selected for TAPE operations is ignored.

2. A SELECT statement with a TAPE parameter can be executed. Cassette statements can then be written without additional specification, so that the selected TAPE class address is used. For example, the statement

    SELECT TAPE 10C

selects the cassette drive whose address is 10C for all subsequent tape operations in which no additional specification is supplied.

3. A SELECT statement may be used to associate a tape cassette address with a "file number". If a cassette statement then contains a "file number" specification, the address SELECTed for that "file number" is used.

For example, this sequence causes the cassette operations to take place at address 10C.

    120 SELECT #1 10C
    130 DATA SAVE #1, B$, X(), Y()
    .
    .
    .
    260 DATA SAVE #1, END
    270 REWIND #1

In the above example, the SELECT statement at line 120 associates the address 10C with "file number" #1. In the subsequent statements, #1 is specified as the file number whose device address is to be used for the operation.

The term "file number" refers to any of the following two-character symbols: #1, #2, #3, #4, #5, #6. In cassette operations, unlike disk operations, a variable may not be used to specify a file number.

File numbers are a useful means of specifying cassette addresses when a single program uses more than one cassette, and when operator selection of cassette address is desirable.

Example 21.6 shows a routine that might be used to allow an operator to select, from three possible addresses, the addresses at which "input" and "output" cassettes are mounted.

295

Example 21.6   Operator Selection of Tape Device Addresses

```
110 REM OPERATOR SELECTION OF TAPE FILE DEVICE ADDRESSES
120 REM SELECT INPUT FILE ADDRESS
130     GOSUB 2010 :REM DISPLAY ADDRESS MENU
140 REM KEYBOARD ENTRY
150     INPUT "ENTER 1, 2 OR 3 TO SELECT INPUT FILE ADDRESS", R
160     IF R = INT(R) THEN 200  :REM ENTRY IS INTEGER?
170     PRINT "INVALID. REENTER"; HEX(ODOC); TAB(64); HEX(OC)
180     GOTO 150
190 REM BRANCH ON ENTERED VALUE
200     ON R GOTO 220, 230, 240
210     GOTO 170 :REM INVALID ENTRY
220     SELECT #1 10A  :GOTO 260
230     SELECT #1 10B  :GOTO 260
240     SELECT #1 10C
250 REM SELECT OUTPUT FILE ADDRESS
260     GOSUB 2010 :REM DISPLAY ADDRESS MENU
270 REM KEYBOARD ENTRY
280     INPUT "ENTER 1, 2 OR 3 TO SELECT OUTPUT FILE ADDRESS", R
290     IF R = INT(R) THEN 330  :REM ENTRY IS INTEGER?
300     PRINT "INVALID. REENTER"; HEX(ODOC); TAB(64); HEX(OC)
310     GOTO 280
320 REM BRANCH ON ENTERED VALUE
330     ON R GOTO 350, 360, 370
340     GOTO 300 :REM INVALID ENTRY
350     SELECT #2 10A  :GOTO 380
360     SELECT #2 10B  :GOTO 380
370     SELECT #2 10C
  .
  .
  .
2000 REM DISPLAY ADDRESS MENU SUBROUTINE
2010     PRINT HEX(030A0A0A0A0A) :REM CLEAR CRT. LINE 5.
2020     PRINT , "1. 10A","3. 10C"
2030     PRINT , "2. 10B"; HEX(01);
2040     RETURN
```

A subroutine (lines 2010-2040) displays the available addresses.   The operator enters 1, 2, or 3 to choose an address (line 150).   An ON...GOTO statement branches on the operator's entry (line 300) to select the desired address for file number #1 (lines 220-240).   The procedure is repeated for the output file, whose address is selected for file number #2.

The program (not shown) that follows this routine would include file number #1 in all cassette statements operating on the "input" cassette and file number #2 in the statements operating on the "output" cassette.

21-9 UPDATING CASSETTE DATA FILES

Any data file may become obsolete. Depending on the purpose of the file, you may then simply destroy it by saving other data or programs over it, or you may update it, so that it is no longer obsolete. Typically, to update a file, you may wish to add new records to the end of the file, add new records in the middle of the file, or change the values saved in existing records.

If, when you create a new data file, you know that more records will sometime have to be added to the end of it, then it is probably a good idea to devote an entire cassette to the file. When new records must be added, your program can simply SKIP END, and begin saving the new records, provided, of course, that the cassette is not full.

When existing records must be updated, or new records added to the middle of a file, there is a preferred approach, and there is a less desirable approach that can be used if absolutely necessary.

The preferred approach to updating a cassette data file is to create a new updated file on another cassette, by copying from the old file all records which are acceptable, and substituting new updated records wherever necessary. This method requires two cassette drives, and is sometimes called a "father-son" approach to file maintenance. In general, the obsoleted file is preserved at least until the new file contributes to a third-generation updated file. Thus, should a file accidentally be destroyed, the last updating of the file is all that has been lost. This form of copy-updating is the only prudent approach for a file that must frequently be updated over a long lifetime, and that would be difficult to reconstruct if accidentally destroyed. Example 21.7 shows a program that performs this type of updating on the array-blocked file discussed in Section 21-7.

Example 21.7 Updating the Array-Blocked Data File

```
110 REM UPDATING THE ARRAY-BLOCKED DATA FILE
120     DIM X(13), Y(13), B$6
130     SELECT PRINT 005, #1 10A, #2 10B
140     PRINT HEX(030A)
150     INPUT "MOUNT OLD FILE - 10A, BLANK CASSETTE - 10B", Z9$
160     DATA LOAD #1, "POINTS2" :REM FIND OLD FILE
170     DATA SAVE #2, OPEN "POINTS2" :REM SAVE NEW HEADER
180 REM
190 REM LOAD NEXT BLOCK
200     DATA LOAD #1, B$, X(), Y()
```

```
210 REM DISPLAY BLOCK
220     PRINT HEX(030A0A) :REM CLEAR CRT
230     IF END THEN 620
240     PRINTUSING 660:  REM HEADINGS
250     CONVERT STR(B$,1,4) TO B :REM BLOCK NUMBER
260     CONVERT STR(B$,5) TO N   :REM NUMBER OF RECDS IN BLOCK
270     PRINT HEX(0C)
280     FOR J = 1 TO N
290         PRINT
300         PRINTUSING 670, J+(B-1)*13, X(J), Y(J);
310     NEXT J
320     PRINT HEX(010D);
330 REM KEYBOARD CHOICE -  UPDATE?    OR OK NOW?
340     INPUT "ENTER NUMBER OF RECORD TO BE CHANGED (0 = NONE)",R1
350 REM OK NOW?
360     IF R1 = 0 THEN 590 :REM SAVE BLOCK
370 REM OPERATOR ERROR?
380     IF R1 <>  INT(R1) THEN 560    :REM NOT INTEGER?
390     IF R1  < 1+(B-1)*13 THEN 560 :REM TOO LOW?
400     IF R1  > N+(B-1)*13 THEN 560 :REM TOO HIGH?
410 REM CHANGE A RECORD
420     S = R1-(B-1)*13 :REM S IS SUBSCRIPT OF DESIRED RECORD
430     PRINT HEX(030A0A0A0A0A0A); "RECORD NOW IS:"
440     PRINTUSING 660
450     PRINTUSING 670, R1, X(S), Y(S)
460     PRINT HEX(010A)
470     INPUT "ENTER NEW X VALUE", X
480     INPUT "ENTER NEW Y VALUE", Y
490     INPUT "ENTER + TO ACCEPT, - TO REJECT NEW ENTRIES", Z9$
500     IF Z9$<> "+" THEN 430 :REM NEW ENTRIES REJECTED?
510     X(S) = X
520     Y(S) = Y
530     GOTO 220 :REM REDISPLAY BLOCK
540 REM
550 REM ERROR ROUTINE
560     PRINT "INVALID. REENTER"; HEX(0D0C); TAB(64); HEX(0C)
570     GOTO 320
580 REM SAVE UPDATED BLOCK ON OUTPUT TAPE
590     DATA SAVE #2, B$, X(), Y()
600     IF N = 13 THEN 200: REM MORE BLOCKS?
610 REM END OF PROGRAM
620     DATA SAVE #2, END
630     REWIND #1
640     REWIND #2
650     PRINT HEX(03); "END OF PROGRAM"
660 %    RECORD NO.       X VALUE        Y VALUE
670 %         ####       #,###.###      #,###.###
```

In this program file number #1 is used to specify the old cassette, in drive 10A; file number #2 is used to specify the new cassette, in drive 10B. At line 200 a block of records is read from the old file. Lines 220 to 320 display the block of records in a manner similar to that used in Example 21.5. The operator can then enter a record number for a record to be changed, or accept the block of records as currently displayed (line 340). If a record is selected to be updated, it is displayed alone on the screen (lines 440, 450),

298

and the operator enters and accepts the new values (lines 470-500). The entered values are assigned to the block array (lines 510, 520), and the entire block, as updated, is redisplayed. After a block is accepted (lines 340, 360), it is saved on the new cassette (line 590). The program continues until the end of the file is found (line 230, 600). Line 620 then saves an end-of-file record in the new file.

The DATA SAVE statement cannot be used to update a record or block of records in the middle of a data file. For example, one might suppose that to update a record in a tape cassette file, a program could simply read the record, or block of records, backspace one record, or block of records, and execute a DATA SAVE to write a new record over the old one. This cannot be done effectively. The DATA SAVE statement, used in this fashion, will not position the new record exactly over the old one. Remnants of the old record will cause errors, when a later DATA LOAD is attempted, and prevent the file from being read.

On systems that have two cassette drives, the copy-update technique as illustrated in Example 21.7 should always be used for file updating. On systems that have only one cassette drive, occasional updating of relatively short-lived data files can be performed with the BASIC statement DATA RESAVE. DATA RESAVE performs the same functions as DATA SAVE, except that it carefully records the new record or block of records exactly over a previously recorded record or block of records. Thus, DATA RESAVE permits single cassette updating. Unlike the two-cassette "father-son" approach, updating in place with DATA RESAVE does not leave any back-up cassette. For this reason, DATA RESAVE should only be used when the following conditions exist.

1. Only one cassette drive is available.

2. The file to be updated need only be updated infrequently. (Specifically the file should not be one requiring regular updating that is intrinsic to the file's purpose.)

3. The file, if accidentally destroyed at any time, can be reconstructed without a disasterous interruption of operations.

Example 21.8 shows a modification of Example 21.7 that uses DATA RESAVE to update a cassette file.

Example 21.8 Using DATA RESAVE to Update a File

```
110 REM UPDATING THE ARRAY-BLOCKED DATA FILE - ONE CASSETTE
120     DIM X(13), Y(13), B$6, F$1
130     SELECT PRINT 005, #1 10A
140     PRINT HEX(030A)
150     INPUT "MOUNT FILE TO BE UPDATED IN DRIVE 10A", Z9$
160     DATA LOAD #1, "POINTS2" :REM FIND OLD FILE
180 REM
190 REM LOAD NEXT BLOCK
200     DATA LOAD #1, B$, X(), Y()
205     F$ = "U"   :REM SET BLOCK FLAG TO "U" = UNCHANGED.
```

```
210 REM DISPLAY BLOCK
220     PRINT HEX(030A0A) :REM CLEAR CRT
230     IF END THEN 630
240     PRINTUSING 660:    REM HEADINGS
250     CONVERT STR(B$,1,4) TO B   :REM BLOCK NUMBER
260     CONVERT STR(B$,5) TO N     :REM NUMBER OF RECDS IN BLOCK
270     PRINT HEX(OC)
280     FOR J = 1 TO N
290         PRINT
300         PRINTUSING 670, J+(B-1)*13, X(J), Y(J);
310     NEXT J
320     PRINT HEX(010D);
330 REM KEYBOARD CHOICE - UPDATE?  OR OK NOW?
340     INPUT "ENTER NUMBER OF RECORD TO BE CHANGED (O = NONE)",R1
350 REM OK NOW?
360     IF R1 = O THEN 585 :REM RESAVE BLOCK, IF CHANGED.
370 REM OPERATOR ERROR?
380     IF R1 <> INT(R1) THEN 560    :REM NOT INTEGER?
390     IF R1 < 1+(B-1)*13 THEN 560 :REM TOO LOW?
400     IF R1 > N+(B-1)*13 THEN 560 :REM TOO HIGH?
410 REM CHANGE A RECORD
415     F$ = "C" :REM SET BLOCK FLAG TO "C" = CHANGED
420     S = R1-(B-1)*13 :REM S IS SUBSCRIPT OF DESIRED RECORD
430     PRINT HEX(030A0A0A0A0A0A); "RECORD NOW IS:"
440     PRINTUSING 660
450     PRINTUSING 670, R1, X(S), Y(S)
460     PRINT HEX(010A)
470     INPUT "ENTER NEW X VALUE", X
480     INPUT "ENTER NEW Y VALUE", Y
490     INPUT "ENTER + TO ACCEPT, - TO REJECT NEW ENTRIES", Z9$
500     IF Z9$<> "+" THEN 430 :REM NEW ENTRIES REJECTED?
510     X(S) = X
520     Y(S) = Y
530     GOTO 220 :REM REDISPLAY BLOCK
540 REM
550 REM ERROR ROUTINE
560     PRINT "INVALID. REENTER"; HEX(ODOC); TAB(64); HEX(OC)
570     GOTO 320
580 REM RESAVE UPDATED BLOCK IF NECESSARY
585     IF F$ = "U" THEN 600   :REM NO CHANGE TO THIS BLOCK
587     BACKSPACE #1, 1    :REM POSITION TAPE FOR UPDATE OF BLOCK
590     DATA RESAVE #1, B$, X(), Y()
600     IF N = 13 THEN 200: REM MORE BLOCKS?
610 REM END OF PROGRAM
630     REWIND #1
650     PRINT HEX(03); "END OF PROGRAM"
660 %   RECORD NO.      X VALUE      Y VALUE
670 %         ####    #,###.###    #,###.###
```

Only minor modifications are required to adapt the two cassette version (Example 21.7) to single cassette (Example 21.8). Line 170 is eliminated. To update the cassette (lines 587 and 590), the tape is backspaced one record and DATA RESAVE is used. With the one cassette system, if a block of records is completely unchanged, there is no need to backspace and resave it, so a procedure for detecting whether a block has been changed is added to this

program. When a new block is read, a flag, F$, is set to "U" to signify that
at this stage the block is unchanged. If the operator immediately accepts the
entire block as is (line 340), then the value "U" in F$ causes a branch (line
585) around the update routine. If any change is made to a block of records,
F$ is set to "C" (line 415) to ensure that an updated block will be resaved.

When DATA RESAVE is used, the record to be saved must be absolutely
identical to the record being recording over; otherwise, the file is
destroyed.

CHAPTER 22
CHAINING PROGRAM MODULES

22-1 <u>OVERVIEW</u>

Sometimes a program cannot fit in the available system memory. In this case it may still be possible to accomplish the program's task with a technique called "chaining" (sometimes also called "overlaying").

To use chaining, a task to be programmed should first be broken down into two or more major sub-tasks or "phases." For example, many tasks may easily be broken down into a set-up or data input phase and a processing phase. Often a distinct third phase may be present in which an output operation takes place.

Once the task has been broken down into several phases that follow upon one another, the program can be broken down into several sub-programs, called "modules", that each accomplish one phase of the total task. We have presupposed that all of these modules will not fit into memory at one time; however, since each accomplishes a distinct phase of the total task, it may be possible to execute these modules successively, letting one module replace the preceding one in memory, and in this manner complete the entire task.

If a task has been programmed using three modules, the operator loads the first module. When the task of the first module is complete, all (or some portion) of the program statements of the first module are automatically cleared; the second module is loaded, and starts to execute. The BASIC statements LOAD or LOAD DC are used to accomplish this linking of one module to the next. Operator action is not required. The LOAD statement appears in one module and, when executed, loads the next module. This second module may contain a LOAD statement which loads a third module. This linking of modules is responsible for the use of the term "chaining". Each time a new module is loaded, some portion of the previous module is cleared to make room for the new.

Four BASIC statements are available to facilitate program chaining. Their purposes are summarized below.

Statement	Purpose
COM	Specifies a variable or array that contains data common to several modules, and that is not to be cleared when a new module is loaded.
LOAD	Clears a specifiable portion of the program currently in memory, and clears all variables not designed as common. Then loads a named program module into memory from cassette tape, and initiates execution at a specified line.
LOAD DC	Same as LOAD except that the program module is loaded from a cataloged disk.
COM CLEAR	Changes the status of a variable from common to non-common, or vice-versa.

An additional command is also available which has not previously been discussed, and which is useful when programming in multiple modules. This is the command

    CLEAR N

The "N" parameter stands for "non-common" variables. CLEAR N clears all variables not designated as common. CLEAR N leaves common variables undisturbed.

22-2 THE LOAD STATEMENTS (LOAD and LOAD DC)

A LOAD statement, LOAD or LOAD DC, is used in a program to load another program module. LOAD is a cassette statement that searches forward through a tape for a named program module, and loads the module. LOAD DC searches a disk catalog index for a named program module, and loads the module from the sector addresses specified in the index.

In Chapter 4 the LOAD and LOAD DC commands were introduced. Though there are some functional similarities between the LOAD commands and the LOAD statements, the operations performed are not identical. Unlike the LOAD commands, the LOAD statements have been specifically designed for chaining program modules. LOAD and LOAD DC are statements when they appear in a numbered program line; otherwise LOAD and LOAD DC are commands.

The general forms of the LOAD and LOAD DC statements are:

LOAD STATEMENT

LOAD $\begin{bmatrix} \text{\#n,} \\ \text{/xyy,} \end{bmatrix}$ ["name"] [1st line number] [,2nd line number]

where: #n = file number, #1-#6, for which a cassette device address has been SELECTed.

           /xyy = a cassette device address.

"name" = name of program module to search for and load (1 to 8 characters).

1st line number = the number of the first line to be deleted from the program currently in memory, prior to loading the new program module. After loading, execution begins again starting with this line number.

2nd line number = the number of the last line to be deleted from the program currently in memory, before the new program module is loaded.

LOAD DC STATEMENT

LOAD DC $\left\{\begin{array}{c} F \\ R \\ T \end{array}\right\}$ $\left[\begin{array}{c} \#n, \\ /xyy, \end{array}\right]$ name [1st line number][,2nd line number]

where:

F = fixed platter. or left platter.

R = removable platter or right platter (middle on 2270-3).

T = "F" or "R" platter depending on device type code in the device address.

#n = file number, for which a disk device address has been selected; n is an integer 1-6 or a numeric variable.

/xyy = a disk device address.

name = the name of the cataloged program file to be loaded into memory, expressed as either an alphanumeric variable or literal string in quotes.

1st line number = the line number of the first line to be deleted from memory, prior to loading the new program module. After loading, execution begins again at this line number.

2nd line number = the number of the last line to be deleted from the program currently in memory, before the new program is loaded.

LOAD and LOAD DC can be thought of as program statements that, in effect, produce the following sequence of BASIC commands:

CLEAR P   Clears program text from memory, beginning at "1st line number" (if specified) and ending at "2nd line number" (if specified). If no line numbers are specified, all program lines are cleared. If only "1st line number" is specified, all program lines from 1st line number to the highest line number are cleared.

CLEAR N
Clears all variables not specifically designated as common variables (see COM statement next section), and all FOR/NEXT and RETURN information.

LOAD (or LOAD DC)
Load named program module.

RUN
Run new program beginning execution at "1st line number" (if specified). If no line number specified, begin at lowest line number in memory.

In summary, then, the ~~LOAD~~ and LOAD DC statements clear a specified portion of the program text currently in memory, clear all variables which have not been designated as common variables, load a specified program module, and begin executing it. The ~~LOAD~~ or LOAD DC statement can itself be included within the area to be cleared. Before execution of the newly loaded module actually begins, the system searches through the entire program text allocating memory area to variables.

For example, the statement

    700 LOAD /10A, "LIN-EQU2" 100

clears all program text from line 100 through the highest numbered line, clears all non-common variables, searches the cassette mounted at device 10A for a program file named "LIN-EQU2" and loads the file. After the file has been loaded, the system scans the entire program text (old statements as well as new) for variables, allocating memory space as needed, and initiates execution at line 100. If there is no line 100 in the program an error results.

The statement

    4040 LOAD DC T#3, "PAYMOD2" 900, 6020

clears all program text from lines 900-6020 inclusive, clears all non-common variables, searches the disk catalog index of the disk mounted at the device and location specified at file number #3, and loads the program module "PAYMOD2". After the file has been loaded, memory space is allocated to variables, and execution begins at line 900.

When a program module is loaded, a line in the incoming module that has the same line number as a line already in memory replaces the old line. However, a LOAD statement executes more quickly if program lines that have the same line numbers as incoming program lines, have first been cleared (by loading into the area cleared by the LOAD or LOAD DC statement).

If used in a multistatement line, LOAD (or LOAD DC) must be the last statement in the line.

305

22-3 THE COM AND COM CLEAR STATEMENTS

In order to accomplish a programming task in several modules that, because of memory limitations, cannot be accomplished in one module, each module must, in some way, pass information to the succeeding module. Broadly construed this "information" might include such things as a temporary data file on tape or disk, or an invoice which has the customer name and address completed. Most frequently, though, information is passed to a succeeding module by assigning it to variables or arrays that are not cleared when the next module is loaded. Since such variables are common to both modules, they are known as "common" variables. In BASIC any variable or array can be established as a common variable or array by means of the COM statement. If a variable has been established as a common variable it is undisturbed by the execution of a LOAD or LOAD DC statement; its value and dimensions pass intact to the next module.

The COM statement designates a variable as a common variable. Otherwise, it can be thought of as exactly like DIM, except for two differences:

1. A "scalar" numeric variable can be included in a COM statement (since a program may require that such a variable be common to several modules).

2. The COM statement (or statements) must be at a lower line number than any line on which there appears a DIM statement or any reference to any non-common variable.

Here is an acceptable use of COM statements:

```
10 COM X, Y, A$20, C$14, D$, R$1, K$(4,4)8
20 COM R(12), S(14,14), L$(6)2
30 DIM T$(14)8, T(14)
40 INPUT "NAME OF FILE", F$
50 DIM R2(12,12)
60 INPUT "ENTER STARTING ARRAY VALUE", R2(1,1)
```

Notice that the scalar numeric variables X and Y have been designated as common (line 10). Scalar alphanumeric variables can be established with any desired character length (1-64). If no length is specified (D$, line 10), the length is 16 characters. Numeric and alphanumeric arrays can be established, just as in a DIM statement (lines 10 and 20). All COM statements must precede any DIM statements and any references to variables (lines 30-50).

A DIM statement need only precede a reference to the variable it dimensions (lines 50 and 60); it may follow other variable references (F$ in line 40). However, a COM statement must precede all references to non-common variables and all DIM statements. If line 40 were changed to line 5, this program would not execute, since, then, a reference to a non-common variable, F$, would precede a COM statement.

306

A common variable can only be cleared from memory by a CLEAR or CLEAR V command, or by Master Initialization. Thus, it is unaffected by successive LOAD operations, and is available in all succeeding modules. However, the BASIC statement COM CLEAR is available to designate common variables as non-common. (COM CLEAR is not included in the 2200S Instruction Set, but can be obtained as part of Option 24.) If COM CLEAR is used to designate common variables as non-common, then the variables designated as non-common will be cleared from memory when a LOAD or LOAD DC statement is executed. COM CLEAR does not itself clear any variables from memory; it merely changes the designation of variables from common to non-common.

To understand what COM CLEAR does, one must have a figurative idea of what it means for a variable to be a common variable or a non-common variable. We have said that during a process called "program resolution", which occurs immediately after RUN(EXEC) is keyed, the system scans the entire program text, in line number sequence, looking for variables and arrays, and allocating memory space to each variable and array used in the program. Each time it finds a variable in the program, it checks whether it has already allocated space to that variable. If it has, it goes on; if not, it allocates the correct amount of space, and assigns the proper initial value. By this process, variables are assigned memory space in the order in which they appear in the program. Let us suppose that Figure 22.1 depicts the memory area used for variables as defined by the program statements of Example 22.1 shown below:

> Example 22.1 Program to Illustrate Memory Allocation to Common and Non-Common Variables

```
10 COM X, A$, R2(6), N$(4)64
20 INPUT K$
30 DIM J(4,4)
40 FOR S=1 TO 4
50    J(S,S)=1
60 NEXT S
   .
   .
   .
```


COMMON VARIABLE POINTER ⟶ K$

(All variables below this pointer are common variables. The COM variable pointer protects them from being cleared when a LOAD statement is executed. All variables above this pointer are non-common and are cleared during execution of a LOAD statement.)

Memory Area Used For Variables

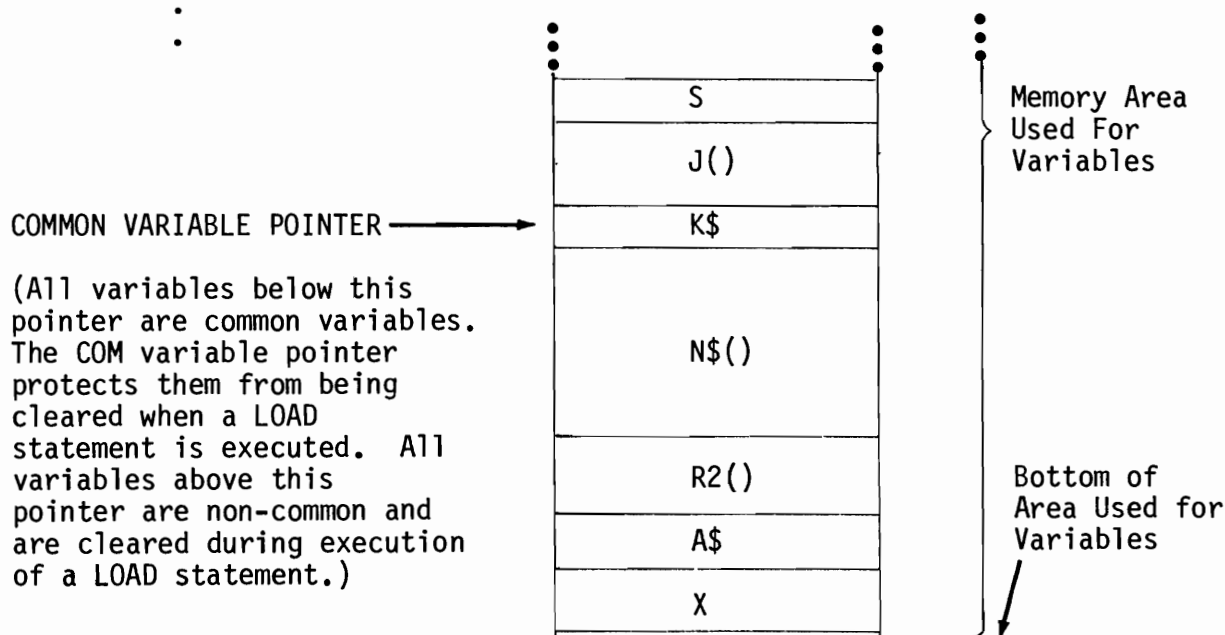Bottom of Area Used for Variables

Figure 22.1 Common and Non-Common Variables Set-up by Sample Program

Since X is the first variable to appear in the program (in the COM statement at line 10), it is the first variable for which space is allocated in memory. In the figure, it appears at the bottom of the memory area used for variables. The other common variables and arrays, A$, R2(6), and N$(4)64, occupy the next successive areas of memory above X. Since common variables must be specified at a lower line number than any DIM statement or reference to a non-common variable, all common variables are allocated contiguous memory space below all non-common variables. In the example above, K$ is the first non-common variable, and all the variables which appear in the program after K$ are non-common. To separate the common variables from the non-common variables, and thereby mark the common variables as common, the system uses the common variable pointer, shown in the illustration as a large arrow. The common variable pointer points to a particular place in memory and says in effect, "All the variables below this location are common variables." Whenever the system clears non-common variables, as it does during a LOAD statement, it clears out all the variables down to the common variable pointer. Variables below the common variable pointer are left undisturbed.

The COM CLEAR statement has the following general form:

$$\text{COM CLEAR} \begin{bmatrix} \text{variable} \\ \text{array designator} \end{bmatrix}$$

If COM CLEAR is used without any variable specified, for example,

70 COM CLEAR

it causes all common variables to become non-common variables. It does this by simply moving the common variable pointer to the bottom of the memory area used for variables. Thus, if statement 70 (above) were appended to Example 22.1, when execution begins, the memory area for variables looks as it does in Figure 22.1, but after statement 70 is executed, it will look this this:

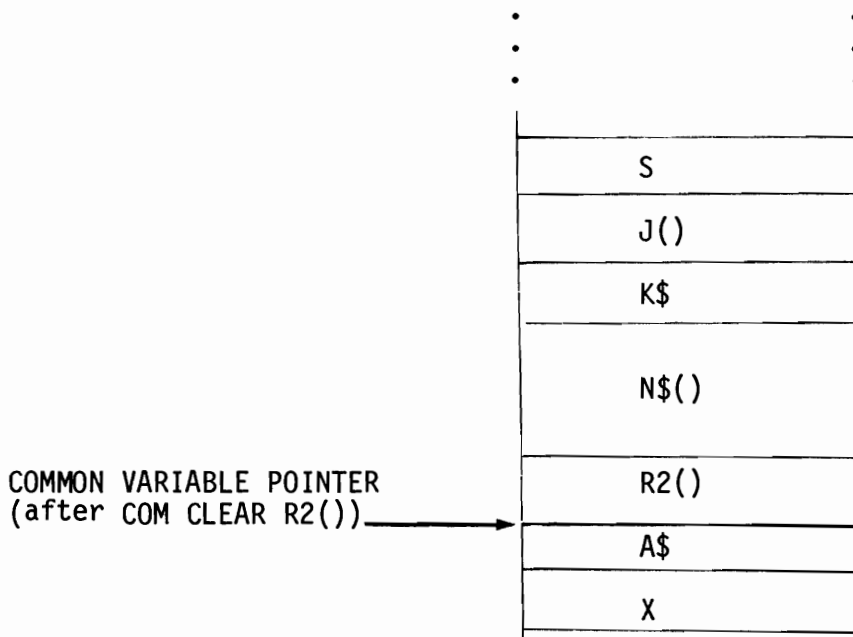COMMON VARIABLE POINTER
(after COM CLEAR)

308
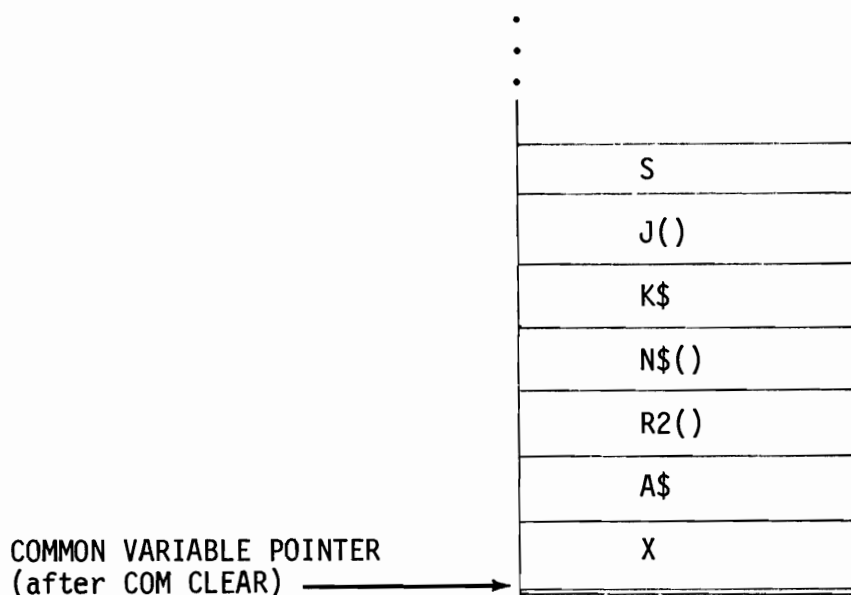
With the common variable pointer moved to the bottom of the memory area, there are now no common variables. The variables which were common are now non-common, and will be cleared from memory if a LOAD statement is executed (or a CLEAR N command is issued).

If a common variable or array designator is specified in the COM CLEAR statement, then the specified variable and all variables defined after it in the program become non-common. For example, if this statement is added to Example 22.1

        70 COM CLEAR R2()

then after this statement is executed, memory looks like this:

```
                                    .           .
                                    .           .
                                    .           .
                              +--------------------+
                              |         S          |
                              +--------------------+
                              |        J()         |
                              +--------------------+
                              |        K$          |
                              +--------------------+
                              |                    |
                              |       N$()         |
                              |                    |
                              +--------------------+
                              |                    |
COMMON VARIABLE POINTER       |       R2()         |
(after COM CLEAR R2())------->+--------------------+
                              |        A$          |
                              +--------------------+
                              |        X           |
                              +--------------------+
```

As can be seen from the figure, R2() and N$() have become non-common as a result of 70 COM CLEAR R2(). N$() is non-common since it was defined in the program after R2() (see line 10).

It is apparent from this example that there is no possible means of making R2() non-common and leaving N$() common. This could be done only if R2() had been originally defined after N$() in a COM statement that looked like this:
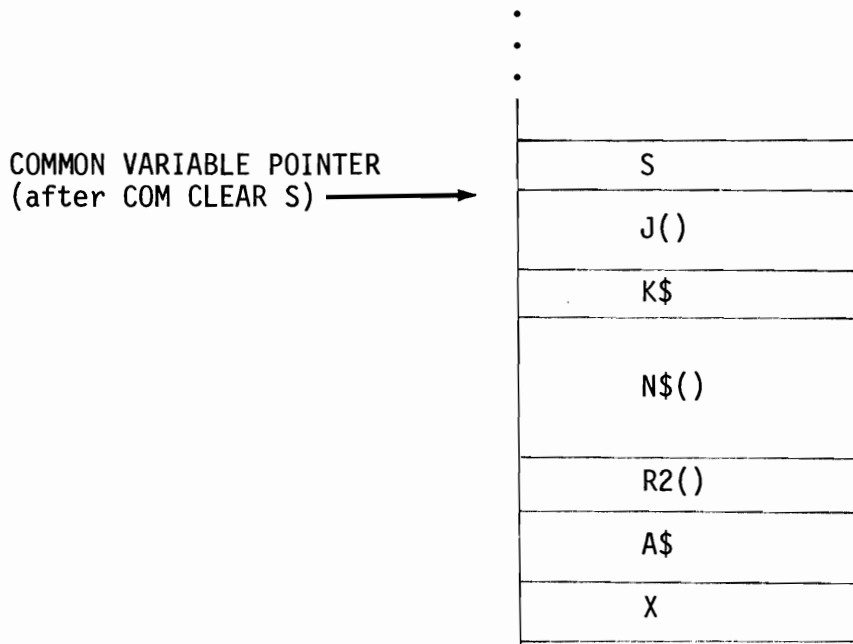
        10 COM X, A$, N$(4)64, R2(6)

Thus, if COM CLEAR is to be used, the sequence of variables and arrays in COM statements can be very important.

We can now see that the COM CLEAR statement simply moves the Common Variable Pointer to change the designation of variables from common to non-common. It should not be altogether surprising, then, to find that the COM CLEAR statement can be used for the opposite effect: to make non-common variables into common variables. For example, if statement 70 is

       70 COM CLEAR S

then, after 70 is executed, the memory area used for variable storage looks like this:

```
                                    .                    .
                                    .                    .
                                    .                    .
                               ┌─────────────────────────┐
  COMMON VARIABLE POINTER      │           S             │
  (after COM CLEAR S) ────────▶├─────────────────────────┤
                               │          J()            │
                               ├─────────────────────────┤
                               │          K$             │
                               ├─────────────────────────┤
                               │                         │
                               │         N$()            │
                               │                         │
                               ├─────────────────────────┤
                               │         R2()            │
                               ├─────────────────────────┤
                               │          A$             │
                               ├─────────────────────────┤
                               │          X              │
                               └─────────────────────────┘
```

After executing COM CLEAR S, J() and K$ have been added to the group of common variables, since J() and K$ were defined in the program before S.

A newly loaded module can add more common variables to the group of common variables which have been passed to it by a previous program. It can do this by beginning with COM statements or by executing a COM CLEAR statement which specifies a non-common variable.

Example 22.2 shows the COM statements and LOAD DC statements for chaining a four-module program. Actual processing to be accomplished by the modules is not shown.

310

Example 22.2   Chaining a Four-Module Program

```
110 REM MODULE 1 OF A 4 MODULE PROGRAM
120     COM N$30, D(8), C$2, F2$(4)8, A$(4)30
130     DIM C$(4,4)2, K$24, R(10,10), S$8
140     SELECT #6 320 :REM PROGRAM MODULES ARE AT 320
150 REM FIRST MODULE PROCESSING STATEMENTS BEGIN HERE
  .
  .
1200 REM END OF FIRST MODULE PROCESSING STATEMENTS
1210 REM CLEAR ALL OF MODULE 1 AND LOAD MODULE 2
1220     LOAD DC T#6, "MOD-2"
```

```
110 REM MODULE 2 OF A 4 MODULE PROGRAM
120     COM M$(4)60
130     DIM Q(2,2)
140 REM SECOND MODULE PROCESSING STATEMENTS BEGIN HERE
  .
  .
730     LOAD DC T#6, "MOD 3" 110, 740
740 REM LAST LINE TO BE CLEARED DURING LOAD
750 REM *** LINES 750 - 1110 ARE RETAINED INTO MODULE 3 ***
  .
  .
1110 REM LAST LINE OF MODULE 2
```

```
110 REM MODULE 3 OF A 4 MODULE PROGRAM
120     DIM K2$(2)24
130 REM MODULE 3 CONTAINS LINES NUMBERED 110 - 700, 1120 - 1300
140 REM LINES 750 - 1110 ARE LEFT IN IT BY MODULE 2
  .
  .
700 REM FIRST SEGMENT OF MODULE 3 LINES ENDS HERE
  .
  .    Lines 750-1110 from module 2 will be here
  .    during execution of module 3.
  .
1120 REM SECOND SEGMENT OF MODULE 3 BEGINS HERE
  .
1270 REM DESIGNATE NON-COMMON ALL VARIABLES EXCEPT N$
1280     COM CLEAR D()
1290 REM CLEAR ALL OF MODULE 3 AND LOAD MODULE 4
1300     LOAD DC T#6, "MOD 4"
```

```
110 REM MODULE 4 OF A 4 MODULE PROGRAM
120     DIM R$1, Z$(2)64
130 REM MODULE 4 INCLUDES LINES 110 - 620
  .
  .
610 REM END OF PROGRAM
620 END
```

311

Module 1, (first box) establishes COM variables and arrays (line 120) as well as non-common variables and arrays (line 130). At line 140 the address of the disk containing the program modules is assigned to file number #6. (It is important for the programmer to be aware that the LOAD DC statement uses a row of the device table much the way a data file does, and that, therefore, if, by default or otherwise, the LOAD DC takes place at a file number already in use for a data file, the data file will be "closed" by the operation of the LOAD DC.) Line 1220 of module 1 clears all of module 1, leaving only the COM variables in memory.

Module 2 (second box) adds the COM array M$() to those passed to it. Line 730 clears line 110 to 740 and loads the third module. This however leaves lines 750-1110 of module 2 in memory to become part of module 3. This partial clearing technique may be used to pass commonly used subroutines from one module to the next, or simply to pass routines which must be executed in both modules.

Module 3 receives all the COM variables established in modules 1 and 2. It has two segments of program lines, numbered 110-700 and 1120-1300 respectively. These segments will surround those left in by module 2 (lines 750-1110). Module 3 does not add any common variables. In fact, it makes non-common all the variables that were passed to it, except N$. It does this at line 1280, by means of the COM CLEAR statement. The array D() specified in line 1280 was declared common by module 1 (line 120). It, and all the variable space allocated after it, are now non-common, which leaves only N$ as common. All program lines are cleared by the statement at line 1300 of module 3 (including those lines left by module 2).

Module 4 is loaded. Only N$ is passed to it from module 3. It executes, and the entire program ends at line 620 of module 4.
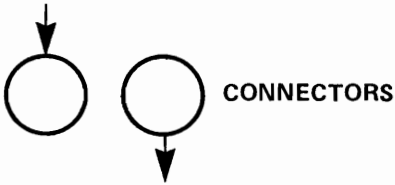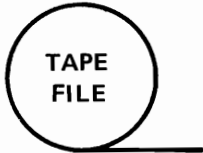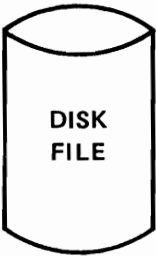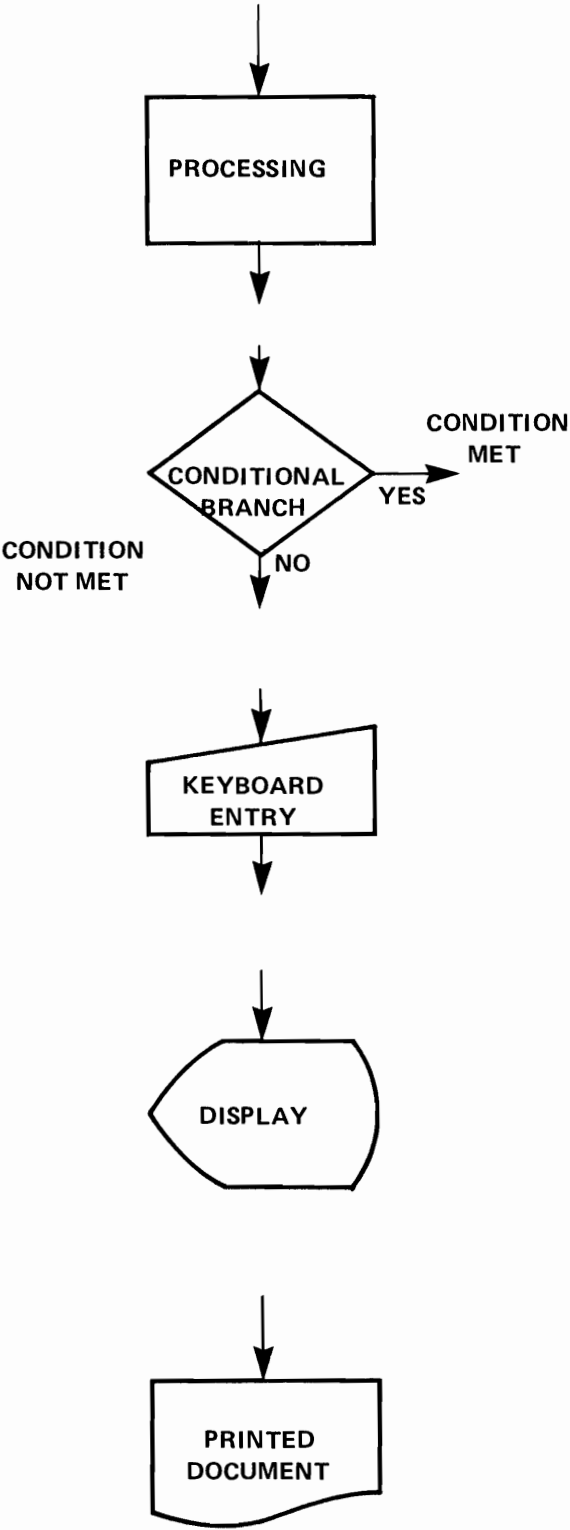
APPENDIX A
ERROR MESSAGES AND THEIR SIGNIFICANCE

ERR 01 TEXT OVERFLOW
ERR 02 TABLE OVERFLOW
ERR 03 MATH ERROR
ERR 04 MISSING LEFT PARENTHESIS
ERR 05 MISSING RIGHT PARENTHESIS
ERR 06 MISSING EQUALS SIGN
ERR 07 MISSING QUOTATION MARKS
ERR 08 UNDEFINED FN FUNCTION
ERR 09 ILLEGAL FN USAGE
ERR 10 INCOMPLETE STATEMENT
ERR 11 MISSING LINE NUMBER OR CONTINUE ILLEGAL
ERR 12 MISSING STATEMENT TEXT
ERR 13 MISSING OR ILLEGAL INTEGER
ERR 14 MISSING RELATION OPERATOR
ERR 15 MISSING EXPRESSION
ERR 16 MISSING SCALAR
ERR 17 MISSING ARRAY
ERR 18 ILLEGAL VALUE
ERR 19 MISSING NUMBER
ERR 20 ILLEGAL NUMBER FORMAT
ERR 21 MISSING LETTER OR DIGIT
ERR 22 UNDEFINED ARRAY VARIABLE
ERR 23 NO PROGRAM STATEMENTS
ERR 24 ILLEGAL IMMEDIATE MODE STATEMENT
ERR 25 ILLEGAL GOSUB/RETURN USAGE
ERR 26 ILLEGAL FOR/NEXT USAGE
ERR 27 INSUFFICIENT DATA
ERR 28 DATA REFERENCE BEYOND LIMITS
ERR 29 ILLEGAL DATA FORMAT
ERR 30 ILLEGAL COMMON ASSIGNMENT
ERR 31 ILLEGAL LINE NUMBER
ERR 33 MISSING HEX DIGIT
ERR 34 TAPE READ ERROR
ERR 35 MISSING COMMA OR SEMICOLON
ERR 36 ILLEGAL IMAGE STATEMENT
ERR 37 STATEMENT NOT IMAGE STATEMENT
ERR 38 ILLEGAL FLOATING POINT FORMAT
ERR 39 MISSING LITERAL STRING
ERR 40 MISSING ALPHANUMERIC VARIABLE
ERR 41 ILLEGAL STR(ARGUMENTS
ERR 42 FILE NAME TOO LONG
ERR 43 WRONG VARIABLE TYPE
ERR 44 PROGRAM PROTECTED
ERR 45 PROGRAM LINE TOO LONG
ERR 46 NEW STARTING STATEMENT NUMBER
　　　　TOO LOW
ERR 47 ILLEGAL OR UNDEFINED DEVICE
　　　　SPECIFICATION
ERR 48 UNDEFINED SPECIAL FUNCTION KEY
ERR 49 END OF TAPE

ERR 50 PROTECTED TAPE
ERR 51 ILLEGAL STATEMENT
ERR 52 EXPECTED DATA (NONHEADER) RECORD
ERR 53 ILLEGAL USE OF HEX FUNCTION
ERR 54 ILLEGAL PLOT ARGUMENT
ERR 55 ILLEGAL BT ARGUMENT
ERR 56 NUMBER EXCEEDS IMAGE FORMAT
ERR 57 ILLEGAL VALUE
ERR 58 EXPECTED DATA RECORD
ERR 59 ILLEGAL ALPHA VARIABLE
ERR 60 ARRAY TOO SMALL
ERR 61 TRANSIENT DISK HARDWARE ERROR
ERR 62 FILE FULL
ERR 63 MISSING ALPHA ARRAY DESIGNATOR
ERR 64 SECTOR NOT ON DISK OR DISK NOT SCRATCHED
ERR 65 DISK HARDWARE MALFUNCTION
ERR 66 FORMAT KEY ENGAGED
ERR 67 DISK FORMAT ERROR
ERR 68 LRC ERROR
ERR 71 CANNOT FIND SECTOR/PROTECTED PLATTER
ERR 72 CYCLIC READ ERROR
ERR 73 ILLEGAL ALTERING OF A FILE
ERR 74 CATALOG END ERROR
ERR 75 COMMAND ONLY (NOT PROGRAMMABLE)
ERR 76 MISSING < OR > (PLOT STATEMENT)
ERR 77 STARTING SECTOR > ENDING SECTOR
ERR 78 FILE NOT SCRATCHED
ERR 79 FILE ALREADY CATALOGED
ERR 80 FILE NOT IN CATALOG
ERR 81 /XYY DEVICE SPECIFICATION ILLEGAL
ERR 82 NO END OF FILE
ERR 83 DISK HARDWARE ERROR
ERR 84 NOT ENOUGH MEMORY FOR MOVE OR
　　　　COPY
ERR 85 READ AFTER WRITE ERROR
ERR 86 FILE NOT OPEN
ERR 87 COMMON VARIABLE REQUIRED
ERR 88 LIBRARY INDEX FULL
ERR 89 MATRIX NOT SQUARE
ERR 90 MATRIX OPERANDS NOT COMPATIBLE
ERR 91 ILLEGAL MATRIX OPERAND
ERR 92 ILLEGAL REDIMENSIONING OF ARRAY
ERR 93 SINGULAR MATRIX
ERR 94 MISSING ASTERISK
ERR 95 ILLEGAL MICROCOMMAND OR FIELD/
　　　　DELIMITER SPECIFICATION
ERR 96 MISSING ARG 3 BUFFER
ERR 97 VARIABLE OR ARRAY TOO SMALL
ERR 98 ILLEGAL ARRAY DELIMITERS
ERR=1 MISSING NUMERIC ARRAY NAME
ERR=2 ARRAY TOO LARGE
ERR=3 ILLEGAL DIMENSIONS
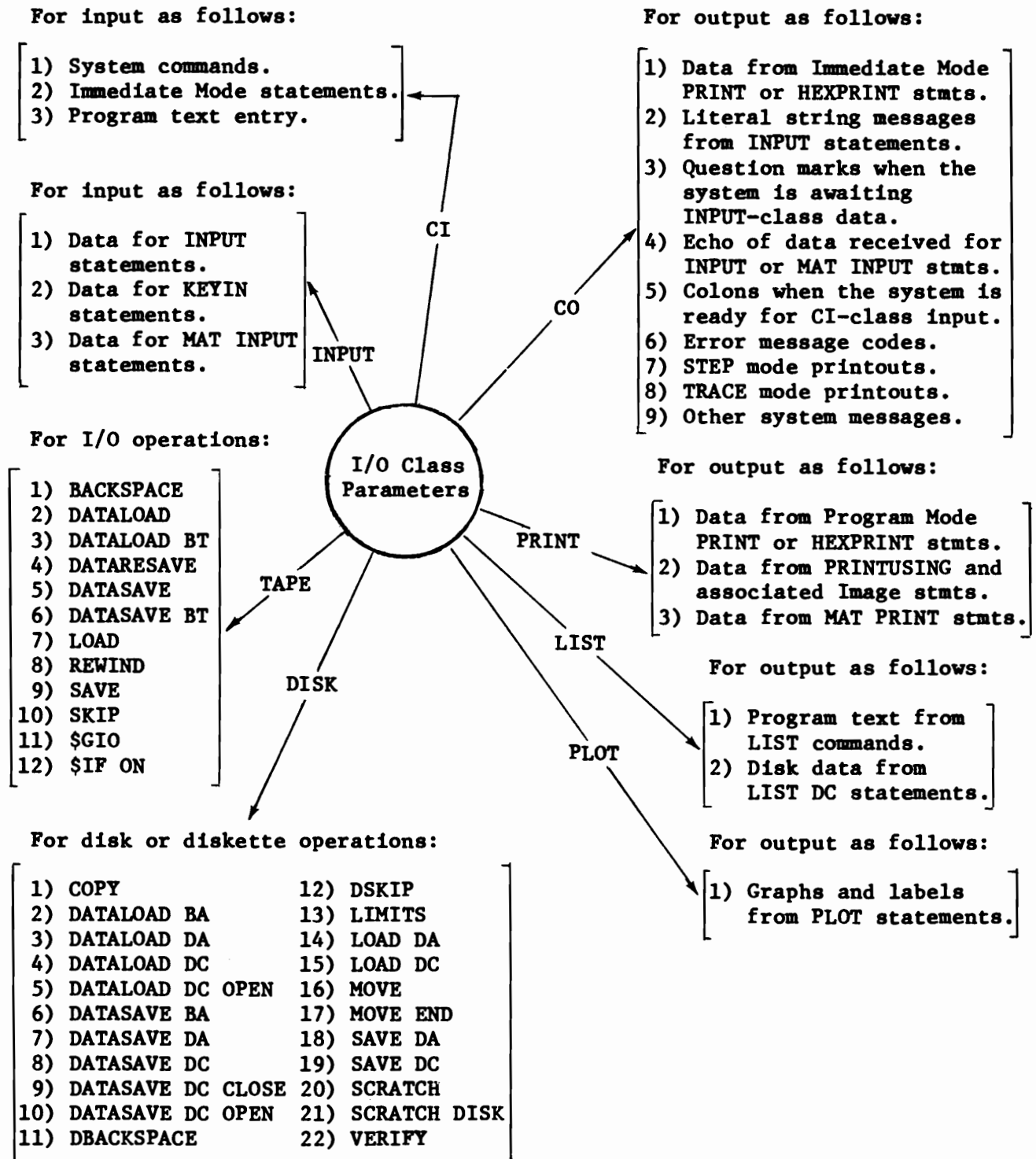
APPENDIX B
STANDARD FLOWCHART SHAPES AND
SYMBOLS

PROCESSING

DISK
FILE

CONDITIONAL
BRANCH

CONDITION
MET

YES

CONDITION
NOT MET

NO

TAPE
FILE

KEYBOARD
ENTRY

PUNCHED
CARDS

DISPLAY

CONNECTORS

PRINTED
DOCUMENT

TERMINAL PRINT,
STOP, DELAY,
START, ETCETERA

APPENDIX C
TABLE OF CHARACTERS AND HEX CODES

High Order Hexadecimal Digit of Code

Low Order Hexadecimal Digit Of Code

	0	1	2	3	4	5	6	7
0	NULL		SPACE	Ø	@	P	prime	p
1	HOME (CRT)		!	1	A	Q	a	q
2			"	2	B	R	b	r
3	CLEAR SCREEN (CRT)		#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7	BELL		(apos)	7	G	W	g	w
8	BACKSPACE (CRT CURSOR ←)		(	8	H	X	h	x
9	CRT CURSOR →	CLEAR TAB(2201)	)	9	I	Y	i	y
A	LINE FEED (CRT CURSOR ↓)	SET TAB(2201)	*	:	J	Z	j	z
B	(VERTICAL TAB) PRINTER		+	;	K	[	k	{
C	FORM FEED OR REV. INDEX (CRT CURSOR ↑)		,	< or [	L	\	l	|
D	CR (CARRIAGE RETURN)		−	=	M	]	m	}
E		¢	.	> or]	N	↑ or ^ or !	n	~
F		° (DEGREE)	/	?	O	← or _	o	■

There is some variation in character sets from one peripheral device to another; therefore, you should consult the reference manual of a particular peripheral device for precise character set definition.

APPENDIX D
I/O CLASSES AND ASSOCIATED STATEMENTS

For input as follows:

```
1) System commands.
2) Immediate Mode statements.
3) Program text entry.
```

For input as follows:

```
1) Data for INPUT
   statements.
2) Data for KEYIN
   statements.
3) Data for MAT INPUT
   statements.
```

For I/O operations:

```
 1) BACKSPACE
 2) DATALOAD
 3) DATALOAD BT
 4) DATARESAVE
 5) DATASAVE
 6) DATASAVE BT
 7) LOAD
 8) REWIND
 9) SAVE
10) SKIP
11) $GIO
12) $IF ON
```

For output as follows:

```
1) Data from Immediate Mode
   PRINT or HEXPRINT stmts.
2) Literal string messages
   from INPUT statements.
3) Question marks when the
   system is awaiting
   INPUT-class data.
4) Echo of data received for
   INPUT or MAT INPUT stmts.
5) Colons when the system is
   ready for CI-class input.
6) Error message codes.
7) STEP mode printouts.
8) TRACE mode printouts.
9) Other system messages.
```

For output as follows:

```
1) Data from Program Mode
   PRINT or HEXPRINT stmts.
2) Data from PRINTUSING and
   associated Image stmts.
3) Data from MAT PRINT stmts.
```

For output as follows:

```
1) Program text from
   LIST commands.
2) Disk data from
   LIST DC statements.
```

For output as follows:

```
1) Graphs and labels
   from PLOT statements.
```

I/O Class Parameters

CI CO INPUT PRINT LIST PLOT TAPE DISK

For disk or diskette operations:

```
 1) COPY                12) DSKIP
 2) DATALOAD BA         13) LIMITS
 3) DATALOAD DA         14) LOAD DA
 4) DATALOAD DC         15) LOAD DC
 5) DATALOAD DC OPEN    16) MOVE
 6) DATASAVE BA         17) MOVE END
 7) DATASAVE DA         18) SAVE DA
 8) DATASAVE DC         19) SAVE DC
 9) DATASAVE DC CLOSE   20) SCRATCH
10) DATASAVE DC OPEN    21) SCRATCH DISK
11) DBACKSPACE          22) VERIFY
```

APPENDIX E
UPDATING ARRAY-BLOCKED DISK FILES

```
110 REM ADDING RECORDS TO THE END OF THE BLOCKED FILE
120     DIM  N$(3)10, D$(3)40, S$(3)6, Q(3), R(3)
130 REM OPEN FILE
140     DATA LOAD DC OPEN F "INVTORY"
150 REM
160 REM READ LAST BLOCK
170     DSKIP END
180     DBACKSPACE 1 S
190     DATA LOAD DC N$(), D$(), S$(), Q(), R()
200 REM LOOK BACK THRU BLOCK FOR LAST DATA RECORD
210          FOR K = 3 TO 2 STEP -1
220              IF N$(K) <> "↑↑↑↑↑↑↑↑↑↑" THEN 260 :REM DATA?
230          NEXT K
240          J = 1 :REM LAST DATA RECORD IS SUBSCRIPT 1
250          GOTO 300
260          J = K  :REM SAVE K
270          K = 2    :REM *   FORCE       *
280          NEXT K   :REM * "NEXT" EXIT *
290 REM J CONTAINS SUBSCRIPT OF LAST DATA RECORD
300     CONVERT STR(N$(J),2) TO N :REM FOR SEQUENCE TEST
310 REM NOW SET J EQUAL TO SUBSCRIPT OF NEW DATA RECORD
320     J = J+1
330     IF J = 4 THEN 390  :REM START NEW BLOCK?
340 REM NEW DATA RECORD MUST GO IN OLD (UNFILLED) BLOCK.
350     DBACKSPACE 1 S
360     GOTO 400
370 REM
380 REM RECORD CREATION LOOP BEGINS HERE
390     J = 1
400     FOR X = J TO 3
410          REM GET NEW PRODUCT NUMBER AND VALIDATE IT.
420              PRINT HEX(03); "KEY S.F. 31 TO END PROGRAM"
430              INPUT "NEW PRODUCT NUMBER", N$(X)
440              IF NUM(STR(N$(X),2)) <> 9 THEN 590 :REM BAD?
450              CONVERT STR(N$(X),2) TO N2
460              IF N+1 <> N2 THEN 590 :REM NOT IN SEQUENCE?
470          REM PRODUCT NUMBER OK. ENTER RECORD VALUES.
480              INPUT "PRODUCT DESCRIPTION", D$(X)
490              INPUT "SUPPLIER CODE", S$(X)
500              INPUT "ON HAND QUANTITY", Q(X)
510              INPUT "REORDER LEVEL", R(X)
520          N = N2 :REM SET-UP FOR NEXT ENTRY
530     NEXT X
540 REM SAVE BLOCK OF DATA RECORDS AND GO BACK FOR MORE
550     DATA SAVE DC N$(), D$(), S$(), Q(), R()
560     GOTO 390
570 REM
580 REM BAD PRODUCT NUMBER ENTERED
590     PRINT "INVALID PRODUCT NUMBER"
600     PRINT
610     GOTO 430
620 REM
630 REM END PROGRAM ROUTINE
640     DEFFN' 31
650     IF X = 1 THEN 720 :REM ALL BLOCKS FULL?
660 REM FILL EXTRA RECORDS IN BLOCK
670     FOR K = X TO 3
680          INIT("↑") N$(K), D$(K), S$(K)
690     NEXT K
700     DATA SAVE DC N$(), D$(), S$(), Q(), R()
710 REM MARK END OF DATA
720     DATA SAVE DC END
```

```
110 REM UPDATING QUANTITY VALUE IN A SPECIFIED RECORD
120 REM BLOCKED RECORDS
130     DIM  N$(3)10, D$(3)40, S$(3)6, Q(3), R(3), N2$10
140     DATA LOAD DC OPEN F "INVTORY"
150 REM
160 REM READ LAST BLOCK
170     DSKIP END
180     DBACKSPACE 1 S
190     DATA LOAD DC N$(), D$(), S$(), Q(), R()
200 REM LOOK BACK THRU BLOCK FOR LAST DATA RECORD
210     FOR K = 3 TO 2 STEP -1
220         IF N$(K) <> "↑↑↑↑↑↑↑↑↑↑" THEN 260 :REM DATA?
230     NEXT K
240     J = 1 :REM LAST DATA RECORD IS SUBSCRIPT 1
250     GOTO 300
260     J = K   :REM SAVE K
270     K = 2     :REM *   FORCE       *
280     NEXT K    :REM * "NEXT" EXIT *
290 REM J CONTAINS SUBSCRIPT OF LAST DATA RECORD
300     CONVERT STR(N$(J),2) TO M :REM M IS MAX PRODUCT NUMBER
310 REM
320 REM GET NEW PRODUCT NUMBER AND VALIDATE IT.
330     PRINT HEX(03)
340     INPUT "PRODUCT NUMBER", N2$
350     IF NUM(STR(N2$,2)) <> 9 THEN 620 :REM NON-NUMERIC?
360     CONVERT STR(N2$,2) TO N2
370     IF N2 > M THEN 620 :REM PRODUCT NUMBER TOO BIG?
380 REM PRODUCT NUMBER OK. FIND PRODUCT RECORD.
390     B = INT((N2-1)/3) :REM B IS NUMBER OF SECTORS TO SKIP
400     K = N2 - (B*3) :REM K IS SUBSCRIPT
410     DBACKSPACE BEG
420     DSKIP B S :REM SKIP TO CORRECT BLOCK
430 REM LOAD AND PRINT PRODUCT RECORD
440     DATA LOAD DC N$(), D$(), S$(), Q(), R()
450     PRINT
460     PRINT D$(K)
470     PRINT "QUANTITY ON HAND ="; Q(K)
480     PRINT
490     DBACKSPACE 1 S
500 REM ENTER TRANSACTION AND UPDATE RECORD
510     INPUT "ENTER AMOUNT RECEIVED (+) OR SOLD (-)", T
520     Q(K) = Q(K) + T
530     DATA SAVE DC N$(), D$(), S$(), Q(), R()
540 REM MORE RECORDS TO UPDATE?
550     R$ = " " :REM NO DEFAULT ENTRY
560     INPUT "MORE RECORDS TO UPDATE (Y/N)", R$
570     IF R$ = "Y" THEN 330
580     IF R$ <> "N" THEN 560   :REM OPERATOR ERROR?
590     DATA SAVE DC CLOSE
600 END
610 REM ERROR ROUTINE
620     PRINT "INVALID. REENTER"
630     PRINT
640     GOTO 340
```

318

NOTES

NOTES

To help us to provide you with the best manuals possible, please make your comments and suggestions concerning this publication on the form below. Then detach, fold, tape closed and mail to us. All comments and suggestions become the property of Wang Laboratories, Inc. For a reply, be sure to include your name and address. Your cooperation is appreciated.

700-3231F

TITLE OF MANUAL: **PROGRAMMING IN BASIC**

COMMENTS:

Fold

Fold

WANG