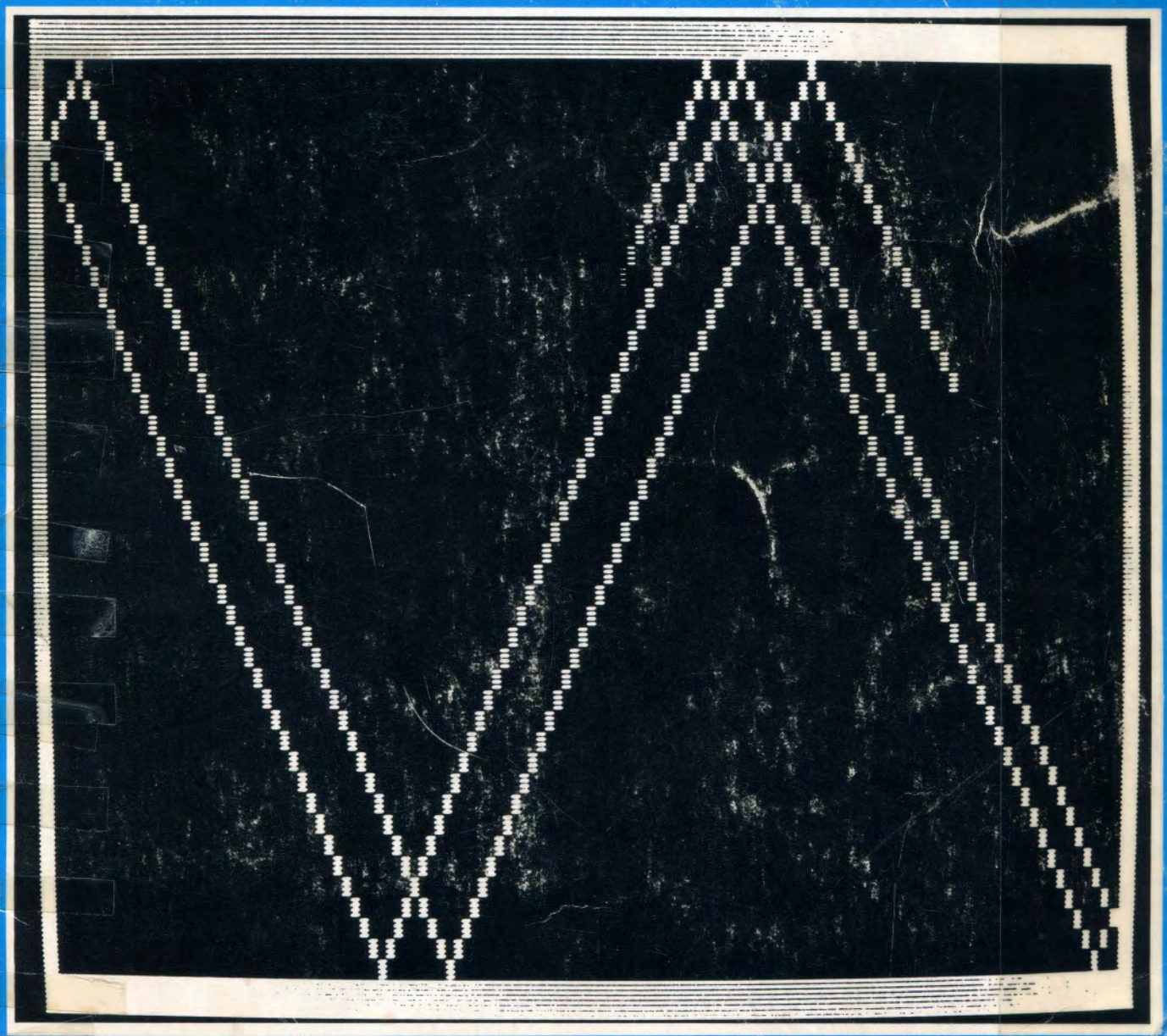
Microcomputer Power

TECHNIQUES OF BASIC

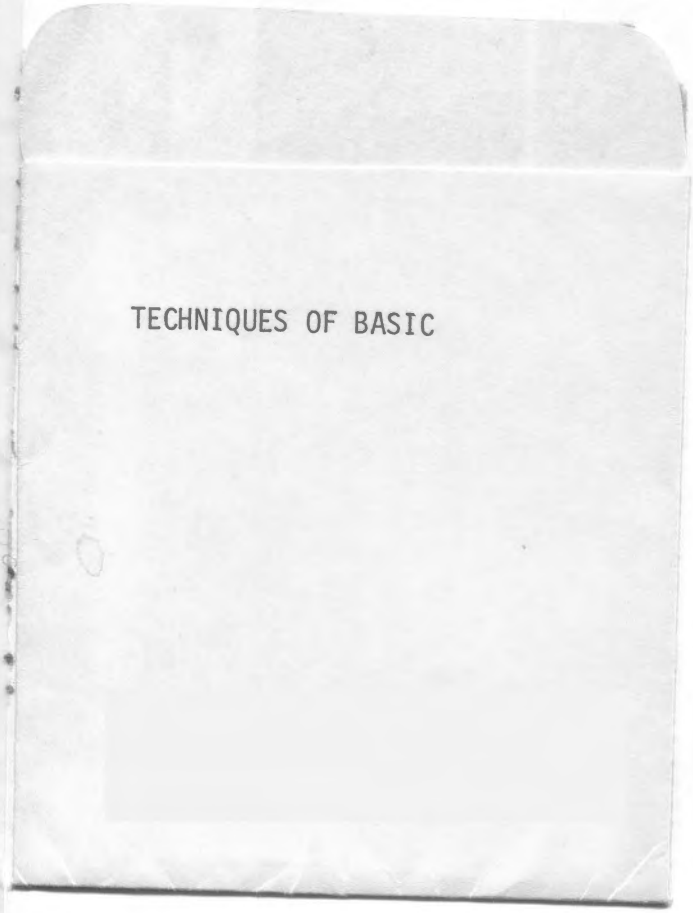
Personal Computer Series/John P. Grillo and J. D. Robertson



Microcomputer Power

TECHNIQUES OF BASIC

John P. Grillo/J. D. Robertson
Bentley College
Waltham, Massachusetts



wcb
Personal Computer Series
Wm. C. Brown Company Publishers
Dubuque, Iowa 52001

Library of Congress Catalog Card Number: 81-65445

ISBN 0—697—09951—2

Third Printing, 1982

Printed in the United States of America

TECHNIQUES OF BASIC

To *Our families*

Contents

Acknowledgements

When two people decide to write a book, the problems of planning, communication, and finally production are compounded. Were it not for the efforts of Ed Bowers and Rita Dunkin of the Wm. C. Brown Business, Professional, and Trade Division, we would not have maintained our enthusiasm during this project. We are indebted to them for their patience and foresight.

We owe special thanks to Steve Grillo, who as a 1980 high school graduate, has demonstrated a level of talent and responsibility beyond his years. He proofread the book, helped type the final manuscript, shot, developed, and printed all pictures in the book, and wrote a substantial portion of the programs.

Introduction

BASIC has come a long way since its first days at Dartmouth College in 1964, when because of its simplicity it helped students to learn about the computer. It has evolved in two stages. The first stage occurred in the early 1970's when minicomputers became standard fixtures in many small business, scientific, and educational environments. At that time BASIC became more than a curiosity. Because of its expanded features, particularly file management, it began to appear as the application language of choice for the popular minis.

The second stage of BASIC's evolution is occurring right now, at the turn of the decade. The popularization of the microcomputer in the last three years of the 1970's has resulted in BASIC being the *de facto* standard as a high-level language for these new devices. Remember that minis were used primarily in small businesses, scientific labs, and schools. The micros have come into the home, and BASIC has come with them. Suddenly the phrase "computer power to the people" means something tangible to millions of individuals. The decade of the '80s is going to see a substantial fraction of the public actively involved in developing programs for their acquisition, and most of these programs will be written in BASIC.

All this is fine, as long as this tool is used for its intended purpose, that being to entertain, educate, calculate, and manage files. However, many purchasers of microcomputers will bring it home, play a few games of Blackjack, Chess, or Star Trek, and perhaps maintain a recipe file. This is not enough. These devices are more powerful than the million-dollar computers of the 1960's, and to use them only for such trivial tasks is to waste their true potential. It's as if you were to buy a TV set and leave it tuned to just one channel. Microcomputer power should be explored and exploited to its fullest, and one way you can do so is to use it for more than repetitive execution of one or a few programs. Program it yourself.

As educators we have exposed many students to the joys of computer programming, and we are continually surprised at the variety of people who exhibit a talent for this science, or art, or craft. No general rule seems to apply; programming talent seems to appear in a fairly large and unpredictable segment of the population.

The microcomputer revolution will add greatly to the growing numbers who know how to write programs. A few of these people will become excellent programmers. Our aim with this book is to increase the ranks of better programmers by exposing them to some techniques for solving problems that are commonly found in a wide variety of applications.

When you write a program, remember that you must consider three different points of view.

1. The *programmer* is the originator of the program, its creator. In many situations, you will find no existing program that even remotely begins to solve your problem. This is when your skill as a programmer is tested to its fullest. You are most of all a problem solver at this stage, and your major task is to decide on the method of solution, or algorithm, for your problem.
2. The *reader* of your program is very possibly also its author, but may also be someone else who wishes to adapt it to his or her own application. A program's reader must understand the fundamentals of the language about as well as the programmer, but is rarely involved in its original creation. The remarks in a program are intended for its reader. During your program's development, you are also its reader, and you can use the remarks effectively to remind you of the program's logic or to help modularize it for easier alteration.
3. The *user* of your program is the intended target for its application. Usually, that person is naive about computers and programs. The program, its advanced techniques, and its wealth of remarks are lost to the user. But you, the programmer, must always keep the user in mind. Here's one of the few general rules that has no exceptions: All well written programs are easy to run.

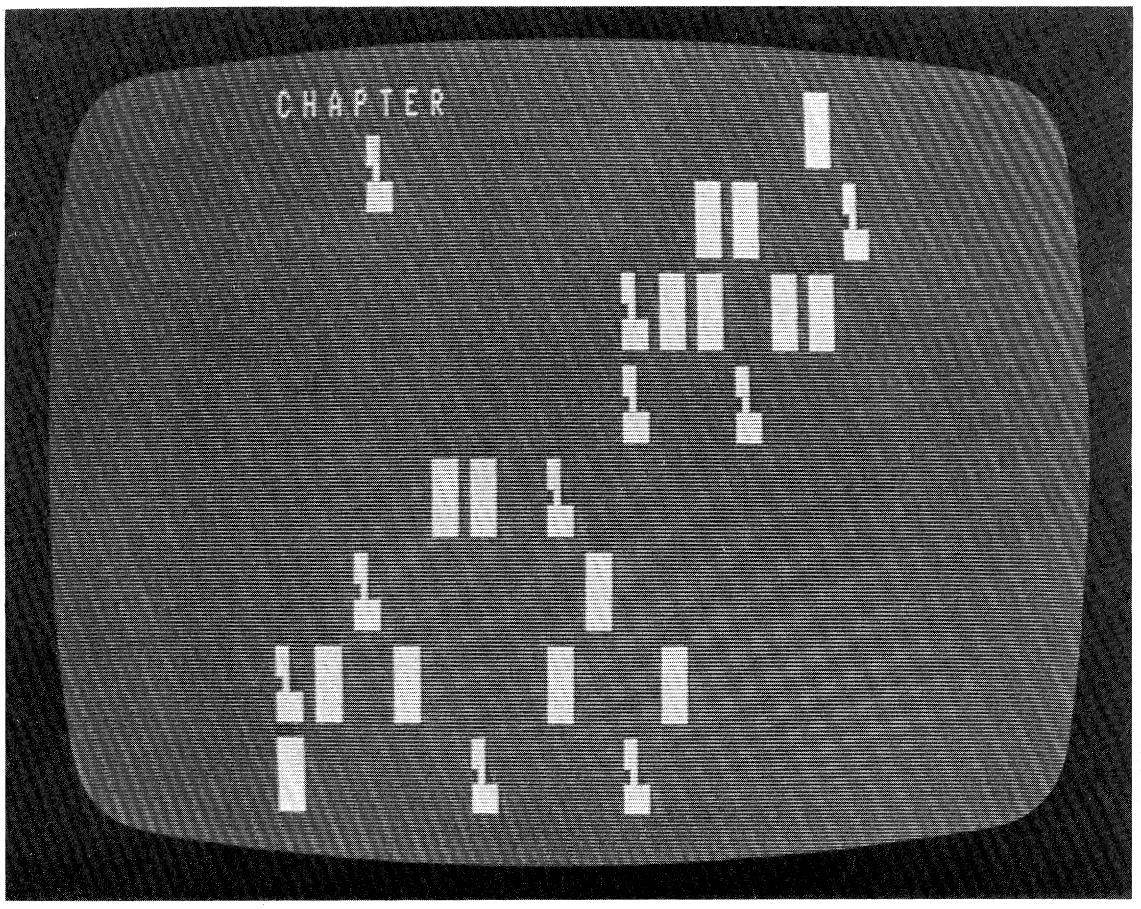
The programs, program segments, and examples contained within this book have been tested on a Radio Shack TRS-80 Level-II system with one minifloppy disk drive. The listings were produced on an Integral Data Systems Model 225 printer. We have purposely oriented this book toward the TRS-80 for the following reasons.

1. The TRS-80 is one of the many microcomputers that use Microsoft BASIC, which is fast becoming a standard of comparison for performance and for variety of extensions.
2. The TRS-80 as of this writing is the most popular microcomputer for personal use, and its Level II BASIC is a subset of the BASIC that is implemented on its bigger brother, the TRS-80 Model II. The Model II gives all indications of becoming a very popular small business computer.
3. The TRS-80 does not have a color display monitor, so its BASIC does not have the variety of commands to manage that aspect of output. This may seem to be a disadvantage,

but the fact is that of all of the sets of BASIC commands that are available on microcomputers, those that deal with color displays are the least standardized.

4. The TRS-80 has a wide range of available peripheral equipment. In addition, the TRS-80's popularity has prompted many peripheral manufacturers other than Tandy to produce competing hardware, including hard disks with capacities on the order of 20 million characters.

We hope that you will try out all of the features that are discussed in this book. Your reward will be a deep understanding of both a fine computer programming language and some excellent programming techniques.



Decisions and Branching

The IF–THEN and the GOTO are certainly simple to learn and understand, but as a person improves in programming techniques, the limitations of these statements become a real burden. This is where the extensions to the language are particularly rewarding. They are very easy to learn and use, and they make any program easier to read.

IF–THEN–ELSE

Primitive BASIC is limited to having just a line number following the THEN, for example:

```
300 IF X=A THEN 820
```

This restriction leads to awkward programs full of GOTOs that force the reader to jump around from one line of code to another. This process of bypassing some lines and tracing the program in various sequences tends to frustrate both the programmer and the reader. As an example of this poor, and all too common type of programming, look at the program below:

```
10 'FILENAME: "C1P1"
20 'FUNCTION: TO FIND LARGEST OF 3 NUMBERS (POORLY STRUCTURED)
30 ' AUTHOR :   JPG                DATE:  12/79
40 DATA 1,2,3,1,3,2,22,11,33,22,33,11,35,15,25,35,25,15,0,0,0
50 'read three values from data block
60 READ A,B,C
70 IF A*B*C=0 THEN 10000
80 'check to see if A is largest
90 IF B > A THEN 130
100 IF C > A THEN 170
110 L = A
120 GOTO 190
130 'check to see if B is largest
140 IF C > B THEN 170
150 L = B
160 GOTO 190
170 'here we know that C is largest
180 L = C
190 'print the value of L; it is the largest
200 LPRINT L; "IS THE LARGEST OF"; A; B; C
210 GOTO 60
10000 END
```

```
 3 IS THE LARGEST OF 1  2  3
 3 IS THE LARGEST OF 1  3  2
 33 IS THE LARGEST OF 22  11  33
 33 IS THE LARGEST OF 22  33  11
 35 IS THE LARGEST OF 35  15  25
 35 IS THE LARGEST OF 35  25  15
```

*Note: The LPRINT command is used here and throughout the book whenever we wish to show the program's output. To run the program on your computer and have the output appear on the screen, just change all LPRINT commands to PRINT.

The program C1P1 is difficult to compose, to trace, and to debug. Extended BASIC lets the programmer instruct the computer to do something after finding out that the condition is true, instead of just branching somewhere else. Look at this rewrite of the same program.

```
10 'FILENAME: "C1P2"
20 'FUNCTION: TO FIND LARGEST OF 3 NUMBERS (BETTER)
30 ' AUTHOR :   JPG              DATE:  12/79
40 '
50 DATA 1,2,3,1,3,2,22,11,33,22,33,11,35,15,25,35,25,15,0,0,0
60 'read three values from data block
70 READ A,B,C
80   IF A*B*C=0 THEN 10000
90 '  store the larsest in L, then print L
100     IF A>B THEN IF A>C THEN L=A
110     IF B>A THEN IF B>C THEN L=B
120     IF C>A THEN IF C>B THEN L=C
130   LPRINT L; "IS THE LARGEST OF"; A; B; C
140 GOTO 70
10000 END
```

Notice that the decisions in this program have no branches. Each IF statement checks the truth of a pair of conditions, and the value of L is set when both conditions within the same statement are true.

But extended BASIC has even more. The THEN clause may be followed by another clause, called the ELSE clause. The resulting compound statement allows the programmer to specify one statement to be executed if the condition is true, and another statement if the condition is false.

```
50 'if discriminant D of quadratic equation is non-negative,
60 'then compute and print the roots; otherwise print the
70 'message, "NO REAL ROOTS"
80 D=B*B-4*A*C '   calculate the discriminant
90 D2=2*A '   calculate denominator of quadratic equation
100 IF D>=0 THEN PRINT "ROOTS=";(-B*SQR(D))/D2;(-B-SQR(D))/D2
                ELSE PRINT "NO REAL ROOTS"
110 '
120 '
200 'let user stop or proceed, but accept only YES or NO answer
210 PRINT "DO YOU WANT TO GO ON (ANSWER YES OR NO)";
220 INPUT A$
230 IF A$="NO" THEN STOP
              ELSE IF A$<>"YES" THEN 210
```

Logical Operators

This feature allows more English-looking conditional statements by the use of OR, AND, and NOT operators.

```
10 'if A is less than B and A is less than C,
20 'then print A as the smallest.
30 IF A<B AND A<C THEN PRINT A; "IS SMALLEST"
40 IF A$="YES" OR A$="SURE" OR A$="OK" THEN 500
```

Program C1P3 shows how much more readable these logical operators are in a program, as opposed to nested IFs or an abundance of GOTOs.

```
10 'FILENAME: "C1P3"
20 'FUNCTION: TO FIND LARGEST OF 3 NUMBERS (BEST)
30 ' AUTHOR :   JPG                DATE:  12/79
40 '
50 DATA 1,2,3,1,3,2,22,11,33,22,33,11,35,15,25,35,25,15,0,0,0
60 'read three values from data block
70 READ A,B,C
80   IF A*B*C=0 THEN 10000
90 'find and print the largest all in one shot
100 'note that the program occupies more space in memory,
110 '   but its operation is very clear.
120   IF A>B AND A>C THEN LPRINT A; "IS THE LARGEST OF";A;B;C
130   IF B>A AND B>C THEN LPRINT B; "IS THE LARGEST OF";A;B;C
140   IF C>A AND C>B THEN LPRINT C; "IS THE LARGEST OF";A;B;C
150 GOTO 70
10000 END

 3 IS THE LARGEST OF 1  2  3
 3 IS THE LARGEST OF 1  3  2
 33 IS THE LARGEST OF 22  11  33
 33 IS THE LARGEST OF 22  33  11
 35 IS THE LARGEST OF 35  15  25
 35 IS THE LARGEST OF 35  25  15
```

The logical operators AND, OR, and NOT can be used for Boolean logic operations. Study this statement:

```
10 ' set A to TRUE (-1) if both conditions are true,
                        otherwise to FALSE (0)
20 A=(X=Y) AND (J>0)
```

The parentheses around each of the conditions are necessary to isolate the conditions from the assignment of the answer to A.

```
10 'set the flag V to TRUE if A is not less than B
20 V=NOT(A<B)
```

This statement sets V to true (-1) if the statement is true; that is, the value of A is not less than B. Notice that the statement

```
10 V=(A>=B)
```

does the same thing, and it is perhaps clearer.

There are some applications for using purely logical operators. Remember that in these cases the values in question are stored by the computer as either true (-1) or false (0). Such applications lead to statements like these:

```
10 'set P to -1 if X is 0 and vice versa
20 P=NOT(X)
30 IF J THEN PRINT "TRUE"
40 IF NOT(A AND B) THEN PRINT "NEITHER IS TRUE"
50 IF (A OR B) THEN PRINT "EITHER ONE OR BOTH IS TRUE"
```

A third possible application of logical operators is in bit manipulation or bit comparison. This could be used in a program to identify the positions of the 1-bits in any variable, in effect representing it in binary form. The program C1P4 exemplifies the problem, converting the variable X that the user input to its binary representation. The test value T starts at the value 16384, which is two to the fourteenth power. Each time through the loop in lines 100-150, T is reduced by a factor of two, in effect shifting the single one-bit to the right one position.

```
10 'FILENAME: "C1P4"
20 'FUNCTION: DISPLAY LAST 15 BITS OF INTEGER X
30 ' AUTHOR :   JPG              DATE: 12/79
40 '
50 ' set t to be the first (largest) power of 2
60 T=16384
70   PRINT "WHAT INTEGER DO YOU WISH CONVERTED (0=STOP)";
80   INPUT X
90     IF X=0 THEN 10000 ELSE LPRINT X, "IN BINARY IS  ";
100   FOR I=1 TO 15
110 '             isolate the single bit from X
```

```
120      B=T AND X
130        IF B>0 THEN LPRINT "1 "; ELSE LPRINT "0 ";
140      T=T/2
150    NEXT I
160    LPRINT
170 GOTO 60
10000 END
```

```
1              IN BINARY IS  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
2              IN BINARY IS  0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
15             IN BINARY IS  0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
3456           IN BINARY IS  0 0 0 1 1 0 1 1 0 0 0 0 0 0 0
32767          IN BINARY IS  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Some of the effects of binary operations using the logical operators can be very misleading. You should study the explanations of these operations in the Level II Manual. We include the examples below more for completeness than for clarification. We suggest that you use these operations only if you feel comfortable with binary representation of values in the computer.

Instruction	Output
10 LPRINT 0 AND 0	0
20 LPRINT 0 AND 1	0
30 LPRINT 1 AND 0	0
40 LPRINT 1 AND 1	1
50 LPRINT 0 OR 0	0
60 LPRINT 0 OR 1	1
70 LPRINT 1 OR 0	1
80 LPRINT 1 OR 1	1
90 LPRINT NOT 0	-1
100 LPRINT NOT -1	0

ON–GOTO and ON–GOSUB

These closely related branching statements allow a great deal of flexibility when a program needs to perform a multiple-way branch. They both use a variable after the ON, and a series of line numbers after the GOTO or GOSUB. The integer value of the variable is calculated, and a branch is taken to the first statement if the variable is 1, the second if 2, the third if 3, and so on. On the TRS-80, as on most other computers, if the integer value of the variable does not correspond to the position of a given line number, the statement following the ON–GOTO or ON–GOSUB is executed.

Example:

```
50 ON X GOTO 80, 300, 750, 10, 80, 900
60 ' fall through if X<0 or X>6
```

The computer obtains the integer portion of X, which must be between 0 and 255.

If the integer portion of X is	then the computer branches to this line number
<0	ERROR MESSAGE
0	60 (the next line—no branch)
1	80
2	300
3	750
4	10 (notice that the line numbers do not need to be in order)
5	80 (notice that the same line can be reached with different values of X)
6	900
7-255	60 (the next line—no branch)
>=256	ERROR MESSAGE

You could write the equivalent of line 50 above without use of an ON–GOTO like this:

```
50 IF INT(X)=1 THEN 80
51 IF INT(X)=2 THEN 300
52 IF INT(X)=3 THEN 750
53 IF INT(X)=4 THEN 10
54 IF INT(X)=5 THEN 80
55 IF INT(X)=6 THEN 900
60.............
```

If the word GOTO was replaced by GOSUB, the multiway branch would become:

```
50 ON X GOSUB 80, 300, 750, 10, 80, 900
62 'fall through if X<0 or X>6
```

which is equivalent to:

```
50 IF INT(X)=1 THEN GOSUB 80 : GOTO 62
52 IF INT(X)=2 THEN GOSUB 300: GOTO 62
54 IF INT(X)=3 THEN GOSUB 750: GOTO 62
56 IF INT(X)=4 THEN GOSUB 10 : GOTO 62
58 IF INT(X)=5 THEN GOSUB 80 : GOTO 62
60 IF INT(X)=6 THEN GOSUB 900
62 ............
```

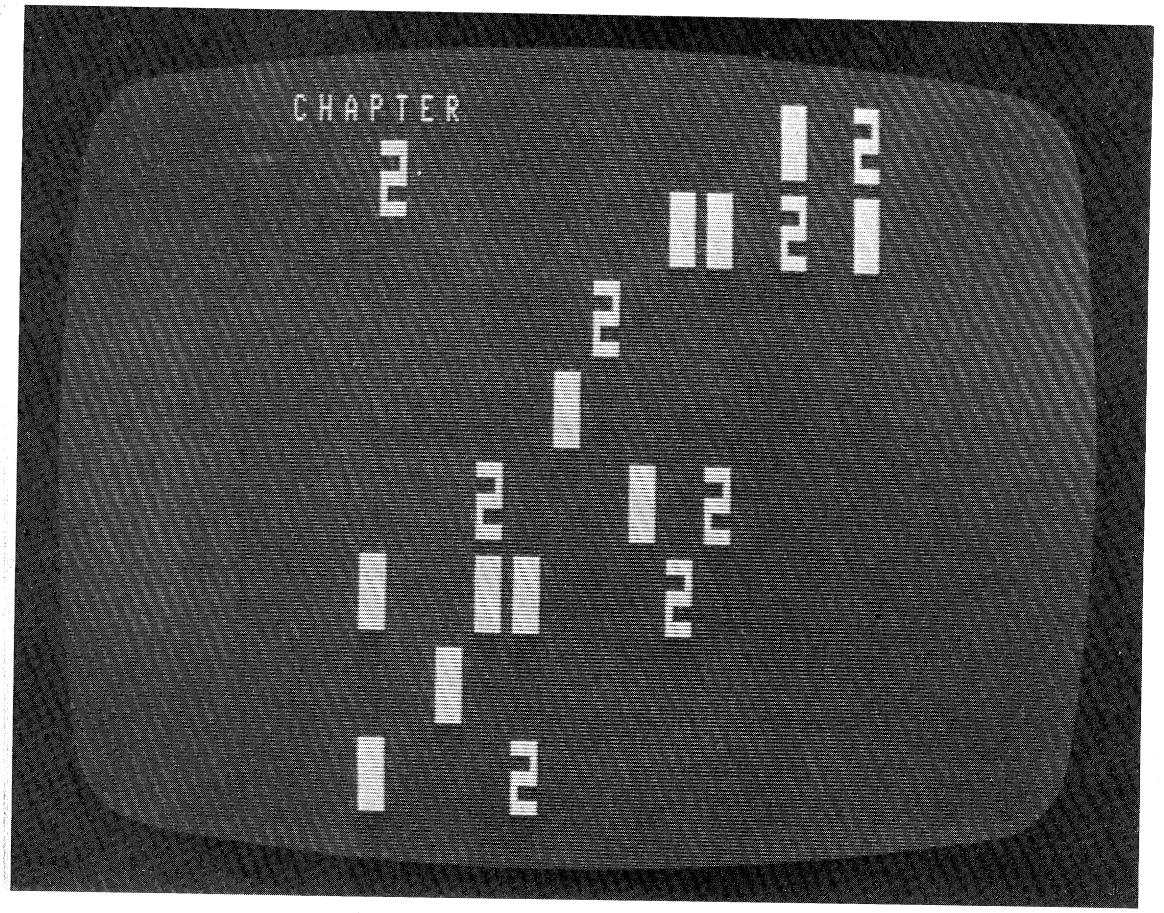
Obviously the economy of the ON–GOTO and ON–GOSUB makes them very attractive to any programmer who has this kind of branching to perform.

One nice application of these statements is in a branch that is based on the sign of a number. Suppose your program must branch (with either a GOTO or a GOSUB) to line 50 if the value of $X^2-4=0$, and to line 150 if X^2-4 is positive. Either one of these statements would do it.

```
20 ON SGN(X*X-4)+2 GOSUB 50, 100, 150
20 ON SGN(X*X-4)+2 GOTO  50, 100, 150
```

This three-way branch uses the fact that the SGN function evaluates its argument, and returns a -1 if the argument is negative, 0 if it is zero, and +1 if it is positive.

In this chapter, we have explored a few of the many extensions that provide flexibility and make the programmer's job easier and in many ways more enjoyable.



Statements and Functions

Language features such as multi-statement lines, RANDOM, DIM, string functions, and many more are incorporated into extended BASIC to make the programmer's job easier and to provide problem solving power seen in other high-level programming languages. This chapter explores these extensions and suggests some possible applications.

Multiple Statements on a Line

Because the variable names in primitive BASIC are limited in size to no more than three characters, many BASIC programs take on the appearance of one long list of tiny statements down the left side of the screen. This is somewhat inconvenient when the screen is limited to displaying 16 lines, as on the TRS-80.

Fortunately, most BASICs allow more than one statement on a given line. The Microsoft Company (Bellevue, Washington) has been a leader in developing languages for microcomputers. Many microcomputers, including the Apple-II and the TRS-80, use some version of BASIC developed by Microsoft. For a full comparison of two popular BASICs on microcomputers, see Appendix A. All versions of Microsoft BASIC, including the TRS-80's Level II, use a

colon (:) to separate statements, but some other BASICs use other symbols. For example, DEC's BASIC-PLUS that is used on the PDP-11 series of computers uses the backslash (\). We will use the colon because it is used by Level II BASIC on the TRS-80, and it is also the most common statement separator for microcomputers.

```
10 Z=0: Y=1: P=3.14159: E=EXP(1)
20 X$=X$+J$: IF I<20 THEN I=I+1: GOTO 20
30 IF J1>J2 THEN T=J1: J1=J2: J2=T
```

Note that these examples do not necessarily improve the readability of the code. In many cases where multiple statements are crowded into one line the readability suffers as far as the understanding of the program logic. This problem is often alleviated through *indentation* and *logical grouping* in its multistatement lines. Consider the two examples that follow:

```
90 'crowded -- saves space at the cost of readability
95 '
100 PRINT: PRINT "WHAT ACTIVITY (0=STOP)";
110 INPUT A: IF A=0 THEN STOP ELSE ON A GOTO 200, 300, 400
120 GOTO 100
```

```
90 'better -- multiple statement lines changed to
95 '          to maintain logical flow of program
100 PRINT: PRINT "WHAT ACTIVITY (0=STOP)";: INPUT A
110   IF A=0 THEN STOP
            ELSE ON A GOTO 200, 300, 400
120 GOTO 100
```

In the first example, the IF statement is crowded into the same line as the INPUT, while in the second example, the IF statement is isolated and indented for high legibility, and the INPUT is relocated with the output to produce one line that generates the message. Note that statement 110 has a line feed (↓) after the word STOP instead of a carriage return (ENTER). This allows statement 110 to be spread over two lines, greatly improving the overall readability of the program.

RND Function

The RND function is commonly found on primitive BASICs as well as the extended versions of the language. Usually, its argument is immaterial or even omitted, and its purpose is to return a pseudo random number between zero and one. A pseudo random number differs from a truly random number in that the former is produced

by an algorithm that uses a seed value to generate a number apparently at "random". For various reasons, most pseudo random number generators use a large prime number as a seed for the first random value, then use that random value, or some modification of it, as a seed for the second value, and so on.

Level II BASIC on the TRS-80 is similar to most BASICs in that if the RND function call uses 0 as an argument, the returned value is a positive real number between 0 and 1.

```
10 'random numbers between 0 and 1
20 '
30 FOR I=1 TO 5: LPRINT RND(0);: NEXT I
10000 END
```

```
 .988032   .64224   .047616   .0655122   .423838
```

A useful extension to this function is its ability to return a pseudo random integer within a given range. You can return random integers between 1 and 6 with the function call RND(6) which might be used to simulate the throw of one die.

```
10 'random integers
20 '
30 LPRINT "1 TO 6", "1 TO 10", "1 TO 52", "0 TO 1"
40 FOR I=1 TO 10
50   LPRINT RND(6), RND(10), RND(52), RND(2)-1
60 NEXT I
10000 END
```

1 TO 6	1 TO 10	1 TO 52	0 TO 1
4	5	30	0
6	5	49	0
4	8	11	1
3	2	7	0
5	9	41	1
5	5	29	1
2	10	38	1
2	2	50	1
5	3	25	0
6	8	48	0

RANDOM

A feature of pseudo random number generators is that the initial seed is always the same. If you turn off the computer, then reload and rerun the program, you get the same output.

```
10 'FILENAME: "C2P1"
20 'FUNCTION: GENERATE 50 REPEATABLE RANDOM NUMBERS
30 ' AUTHOR :   JPG                DATE:  12/79
40 '
50 FOR I = 1 TO 50
60   LPRINT RND(1000);
70 'return the carriage after every 10th value
80     IF INT(I/10)*10=I THEN LPRINT
90 NEXT I
10000 END
```

```
 368  72  219  825  52  891  119  981  911  616
 912  844  856  153  324  550  271  532  832  409
 500  199  519  75  83  62  688  474  736  292
 219  470  92  866  677  290  178  867  173  697
 655  528  976  907  798  31  866  266  234  53
```

The purpose of the RANDOM statement is to seed the generator with a different value, based on some varying internal value in the computer. Add a new line

```
45 RANDOM
```

and run the program after turning off the computer and then powering it back up. Notice that the output is different.

```
250  669  426  63  16  733  905  368  147  718
255  913  75  201  849  823  798  494  334  705
731  820  860  689  321  232  73  232  405  287
840  658  317  465  741  461  752  345  299  345
370  523  966  29  162  285  858  787  552  31
```

DIM and Subscripted Variables

BASIC is not limited to managing its variables in memory one at a time. It can allow the programmer to set up lists and tables in memory by name, and access specific positions of those lists and tables by using subscripts.

The DIM statement does two things:

(1) It names a list or table.
(2) It sizes that list or table.

Examples:

```
10 DIM A(50)
```

The variable named A is a list 50 values long.

```
20 DIM X(50,20)
```

The variable X is a table with 50 rows and 20 columns.

```
30 DIM V1$(30)
```

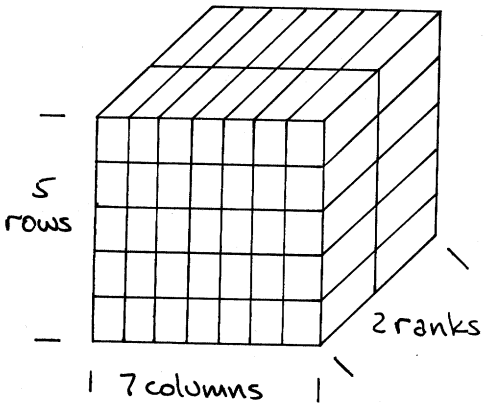
The string variable V1$ is a list of 30 strings.

Most people call dimensioned variables *arrays*, so that a list is called a one-dimensional array and a table is a two-dimensional array.

The statement

```
10 DIM X(5,7,2)
```

creates a three-dimensional array. You can imagine it as a cube, with 5 rows, 7 columns, and 2 ranks.



Most microcomputer BASICs, including Level II for the TRS-80, allow the creation of arrays with multiple dimensions.

When a program refers to a particular element of an array, it does so by subscript. In mathematics, a matrix element is referred to by its subscripts, such that the matrix A has an element $A_{i,j}$.

In BASIC, a true subscript cannot be written below the line, so it is parenthesized. Thus the programmer can refer to the seventh element of the one-dimensional array X as X(7).

Examples:

```
10 DIM A(5,20) 'set up the array, 5 rows, 20 columns
20 A(1,12)=50 'place the value 50 in row 1, column 12
30 A(2,20)=A(1,1) 'copy row 1, col. 1 into row 2, col. 20
40 X=3: Y=7: A(X,Y)=8.3 'place the value 8.3 in row 3, col. 7
50 'set all values to -1
60 FOR I=1 TO 5: FOR J=1 TO 20: A(I,J)=-1: NEXT J, I
```

Notice in line 10 the use of the apostrophe as a substitute for the REM to denote a remark. Also notice in line 20 that the apostrophe does not have to follow a colon to start a remark in the middle of a line. These niceties allow more flexibility in the practice of writing remarks.

One common use of multiply subscripted arrays is in the statistical analysis of polls and questionnaires. Suppose these are the characteristics that one wishes to analyze:

Characteristics	Number of possible responses
Sex	2 (male and female)
Age	3 (under 20, 20 to 50, over 50)
Geographical Location	5 (Pacific, Mountain, Midwest, South, Atlantic)
Political Preference	3 (Republican, Democratic, Other)
Answer to Poll Question	5 (Strongly disagree, disagree, no opinion, agree, strongly agree)

Now suppose that 50,000 observations are made all over the country, and that they are encoded as numeric values. For example, the coded response 13225 represents a male more than 50 years old, living in the Mountain states, a Democrat, whose answer was "strongly agree", The encoded value is generated this way: The response to "sex", a 1 or a 2, is multiplied by 10,000. The age response is multiplied by 1000 and added to the previous number, 10,000 or 20,000. The remaining three responses are multiplied by 100, 10, and finally 1. The resultant integer sum lies between 11,111 and 23,535. Note that this encoding scheme won't work for polls of 6 or more questions, or with 5-question polls in which the first response is greater than 3. An interesting challenge might be to develop a coding scheme in octal integer representation, which would allow values of 37777, or even 77777.

The results of the entire poll could be stored on one or two cassette tapes for input to a microcomputer for analysis. The program segment below transfers all of the information on tape into a 5-dimensional array for analysis. Rather than storing one questionnaire's results as a single value, say the integer 13225, this program adds one to the tally of like responses in the 5-dimensional array. The DIM A(2,3,5,3,5) statement names the array A and sizes it as 2x3x5x3x5, or 450 values. A program such as the one below could analyze almost any number of questionnaire responses of this type, and yet run on a moderately configured microcomputer system.

The program C2P2 listed below does such an analysis. It generates its own test data, because it is the analysis technique rather than the data which is the issue here. The output that follows the program shows a sample run. Notice how difficult it is to tally like responses by scanning the sea of numbers. Imagine how error-prone such a visual analysis would be with a sample size in the thousands, instead of just 297.

```
10 'FILENAME: "C2P2"
20 'FUNCTION: QUESTIONNAIRE ANALYSIS WITH LARGE ARRAY
30 ' AUTHOR :   JPG                    DATE: 12/79
40 '
50 DEFINT A-Z
60 DIM A(2,3,5,3,5), B(5)
70 'input from tape is simulated with artificial data
80 LPRINT"T A B L E   O F   A R T I F I C I A L   D A T A"
90 LPRINT                'assume a total of 297 responses
100 FOR OBS = 1 TO 297
110   GOSUB 500: LPRINT X;'generate one 5-answer response
120    IF INT(OBS/9)*9=OBS THEN LPRINT 'new line every 9th
130   FOR J=5 TO 1 STEP -1    'extract each digit as B(J)
140     B(J)=X-INT(X/10)*10: X=X/10
150   NEXT J
160   ' add 1 to proper position of A
170   '      A(B(1),B(2),B(3), ... is tedious
180   D=B(1): E=B(2): F=B(3): G=B(4): H=B(5)
190   A(D,E,F,G,H) = A(D,E,F,G,H) + 1
200 NEXT OBS: LPRINT: LPRINT
210 'sum up all responses according to sex
220 GOSUB 600
230 LPRINT "SUM OF MALE RESPONDENTS =";S1
240 LPRINT "SUM OF FEMALE RESPONDENTS =";S2
250 'print user-selected elements of the array A
260 INPUT "SEX:  M=1  F=2  STOP=0"; N1
270   IF N1=0 THEN 10000
280 INPUT "AGE:   <30=1  30-50=2  >50=3"; N2
290 INPUT "AREA:   PAC=1  MTN=2  MDW=3  S=4  ATL=5"; N3
300 INPUT "PARTY:   REP=1  DEM=2  OTHER=3"; N4
310 T=0     'total all respondents for this category
320 FOR I=1 TO 5: B(I)=A(N1,N2,N3,N4,I): T=T+B(I): NEXT I
330 'report the results of the tally
340 LPRINT"SEX =";N1,"AGE =";N2,"AREA =";N3,"PARTY =";N4
350 FOR I=1 TO 5
360   LPRINT B(I); "OF THIS GROUP ANSWERED";I
370 NEXT I
380 LPRINT "TOTAL OF RESPONDENTS";N1;N2;N3;N4;"X =";T
390 LPRINT
400 GOTO 260
```

```
500 '*****  make artificial data using random function
510 J1=RND(2): J2=RND(3): J3=RND(5): J4=RND(3): J5=RND(5)
520 X=(((J1*10+J2)*10+J3)*10+J4)*10+J5
530 RETURN
600 '*****  sum the sexes
610 FOR N2 = 1 TO 3          'N2=subscript for age
620   FOR N3 = 1 TO 5        'N3=subscript for location
630     FOR N4 = 1 TO 3      'N4=subscript for pol. pref.
640       FOR N5 = 1 TO 5    'N5=subscript for answer
650         S1 = S1 + A(1,N2,N3,N4,N5)  'sum of males
660         S2 = S2 + A(2,N2,N3,N4,N5)  'sum of females
670 NEXT N5,N4,N3,N2
680 RETURN
10000 END
```

T A B L E O F A R T I F I C I A L D A T A

13224	23321	13215	11411	13223	23524	22422	22131	13225
11122	23435	13135	21521	21321	23114	13311	11131	21224
12312	22322	23515	13215	13213	13512	11114	21235	13331
22122	12212	12525	23313	12233	11521	12533	21213	21222
22123	13325	22421	22434	22333	11511	21334	23324	12535
22421	12335	21422	11332	23231	13234	22434	22531	12411
23433	13335	21422	22521	11322	21435	13422	11334	23515
13312	21132	13531	12315	12225	22311	12211	23112	11114
13125	21521	21222	21135	13321	11112	12122	13134	12231
23135	21334	11334	22133	11113	22323	22521	21135	11533
11235	13431	22525	11422	12335	23424	13423	22425	22221
21522	23331	22134	12533	11132	11112	22231	22315	11513
13523	11311	23334	23532	13123	12314	13314	13421	21434
21313	11322	22324	12524	22125	21121	12312	23413	21211
12421	13521	11523	21235	21211	23523	11313	21325	13531
11533	11533	12231	23314	11311	12531	23424	23424	21332
12312	23135	21424	22134	13423	23132	11333	21412	21132
21312	12225	13422	21211	12534	11512	13522	11412	12535
21411	13211	21335	23212	23515	22134	23132	13121	13313
12222	22534	23125	12225	13425	13124	22425	11314	21115
11131	21114	22424	13111	13525	11224	23112	12325	21424
22213	23125	21415	13212	23521	23432	12115	23322	22123
11433	23132	11312	23531	22534	23125	11435	22435	21335
23323	13315	12512	11124	23313	12135	21121	13312	13511
23514	22431	23234	12435	11121	22115	22233	22425	23414
11423	11534	21213	21333	23514	13112	13311	21225	23511
11131	23415	11311	22233	21121	21421	22222	21422	13521
13113	22515	11424	13535	23533	13533	21522	13424	23235
12431	12224	12134	23524	22112	13134	21213	21434	22523
13124	11231	12433	13223	12335	21321	22333	23333	13324
23411	12435	22113	22315	12315	12122	21222	22412	21432
22324	11331	22233	23535	12313	23323	23215	21221	23222
12235	22331	13323	22321	13325	22334	23432	22135	21211

```
 SUM OF MALE RESPONDENTS = 140
 SUM OF FEMALE RESPONDENTS = 157
 SEX = 1          AGE = 2          AREA = 3          PARTY = 1
  0 OF THIS GROUP ANSWERED 1
  3 OF THIS GROUP ANSWERED 2
  1 OF THIS GROUP ANSWERED 3
  1 OF THIS GROUP ANSWERED 4
  2 OF THIS GROUP ANSWERED 5
 TOTAL OF RESPONDENTS 1  2  3  1 X = 7

 SEX = 2          AGE = 3          AREA = 5          PARTY = 3
  1 OF THIS GROUP ANSWERED 1
  1 OF THIS GROUP ANSWERED 2
  1 OF THIS GROUP ANSWERED 3
  0 OF THIS GROUP ANSWERED 4
  1 OF THIS GROUP ANSWERED 5
 TOTAL OF RESPONDENTS 2  3  5  3 X = 4

 SEX = 1          AGE = 1          AREA = 1          PARTY = 1
  0 OF THIS GROUP ANSWERED 1
  2 OF THIS GROUP ANSWERED 2
  1 OF THIS GROUP ANSWERED 3
  2 OF THIS GROUP ANSWERED 4
  0 OF THIS GROUP ANSWERED 5
 TOTAL OF RESPONDENTS 1  1  1  1 X = 5

 SEX = 2          AGE = 2          AREA = 2          PARTY = 2
  1 OF THIS GROUP ANSWERED 1
  1 OF THIS GROUP ANSWERED 2
  0 OF THIS GROUP ANSWERED 3
  0 OF THIS GROUP ANSWERED 4
  0 OF THIS GROUP ANSWERED 5
 TOTAL OF RESPONDENTS 2  2  2  2 X = 2

 SEX = 2          AGE = 3          AREA = 3          PARTY = 3
  1 OF THIS GROUP ANSWERED 1
  0 OF THIS GROUP ANSWERED 2
  1 OF THIS GROUP ANSWERED 3
  1 OF THIS GROUP ANSWERED 4
  0 OF THIS GROUP ANSWERED 5
 TOTAL OF RESPONDENTS 2  3  3  3 X = 3
```

The two programs that follow exemplify the Monte Carlo technique, which is a common way to establish randomness in a series of observations.

The first program, C2P3, demonstrates how to shuffle a deck of 52 cards quickly and effectively on the computer by picking two cards at random and switching their positions. The pick-and-switch is repeated 100 times, although fewer switches might be just as effective.

```
10 'FILENAME: "C2P3"
20 'FUNCTION: SHUFFLE A DECK OF CARDS
30 ' AUTHOR : JPG                    DATE: 12/79
40 DIM C(52)
50 ' generate the deck and deal it in order
60 FOR I = 1 TO 52: C(I) = I: NEXT I: GOSUB 500
70 'shuffle the deck
80 RANDOM
90 'switch two cards at random 100 times
100 FOR I = 1 TO 100: X = RND(52): Y = RND(52)
110 T = C(X): C(X) = C(Y): C(Y) = T: NEXT I
120 'now deal it shuffled
130 GOSUB 500: GOTO 10000
500 'dealing subroutine (4 hands)
510 FOR I=1 TO 52
520 LPRINT C(I);: IF INT(I/13) * 13 = I THEN LPRINT
530 NEXT I: LPRINT: RETURN
10000 END
```

```
 1  2  3  4  5  6  7  8  9  10  11  12  13
14  15  16  17  18  19  20  21  22  23  24  25  26
27  28  29  30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49  50  51  52

18  3  17  27  37  19  14  2  10  8  51  26  31
4  25  5  13  30  22  49  36  20  41  15  50  35
24  23  52  6  11  32  29  44  21  9  42  45  28
39  33  43  34  46  16  48  47  7  1  40  12  38
```

The second Monte Carlo demonstration shows how the technique can be used to balance two (or more) series of events. Suppose a record company wishes to equate as closely as possible the total playing time on each of two sides of an LP record. The first side must contain 12 selections while the second side must contain 13 selections for a total of 25 selections with a total playing time of 93 minutes. Ideally, each side should be recorded with 46 minutes and 30 seconds of playing time. The problem arises when trying to find the 12 or 13 tunes whose playing time most closely approximates that figure. The Monte Carlo technique selects 12 tunes at random, adds their playing time, and keeps the "best" schedule.

```
10 'FILENAME: "C2P4"
20 'FUNCTION: MONTECARLO SELECTION OF SONG PROGRAMS
30 'AUTHOR  : JPG/JDR  12/79
40 '
50 DIM T$(25), M(25), S(25), T(25), K(25), L(25)
60 ' T$= song title              M = minutes per cut
70 ' S = seconds per cut         T = time per cut, in seconds
80 ' K = random pointer          L = second random pointer
90 '
100 ' read in the titles, minutes, and seconds
110 ' N = total number of songs, both sides
120 INPUT "HOW MANY OF THE 25 SONGS ON THIS ALBUM";N
130 LPRINT "THIS ALBUM HAS";N;"OF THE 25 SONG TITLES"
140 FOR I=1 TO 25
150   READ T$(I), M(I), S(I)
160   T(I)=60 * M(I) + S(I):  K(I) = I
170 NEXT I
180 C=60000: ' set smallest diff. between sides very high
190 INPUT "HOW MANY SCRAMBLES";N5
200 LPRINT "SELECTED NUMBER OF SCRAMBLES=";N5
210 FOR Q=1 TO N5              ' scramble songs N5 times
220   FOR I=1 TO N
230     J=RND(N): Z=K(I): K(I)=K(J): K(J)=Z
240   NEXT I
250 ' sum times for sides
260   Z1=0: Z2=0: N2=INT(N/2)
270   FOR I=1 TO N2
280     J=K(I): Z1=Z1+T(J): J=K(I+N2): Z2=Z2+T(J)
290   NEXT I
300     IF N/2<>INT(N/2) THEN Z2=Z2+T(N)
310   B=ABS(Z1-Z2): ' B = diff. in time between sides
320     IF B>=C THEN 380: ' C = previously smallest diff.
330   C=B: C1=Z1: C2=Z2
340   FOR I=1 TO N: L(I)=K(I): NEXT I
350   PRINT "LEAST=";C;"IN TRY";Q
360  LPRINT "LEAST=";C;"IN TRY";Q
370 IF C=0 THEN 390      'great! 0 is difference!
380 NEXT Q
390 PRINT: PRINT         'set next shuffle of times
400 LPRINT:LPRINT
410 M1=INT(C1/60): S1=C1-60*M1
420 PRINT "SIDE 1",M1;":";S1: LPRINT "SIDE 1",M1;":";S1
430 FOR I=1 TO N2: J=L(I)
440   PRINT T$(J)TAB(40)M(J)":"S(J)
450   LPRINT T$(J)TAB(40)M(J)":"S(J)
460 NEXT I
470 PRINT:PRINT:LPRINT:LPRINT
480 M2=INT(C2/60): S2=C2-60*M2
490 PRINT "SIDE 2", M2; ":"; S2
```

```
500 LPRINT "SIDE 2"; M2; ":"; S2
510 FOR I=N2+1 TO N: J=L(I)
520   PRINT T$(J)TAB(40)M(J)":"S(J)
530   LPRINT T$(J)TAB(40)M(J)":"S(J)
540 NEXT I
550 DATA "MAGMA COME LOUDLY",2,45,"CRAMP MY STYLE",2,20
560 DATA "PORKY AND TESS",5,21,"PUSHBUTTON POLKA",6,23
570 DATA "THE GODMOTHER THEME",4,33,"FIG NEWTON",3,2
580 DATA "MOTEL COLORADO",2,43,"YELLOW FEVER",2,10
590 DATA "STAGNANT",3,55,"NEW HAVEN NEW HAVEN",8,23
600 DATA "LIFE IN THE FAT LANE",3,21,"HOT SNUFF",5,31
610 DATA "FIRST RATE ROMANCE RITZ RENDEZVOUS",3,33
620 DATA "IRON ORCHID",3,36,"FIFTY-FIRST STREET",1,23
630 DATA "BLACK HOLE BLUES",2,51,"YELLOW PILLOW",6,0
640 DATA "PINACOLATAVILLE",2,44,"SALADA CANTATA",4,25
650 DATA "FLORIBUNDA",3,8,"TAKE THIS JOBBIN SHOVEL",3,8
660 DATA "STAR TRUCK",3,40,"SEMIHEMIDEMIQUAVER",2,2
670 DATA "IPHEGENIA IN QUEENS",2,34,"CABINETWORKS",3,42
10000 END

THIS ALBUM HAS 25 OF THE 25 SONG TITLES
SELECTED NUMBER OF SCRAMBLES= 50
LEAST= 46 IN TRY 1
LEAST= 36 IN TRY 7
LEAST= 8 IN TRY 28

SIDE 1            47 : 1
THE GODMOTHER THEME                4 : 33
HOT SNUFF                          5 : 31
IPHEGENIA IN QUEENS                2 : 34
FIFTY-FIRST STREET                 1 : 23
IRON ORCHID                        3 : 36
SALADA CANTATA                     4 : 25
PORKY AND TESS                     5 : 21
PUSHBUTTON POLKA                   6 : 23
BLACK HOLE BLUES                   2 : 51
TAKE THIS JOBBIN SHOVEL            3 : 8
STAGNANT                           3 : 55
LIFE IN THE FAT LANE               3 : 21
```

```
SIDE 2          47 : 9
PINACOLATAVILLE                              2 : 44
FIRST RATE ROMANCE RITZ RENDEZVOUS           3 : 33
FLORIBUNDA                                   3 : 8
CRAMP MY STYLE                               2 : 20
STAR TRUCK                                   3 : 40
YELLOW PILLOW                                6 : 0
SEMIHEMIDEMIQUAVER                           2 : 2
NEW HAVEN NEW HAVEN                          8 : 23
CABINETWORKS                                 3 : 42
YELLOW FEVER                                 2 : 10
FIG NEWTON                                   3 : 2
MOTEL COLORADO                               2 : 43
MAGMA COME LOUDLY                            2 : 45

THIS ALBUM HAS 17 OF THE 25 SONG TITLES
SELECTED NUMBER OF SCRAMBLES= 50
LEAST= 200 IN TRY 1
LEAST= 145 IN TRY 2
LEAST= 17 IN TRY 4
LEAST= 8 IN TRY 14

SIDE 1          35 : 11
NEW HAVEN NEW HAVEN                          8 : 23
FIG NEWTON                                   3 : 2
PORKY AND TESS                               5 : 21
YELLOW PILLOW                                6 : 0
YELLOW FEVER                                 2 : 10
LIFE IN THE FAT LANE                         3 : 21
HOT SNUFF                                    5 : 31
FIFTY-FIRST STREET                           1 : 23

SIDE 2          35 : 3
STAGNANT                                     3 : 55
CRAMP MY STYLE                               2 : 20
THE GODMOTHER THEME                          4 : 33
MAGMA COME LOUDLY                            2 : 45
PUSHBUTTON POLKA                             6 : 23
FIRST RATE ROMANCE RITZ RENDEZVOUS           3 : 33
BLACK HOLE BLUES                             2 : 51
MOTEL COLORADO                               2 : 43
IRON ORCHID                                  3 : 36
```

String Functions

String functions are designed to return strings or information concerning the strings that are referenced in the argument.

LEN

The LEN function returns the length of the string argument.

```
Instruction                                  Output
-----------                                  ------

10 PRINT LEN("ABC")                          3
20 PRINT LEN("COUNT"+"DRACULA")              12
30 PRINT LEN("MAC")+LEN("HINES")             8
40 A$="SCR": B$="AM"
50 IF LEN(A$)<10 THEN A$=A$+"E": GOTO 50
60 A$=A$+B$: PRINT LEN(A$);A$                12  SCREEEEEEEAM
```

Line 50 above pads A$ with as many "E"s as it takes to make it 10 characters long, then "adds" (concatenates) B$ to the result.

LEFT$ and RIGHT$

LEFT$ and RIGHT$ return substrings of the string argument for the length specified by the numeric argument.

LEFT$(X$,N) returns the N leftmost characters of X$.

RIGHT$(X$,N) returns the N rightmost characters of X$.

```
Instruction                                  Output
-----------                                  ------

70 PRINT LEFT$("ABCD",3)                     ABC
80 PRINT RIGHT$("ABCD",2)                    CD
90 A$="BAN": PRINT A$+RIGHT$(A$,2)+"A"       BANANA
100 PRINT LEFT$("FIRE"+"     ",6)+"Z"        FIRE  Z
```

MID$

MID$(A$,P,L) extracts a substring of the argument string A$ starting with the character at position P for a length of L characters. If the third argument L is omitted, the function returns all of the string starting at position P.

```
Instruction                                  Output
-----------                                  ------

100 PRINT MID$("ABCDE",3,2)                  CD
110 PRINT MID$("ABCDE",3,3)                  CDE
120 PRINT MID$("ABCDE",2)                    BCDE
```

If the position argument exceeds the length of the string, the returned ("extracted") string is null.

Instruction	Output
`130 B$=MID$("ABCDE",8): PRINT B$; LEN(B$)`	0

The MID$ function is useful for searching a string for a particular character or substring. This example shows how the length of a person's first name could be determined in order to reposition the last name.

```
10 'FILENAME: "C2P5"
20 'FUNCTION: REVERSE LAST AND FIRST NAME
30 ' AUTHOR : JPG                DATE: 6/79
40 '
50 CLEAR 100: INPUT "TYPE YOUR NAME -- LAST FIRST MI";N$
60 LPRINT N$
70 FOR I=1 TO LEN(N$)
80   IF MID$(N$,I,1)<>" " THEN NEXT I
90   IF I=LEN(N$) THEN 40
100 LPRINT MID$(N$,I+1)+" "+LEFT$(N$,I-1)
10000 END

Clone Bozo T
Bozo T Clone
```

ASC and CHR$

The ASC function returns the ASCII (American Standard Code for Information Interchange) code equivalent in decimal of the first character of its string argument, which cannot be null. See Appendix B for a complete listing of all ASCII codes.

Instruction	Output
`10 PRINT ASC("ABC")`	65
`20 PRINT ASC("1979")`	49
`30 PRINT ASC(" ")`	32
`40 PRINT ASC("1")`	49

The CHR$ function is the reverse of the ASC function. Its argument is a value that is taken to be the decimal equivalent of an ASCII code, and the character it represents is returned.

```
          Instruction                Output
          -----------                ------

50 PRINT CHR$(77)                    M
60 PRINT CHR$(65)                    A
70 FOR I=40 TO 63
80 PRINT CHR$(I);                    ()*+,-./0123456789:;<=>?
90 NEXT I
```

The TRS-80 has an unusual extension to the character set. As is the case with all 8-bit micros, the 128-character ASCII code uses only half of the possible bit patterns, and so Radio Shack uses the values 128 to 255 for a variety of reasons. Half of these, from 192 to 255 inclusive, can be printed using the CHR$ function, and the result is a variable-length TAB argument.

Examples—Various Ways to Tab

```
 Tabbing without TAB          with TAB
 ------- ------- ---          ---- ---

10 PRINT CHR$(192+5);"X"      10 PRINT TAB(5);"X"
50 PRINT CHR$(255);"Y"        50 PRINT TAB(63);"Y"
100 V=192: PRINT CHR$(V);"Z"  100 PRINT TAB(0);"Z"
```

VAL and STR$

The VAL and STR$ are two companion functions used for string-to-numeric and numeric-to-string conversion. The VAL function uses a string as its argument. It returns the value that is represented in the string. If the string is mixed, and starts with numeric characters, the value of the leading number is returned. If the string starts with non-numeric characters the value returned is 0.

```
          Instruction                Output
          -----------                ------

130 LPRINT VAL("2E3")                2000
140 LPRINT VAL("12.34")              12.34
150 LPRINT VAL("8 O'Clock")*100      800
160 LPRINT VAL("B29")                0
170 X$="-8.765": LPRINT VAL(X$)      -8.765
```

The STR$ function performs the opposite of the VAL function. It converts a value to a string. Its argument is a constant, a numeric variable, or a numeric expression. Note that 9-digit accuracy is maintained.

```
      Instruction                                          Output
      -----------                                          ------

160 LPRINT STR$(5)                                       5
170 LPRINT STR$(123456789)                               123456789
180 LPRINT VAL(STR$(123457689))                          123456789
190 LPRINT STR$(VAL("123457689"))                        123456789
200 LPRINT RIGHT$(STR$(5),1)+RIGHT$(STR$(7),1)           57
210 LPRINT RIGHT$(STR$(5),1)+LEFT$(STR$(7),1)            5
220 LPRINT RIGHT$(STR$(5),1)+STR$(7)                     5 7
```

Note that STR$(5) is two bytes long, representing the sign and the digit.

```
10 'FILENAME: "C2P6"
20 'FUNCTION: VERY LONG ADDITION
30 ' AUTHOR : JPG                    DATE: 12/79
40 '
50 CLEAR 1000'   clear enough string space
60 '     input both long integers as strings, A$ and B$
70 PRINT "TYPE THE FIRST NEMBER, UP TO 60 DIGITS LONG."
80 PRINT "TYPE 'STOP' TO EXIT."
90 INPUT A$: L1 = LEN(A$): C=0
100 IF A$ = "STOP" THEN STOP
110 PRINT "TYPE THE SECOND NUMBER, UP TO 60 DIGITS LONG."
120 INPUT B$: L2 = LEN(B$)
130 C$="": C=0'  set answer string to null, carry to 0
140 '     pad both strings at left with blanks
150 A$=STRING$(60-L1," ")+A$
160 B$=STRING$(60-L2," ")+B$
170 '     add one digit at a time; keep track of carry
180 FOR I=60 TO 1 STEP -1
190     S=VAL(MID$(A$,I,1)) + VAL(MID$(B$,I,1)) + C
200     IF S>9 THEN C=1: S=S-10  ELSE C=0
210     C$=RIGHT$(STR$(S),1) + C$
220 NEXT I
230 ' now set rid of leading zeros in answer
240 '    first find position of first non-zero character
250 FOR I = 1 TO LEN(C$): IF MID$(C$,I,1)="0" THEN NEXT I
260 '    then concatenate blanks where there were zeros
270 C$=STRING$(I-1," ") + RIGHT$(C$,61-I)
280 ' print both the input and the answer
290 LPRINT " "+A$: LPRINT "+" + B$
300 LPRINT STRING$(61,"-"): LPRINT " "+C$
310 LPRINT: LPRINT: GOTO 20
10000 END
```

```
                    7645685435807335437453311285003
+                           3468443432226092121 44399
-----------------------------------------------------------
                    7645685782651678660062523429402


     98765498765443321254398709877654323276548765432545766565
+    11111111111111111111111111111111111111111111111111111111
-----------------------------------------------------------
    109876609876554432365509820988765434387659876543656877676


                  1234567876543212345678765432123456787654321
+                 8765432123456787654321234567876543212345678
-----------------------------------------------------------
                  9999999999999999999999999999999999999999999


     77777777777777777777777777777777777777777777777777777777
+    77777777777777777777777777777777777777777777777777777777
-----------------------------------------------------------
    155555555555555555555555555555555555555555555555555555554


     99999999999999999999999999999999999999999999999999999999
+                                                           1
-----------------------------------------------------------
    100000000000000000000000000000000000000000000000000000000
```

User-defined Functions

Some programs use a few simple routines so often that they are best written as single-line functions. This is possible in most versions of extended BASIC. The format for defining single-line functions is:

DEF FNX(V1, V2, . . .) = expression
or
DEF FNX$(V1, V2, . . .) = string expression

The X or X$ is any legal numeric or string variable that will identify whether the function returns a numeric or string result. The V1, V2, and others if necessary are *dummy arguments* to the function definition. That is, their name is immaterial, except for type and position. They represent the type and position of the actual arguments as the function is invoked within the program. The expression shows the way the dummy arguments interrelate to produce the single answer.

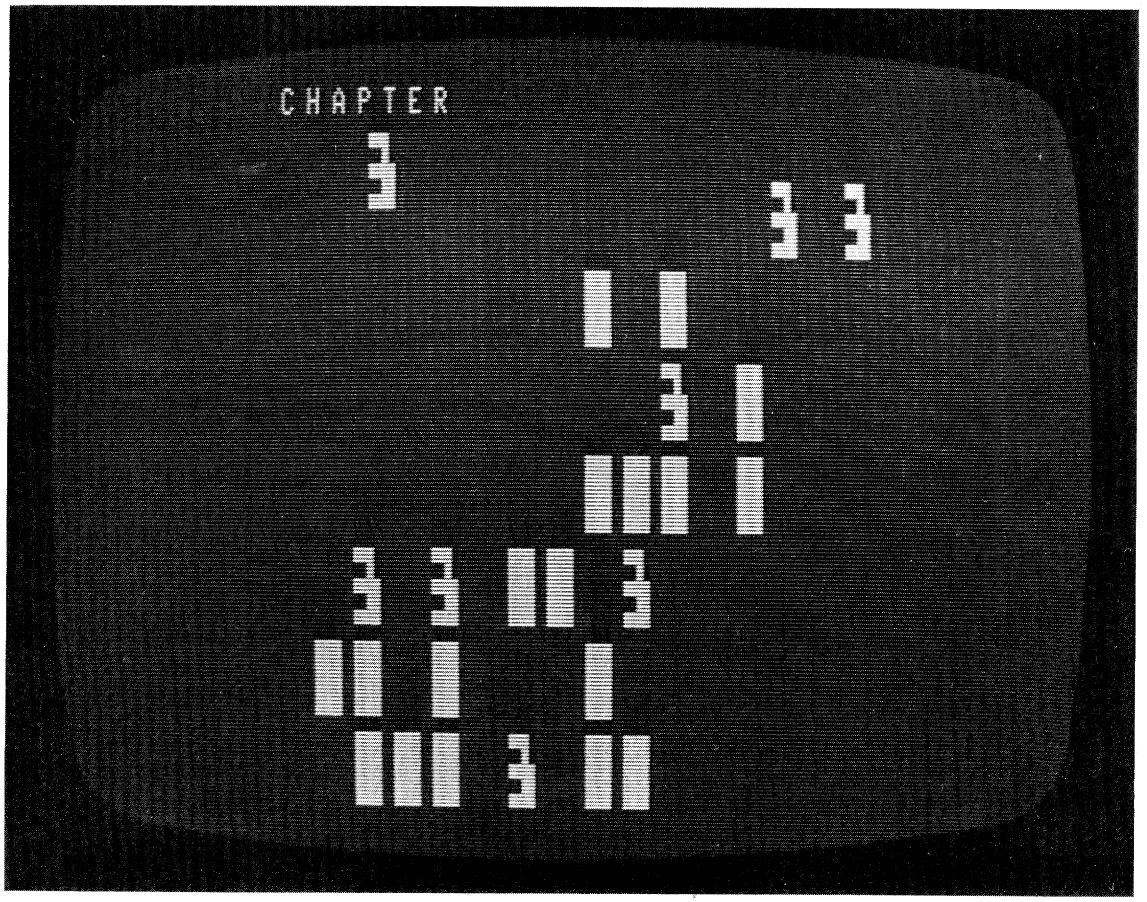
There are two distinctly different occurrences of user-defined functions in any program. One occurrence is its *definition*, and this must precede all other occurrences, which are called its *invocations*. Your study of these examples will help you to appreciate this feature of BASIC.

```
          Instruction                                    Output
          -----------                                    ------

5 T1=5: R7=8: B2=75
10 DEF FNA(X,Y,Z)=X+Y+Z
20 LPRINT FNA(T1,R7,B2)                                   88
30 DEF FND(A,B,C)=B*B-4*A*C
40 LPRINT FND(T1,B2,R7)                                   5465
50 DEF FNR(A,B,C)=(-B+SQR(FND(A,B,C))/(A+A))
60 LPRINT FNR(T1,B2,R7)                                  -67.6074
70 DEF FNX$(N$)=LEFT$(N$,2)+RIGHT$(N$,2)
80 LPRINT FNX$("FINS SPLASH")                            FISH
90 DEF FNL#(A#,B#)=A#/B#
100 LPRINT FNL#(R7,B2)                                    .1066666666666667
110 LPRINT FNA(45,67,89)                                  201
120 LPRINT FND(3243,234,123)                             -1.5408E+06
130 LPRINT FNR(34,567,89)                                -558.82
140 LPRINT FNL#(1,7)                                      .1428571428571429
```

Notice that the variables used when the function was invoked are not the same as those used in the definition, but that they are used in a one-to-one substitution.

Double precision is explained in Chapter 4, but this example is used here to show that user-defined functions can return integer or double precision answers, unlike the library functions, which return only single precision answers.

In this chapter, you have learned that strings can be manipulated in a variety of ways to ease the burden of character processing. Also, you have seen some unusual features of the TRS-80 Level II BASIC that further extend its flexibility in programming. The next chapter discusses some extensions of BASIC that the TRS-80 uses to communicate to the user through video screen and line printer.



Input and Output

Programmers are often quick to point out that programming a computer to process information is only one phase of the work. Often the input of data into the computer for future processing or the output of the processed information is at least as troublesome. As college teachers we have found that what is obvious to the professional programmer is not at all obvious to the beginning programmer: Good output is by definition highly readable and well organized, and requires a great deal of prior planning to produce.

In this chapter we will discuss the PRINT USING, a statement that greatly simplifies the task of making output readable. We will also discuss the PEEK and POKE instructions that read or alter memory directly, and the INP and OUT instructions that control the interface ports of the microcomputer.

Cued INPUT

Most extended BASICs have a feature that allows a message to be printed along with the usual question mark prompt upon execution of an INPUT statement. The programmer simply places the message in quotes after the word INPUT, then a semicolon and the list of variables.

Examples:

Instruction	Output
`10 INPUT "NAME";N$`	`NAME?`
`20 INPUT "WEIGHT";W`	`WEIGHT?`
`30 INPUT "SEX";S$`	`SEX?`
`40 INPUT "VALUE";A`	`VALUE?`

LPRINT

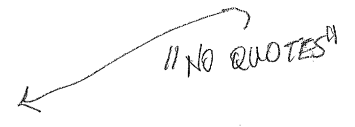
One of the signs that BASIC has matured as a computer language is its ability to use an attached printer as an output device. The LPRINT command acts exactly like all versions of the PRINT command, except the output is sent to the printer. Most of the examples and programs in this book have used and will use the LPRINT as well as the PRINT.

PRINT USING

If any one feature has enhanced the reputation of BASIC as a language in the professional community, it is the PRINT USING statement. This feature allows a great deal of flexibility in the formatting of output, and for this reason is used extensively in the printing of reports and in increasing the readability of screen output.

Some examples should clarify its use. Suppose you want to produce a chart of the values of the sine, cosine, and tangent for angles between 0 and 45 degrees in increments of 5 degrees.

```
10 'FILENAME: "C3P1"
20 'FUNCTION: CHART FOR VARIOUS FUNCTIONS
30 ' AUTHOR : JPG                    DATE:   3/80
40 '   print column headings
50 LPRINT "DEGREES", "SINE", "COSINE", "TANGENT"
60 FOR I = 0 TO 45 STEP 5
70 '    convert radians to degrees
80   A = .0174533 * I
90   LPRINT I, SIN(A), COS(A), TAN(A)
100 NEXT I
10000 END
```



```
DEGREES         SINE            COSINE          TANGENT
 0               0               1               0
 5               .0871558        .996195         .0874887
 10              .173648         .984808         .176327
 15              .258819         .965926         .267949
 20              .34202          .939693         .36397
 25              .422618         .906308         .466308
 30              .5              .866025         .577351
 35              .573577         .819152         .700208
 40              .642788         .766044         .8391
 45              .707107         .707107         1
```

Wouldn't it be nice if the same chart could contain the square root and cube root of these values? Unfortunately, this would print six values causing overflow of the four 16-column zones that make up the TRS-80 screen. The first four values would appear on one line, then the last two on the second line, as shown here.

```
DEGREES         SINE            COSINE          TANGENT
SQ ROOT         CUBE ROOT
 0               0               1               0
 0               0
 5               .0871558        .996195         .0874887
 .295409         .443557
 10              .173648         .984808         .176327
 .417772         .558847
 15              .258819         .965926         .267949
 .511663         .63972
 20              .34202          .939693         .36397
 .590818         .704103
 25              .422618         .906308         .466308
 .660555         .758471
 30              .5              .866025         .577351
 .723601         .805996
 35              .573577         .819152         .700208
 .781579         .848494
 40              .642788         .766044         .8391
 .835543         .887114
 45              .707107         .707107         1
 .886227         .922635
```

A clever programmer could use the TAB function and a rounding function to produce some much better looking output. Consider this alteration of the program.

```
10 'FILENAME: "C3P2"
20 'FUNCTION: PRINT NEAT TABLE WITH NOT-SO-NEAT PROGRAM
30 ' AUTHOR : JPG                    DATE:   4/80
40 '    print column headings
50 LPRINT "DEG"; TAB(10); "SINE"; TAB(20); "COSINE";
60 LPRINT TAB(30); "TANGENT"; TAB(40); "SQ ROOT";
70 LPRINT TAB(50); "CUBE ROOT"
80 FOR I=0 TO 45 STEP 5
90  A=.0174533*I '   convert radians to degrees
100 ' convert all values to 3-place numbers
110 X=SIN(A): GOSUB 190: S=X
120 X=COS(A): GOSUB 190: C=X
130 X=TAN(A): GOSUB 190: T=X
140 X=SQR(I): GOSUB 190: U=X
150 X=I^(1/3): GOSUB 190: V=X
160 ' note the minimal punctuation
170 LPRINT I TAB(10)S TAB(20)C TAB(30)T TAB(40)U TAB(50)V
180 NEXT I: GOTO 10000
190 X=INT(1000*X+.0005)/1000
200 RETURN
10000 END
```

DEG	SINE	COSINE	TANGENT	SQ ROOT	CUBE ROOT
0	0	1	0	0	0
5	.087	.996	.087	2.236	1.709
10	.173	.984	.176	3.162	2.154
15	.258	.965	.267	3.872	2.466
20	.342	.939	.363	4.472	2.714
25	.422	.906	.466	5	2.924
30	.5	.866	.577	5.477	3.107
35	.573	.819	.7	5.916	3.271
40	.642	.766	.839	6.324	3.419
45	.707	.707	1	6.708	3.556

The real problem with this kind of programming is not that it doesn't do the job. Rather, the job it does is not obvious to the reader of the program. One doesn't "see" the layout of the output line by studying the program. Also, a small change in layout format would be difficult to implement.

The PRINT USING statement allows the programmer to define an *image* of the output line as a string variable, and then *print* the variables *using* that image. There two ways this can be done:

(1) The PRINT USING statement can contain the image (without any variables).

(2) The PRINT USING statement can contain a string variable that defines the image.

The image is a string that acts as a mask for the output line.

There are five possible contents to an image statement.

(1) Spaces, used to spread out the values of the variables.
(2) Literals, used to place headings or messages.
(3) Digit specifiers (#), used to mask digit positions.
(4) String specifiers (% and !), used to mask characters in a string.
(5) Special characters (,.$*+-), used to designate punctuation, fill characters, signs, or exponents in scientific notation.

See Table 3.1 for a summary of these image specifiers and their effects.

The PRINT USING statement has the following form:

PRINT USING *string*; *values*

where *string* is the image, either as a string constant or variable, and *values* is the list of variables or constants to be printed.

The following examples show the use of spaces, literals, and digit specifiers.

```
Suppose A=25, B=368, and C=71904

          Instruction                          Output
          -----------                          ------

30 PRINT USING "#####";A                         25
40 PRINT USING "#####";B                        368
50 PRINT USING "#####";C                      71904
60 PRINT USING "#####  #####";A,B                25    368
70 PRINT USING "####  ####  ####";A,B,C         25   368  %71904
75 A$="###  ###  #####"
80 PRINT USING A$;A,B,C                        25  368  71904
85 B$="A=##  B=####"
90 PRINT USING B$;A,B                         A=25  B= 368
95 C$="VALUES:"
100 PRINT USING C$+A$;A,B,C                   VALUES: 25  368  71904
```

SPECIFIER	CHARACTER	FUNCTION	EFFECT
Space	(sp)	spreads output	a space
Literal	any printable character but # or special characters	appears exactly in image	the literal itself
Special Characters	#	marks a digit position	digit or a space
	,	separates every three digits in the correct positions (475,346,257)	, or a blank
	.	marks the position of the decimal point in a numeric field	the . is always printed
	$	prints a floating dollar sign	printed just in front of the first digit in the numeric field
	*	gives check protection	is printed instead of the leading zeros or spaces
	+	prints correct sign of the followns number	positive value prints a + negative value prints a -
	-	print a - if value is negative	positive value prints a space; negative value prints a - sign
		four of them denote scientific notation using exponents	the letter E plus the sign of the exponent plus two exponent digits
	%	marks the boundaries of an alphanumeric field	allows string variables to be used in the image
	!	marks the position of a character	allows single characters

Table 3.1 Image Specifiers and Their Effects

A number of features enhance the value of the PRINT USING.

(1) It prints the value even if it is too large for its corresponding image.
(2) Formats can be changed during the execution of the program.
(3) Trailing zeros are printed.
(4) Values are printed in rounded form.
(5) If the number to be printed is too large for its image, the entire value is printed, but with a leading percent sign to signal the user that the number was too large for its image. An example of this was shown in the output from line 70 in the above examples. For more, see the following examples.

```
          Instruction                          Output
          -----------                          ------

10 PRINT USING "##.##";450                     %450.00
20 PRINT USING "##.##";830.375                 %830.38
```

Punctuation, such as decimal points, commas separating thousands and millions, plus and minus signs, dollar signs, and fill characters, can be inserted in an output field with very little trouble.

Suppose E=2.718282, G=-65.432, H=7492835, and P=3.141593

```
          Instruction                          Output
          -----------                          ------

10 PRINT USING "$.####";E                      2.7183
20 PRINT USING "$,###,###.##";H                7,492,840.00
25 A$="##.##"
30 PRINT USING A$;G                           %-65.43
40 PRINT USING A$;P                             3.14
45 B$="##.##-"; C$="##.##+"
50 PRINT USING B$;G                            65.43-
60 PRINT USING C$;G                            65.43-
70 PRINT USING B$;P                             3.14
80 PRINT USING C$;P                             3.14+
```

The examples above show that when a plus sign is placed at the end of a digit specifier field, it forces the printing of a sign at that position: + for positive numbers and - for negative numbers. When a minus sign is placed at the end of a digit specifier field, it forces the printing of a space for positive and a - for negative numbers.

Now let's look at that table-printing program again, only this time it will have a PRINT USING statement.

```
10 'FILENAME: "C3P3"
20 'FUNCTION: PRINT A TABLE WITH PRINT USING STATEMENTS
30 ' AUTHOR : JPG                    DATE:   4/80
40 '  print column headings
50 LPRINT "    DEG      SIN      COS      TAN     SQ RT    CU RT"
60 ' define the image for the table
70 A$ = "    ###     ##.###  ##.###  ##.###  ##.###  ##.###"
80 FOR I = 0 TO 45 STEP 5
90  A=.0174533#I   '  convert radians to degrees
100 '   note the minimal punctuation
110 LPRINT USING A$; I,SIN(A),COS(A),TAN(A),SQR(I),I^(1/3)
120 NEXT I
10000 END
```

DEG	SIN	COS	TAN	SQ RT	CU RT
0	0.000	1.000	0.000	0.000	0.000
5	0.087	0.996	0.087	2.236	1.710
10	0.174	0.985	0.176	3.162	2.154
15	0.259	0.966	0.268	3.873	2.466
20	0.342	0.940	0.364	4.472	2.714
25	0.423	0.906	0.466	5.000	2.924
30	0.500	0.866	0.577	5.477	3.107
35	0.574	0.819	0.700	5.916	3.271
40	0.643	0.766	0.839	6.325	3.420
45	0.707	0.707	1.000	6.708	3.557

The output of program C3P3 shows the advantage of the PRINT USING in printing trailing zeros to fill the image. The computer prints the sine of 30° as 0.500, and not just 0.5.

You can build image strings during execution of the program that depend upon certain features of the variable values to be printed. For example, suppose your program is to print an amount with varying size embedded within text without any extra blanks. Study the following program to see how this is done.

```
10 'FILENAME: "C3P4"
20 'FUNCTION: ANOTHER EXAMPLE OF HOW IMAGES CAN BE USED
30 ' AUTHOR : JPG                    DATE:   4/80
40 CLEAR 300
50 INPUT "AMOUNT TO BE EMBEDDED (DOLLARS AND CENTS)";X
60 IF X=0 THEN 10000
70 F$="": A$=STR$(X): L=LEN(A$)
80 FOR I=1 TO L-3: F$=F$+"#":NEXT I
90 F$="$"+F$+".##"
100 LPRINT "OUR ACCOUNTS SHOW YOU TO BE IN ARREARS"
110 LPRINT USING "BY THE AMOUNT OF "+F$+" DOLLARS"; X
120 LPRINT " FOR THE MONTH OF AUGUST."
130 GOTO 50
10000 END
```

```
OUR ACCOUNTS SHOW YOU TO BE IN ARREARS
BY THE AMOUNT OF $ 4338.24 DOLLARS
 FOR THE MONTH OF AUGUST.
OUR ACCOUNTS SHOW YOU TO BE IN ARREARS
BY THE AMOUNT OF $ 1.23 DOLLARS
 FOR THE MONTH OF AUGUST.
```

When very large or very small values are to be printed, it is sometimes better, either for convenience or for appearance, to print these values in scientific notation. The up-arrow (↑) indicates the exponentiation operation in BASIC, but in a PRINT USING statement it also serves as an image specifier for the exponent portion of a number. Four up-arrows (↑↑↑↑) are used in an image that serves for scientific notation output.

The first ↑ masks the letter E.

The second ↑ masks the sign of the exponent.

The last two ↑↑ mask the value of the exponent.

Study the following examples to see how very large values can be printed.

```
          Instruction                          Output
          -----------                          ------

10 CLEAR 200
20 A=6.023E-8: B=.000000000012345
30 C=5.43E12: D=4920450000000000
40 S$="#.##################"
50 L$="###,###,###,###,###,###"
60 E$="#.####^^^^"
70 PRINT USING S$;A                    0.000000060230000000
75 PRINT USING S$;B                    0.000000000012345000
80 PRINT USING E$;A                    0.6023E-07
85 PRINT USING E$;B                    0.1235E-10
90 PRINT USING L$;C                          5,430,000,000,000
95 PRINT USING L$;D                      4,920,450,000,000,000
100 PRINT USING E$;C                   0.5430E+13
105 PRINT USING E$;D                   0.4920E+16
```

The $ and * characters can be used as fill characters in an image statement. This is useful in payroll programs that contain a check protection feature.

Suppose A=3.75, B=-4.86, X$="###.##", and Y$="###.##-"

```
          Instruction                          Output
          -----------                          ------

10 PRINT USING "***"+X$;A              *****3.75
20 PRINT USING "***"+Y$;A              *****3.75*
30 PRINT USING "***"+X$;B              ****-4.86
40 PRINT USING "***"+Y$;B              *****4.86-
50 PRINT USING "$$#"+X$;B                 -$4.86
60 PRINT USING "$$#"+Y$;B                  $4.86-
70 PRINT USING "$ #"+X$;B              $   -4.86
80 PRINT USING "$ #"+Y$;B              $    4.86-
90 PRINT USING "**$"+X$;B              ***-$4.86
100 PRINT USING "**$"+Y$;B             ****$4.86-
```

String specifiers in a PRINT USING image statement can control the positioning of strings. There are two string specifiers:

! is used to denote a single character.
% % is used to denote an enclosed string.

The ! specifier positions just the first character of a string.

```
                Instruction                                    Output
                -----------                                    ------

10 A$="ABC": X$="XYZ": I$="!"
20 PRINT USING I$;A$                                           A
30 PRINT USING "!";X$                                          X
40 PRINT USING "!!";A$,X$                                      AX
50 PRINT USING "!   !   !   !";A$,X$,"LIVID","ENACT"           A  X  L  E
60 PRINT USING "!! !!";"BOZO",",","THE",".";
70 PRINT " CLONE"                                              B. T. CLONE
```

The % % specifiers are always used in pairs, each pair enclosing the string that it is masking. The spaces between the % characters, plus the % characters themselves, provide the mask.

```
                Instruction                                    Output
                -----------                                    ------

90 A$="%    %": B$="%%"
100 X$="MONTANA": Y$="OHIO"
110 PRINT USING A$;X$                                          MONTA
120 PRINT USING A$;Y$                                          OHIO
130 PRINT USING B$+B$;X$,Y$                                    MOOH
140 PRINT USING B$+" "+B$;Y$,Y$                                OH OH
```

The four features that follow—INP, OUT, PEEK, and POKE—are not found just in the TRS-80 Level II BASIC. This is why they are explained in this chapter. However, the reader should be cautioned that some microcomputers handle these features in slightly different ways than the TRS-80. We have explained them as they operate on the TRS-80 because that is the target system for this book.

INP and OUT

The INP is a function that returns a single byte from the TRS-80's I/O (input/output) port specified in the argument. For example, the statement

```
30 X=INP(127)
```

returns the byte that is at port 127 and stores it in the variable X. The expansion interface is necessary to make full use of this function.

The OUT statement acts much like the INP, but in reverse. It requires two values, the first being the port number in decimal and the second being the byte that is to be transmitted to that port. For example, the statement

```
40 OUT 127,60
```

transfers a 60 to port 127.

PEEK and POKE

The PEEK function and POKE statement are very much like the INP function and OUT statement respectively. The difference is that they pick up or deposit single bytes in memory, rather than at the I/O ports.

The function PEEK has a single argument which is a decimal memory address. An example statement using the PEEK function is

```
50 A=PEEK(14520)
```

This statement causes the variable A to take on the value of the byte stored at the decimal address 14520.
The statement

```
60 PRINT PEEK(16650)
```

prints the byte at memory address 16650.
The statement

```
70 X=PEEK(15360+I)
```

places in X the value at location 15360 displaced by an increment I.

The POKE statement has two arguments, an address and a value. For example, the statement

```
80 POKE 15650,65
```

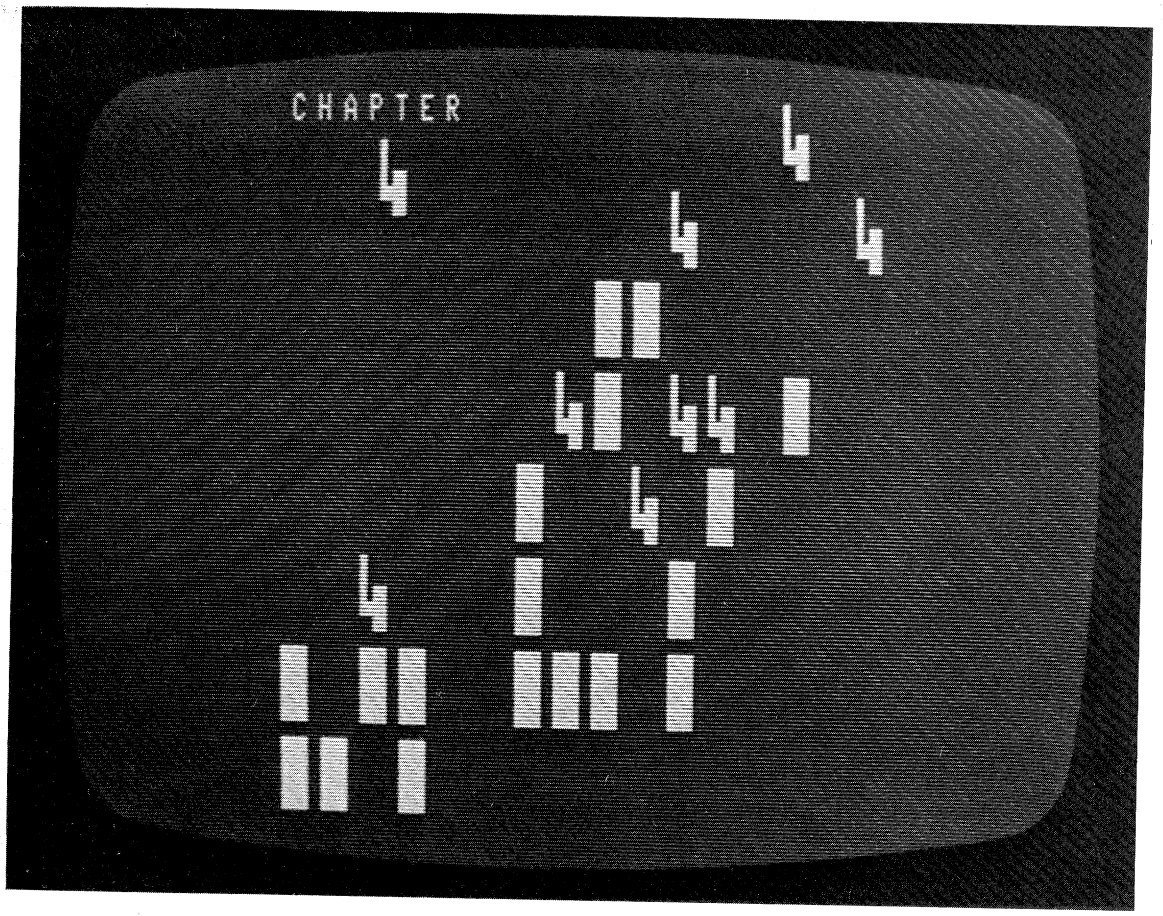
places the character "A" in a location of memory which happens to be in the portion of memory that is displayed on the video screen (see Appendix C for a complete memory map for the TRS-80). The memory positions with addresses 15360 to 16383 represent the 1024 specific positions on the screen, that section of memory which

serve as a buffer to the screen. The image of the screen at any time is in memory at those addresses, and a programmer can "read" the screen with a PEEK or "write" to the screen with a POKE into that area.

This program accesses (and does a quick read/write check of) the memory addresses 17129 to 20479, the 3300-byte area of memory that a 4K Level II system has reserved for user programs.

```
10 'FILENAME: "C3P5"
20 'FUNCTION: MEMORY TESTER
30 ' AUTHOR : JPG                    DATE:   4/80
40 FOR I = 17129 TO 20479
50 X = PEEK(I):  Y = X
60 IF INT(I/100)*100=I THEN PRINT I;
70 POKE I,Y:  Y = PEEK(I)
80 IF X <> Y THEN PRINT "SOMETHING WRONG AT";;I
90 NEXT I
10000 END
```

This chapter concludes the coverage of the extensions of BASIC that are commonly found on most microcomputers. The following chapters discuss in detail the extensions of BASIC that are found in the TRS-80's version of Microsoft BASIC.



Variables

Level II BASIC on the TRS-80 is rich with features that makes it perform like other popular high-level languages. This chapter will discuss how variables can be used in various ways to make them more appropriate for their application.

Long Variable Names

Most BASICs hold to the rule of letter or letter-and-digit as the only permissible variable names. This stringent requirement reduces the total possible number of variable names to 286. (26 for A to Z, 26 for A0 to Z0, . . . , 26 for A9 to Z9). A serious flaw with this scheme is that it greatly reduces the meaning that can be attached to a particular variable. For example, the high-level language COBOL allows up to 30 characters per variable name, so that names like NET-AFTER-TAXES, NAME-TABLE-POINTER, and DEPRECIATION are allowed. FORTRAN allows up to 6 characters, and names like SUM, SPEED, and BALNCE can be used in programs. The majority of BASIC programs, because of the variable naming limitation, are terse and harder to understand.

Many of the best modern BASICs for microcomputers have incorporated the feature of allowing long variable names. Microsoft's

BASIC, as incorporated in the TRS-80's Level II, has adopted a compromise. A variable name can be a single letter, or it must begin with a letter and be followed by either a letter or a digit, so there are exactly 962 possible distinct variable names. Level II BASIC allows longer names, but only the first two characters are used by the computer to distinguish between variables. Also, a Level II reserved word cannot be contained within a variable name. Appendix D has a full list of all reserved words in TRS-80's BASIC.

Examples:

Legal in Level II	Illegal in Level II
X	8J (digit first)
V7	STAB (contains TAB)
AB	COST (contains COS)
ABC (same as AB)	IRON (contains ON)
SUM	STIFF (contains IF)
SUPER (same as SUM)	POST (contains POS)
SHUPER	FORMIDABLE (contains FOR)

The programs in this book will use names of one or two characters only, except for a few carefully chosen names that are longer when their use clarifies the meaning of the programs significantly.

Variable Types

Variables are named according to their application by using a *type declaration character* as part of the variable name. The string variable A$ is distinct from the numeric variable A because it is *declared* as a string by the dollar sign ($), which is its type declaration character.

Integer Variables

Integers in Level II BASIC use the percent sign (%) as a type declaration character. Integers are whole numbers that vary in size between -32768 and +32767 inclusive.

Examples:

Integer variables	Typical values
X%	5
SUM%	0
NUM%	-8
I%	7982
J2%	-3000
COUNT%	-1

Single Precision Real Variables

Single precision real variables use the exclamation point (!) as a type declaration character, or they use nothing at all, since numeric variables are declared by default to be single precision real variables. These variables are accurate to seven digits, and vary in size from

about -1.7×10^{38} to about $+1.7 \times 10^{38}$.

Examples:

Single precision real variables	Typical values
A	372.871
V7	-6.5
XX	-.09
J!	142.
B2!	4E-12
GROUP!	2.5E-34

Double Precision Real Variables

Double precision real variables use the pound sign (#) to signify the ability to represent 16 decimal digits of accuracy. Like single precision values, they vary from about -1.7×10^{38} to about $+1.7 \times 10^{38}$ in magnitude.

Examples:

Double precision real variables	Typical values
F#	32984532891.77
V8#	3.141592653589793
JA#	.3333333333333333
GR#	1.000000000000001
NUM#	2.718281828459045

The flexibility of programming that these three numeric precision types allow is bought at a price. Integer variables have a limited range; single precision real variables are accurate to just 7 digits; and double precision real variables take up more than twice as much memory as single precision variables. Each type has various characteristics, favorable and unfavorable, as table 4.1 shows.

Conversion of Constants

The conversion of a constant to internal representation can be a time-consuming process, and a good programmer should always be on guard for ways to speed up particularly slow sections of code. This is where some basic knowledge about the particular characteristics for the TRS-80 and its Level II BASIC can be very helpful.

The program below does some simple additions in a loop that can be varied in the number of times it is executed. Both constants and variables appear in the calculations within the loop. The timings for the program's execution are shown in table 4.2.

Type	Integer	Single Precision	Double Precision
Declaration character	%	! (default)	#
Range	-32767 to +32767	1.701411-E38 to 1.701411+E38	1.70141183460469221-E38 to 1.70141183460469221+E38
Size, bytes	2	4	8
Precision (stored digits)	all in range	7	17
Precision (printed digits)	all in range	6	16
Run time memory allocation	5	7	11
Conversion time to binary, msec	.05-.1	5-10	500-1000

Table 4.1 Variable Types and Their Characteristics

```
10 'FILENAME:  "C4P1"
20 'FUNCTION:  DEMONSTRATE DECLARED VARIABLE, CONSTANT TIMING
30 'AUTHOR   :  JPG                DATE:  12/79
40 '
50 X%=0: X=0: X#=0:  'declare three types, initialize to zero
60 INPUT "LOOP SIZE= (0=STOP)";N: IF N=0 THEN STOP
70 M%=1%: S=1.1: D#=.123456789012345:  'increments for loop
80 '   let user select the type to be added.
90 '     IC=integer constant,          IV=integer variable
100 '    SC=single precision constant, SV=single prec. var.
110 '    DC=double precision constant, DV=double prec. var.
120 INPUT "TYPE (1=IC, 2=IV, 3=SC, 4=SV, 5=DC, 6=DV)";TY
130 PRINT "TIME BEFORE=";RIGHT$(TIME$,5)
140 '    now execute a simple addition N times, according to
150 '   the user's requested type.  note that the loop
160 '   overhead of ON-GOTO and GOTO instructions is the same
170 '   for all types of conversion or addition.
```

```
180 FOR I=1 TO N
190    ON TY GOTO 210,220,230,240,250,260
200    GOTO 40
210            X%=X%+1:   GOTO 270
220            X%=X%+M%:  GOTO 270
230            X=X+1.1:   GOTO 270
240            X=X+S:     GOTO 270
250            X#=X#+.12345678901234:  GOTO 270
260            X#=X#+D#:  GOTO 270
270 NEXT I
280 '   Print the values and the current time.
290 PRINT X%, X, X#
300 PRINT "TIME AFTER=";RIGHT$(TIME$,5)
310 GOTO 50
10000 END
```

	Form within loop					
	Variable			Constant		
	INT	SNG	DBL	INT	SNG	DBL
5	-	-	-	-	-	4
10	-	-	-	-	-	8
20	-	-	-	-	1	15
50	1	1	1	1	2	38
100	2	2	3	2	3	-
200	3	4	5	4	5	-
500	8	8	13	10	12	-
1000	17	18	27	20	23	-
Execution time, per instruction (msec)	17	18	27	20	23	800

Table 4.2 Timings from Program C4P1, in Seconds

Notice that the time it takes to deal with double precision is negligible, so long as conversion is kept to an absolute minimum. This can be achieved by doing all arithmetic within the program on values that are stored as variables. For example, both of these program segments produce double precision answers to the same problem, but their execution times are vastly different. The first one, which executes a loop that contains some double precision conversion, executes in 80 seconds, while the second, which contains only previously defined variables, executes in 2 seconds.

```
10 PRINT TIME$
20 B#=1.00000000000001
25 '
30 FOR I=1 TO 100
40    S#=S#+.100000000000001
50 NEXT I
60 PRINT S#, TIME$
10000 END
```

```
10 PRINT TIME$
20 B#=1.00000000000001
25 C#=.100000000000001
30 FOR I=1 TO 100
40    S#=S#+C#
50 NEXT I
60 PRINT S#, TIME$
10000 END
```

Implicit Conversion

A program may have numerous statements that mix different types of variables, and a programmer who writes such mixed mode expressions and statements must be able to predict exactly what the computer will do in all circumstances.

These rules show how the TRS-80 will treat various constants that appear in the program, either as a part of an expression or as created during the actual execution of an expression. Notice that in general, the precision of an answer maintains the maximum precision of any operand.

Rule 1: Any constant with more than 7 digits, with a # type declaration character, or with a D exponent forces storage as a double precision value. In the examples that follow, the values in the PRINT statement are first converted as necessary, then operated upon and stored temporarily, and finally printed as shown.

```
          Instruction                    Output
          -----------                    ------

10 PRINT 12345678+.12345                12345678.12345
20 PRINT 12345D0+12345                  12345.12344999611
30 PRINT 123.45678                      123.45678
40 PRINT .0000000000001#                1D-13
50 PRINT -6000000000001                 -6000000000001
60 PRINT 5000000000000                  5000000000000
70 X#=50000: PRINT X#+.00005            50000.000049999995
```

Rule 2: Any non-double precision constant less than -32768 or more than +32767 or containing a decimal point forces storage as a single precision value.

```
          Instruction                    Output
          -----------                    ------

10 PRINT 12345+.12345                   12345.1
20 PRINT 123.456                        123.456
30 PRINT 123.4567                       123.457
40 PRINT -81245                         -81245
```

Note that if a value that is printed cannot be represented with either six digits for single precision, or sixteen digits for double precision, the value is printed in scientific notation. The following examples show this.

```
          Instruction                          Output
          -----------                          ------

10 PRINT .00000000000001#                      1D-13
20 PRINT .00000000000012345                    1.2345E-13
```

Rule 3: Any constant between -32768 and +32767 inclusive not containing a decimal point and not declared single precision with a ! or double precision with a # is stored as an integer value.

Memory Storage After Mixed Operations

When an operation is performed on two or more operands in a statement, the result must be predictable even though the operands are not all the same type. Microsoft's Level II BASIC is not only predictable, but it usually produces the result that a programmer would have liked to get.

(1) *Most Precise Operand Rule*
The result of a +, -, or * operation has the precision of its most precise operand.

```
Instruction                 Output      Comments
-----------                 ------      --------

10 PRINT 2*3.6              7.2         INT * SNG   -> SNG
20 PRINT 2*4                8           INT * INT   -> INT
30 PRINT 3+2.718281828      5.718281828 INT + DBL   -> DBL
40 PRINT 3.3+2.718281828    6.018281828 SNG + DBL   -> DBL
```

Note that when two different precision operands are compared, the computer actually compares temporary versions of the operands, and these temporary versions have a precision of the most precise operand.

```
          Instruction                                    Output
          -----------                                    ------

10 IF 2=2.00001 THEN LPRINT "YES"
                ELSE LPRINT "NO"                         NO
20 IF 2.5=2.5000000000001 THEN LPRINT "YES"
                          ELSE LPRINT "NO"               NO
30 IF 2=2.00000000001 THEN LPRINT "YES"
                      ELSE LPRINT "NO"                   NO
```

(2) *No Integer Division Rule*
The result of a division obeys the Most Precise Operand Rule, except that when it is indicated between two integers, both operands are converted to single precision before the division is performed.

```
Instruction                 Output          Comments
-----------                 ------          --------

10 PRINT 2/3                .666667         INT/INT  ->  SNG
20 PRINT 2%/3%              .666667         INT/INT  ->  SNG
30 PRINT 2/3%               .666667         INT/INT  ->  SNG
40 PRINT 2/3.5              .571429         INT/SNG  ->  SNG
50 PRINT 2/3.12345678       .64031620760893 INT/DBL  ->  DBL
60 PRINT 3.5/4.12345678     .848802397293467 SNG/DBL  ->  DBL
```

(3) *Integer Truncation, Otherwise Rounding Rule*
During conversion from single precision to an integer, a number is reduced to the largest integer not greater than the original number, exactly as the INT function does it. During conversion from double precision to single precision, the number is rounded up. Conversion from double precision to integer goes through conversion to single precision.

```
Instruction                          Output     Comments
-----------                          ------     --------

10 I%=7.999: PRINT I%                7          SNG -> INT (truncation)
20 S!=7.999999999: PRINT S!          8          DBL -> SNG (roundins)
30 J%=7.999999999: PRINT J%          8          DBL -> SNG -> INT
                                                (roundins,
                                                then truncation)
```

(4) *Integer Boolean Operations Only Rule*
A Boolean AND, OR, or NOT between two unlike operands forces conversion to integer first. This rule is included here only for the sake of completeness. It is unlikely that you would ever need to use this feature.

```
Instruction                          Output     Comments
-----------                          ------     --------

10 PRINT 2 AND 3.5                     2        same as 2 AND 3
20 PRINT 2 OR 4.999999999              7        same as 2 OR 5
30 PRINT 1.9999 OR 7.999999999         9        same as 1 OR 8 (7.9...
                                                rounded up first, then
                                                converted to inteser)
```

DEFINT, DEFSNG, DEFDBL, DEFSTR

Level II BASIC allows a programmer to reserve sections of the alphabet for various types of variables. This is in effect a form of implicit type declaration. Instead of having to type a # character at every occurrence of a double precision variable, the programmer can declare any variable starting with that letter of the alphabet to be double precision.

The general form of these statements is

DEF*typ letter-range*

where *typ* is:
INT for integer,
SNG for single precision real,
DBL for double precision real, or
STR for string variables.
and *letter-range* is:
a single letter from A to Z,
two or more letters separated by commas,
two letters separated by a hyphen, or
any combination of the above.
These statements are normally used at the beginning of a program.

Examples:

```
100 DEFINT I-N
```

All variables beginning with any letter I through N are integer, such as I, J2, I8, MM, NICE.

```
110 DEFDBL B, C, D
```

All variables beginning with the letters B, C, or D are double precision, such as DIG, C, COUNT, BB, B5.

```
120 DEFSNG X-Z, V, R
```

All variables beginning with the letters X through Z, V, or R are single precision. Note that single precision is the default condition of the computer, so this statement is not necessary except as a reminder to the programmer, unless it is used to override a previous declaration.

```
130 DEFSTR A, P
```

All variables beginning with the letters A or P are strings, even though they are not followed by a $.

```
140 DEFINT I, K-R, T-Z
150 DEFSNG A, C, T-Z
160 DEFDBL D,E
170 DEFSTR M-O
180 DEFINT A-Z
```

The use of the DEF statements does not preclude the use of type declaration characters. For example, if your program has the statement:

```
25 DEFDBL D
```

the variable D in the program is double precision, the variable D% is a distinctly different integer variable, D$ is a string, and D! can be used for single precision. If the program uses D#, it will be considered the same variable as D when it is used.

Hexadecimal and Octal Constants

Radio Shack's Disk BASIC allows the programmer to define constants in hexadecimal (base 16) or octal (base 8) as well as decimal. This feature is convenient for dealing with memory addresses or for manipulating specific bytes in memory.

The prefix &H signifies a hex constant and the prefix &O or & (the O is optional) signifies an octal constant. These constants represent signed integers in memory, so they are two bytes (16 bits) long. See table 4.3 for examples of this feature.

Constant (octal)	Constant (hex)	stored as hex bytes	decimal equivalent
&0	&H0	0000	0
&1	&H1	0001	1
&17	&HF	000F	15
&20	&H10	0010	16
&255	&HAD	00AD	173
&377	&HFF	00FF	255
&77777	&H7FFF	7FFF	32767
&100000	&H8000	8000	-32768
&100001	&H8001	8001	-32767
&100002	&H8002	8002	-32766
&177776	&HFFFE	FFFE	-2
&177777	&HFFFF	FFFF	-1

Table 4.3 Octal and Hexadecimal Conversions

The following example illustrates the use of hex conversion as a possible aid in program writing. If you become more familiar with the hexadecimal representation of certain memory addresses, such as hex 3C00 being the first address of the screen buffer area, you may take advantage of this feature as illustrated here.

Example

```
110 FOR I=(&H3C00) TO (&H3FFF)
120    POKE I, RND(64)+127
130 NEXT I
```

Result

The screen buffer is at memory addresses from 3C00 to 3FFF, so this segment places a random graphic character (see the next chapter) on every position of the screen.

VARPTR

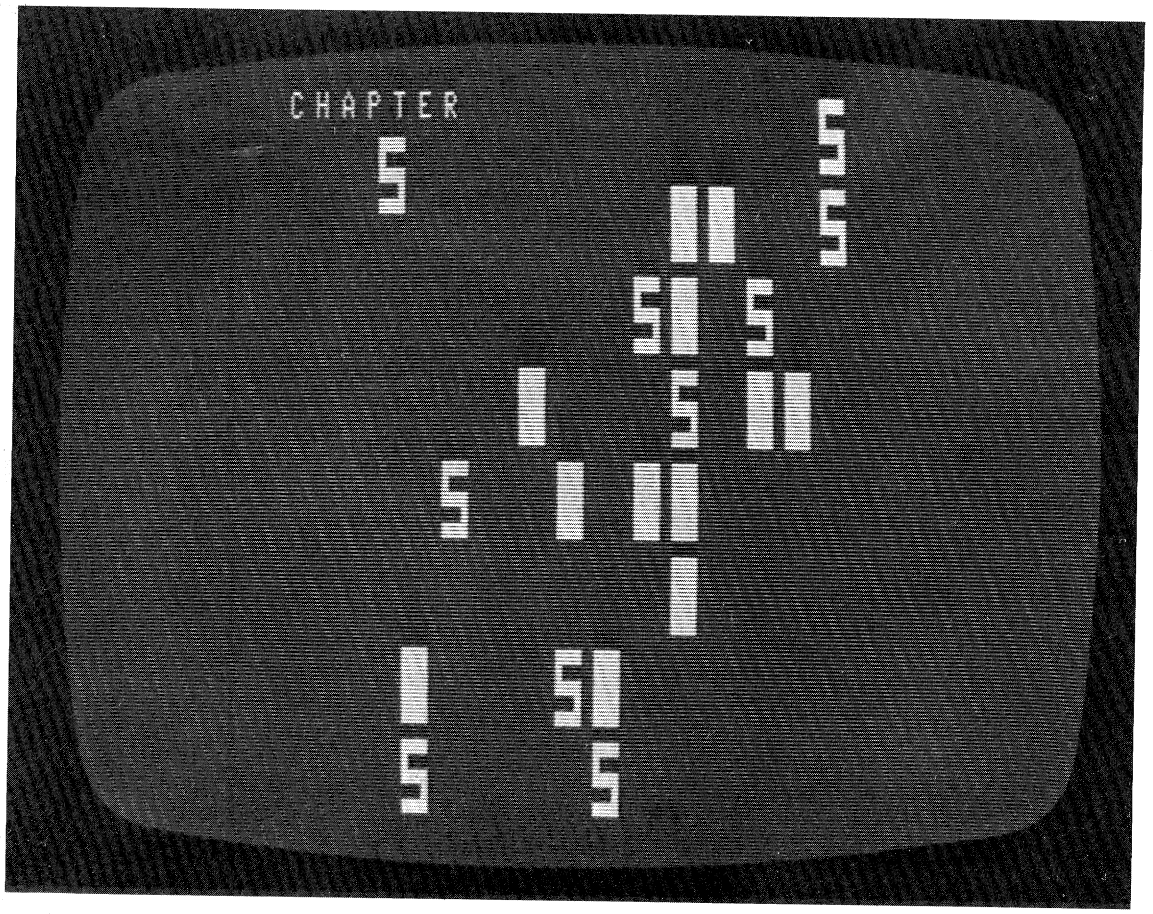
This function of Level-II BASIC returns the address of its argument. The argument is a variable name and if it has not been defined, the computer prints an error message.

When the argument's variable is numeric, whether it is integer, single precision, or double precision, the address that is returned is that of the least significant byte (LSB) of the variable. The other bytes are from 1 to 7 bytes past the returned address, with the address of the most significant byte (MSB) being the largest.

When the argument of the VARPTR function is a string variable, the value returned is the address of the length of the string and the two-byte address for the string itself is in the next two bytes.

The VARPTR function is useful for passing the addresses of variables back and forth between assembly language routines and BASIC programs, but it finds little use elsewhere.

With what you know now about the screen's buffer area and the computer's memory representation of variables, you are ready to explore graphics, which many programmers consider to be the most exciting challenge in a microcomputer. The following chapter discusses graphics in detail, and includes a wealth of examples for you to try.



Graphics

There are three distinctly different ways to produce pictures on the screen of a TRS-80. The first is as old as computers; lines are printed one at a time on the screen, just as if it were paper. The other two methods are more accommodating to the programmer, allowing considerable flexibility and some rather stunning graphical displays.

This chapter will discuss all three methods and show by example what can be done with a little care and imagination.

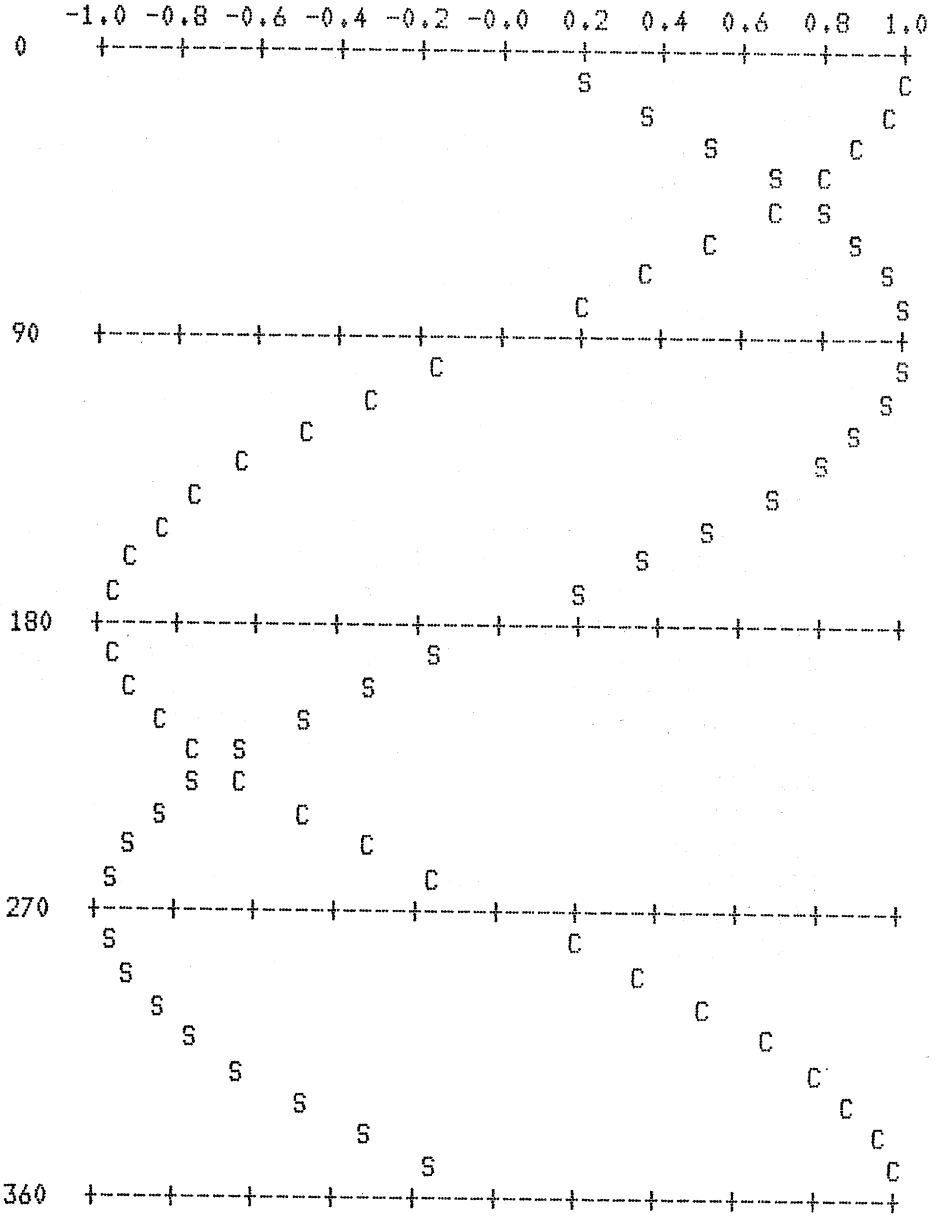
Line Printer Graphics

Line printer graphics is called what it is because anything that can be done on the screen can also be done on a line printer. It can be used on almost any computer and with most computer languages. In many ways it is the most powerful method of graphing. We propose to show you by example some of the various pictures that can be produced with line printer graphics.

Graphing with Tabs

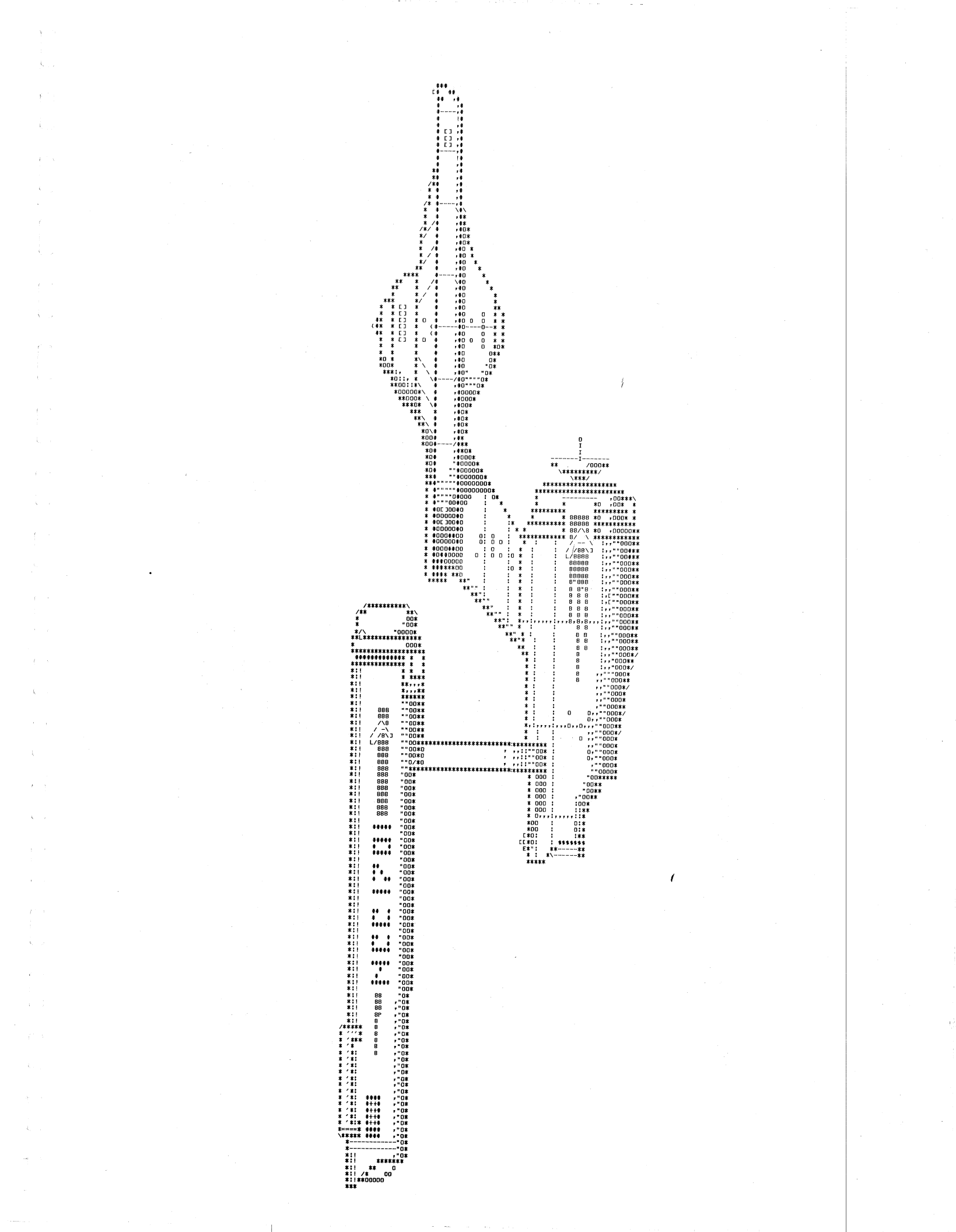
Problem: Represent the sine and cosine functions graphically for all values between 0 and 360 degrees.

```
10 'FILENAME: "C5P1"
20 'FUNCTION: GRAPH SINE AND COSINE FUNCTIONS
30 'AUTHOR  : JPG                DATE: 12/79
40 '
50 ' K=conversion constant for degrees to radians
60 K=3.14159/180
70 ' draw axis of values -1 to 1
80 LPRINT "    ";
90 FOR I=-1 TO 1 STEP .2
100 LPRINT USING "##.# ";I;
110 NEXT I: LPRINT
120 'print S for sine, C for cosine at appropriate position.
130 'each point is 10 degrees apart.
140 FOR D=0 TO 360 STEP 10
150 'convert degrees to radians
160 R=D*K
170 'determine positions of the S and C characters.
180 S=25*SIN(R)+32: C=25*COS(R)+32
190 '   if angle is a quadrant divider then print its value
200 '      and print the grid line
210 IF INT((D+90)/90)=(D+90)/90 THEN LPRINT D;: GOSUB 260: GOTO 240
220 'if cosine is less, print C then S, otherwise reverse.
230 IF C<S THEN LPRINT TAB(C)"C" TAB(S)"S"
            ELSE LPRINT TAB(S)"S" TAB(C)"C"
240 NEXT D
250 STOP
260 LPRINT TAB(5);" ";
270 FOR I=-1 TO .8 STEP .2: LPRINT "+----";: NEXT I
280 LPRINT "+": RETURN
10000 END
```


-1.0 -0.8 -0.6 -0.4 -0.2 -0.0 0.2 0.4 0.6 0.8 1.0
0
90
180
270
360

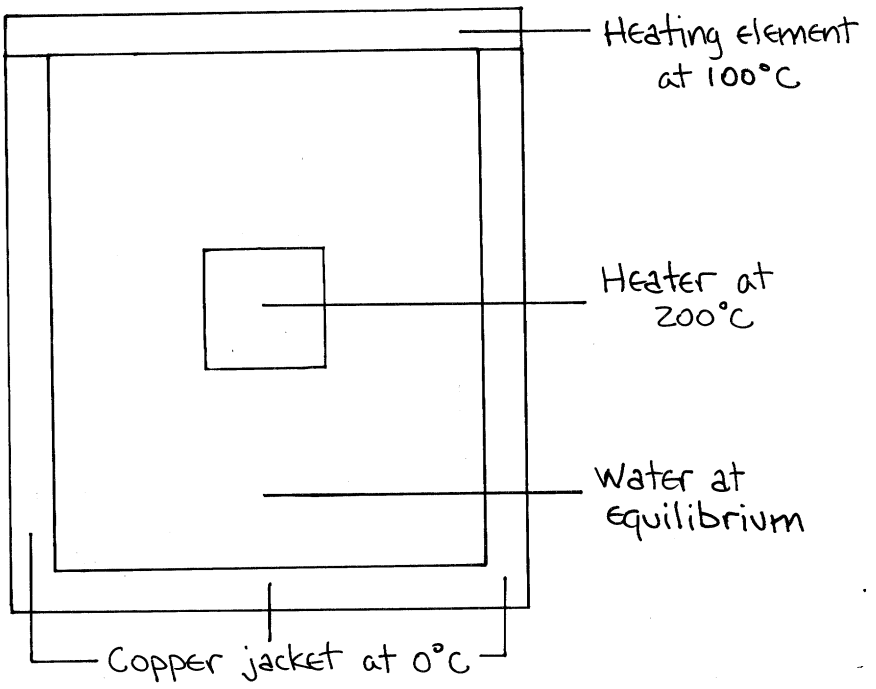
Another way that tabs can be used in a design is to symbolize them in a list of data statements. The program below was written by Steve Grillo to draw the Starship Enterprise. We have included just a portion of its three pages of data statements. Note that the program analyzes the data that it reads and executes its LPRINT statements according to the contents of the DATA statements.

```
10 'FILENAME: "C5P2"
20 'FUNCTION: PRINT A PICTURE OF THE USS ENTERPRISE
30 ' AUTHOR : SPG                    DATE:   6/79
40 '
50 ' Here are a few remarks concernins the prosram:
60 '     About the data statements:
70 '              T10   = TAB (10)
80 '       ANY STRING   = Print the followins text
90 '                R   = Carriage Return
100 '               Q   = Left Bracket
110 '               W   = Right Bracket
120 '               A   = Top to bottom diasonal (opposite of "/")
130 '               M   = A comma (,)
140 '               S   = A quotation mark (")
150 CLEAR 300: DEFINT A-Z
160 READ A$: IF A$="END" THEN STOP
170 IF A$="R" THEN LPRINT CHR$(13);: GOTO 160
180 READ B$: IF B$="END" THEN STOP
190 FOR N=1 TO LEN(B$): C$=MID$(B$,1)
200   IF C$="Q" THEN B$=MID$(B$,1,N-1)+CHR$(91)+MID$(B$,N+1)
210   IF C$="W" THEN B$=MID$(B$,1,N-1)+CHR$(93)+MID$(B$,N+1)
220   IF C$="A" THEN B$=MID$(B$,1,N-1)+CHR$(92)+MID$(B$,N+1)
230   IF C$="M" THEN B$=MID$(B$,1,N-1)+CHR$(44)+MID$(B$,N+1)
240   IF C$="C" THEN B$=MID$(B$,1,N-1)+CHR$(58)+MID$(B$,N+1)
250   IF C$="S" THEN B$=MID$(B$,1,N-1)+CHR$(34)+MID$(B$,N+1)
260 NEXT N: IF LEFT$(A$,1)="S" THEN 280
270 LPRINT TAB(1+VAL(MID$(A$,2)));B$;: GOTO 290
280 LPRINT STRING$(VAL(MID$(A$,2)),"");B$;
290 READ A$: IF LEFT$(A$,1)="T" THEN 190 ELSE 170
900 '
910 '          This is only a partial data listing
920 '                    for the Enterprise
930 '
1000 DATA T22,###,R,T21,Q#  ##,R,T22,##  M#,R,T22,#    M#
1010 DATA R,T22,#----M#,R,T22,#    !#,R,T22,#    M#,R,T22
1020 DATA # QW M#,R,T22,# QW M#,R,T22,# QW M#,T22,"----M#
1030 DATA R,T22,#    !#,R,T22,#    M#,R,T21,*#    M#,R,T21
1040 DATA *#    M#,R,T20,/*#    M#,R,T20
1050 DATA T20,* #    M#,R,T19,/* #----M#,R,T19
1060 DATA *  #    A#A,R,T19,*  #    M#*,R,T19,* /#    M#*
1070 DATA R,T18,/*/ #    M#O*,R,T18,*/  #    M#O*,R,T18
1080 DATA *   #    M#O*,R,T18,*  /#    M#O *,R,T18
1090 DATA * / #    M#O *,R,T18,*/  #    M#O  *,R,T17
```



Using Memory to Hold the Picture

Problem: Display the thermal gradient at equilibrium throughout a water-carrying rectangular copper pipe held at 0°C if it is covered with a heated lid at 100°C, and contains a rectangular heater at its center heated at 200°C.



This problem is a modification of an old FORTRAN problem found in *A Guide to FORTRAN IV Programming,* Daniel D. McCracken, p. 98 (Wiley, 1965).

Solution: (1) Consider the pipe's dimensions to be 40 units wide and 30 units deep; both sides and the bottom are the cold copper, and the top is the hot heating element. (2) Reserve a 30x40 integer array X in memory, with X(0,1) to X(0,39) held at a value of 100; X(30,0) to X(30,40), X(0,0) to X(0,30), and X(0,40) to X(30,40) held at a value of 0. These are the edges of the pipe. The 6-unit wide by 4-unit deep center heating element is held at 200 degrees. (3) Proceed throughout the array wherever there is water, from X(1,1) to X(1,39), then X(2,1) to X(2,39), . . . through X(29,1) to X(29,39), modifying each point on the basis of its neighbors according to the formula:

$$X(I,J) = (X(I-1,J)+X(I,J-1)+X(I+1,J)+X(I,J+1))/4$$

This formula calculates the temperature of a point by averaging the temperatures of that point's four neighbors. (4) Repeat Step 3 for as many iterations as the user wishes. (5) Convert all numeric values of X, one 64-character line at a time, from numeric to graphic symbols using this chart of symbols to indicate various temperature

ranges.

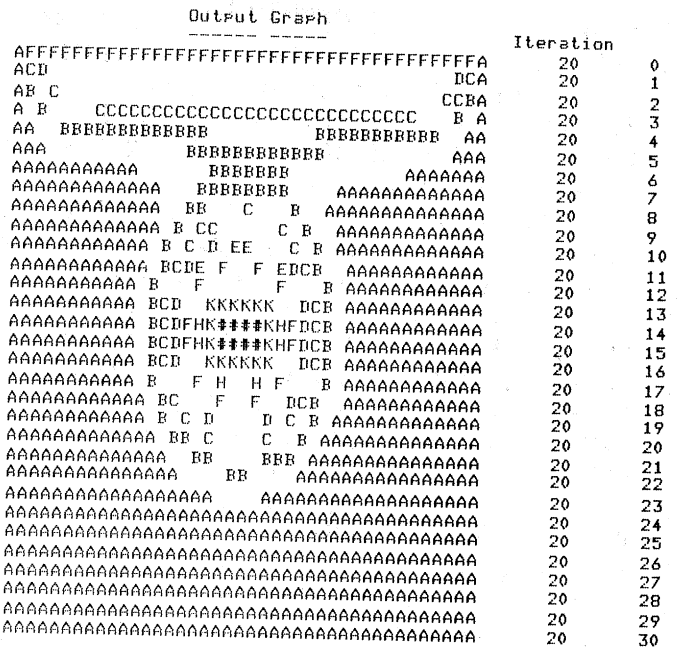
Integer value	Graphic symbol
0-9	A
20-29	B
40-49	C
60-69	D
80-89	E
100-109	F
120-129	G
140-149	H
160-169	I
180-189	J
200	K

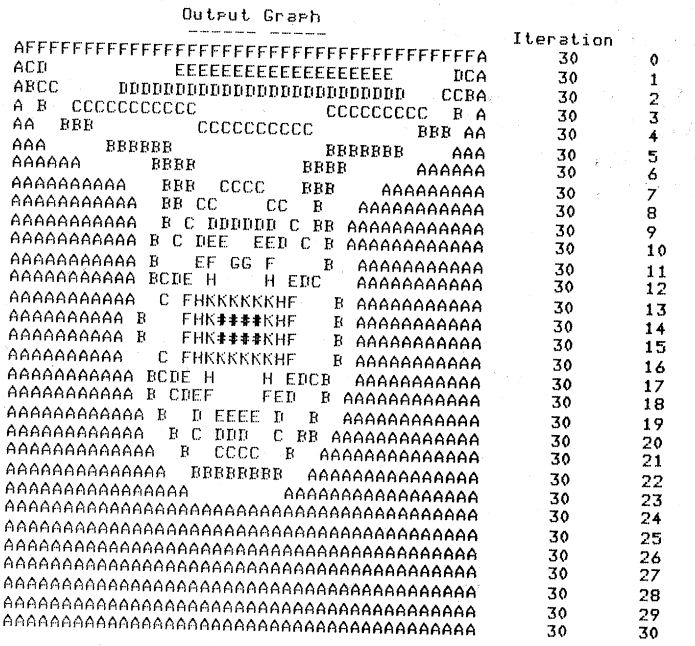
All temperatures in the unspecified intervals are symbolized with a blank. (6) Paint the picture on the screen one line at a time.

```
10 'FILENAME: "C5P3"
20 'FUNCTION: SHOW THERMAL GRADIENTS USING RELAXATION
30 ' AUTHOR : JPG                    DATE:   3/80
40 CLEAR 300: DEFINT A-Z          'all inteser math
50 DEF FNA(X) = 5 * X / 12        'FNA is proportion of pipe width
60 DEF FNB(X) = 7 * X / 12        'FNB is proportion of pipe depth
70 IT = 100                       'IT is total iterations
80 DEPTH = 30:  WIDTH = 40        'set up sraph size
90 LL = FNA(WIDTH):  LR = FNB(WIDTH)  'set four corners of pipe
100 LT = FNA(DEPTH):  LB = FNB(DEPTH)
110 INPUT "LID, EDGE, CENTER TEMPERATURES";LID,EDGE,CENTER
120 LPRINT "Temperatures:  Lid =";LID;TAB(26)"Edse =";EDGE;
130 LPRINT TAB(43)"Center =";CENTER
140 LPRINT
150 LPRINT
160 LPRINT "Dimensions:  Depth =";DEPTH;
170 LPRINT TAB(26)"Width =";WIDTH;TAB(39)"Iterations =";IT
180 LPRINT
190 DIM X(DEPTH,WIDTH), S$(21)   'S$ is symbol for temperature
200 FOR I = 0 TO 20 STEP 2:  READ S$(I):  NEXT I
210 DATA A, B, C, D, E, F, G, H, I, J, K
220 ' every other symbol is blank
230 FOR I = 1 TO 19 STEP 2:  S$(I) = " ":  NEXT I
240 ' set starting temperature from user
250 INPUT "INITIAL TEMPERATURE (0 TO 200, NEG. = RANDOM)";T
260 IF T>=0 THEN 310
270 ' set temperature to random - begin
280 FOR I=1 TO DEPTH-1: FOR J=1 TO WIDTH-1: X(I,J)=RND(200)
290 NEXT J,I: GOTO 330
```

```
300 ' set temperature to user's suggestion
310 FOR I=1 TO DEPTH-1: FOR J=1 TO WIDTH-1: X(I,J)=T: NEXT J,I
320 ' set lid and edge temperatures
330 FOR I=0 TO WIDTH: X(0,I)=LID: X(DEPTH,I)=EDGE: NEXT I
340 FOR I=0 TO DEPTH: X(I,0)=EDGE: X(I,WIDTH)=EDGE: NEXT I
350 ' set pipe temperatures
360 FOR I=LT TO LB: FOR J=LL TO LR: X(I,J)=CENTER: NEXT J,I
370 CLS  '          now comes the tedious portion
380 FOR N=1 TO IT:  I = 0:  GOSUB 460
390 '  calculations.   The busy loop
400 FOR I = 1 TO DEPTH - 1:  FOR J = 1 TO WIDTH - 1
410 IF I > LT AND I < LB AND J > LL AND J < LR THEN NEXT J
420 X(I,J)=(X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1))* .25
430 NEXT J: GOSUB 460: NEXT I:  GOSUB 460:  NEXT N
440 GOTO 440  '  freeze screen
450 '  subroutine to calculate and print one line
460 A$="": FOR K=0 TO WIDTH
470 '  check if center portion - don't calculate it
480 IF K > LL+1 AND K < LR-1 AND I > LT+1 AND I < LR-1
      THEN A$=A$+"*"   ELSE A$=A$+S$(X(I,K)/10)
490 NEXT K
500 '
510 IF DEPTH >15 PRINT A$, N; I ELSE PRINT @ I*64, A$, N; I;
520 IF INT(N/10)*10<>N OR I<>0 THEN 560
530 LPRINT: LPRINT: LPRINT
540 LPRINT TAB(15)"Output Graph"
550 LPRINT TAB(15)"------ -----"TAB(43)"Iteration"
560 IF INT(N/10)*10=N THEN LPRINT A$;TAB(46);N;TAB(54);I
570 RETURN
10000 END
```

```
Temperatures:  Lid = 100  Edge = 0          Center = 200

Dimensions:  Depth = 30   Width = 40   Iterations = 30

                Output Graph
                ------ -----                    Iteration
AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA        10      0
ACDD                                   DDCA      10      1
AB CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC BA        10      2
A  BBB                               BBB A       10      3
AAA                                    AA        10      4
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10      5
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10      6
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10      7
AAAAAAAAAAAAAAAA    B   AAAAAAAAAAAAAAAA         10      8
AAAAAAAAAAAAAAA B       B  AAAAAAAAAAAAAAA       10      9
AAAAAAAAAAAAAA BC DDDD C   AAAAAAAAAAAAAA       10     10
AAAAAAAAAAAA BCDE FF EDC  AAAAAAAAAAAAAA        10     11
AAAAAAAAAAA BCD  HHHH  DCB AAAAAAAAAAAAA        10     12
AAAAAAAAAAA B E KKKKKKGE B AAAAAAAAAAAAA        10     13
AAAAAAAAAA    HK#####K E   AAAAAAAAAAAAA        10     14
AAAAAAAAAA     K#####K E   AAAAAAAAAAAAA        10     15
AAAAAAAAAA B EGKKKKKKGE   AAAAAAAAAAAAA        10     16
AAAAAAAAAA BCD GHHHHG DCB AAAAAAAAAAAAA        10     17
AAAAAAAAAAA BCDE    EDC   AAAAAAAAAAAAA        10     18
AAAAAAAAAAA  BC  DDD C B AAAAAAAAAAAAA        10     19
AAAAAAAAAAA  B       B  AAAAAAAAAAAAA        10     20
AAAAAAAAAAAAA   BBB   AAAAAAAAAAAAAA          10     21
AAAAAAAAAAAAAAAAA   AAAAAAAAAAAAAAAA          10     22
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10     23
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10     24
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10     25
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10     26
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10     27
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10     28
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10     29
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         10     30
```

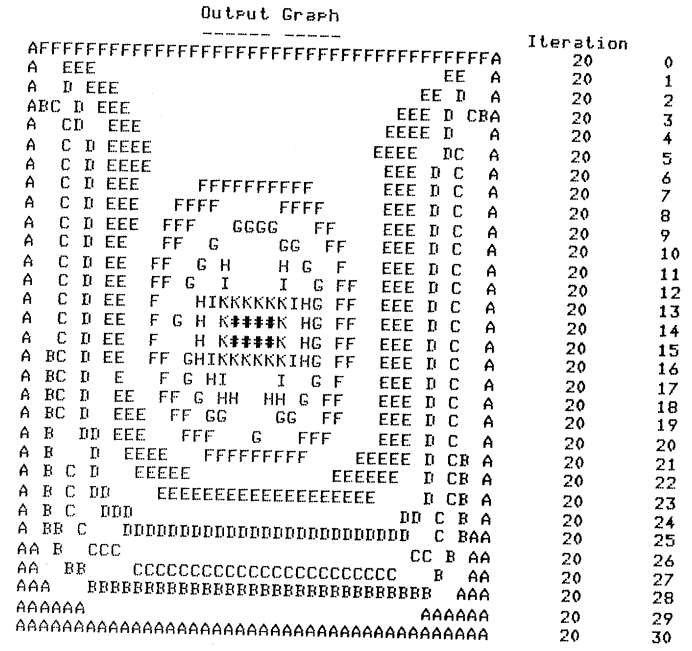

Output Graph
Iteration

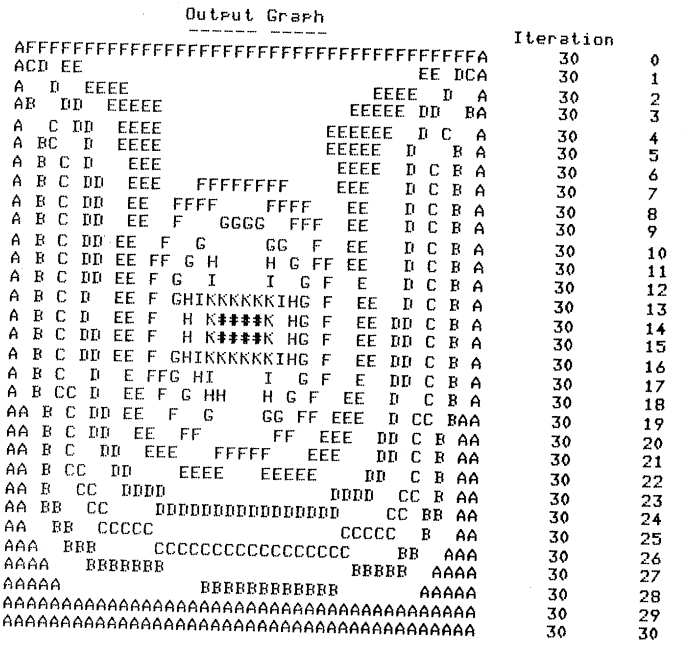

Output Graph
Iteration

The previous output shows the effect of starting the pipe's water temperature at 0 degrees. In McCracken's original work, no mention was made of initializing the water at any other temperature. We tried it at an average setting, that is, half way between heater and edge, then at random settings from 1 to 200 degrees. Although the latter case is not realistic, it seems to be the most effective in reaching equilibrium quickly. Remember that the goal of this problem is to produce a visual representation of the equilibrium condition in the pipe, so the starting temperature can be anything, even an unrealistic random value.

```
Temperatures:  Lid = 100  Edge = 0          Center = 200

Dimensions:  Depth = 30   Width = 40   Iterations = 30
```

```
                Output Graph
                ------ -----                    Iteration
AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA      10        0
A  E            FF                    E  A      10        1
A   EE        FFFF                   E D A      10        2
ABCD EE      FFFFFF                   E  BA      10        3
ABCD EE     FFFFFFF                  EEDCBA      10        4
ABC  EE     FFFFFFF                  EEDCBA      10        5
ABC  EE     FFFFFFF                  EEDCBA      10        6
ABC  EE     FFFFFFFF   FFFFFFFFF     E DCBA      10        7
ABCD EE     FFFFFFFFFFFFFFFFFFFF     E DCBA      10        8
ABCD EE    FFFF           FFFFFFF    E DCBA      10        9
ABCD E    FFFF   GGG  GGG  FFFF      E DCBA      10       10
ABCD E    FFF  GG HHHHHH G  FF       E DCBA      10       11
ABCD E  FFF  GG  I    IH G FF       EEDCBA      10       12
ABCD E  FFF  G HIKKKKKKIHG FF       EEDCBA      10       13
ABCD E  FF   G H K####KIHG FF        EDCBA      10       14
ABCD E  FFF  G H K####KIHG FF        EDCBA      10       15
ABCD E  FFF   GHIKKKKKKI G FF        EDCBA      10       16
ABCD E  FFFFF  GHIII IIH G FF        E CBA      10       17
ABC  E   FFFFF  G HHHHH G FF         E  BA      10       18
ABC  EE     FFFF  GGGGGG  FF         E  BA      10       19
ABC  EE        FFF       FF          E  BA      10       20
ABC D E          FFFFFFFFFF       FF E  BA      10       21
ABC D E            FFFFFFF        FF E  BA      10       22
ABC D E                           FF E  BA      10       23
AB  D EE                             E CBA      10       24
AB  D EEEEEEEEEEEEEEEEEEEEEEEEE  EEDCBA         10       25
A  C D  EEE        EEEE          EE     A       10       26
A B C DDDDDDDDDDDDDDDDDDDDDD  DDDDDD C  A      10       27
A  B  CCCCCCCCCCCCCCCCCCCCCCCCCCCCC  B A        10       28
AAA   BBBBBBBBBBBBBBBBBBBBBBBBBBBBB   AA        10       29
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA          10       30
```


Output Graph
Iteration


Output Graph
Iteration

Histograms or Bargraphs

The pictorial representations variously called histograms or bargraphs are easy to understand but in some ways difficult to program. If the bars vary in length horizontally, they are trivial to program, but the descriptive text is arranged contrary to custom.

Problem: Draw a histogram of monthly rainfall for one year. The rainfall in inches is stored in the array R dimensioned 12.

Solution 1: Horizontal arrangement of bars.

```
10 'FILENAME: "C5P4"
20 'FUNCTION: BARGRAPH OF RAINFALL
30 ' AUTHOR : JPG                    DATE:   3/80
40 CLEAR 500: DIM M$(12), R(12)
50 '  input section
60 FOR I=1 TO 12: READ M$(I): NEXT I
70 FOR I=1 TO 12: READ R(I): NEXT I
80 DATA Jan, Feb, Mar, Apr, May, Jun
90 DATA Jul, Aug, Sep, Oct, Nov, Dec
100 DATA 5.23, 5.79, 3.02, 2.44, 2.62, 3.99
110 DATA 4.75, 4.20, 2.01, 3.49, 4.21, 4.05
120 LPRINT "Month     Inches of rainfall"
130 '  calculate constant for keeping bar under 40 *'s
140 L=0
150 FOR I=1 TO 12: IF L<=R(I) THEN L=R(I)
160 NEXT I: K=40/L
170 ' loop to print graph
180 FOR I=1 TO 12
190 LPRINT " ";M$(I);" ";STRING$(R(I)*K,"*");
200 LPRINT TAB(55) USING "##.##";R(I)
210 S=S+R(I): NEXT I
220 LPRINT STRING$(60,"=")
230 LPRINT "Ave";: A = S/12
240 LPRINT STRING$(A*K-2,"*");
250 LPRINT TAB(55) USING "##.##";A
260 LPRINT "Twelve month total";
270 LPRINT TAB(54) USING "###.##";S
10000 END
```

```
Month      Inches of rainfall
 Jan ************************************                  5.23
 Feb ****************************************              5.79
 Mar *********************                               3.02
 Apr ****************                                      2.44
 May ******************                                    2.62
 Jun ****************************                          3.99
 Jul ***********************************                   4.75
 Aus *****************************                         4.20
 Sep ************                                          2.01
 Oct ************************                              3.49
 Nov *******************************                       4.21
 Dec ****************************                          4.05
=============================================================
Ave************************                                3.82
Twelve month total                                         45.80
```

Another bargraph could be arranged so that the bars are vertical, which is the more typical display. The following program does this, and also shows the use of a list of sentinels, or flags, in the FLAG array.

Solution 2: Vertical arrangement of bars.

```
10 'FILENAME: "C5P5"
20 'FUNCTION: REVISED RAINFALL GRAPH
30 ' AUTHOR : JPG                    DATE:   4/80
40 DIM FLAG(13), M$(13), R(13)
50 '    read in the data
60 FOR I=1 TO 13: READ M$(I): NEXT I
70 FOR I=1 TO 12: READ R(I): NEXT I
80 DATA Jan, Feb, Mar, Apr, May, Jun
90 DATA Jul, Aus, Sep, Oct, Nov, Dec, Ave
100 DATA 5.23, 5.79, 3.02, 2.44, 2.62, 3.99
110 DATA 4.75, 4.20, 2.01, 3.49, 4.21, 4.05
120 LPRINT TAB(20);"Rainfall by month"
130 LPRINT TAB(20);"-------- -- -----"
140 LPRINT
150 ' calculate constant for keeping the heisht under 20 lines
160 L=0
170 FOR I=1 TO 12: IF L<R(I) THEN L=R(I)
180 S=S+R(I)     'calculate the sum
190 NEXT I: K=L/20: R(13)=S/12
200 FOR J=20 TO 1 STEP -1
210 FOR I=1 TO 13
220 IF R(I)<J*K THEN 250
230 IF FLAG(I)=1 LPRINT TAB(I*4-4)USING "% % ";M$(I);: GOTO 250
240 IF FLAG(I)=0 LPRINT TAB(I*4-4)USING "#.# ";R(I);: FLAG(I)=1
250 NEXT I: LPRINT
260 NEXT J
10000 END
```

```
                    Rainfall by month
                    -------- -- -----

    5.8
    Feb
5.2 Feb
Jan Feb
Jan Feb                      4.8
Jan Feb                      Jul
Jan Feb                      Jul 4.2          4.2
Jan Feb                 4.0  Jul Aug          Nov 4.1 3.8
Jan Feb                 Jun  Jul Aug      3.5 Nov Dec Ave
Jan Feb                 Jun  Jul Aug      Oct Nov Dec Ave
Jan Feb 3.0             Jun  Jul Aug      Oct Nov Dec Ave
Jan Feb Mar      2.6    Jun  Jul Aug      Oct Nov Dec Ave
Jan Feb Mar 2.4  May    Jun  Jul Aug      Oct Nov Dec Ave
Jan Feb Mar Apr  May    Jun  Jul Aug      Oct Nov Dec Ave
Jan Feb Mar Apr  May    Jun  Jul Aug 2.0  Oct Nov Dec Ave
Jan Feb Mar Apr  May    Jun  Jul Aug Sep  Oct Nov Dec Ave
Jan Feb Mar Apr  May    Jun  Jul Aug Sep  Oct Nov Dec Ave
Jan Feb Mar Apr  May    Jun  Jul Aug Sep  Oct Nov Dec Ave
Jan Feb Mar Apr  May    Jun  Jul Aug Sep  Oct Nov Dec Ave
Jan Feb Mar Apr  May    Jun  Jul Aug Sep  Oct Nov Dec Ave
```

Table-Driven Pictures

Many graphic designs can be summarized in the form of a table of starting addresses and lengths. For example, the design of the numeral 1 could be stored in a two-dimensional integer array D(12,2) in which the first subscript represents the starting column and the second subscript represents the string length. The following program prints the numeral 1 shifted right approximately 30 spaces.

```
10 'FILENAME: "C5P6"
20 'FUNCTION: TABLE-DRIVEN DIGIT
30 ' AUTHOR : JPG                    DATE:   12/79
40 DEFINT A-Z: DIM D(12,2)
50 FOR I=1 TO 12: FOR J=1 TO 2: READ D(I,J): NEXT J,I
60 DATA 31, 3, 30, 4, 29, 5, 31, 3, 31, 3, 31, 3
70 DATA 31, 3, 31, 3, 31, 3, 30, 5, 30, 5, 30, 5
80 FOR I=1 TO 12
90 LPRINT TAB(D(I,1)); STRING$(D(I,2),"*")
100 NEXT I
10000 END
```

The graph of this digit is hardly representative of the many such designs that could be done, particularly since it is solid throughout; each printed line is made up of just one string. The Starship Enterprise printing program shown earlier (program C5P2) includes a number of features that lend flexibility to this technique. That program requires the printing of one or more strings per line, so it uses a signal value which has to be included in the table. This table is really a series of DATA statements, and the signal value tells the computer to return the cursor for the production of another line.

Picture Within Program

The simplest procedure for painting a picture is to contain the design itself within the program. For example, the output of program C5P7 is obvious.

This technique wastes memory, because each line that is printed is stored in its entirety. However, it can be useful as an intermediate tool to develop pictures, from which data tables can be designed.

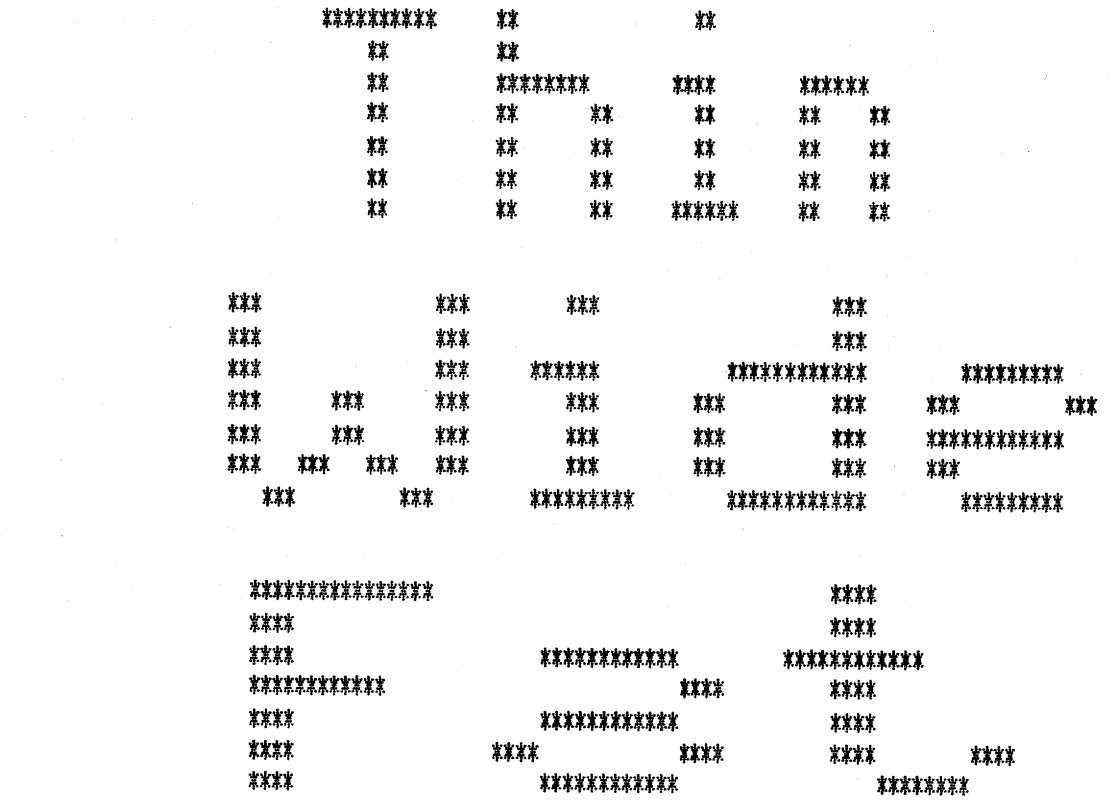
Program C5P8 is one last example of table driven graphics. It was written by Steve Grillo and his goal was to write a program that would produce banners of his choice on the screen. It does this, and more: You can select whether you want the banner displayed on the screen or printed on paper, and you can select the width of the letters. The output hints at the power of the program.

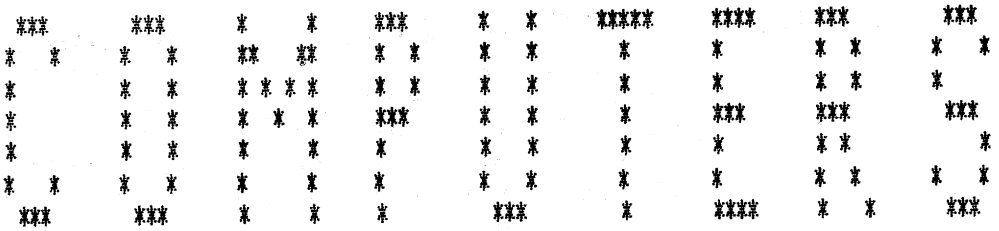
```
10 'FILENAME: "C5P7"
20 'FUNCTION: PICTURE OF SNOOPY
30 ' AUTHOR : SPG                    DATE:   2/80
40 CLS: CLEAR 1000
100 LPRINT "               XXXX"
110 LPRINT "             X    XX"
120 LPRINT "            X  ***  X                XXXXX"
130 LPRINT "           X  *****  X            XXX     XX"
140 LPRINT "         XXXX ******* XXX      XXXX          XX"
150 LPRINT "       XX   X ******  XXXXXXXXX                XX XXX"
160 LPRINT "     XX      X ****  X                           X** X"
170 LPRINT "   X         XX    XX     X                      X****X"
180 LPRINT "  X          //XXXX       X                      XXXX"
190 LPRINT " X          //   X                              XX"
200 LPRINT "X          //    X          XXXXXXXXXXXXXXXXXX"
210 LPRINT "X     XXX//     X          X"
220 LPRINT "X    X   X      X         X"
230 LPRINT "X    X    X     X        X"
240 LPRINT " X   X    X     X        X                    XX"
250 LPRINT " X    X   X     X        X                 XXX  XX"
260 LPRINT "  X    XXX       X        X              X  X X  X"
270 LPRINT "  X              X         X             XX X  XXXX"
280 LPRINT "   X             X          XXXXXXXXA     XX   XX  X"
290 LPRINT "    XX            XX              X     X    X  XX"
300 LPRINT "      XX            XXXX   XXXXXX/     X     XXXX"
310 LPRINT "        XXX             XX***          X     X"
320 LPRINT "           XXXXXXXXXXXXX *   *       X     X"
330 LPRINT "                         *---* X     X     X"
340 LPRINT "                        *-* *   XXX X     X"
350 LPRINT "                        *- *       XXX   X"
360 LPRINT "                       *- *X          XXX"
370 LPRINT "                       *- *X  X          XXX"
380 LPRINT "                      *- *X    X             XX"
390 LPRINT "                      *- *XX    X              X"
400 LPRINT "                     *  *X* X    X             X"
410 LPRINT "                     *  *X * X    X             X"
420 LPRINT "                    *  * X**  X   XXXX          X"
430 LPRINT "                    *  * X**  XX     X          X"
440 LPRINT "                   *  ** X** X     XX          X"
450 LPRINT "                   *  **  X*  XXX   X          X"
460 LPRINT "                  *  **    XX   XXXX        XXX"
470 LPRINT "                 *  * *      XXXX      X     X"
480 LPRINT "                *   * *          X     X     X"
490 LPRINT " =======*******   * *          X     X      XXXXXXXXA"
500 LPRINT "        *         * *      /XXXXX      XXXXXXXXA      )"
510 LPRINT "   =====**********  *     X                    )  A  )"
520 LPRINT "     ====*          *     X                A  A   )XXXXX"
530 LPRINT "=========**********      XXXXXXXXXXXXXXXXXXXXXX"
10000 END
```

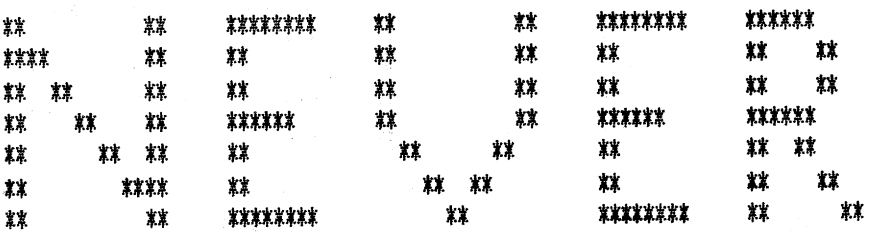
```
10 'FILENAME: "C5P8"
20 'FUNCTION: TO PRINT MESSAGES ON THE SCREEN OR PRINTER
30 ' AUTHOR : SPG                DATE:   6/79
40 '
45 '   data lines:       1000-1070
50 '   major subroutines:
60 '      2000    converts the decimal number to binary and
70 '              plots the proper pixels on the screen
80 '      3000    plots a large word (E$) on the screen at X,Y
90 '      4000    initialize matrix P(N,B) using the data
100 '     5000    stores 6 power of 2 in array O
110 '     6000    accepts a temporary string from the user and
120 '             if desired, copies the screen to the printer
130 '
500 CLEAR 200: DEFSTR A: DEFINT B-Z: DIM P(91,8)
510 GOSUB 4010: GOSUB 5000
520 CLS: INPUT "WOULD YOU LIKE A SAMPLE MESSAGE";A
530 IF A<>"YES" THEN 700
600 CLS: WIDTH=1: E$="LETTER": X=30: Y=0: GOSUB 3000: E$="By:":
    X=30: Y=15: GOSUB 3000: E$="Steve": X=0: Y=25: GOSUB 3000:
    E$="Grillo": X=35: Y=36: GOSUB 3000: PRINT@950,"<<CR>>";:
    INPUT S$
700 Y=0: H=0: CLS
710 PRINT "   Note:"
720 PRINT "      To print lower case messages, hold the"
730 PRINT "      shift key while typing."
740 PRINT
750 H=H+1: IF H>5 THEN 810
    ELSE LINE INPUT "TYPE IN THE WORD ('END' TO STOP): ";E$(H):
    IF E$(H)="END" THEN 810
760 WIDTH(H)=0
770 INPUT "WHAT WIDTH (1-4) WOULD YOU LIKE";WIDTH(H):
    IF WIDTH(H)=0 THEN WIDTH(H)=1
780 X(H)=0
790 INPUT "FIRST HORIZONTAL POSITION OF THIS WORD (>=0)";X(H)
800 PRINT: GOTO 740
810 PRINT: PRINT
850 PRINT "DO YOU WANT THE MESSAGE SENT TO THE PRINTER"
860 INPUT "            1=YES       2=NO";P1
900 CLS: FOR I=1 TO H-1: X=X(I): WIDTH=WIDTH(I): E$=E$(I):
    GOSUB 3000: Y=Y+9: NEXT I: GOSUB 6000
910 GOTO 520
920 '        lines 1000-1070  hold all of the data
930 '                for the characters
940 '
```

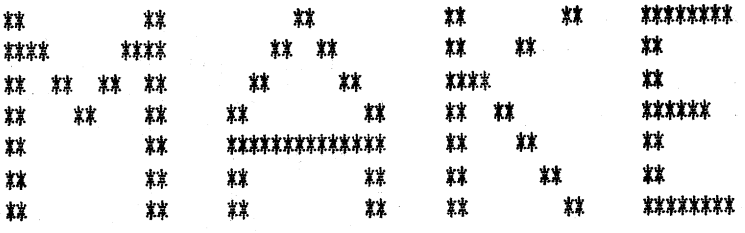
```
1000 DATA -2,-2,-2,-2,-1,95,-1,7,-2,7,-1,20,127,20,127,20,
-1,4,42,127,42,16,-1,67,32,19,8,100,2,97,-1,48,74,69,18,32,64,
-1,4,2,1,-1,28,34,65,-1,65,34,28,-1,34,20,127,20,34,-1,8,8,127,
8,8,-1,88,56,-1,8,8,8,8,-1,32,80,32,-1,64,32,16,8,4,2,-1
1010 DATA 28,34,65,65,34,28,-1,66,127,64,-1,66,33,64,17,64,9,
70,-1,33,64,1,72,5,74,49,-1,16,8,20,2,125,16,-1,39,64,5,64,5,
64,57,-1,48,72,4,74,1,72,48,-1,65,32,17,8,5,2,1,-1,54,73,-2,73,
-2,73,54,-1,6,9,64,41,16,9,6,-1,34,85,34,-1,90,33,26,-1
1020 DATA 8,-2,20,-2,34,-2,65,-1,20,20,20,20,-1,65,-2,34,-2,20,
-2,8,-1,2,1,-2,93,-2,5,2,-1,62,65,-2,89,84,14,-1,120,20,18,17,
18,20,120,-1,65,127,73,73,54,-1,62,65,65,65,34,-1,65,127,65,65,
62,-1,127,73,73,65,-1,127,9,9,1,-1,62,65,65,81,50,-1
1030 DATA 127,8,8,127,-1,65,127,65,-1,48,64,65,63,1,-1,127,4,8,
18,32,65,-1,127,64,64,64,-1,127,2,4,8,4,2,127,-1,127,2,4,8,16,
32,127,-1,62,65,65,65,62,-1,127,9,9,6,-1,62,65,65,17,33,
94,-1,127,9,25,38,64,-1
1040 DATA 38,73,73,73,50,-1,1,1,127,1,1,-1,63,64,64,64,63,-1,
15,16,32,64,32,16,15,-1,63,64,32,24,32,64,63,-1,65,34,20,8,20,
34,65,-1,1,2,4,120,4,2,1,-1,65,32,81,8,69,2,65,-1,4,6,127,6,4,
-1,16,48,127,48,16,-1,8,20,42,8,-1,8,42,20,8,-1,64,64,64,64,-1
1050 DATA 62,65,-2,89,84,14,-1,32,84,84,84,40,-1,127,68,68,68,
56,-1,56,68,68,68,72,-1,56,68,68,68,127,-1,56,84,84,84,8,-1,4,
126,5,1,2,-1,72,84,84,84,106,-1,127,4,4,4,120,-1,68,125,64,-1,
32,64,64,64,61,-1,127,8,20,34,64,-1,65,127,64
1060 DATA -1,124,4,120,4,120,-1,124,4,4,120,-1,56,68,
68,68,56,-1,124,20,20,20,8,-1,56,68,68,20,36,88,
-1,124,4,4,4,8
1070 DATA -1,8,84,84,84,32,-1,4,63,68,64,32,-1,60,64,64,124,-1,
12,16,32,64,32,16,12,-1,60,64,32,16,32,64,60,-1,68,40,16,40,68,
-1,12,80,80,80,44,-1,68,32,68,16,68,8,68,-999
2000 '       subroutine to change decimal # to binary
2010 '      and plot the discrete pixels on the screen
2020 FOR V=1 TO 7: W=O(V): IF D-W<0 THEN 2090
2030 D=D-W
2040 FOR R=1 TO WIDTH
2050      IF X+R<128 AND Y-V+7<48 THEN 2070
2060      PRINT: PRINT "    *** OUT OF BOUNDS ERROR ***": STOP
2070      SET(X+R,Y-V+7)
2080 NEXT R
2090 NEXT V: X=X+WIDTH-1: RETURN
3000 '    subroutine to plot a word (E$) on the screen at X,Y
3010 FOR G=1 TO LEN(E$): M=ASC(MID$(E$,G,1))-31: B=0
3020 B=B+1: IF P(M,B)=0 THEN 3030 ELSE D=P(M,B): X=X+1:
     GOSUB 2000: GOTO 3020
```

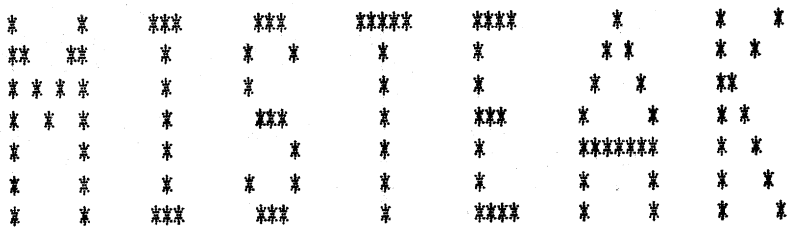
```
3030 X=X+5: NEXT G: RETURN
4000 '   subroutine to initialize matrix P(N,B) using
4005 '            the data in lines 1000-1070
4010 CLS: B=0: PRINT: PRINT "LOADING CHARACTER - ";
4020 FOR N=1 TO 91
4030      READ C$: C=VAL(C$): IF C$="-1" THEN 4050 ELSE
          IF C>0 PRINT@85,CHR$(N+31);
4040      IF C=-999 GOTO 4070 ELSE B=B+1: P(N,B)=C: GOTO 4030
4050      B=0
4060 NEXT N
4070 RETURN
5000 '     subroutine to store 6 powers of 2 in array O
5010 FOR M=1 TO 7: O(M)=2^(7-M): NEXT M: RETURN
6000 PRINT@950,"<<CR>>";:INPUT S$
6010 IF P1=2 THEN RETURN ELSE PRINT@950,"        ";
6020 '   the next line sets the IP-225 printer to 16.5 cpi
6030 LPRINT CHR$(31)
6040 '       copy the screen graphics to the printer
6050 FOR Y=0 TO 47: FOR X=0 TO 123
6060     IF POINT(X,Y) THEN LPRINT "*"; ELSE LPRINT " ";
6070 NEXT X: LPRINT " ": SET(0,Y): NEXT Y: RETURN
10000 END
```











Character Graphics

The second major graphical method is character graphics. This method is realized on the TRS-80 through the use of the PRINT @ instruction which can address any of the video screen's 1024 positions. The positions in the screen's top row have addresses 0 to 63, the second row 64 to 127, the third row 128 to 191, and so on until the last row, which has addresses 960 to 1023.

PRINT @

The PRINT @ instruction uses the screen address to position the leftmost character to be printed. Caution: The @ symbol is represented differently in memory when the shift key is depressed, so be sure to use the unshifted @, otherwise a syntax error results when the program is executed.

```
Instruction                          Output
-----------                          ------

10 PRINT @ 0,"X"                     X at the top left of the screen
20 PRINT @ 95,"ZOT"                  ZOT centered on the second line
30 PRINT @ 510,"ZOTZOT"              ZOTZOT centered on the screen
40 PRINT @ 1023,"Z";                 Z at the bottom right of the screen
```

If the PRINT@ instruction ends with no punctuation, the cursor returns to the beginning of the next line, and this may produce undesirable results. If the address is between 960 and 1023, the string prints on the last line of the screen, then the screen scrolls one line. The effect of a PRINT @960 is the same as a PRINT @896, which of course is not what was intended.

If the PRINT @ instruction ends with a semicolon (;) the result is predictable. We recommend that you always end all PRINT @ instructions with a semicolon.

STRING$

A Level II BASIC function which is useful in graphing is the STRING$ function. This function has two arguments: The first is the number of characters desired, up to 255, and the second is the character itself.

Instruction	Output
`10 PRINT STRING$(10,"*")`	`**********`
`20 PRINT STRING$(5,CHR$(65))`	`AAAAA`
`30 PRINT STRING$(8,CHR$(13))`	Cursor moves down 8 lines, positions itself at left of line. CHR$(13) is a carriage/cursor return which is the character produced by the ENTER key (/EN/).
`40 PRINT @960, STRING$(64,"Z");`	ZZZ...ZZZ (64 of them) on the bottom of the screen
`50 A=128: B$="9": PRINT STRING$(A,B$)`	Two rows of 9s

Note: Since the computer builds the string in its memory first before displaying it, your program may require additional string space to be reserved for it. Use the CLEAR instruction to allow for more string space.

The Character Set

A look at Appendix B reveals that the TRS-80 has a total of 256 possible characters, and the effect of each can be displayed by using the instruction PRINT CHR$(N) where N is a number from 0 to 255. A few of these values of the TRS-80's character code have no effect on the TRS-80, but most do, and they are certainly more numerous than would be necessary for just the alphabet, digits, and special characters that BASIC needs. The table below is a summary of the expanded table in Appendix B.

Code	Function
0-7	None
8-31	Carriage/Cursor Control
32-47	Special Characters
48-57	Digits
58-64	Special Characters
65-90	Alphabet (Upper Case)
91-95	Carriage/Cursor Control
96	Lower Case @
97-122	Alphabet (Lower Case)
123-127	Lower Case of Codes 91-95
128-191	Graphics Characters
192-255	Tabs for 0 to 63 Spaces

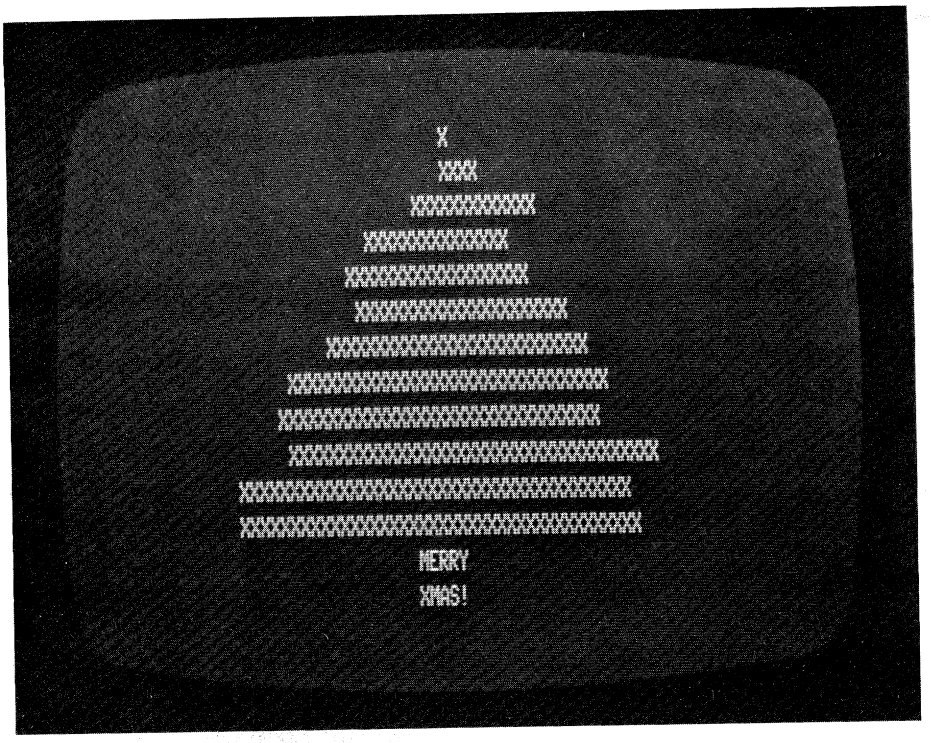
Tabulation Codes

The last two groups of codes represent half of the possible printable characters, and they deserve special mention. The last 64 codes allow tabbing without the TAB function.

```
Instruction                              Output
-----------                              ------

10 PRINT CHR$(202);"*"                   * in the 10th position
20 A$=CHR$(192+I): PRINT A$;"*"          * in the Ith position
```

These codes make possible some very simple graphics programs, such as the one below. Note that this technique is restricted to screen graphics because printers don't respond to the tabulation codes.

```
10 'FILENAME: "C5P9"
20 'FUNCTION: DRAW A VERY SIMPLE CHRISTMAS TREE
30 ' AUTHOR : JPG                    DATE:   6/79
40 CLEAR 200
50 M=32
60 FOR I=1 TO 12
70    IF I>1 THEN M=M-RND(3): R1=RND(5)-3: R2=RND(5)-3
80    P$=CHR$(192+M-R1)
90    L=65-M*2+R2
100   PRINT P$; STRING$(L,"X")
110 NEXT I
120 PRINT TAB(29)"MERRY"
130 PRINT TAB(29)"XMAS!"
140 GOTO 140    '    freeze the screen
10000 END
```



Graphics Codes

The codes from 128 to 191 are graphic characters that are made up of six small rectangles, or *pixels,* arranged in three rows and two columns for each character. Each character entirely fills one of the 1024 print positions on the screen. Each pixel is either on (bright) or off (blank), depending on the code. For example, the graphic character 134 looks like this:

where  is off, and

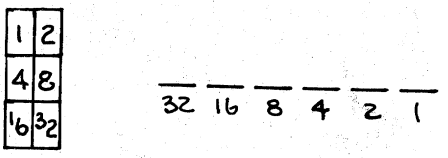
and  is on.

The result of PRINT CHR$(134) would be the graphics character shown above.

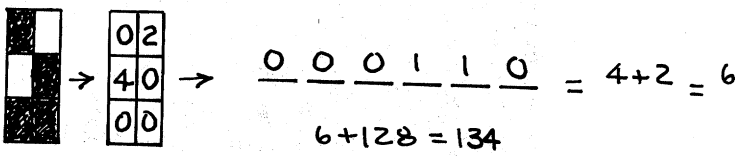
Graphic to Binary Conversion

Each character code can be thought of as a visual representation of a six-bit binary number from 000000 to 111111, corresponding to all 64 possible combinations of bits. The character's code value can be computed by translating each off pixel to a 0 and each on pixel to a 1. Then the positions of the 1s can be masked into the six-bit

binary number. The value of that number plus 128 is equal to the code value.



Thus the graphics character whose code is 134 is



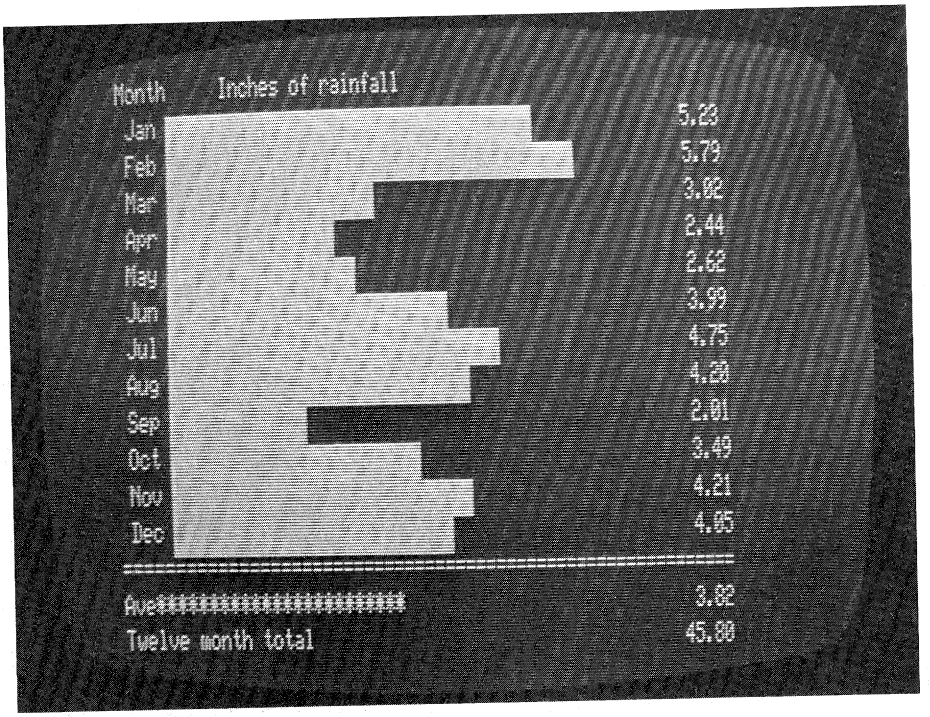
and so the statement PRINT CHR$(134) produces that graphic symbol. The graphic character whose code is 191 is the one in which all pixels are on, resulting in a large rectangle of light. This graphic symbol is one of the most useful, as shown in the following discussion.

Uses for Graphics Characters

All pictorial printer graphic applications that have been mentioned so far in this chapter are fair game for these characters. For example, the histogram application (program C5P4) requires this line change and all LPRINTs to be changed to PRINTs to modify the output dramatically.

```
190 PRINT " ";M$(I);" ";STRING$(R(I)*K,CHR$(191));
```

Program C5P4 will now produce output like:

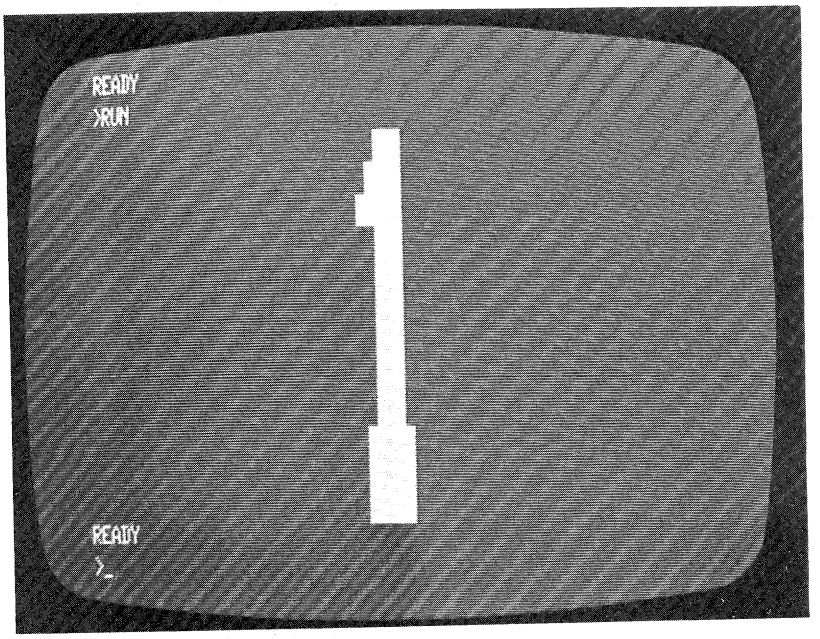


Other programs can be changed very simply, and their output reflect the use of graphics characters.

Program C5P6 with the change

```
50 PRINT TAB(D(I,1))STRING$(D(I,2),CHR$(191))
```

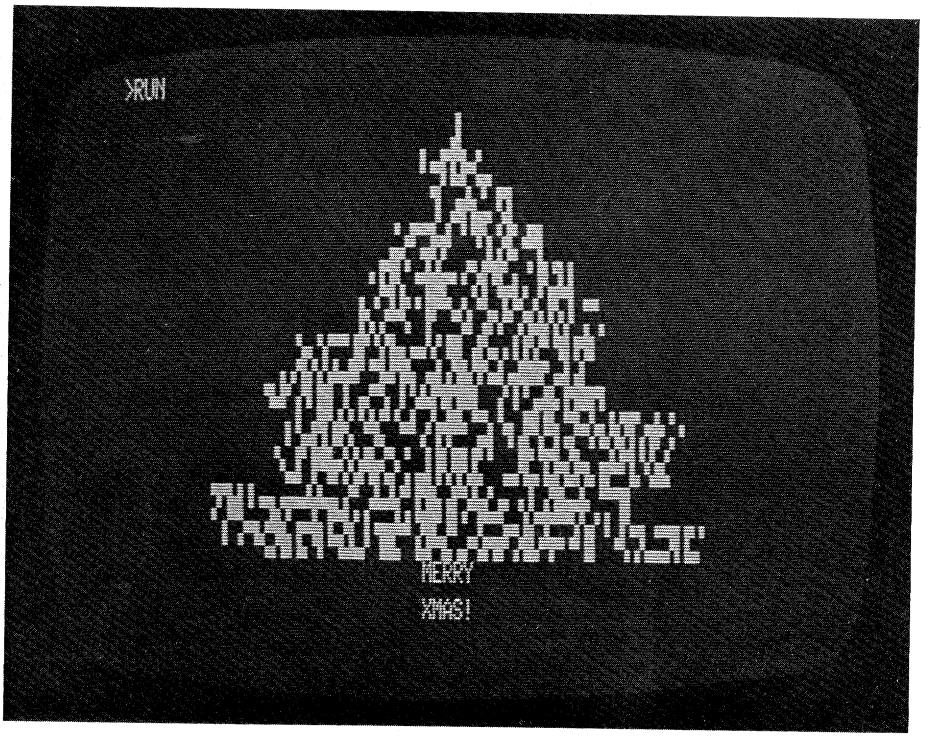
yields output like:



Program C5P9 with the modification

```
60 PRINT P$;: FOR J=I TO L: PRINT CHR$(RND(64)+128);: NEXT J: PRINT
```

now produces output like:

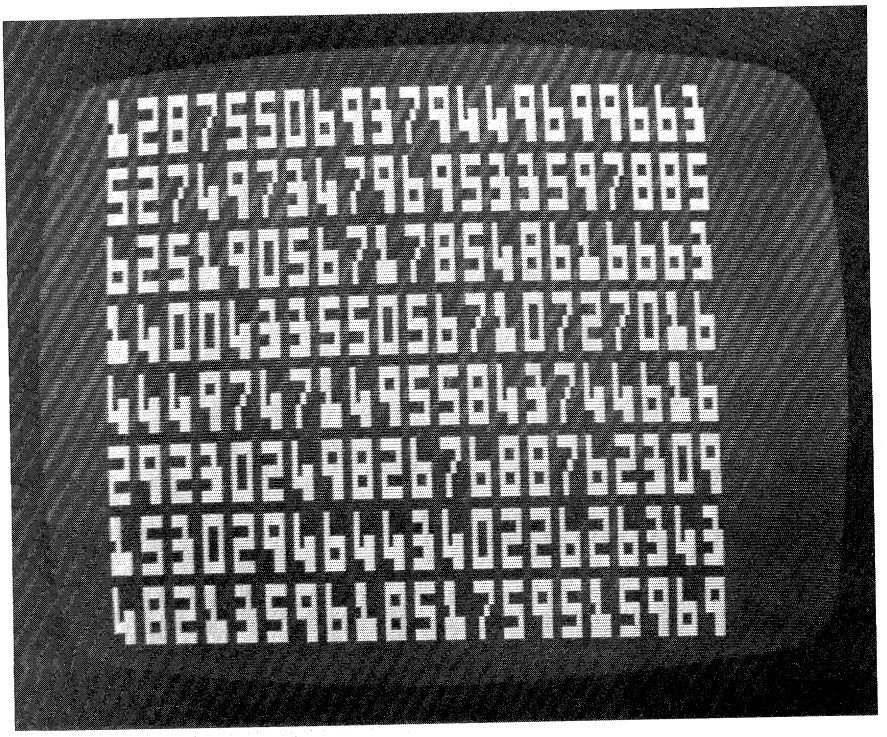


Cartooning is possible with these graphic characters. This technique requires very fast display rates, so the POKE commands should be used to transfer the characters directly to the screen. POKE is about six times faster than PRINT, so it's worth the effort.

To give the semblance of motion, it is necessary to display a sequence of slightly altered pictures in rapid order. The time it takes to plan this sequence is truly imposing.

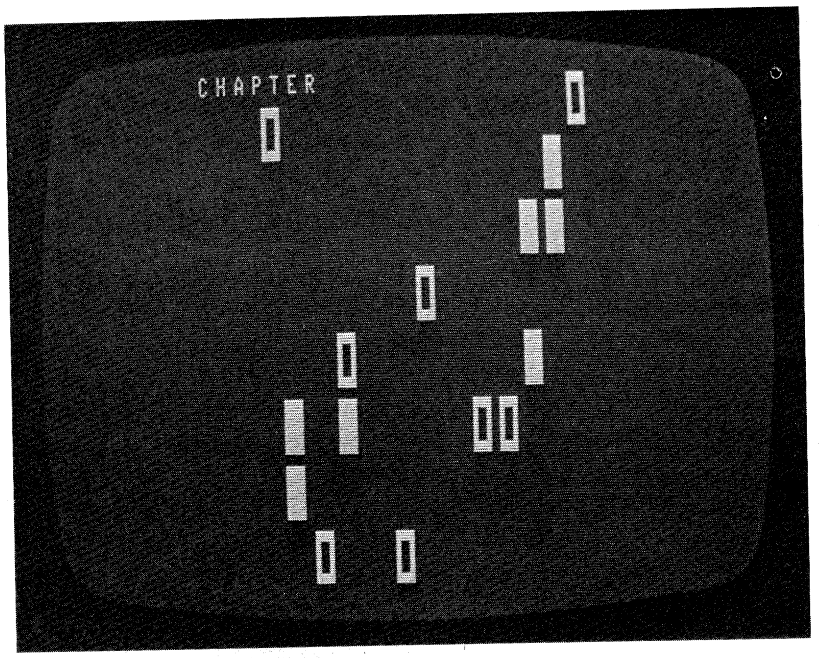
The following program uses the character graphics, and its purpose is to draw a full screen of digits on the screen very quickly. The digits are made up of four screen positions each, so they appear larger than normally printed character digits. We show you this program and its output here to prompt you into thinking of uses for this "enhanced digits" display.

```
10 'FILENAME: "C5P10"
20 'FUNCTION: PRINTS DIGITS USING CHARACTER GRAPHICS
30 ' AUTHOR : JDR                    DATE: 9/79
40 CLS: DEFINT A-Z: CLEAR 100: DIM D$(10)' initialize
50 '  compose strins to linefeed, then backspace twice
60 E$=CHR$(26)+CHR$(8)+CHR$(8)
70 '  this loop forms all of the ten disits
80 FOR I=1 TO 10: A$=""
90       FOR J=1 TO 4: READ X: A$=A$+CHR$(128+X)
100      IF J=2 THEN A$=A$+E$' add LF & backspace twice
110      NEXT J
120 D$(I)=A$: NEXT I
130 DATA 23,43,13,14,47,0,15,15,51,59,13,12
140 DATA 51,53,12,15,21,48,3,15,55,51,12,14
150 DATA 53,48,13,14,3,27,10,0,55,59,13,14,55,59,0,15
160 '     print the disits on the screen randomly
170 FOR K=0 TO 62 STEP 3
180      FOR Y=K TO 896+K STEP 128
190      C=RND(10)-1: PRINT@Y,D$(C+1);
200      NEXT Y
210 NEXT K
220 GOTO 220'   freeze the screen
10000 END
```



The output of the following program should look familiar. This program's output is used as the chapter heads, or first page illustrations.

```
10 'FILENAME: "C5P10A"
20 'FUNCTION: CHAPTER HEAD PROGRAM USING CHARACTER GRAPHICS
30 ' AUTHOR : JDR                          DATE: 10/79
40 RANDOM: DEFINT A-Z: CLEAR 100: DIM D$(10)' initialize
50 '  compose strins to linefeed, then backspace twice
60 E$=CHR$(26)+CHR$(8)+CHR$(8)
70 '   compose strins for simple desisn
80 B$=CHR$(191)+CHR$(191)+E$+CHR$(143)+CHR$(143)
90 '  this loop forms all of the ten disits
100 FOR I=1 TO 10: A$=""
110     FOR J=1 TO 4: READ X: A$=A$+CHR$(128+X)
120      IF J=2 THEN A$=A$+E$' add LF & backspace twice
130     NEXT J: D$(I)=A$
140 NEXT I
150 '      data for disits 0-9
160 DATA 23,43,13,14,47,0,15,15,51,59,13,12
170 DATA 51,53,12,15,21,48,3,15,55,51,12,14
180 DATA 53,48,13,14,3,27,10,0,55,59,13,14,55,59,0,15
190 '    main loop for printins 10 chapters is below
200 FOR K=0 TO 9: CLS
210 '    print headins on left portion of screen
220 PRINT@138,"CHAPTER";: PRINT@205,D$(K+1);
230 '   loops to print desisns on the screen
240 FOR I=1 TO 8: B=ABS(I+I-3): E=I+9-INT(I/8)
250    FOR J=B TO E
260      IF RND(3)=3 THEN PRINT@1031-128*I+3*J,B$;
270    NEXT J
280 NEXT I
290 '     loop to print disits on the screen
300 FOR J=1 TO 19: R=RND(8)
310    B=ABS(R+R-3): E=R+9-INT(R/8): S=B-1+RND(E-B+1)
320    PRINT@1031-128*R+3*S,D$(K+1);
330 NEXT J
340 '   hold the screen before settins next chapter
350 FOR I=1 TO 1000: NEXT I
360 '     set next chapter
370 NEXT K
10000 END
```



Pixel Graphics

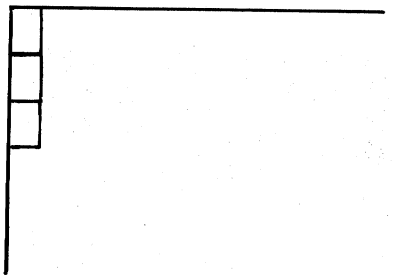
The third major technique for programming graphic displays is pixel graphics, so-called because the programmer controls the on-off state of every single pixel. Whereas the screen contains 64 columns and 16 rows of characters, whether they are script or graphic, the same screen contains six times as many pixels. There are 128 columns and 48 rows of pixels, and their screen addresses are radically different from the addresses of the graphic characters they generate. A programmer can address a single pixel with X and Y coordinates, using the column numbers 0 to 127 for X and the row numbers 0 to 47 for Y.

SET

The SET command turns on the pixel whose screen address is in X,Y coordinate form in parentheses immediately following the word SET.

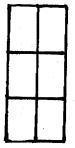
Instruction	Output
`10 SET(0,0)`	Upper left pixel is turned on
`20 SET(2,4)`	Second pixel on fourth row
`30 SET(127,47)`	Last pixel on 47th row
`40 SET(0,47)`	Bottom left pixel
`50 SET(127,0)`	Top right pixel
`60 POKE 15360, 128+21`	See text below
`70 SET(0,0): SET(0,1): SET(0,2)`	See text below

The last two statements above have the same effect, except that the POKE is much faster. Both light up the top left pixels like this:



RESET

The RESET command turns off the pixel whose screen address is in X,Y coordinate form.



```
10 PRINT @0, CHR$(191)
20 RESET(1,0)
```

```
10 'FILENAME: "C5P11"
20 'FUNCTION: SHOOT A DIAGONAL LINE ACROSS THE SCREEN
30 ' AUTHOR : JPG                    DATE:   6/79
40 CLS
50 FOR I=0 TO 119: X=I: Y=I*.4: J=I-10: K=J*.4
60    SET(X,Y)
70    IF I>9 THEN RESET(J,K)
80 NEXT I
10000 END
```

The program above "shoots" a line from the top left to the bottom right of the screen. Try it.

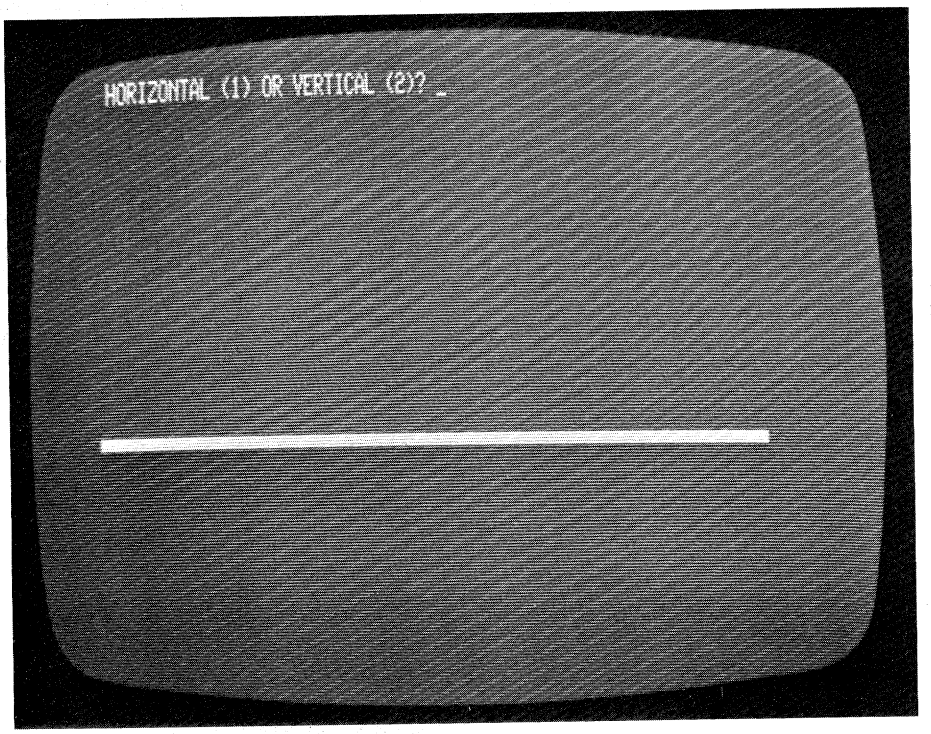
POINT

The POINT function returns a zero (false) if the pixel at the X,Y coodinates that make up its argument is off, and a -1 (true) if that pixel is on. Thus it is useful when testing if a particular spot is on or off.

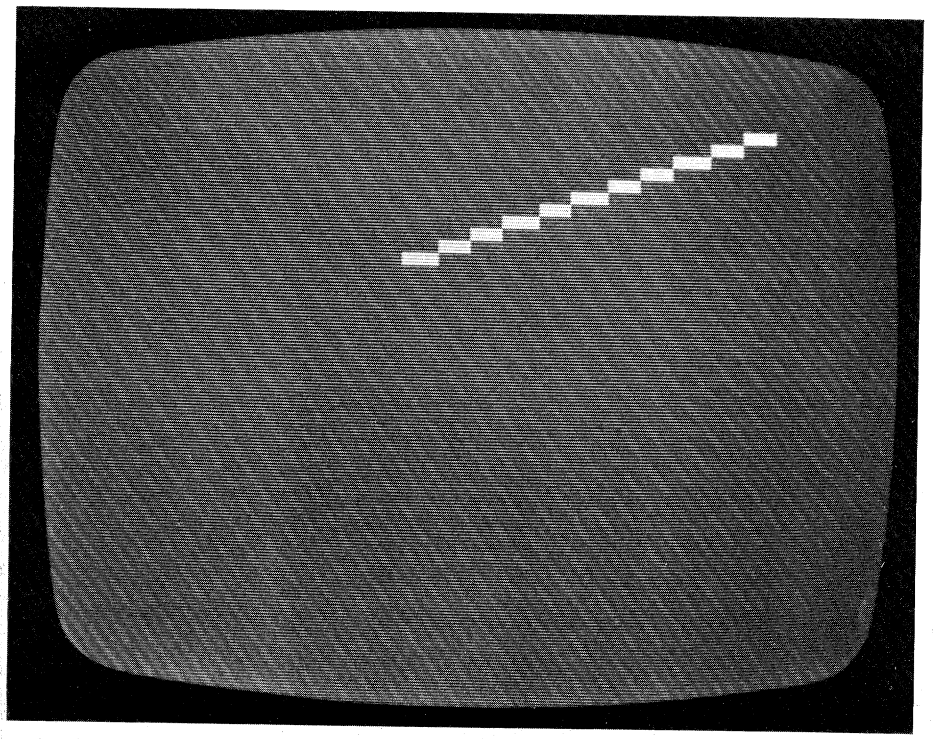
```
10 IF POINT(0,0) THEN RESET(0,0)

20 IF POINT(X,Y) THEN SET(X+1,Y+1): RESET(X,Y)
```

Pixel graphics give the programmer a true graphing capability on the screen with 128-column, 48-row resolution. This may not be the finest resolution you can find on the graphic terminals and computers, but the latter generally cost about five times as much as a TRS-80. With pixel graphics you can draw horizontal, vertical, and angled lines, and even graph geometric shapes.

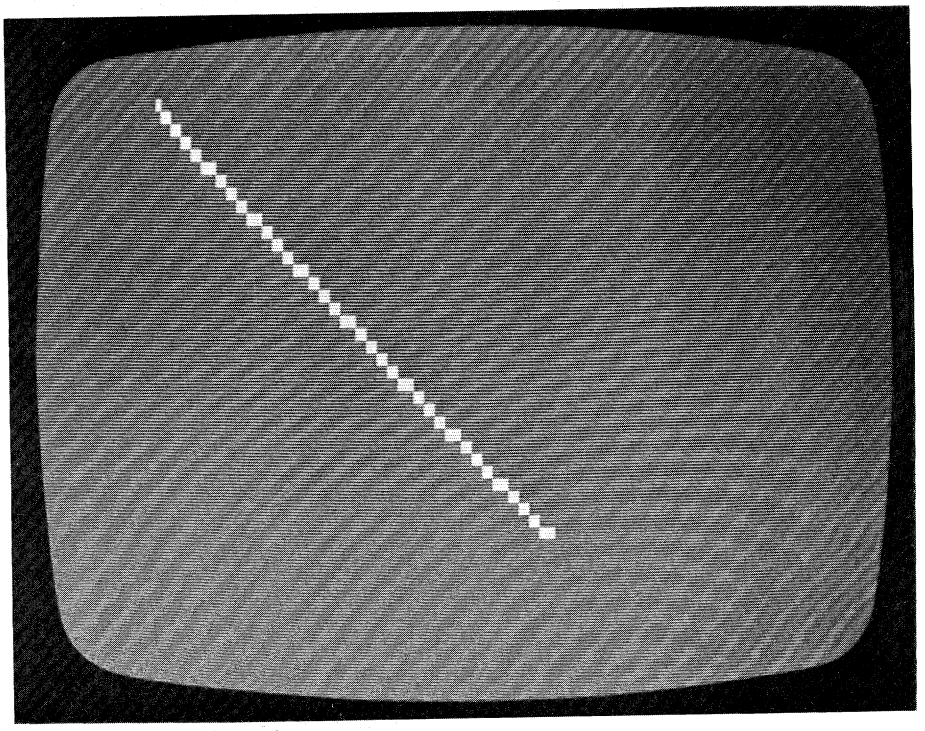
```
10 'FILENAME: "C5P12"
20 'FUNCTION: DRAW A HORIZONTAL OR VERTICAL LINE
30 ' AUTHOR : JPG                    DATE:   6/79
40 CLS
50 INPUT "HORIZONTAL (1) OR VERTICAL (2)"; A
60 INPUT "STARTING COORDINATE"; B
70 CLS: ON A GOTO 80,90: GOTO 50
80 FOR X=0 TO 127: SET(X,B): NEXT X: GOTO 50
90 FOR Y=0 TO 47: SET(B,Y): NEXT Y: GOTO 50
10000 END
```



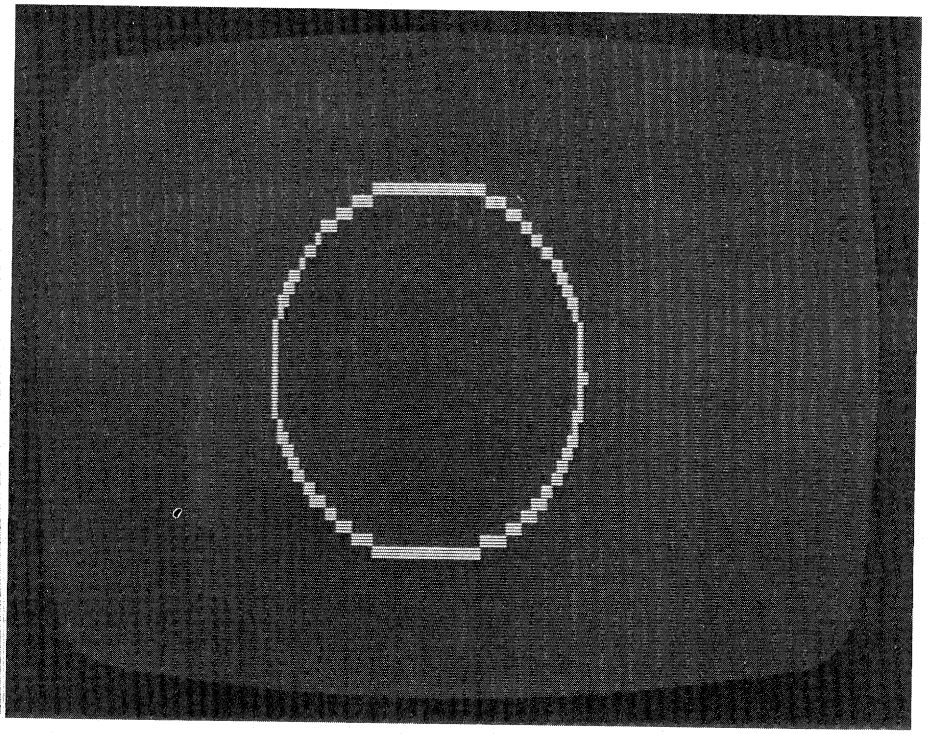
```
10 'FILENAME: "C5P13"
20 'FUNCTION: DRAW AN ANGLED LINE (POLAR METHOD)
30 ' AUTHOR : JPG                    DATE:   6/79
40 CLS
50 INPUT "X AND Y OF ORIGIN"; X,Y
60 INPUT "ANGLE IN DEGREES";TH: A=TH/57.3: S=SIN(A): C=COS(A)
70 INPUT "LENGTH";R
80 CLS
90 '   draw the line
100 FOR N=1 TO R
110   SET(X+N*C,47-(Y+N*S))
120 NEXT N
130 GOTO 130 '  freeze the screen
10000 END
```



```
10 'FILENAME: "C5P14"
20 'FUNCTION: DRAW ANY LINE BY CARTESIAN METHOD
30 ' AUTHOR : SPG                    DATE:   5/79
40 CLS: INPUT "  POINT OF ORIGIN (X,Y)";X1,Y1
50 INPUT "  POINT OF TERMINATION (X,Y)";X2,Y2
60 CLS
70 '  switch variables if needed to reverse direction
80 IF Y2<Y1 THEN X3=X2: X2=X1: X1=X3: Y3=Y2: Y2=Y1: Y1=Y3
90 '    checks for a vertical angle and acts accordingly
100 IF X1<>X2 THEN 180
110 IF Y1<Y2 THEN ST=1 ELSE ST=-1'   set step size
120 '   loop to draw vertical line
130 FOR Y=Y1 TO Y2 STEP ST
140    SET (X1,Y)
150 NEXT Y
160 GOTO 260
170 '     takes care of all other angles
180 M=(Y2-Y1)/(X2-X1)'    calculate slope of angle
190 '       set step size
200 IF M<-1 OR M>1 THEN ST=1/M-1/(M*90) ELSE ST=SGN(M)
210 IF ST=0 THEN ST=1' set step size if M=0
220 '   draw line between points
230 FOR X=0 TO X2-X1 STEP ST
240    SET (X+X1,M*X+Y1)
250 NEXT X
260 GOTO 260'    freeze screen
10000 END
```



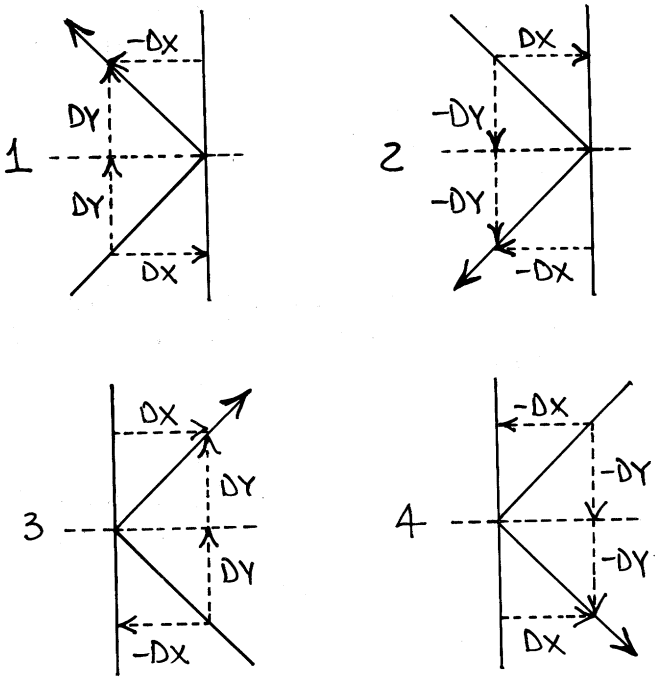
```
10 'FILENAME: "C5P15"
20 'FUNCTION: DRAW A CIRCLE USING SINE AND COSINE FUNCTIONS
30 ' AUTHOR : JPG                    DATE:   10/79
40 CLS
50 INPUT "RADIUS";R:  INPUT "CENTER COORDINATES";X1,Y1
60 CLS
70 FOR TH=0 TO 6.3 STEP 1/R
80    X=R*COS(TH)+X1: Y=.5*R*SIN(TH)+Y1: SET(X,Y)
90 NEXT TH
100 GOTO 100'   freeze screen
10000 END
```



Bouncing Dots

In many games, as well as in serious graphing applications, you will want to have a dot move in a straight direction until it meets an obstacle and keep moving in a new direction. Consider the possible bounces:

Vertical Wall Collision



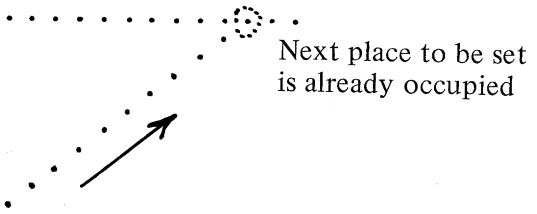
Each moving dot changes its X direction by an amount DX, and its Y direction by an amount DY. In all the conditions above, the Y direction increment DY never changes sign. In conditions 1 and 2, DX starts as positive, but changes sign to negative on the bounce.

If you investigate the four possible bounces from a horizontal wall, you will find that, as expected, the only change is the reversal of the sign of DY.

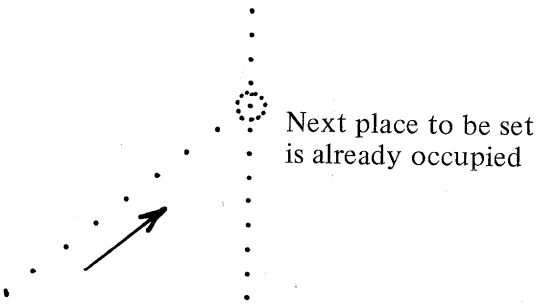
This understanding is all that is necessary to graph a moving and bouncing dot. The dot is a pixel drawn with a SET command. The coordinates of the SET are X and Y. The movement is provided with a SET at a new position X+DX, Y+DY followed with a RESET of the old dot at X,Y.

```
100 SET(X,Y)
110 X=X+DX: Y=Y+DY
120 SET(X,Y)
130 RESET(X-DX,Y-DY)
140 GOTO 110
```

To bounce the dot off the wall, you must "feel" ahead to see if any part of the surrounding territory is occupied. It is not enough to just sense the status of the next point. For example, suppose the dot is moving up and to the right, and it encounters a horizontal wall.

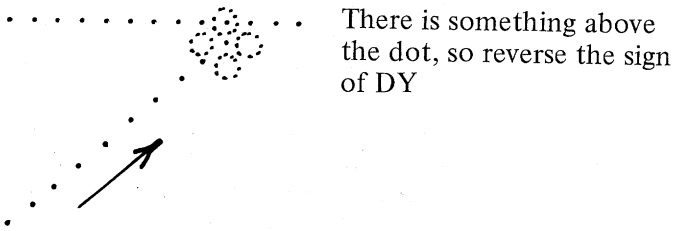


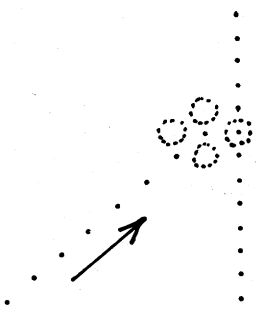
If the "next place" is the only point that is considered on a bounce, the computer couldn't tell if the wall were horizontal or vertical.



The only way to judge which direction increment, the DX or the DY, needs to change sign is to sense in all four directions.

Examples:

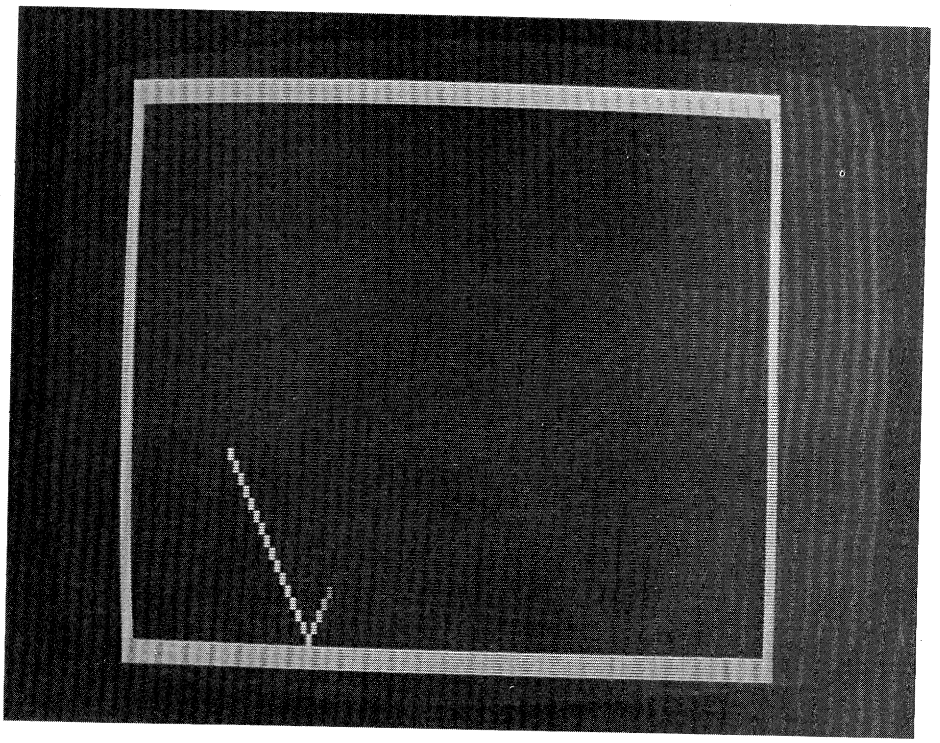


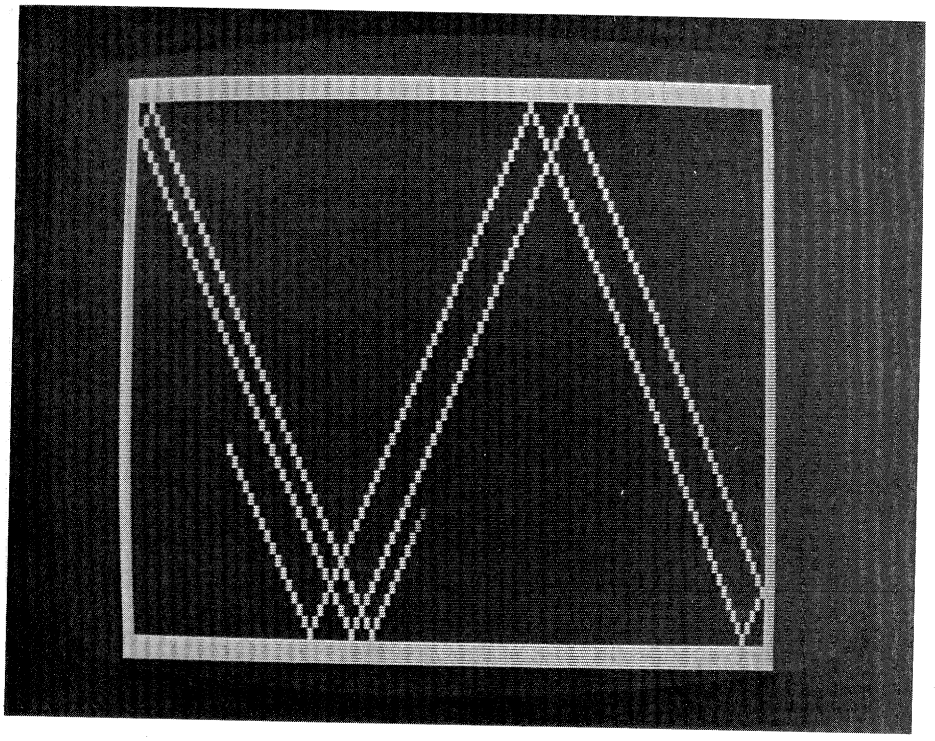


There is something to the right of the dot, so reverse the sign of DX

This program bounces a dot off any wall set up in either a horizontal or vertical position. It allows DX and DY to be defined for any value between 1 and 2, and builds its walls on the four sides of the screen.

```
10 'FILENAME: "C5P16"
20 'FUNCTION: BOUNCING DOT PROGRAM
30 ' AUTHOR : JPG                    DATE:   2/80
40 INPUT "DX, DY BETWEEN 1 AND 2 INCLUSIVE";DX,DY
50 INPUT "STARTING COORDINATES";X,Y: CLS
60 FOR I=0 TO 127 ' this loop draws horizontal boundaries
70   SET(I,0): SET(I,1): SET(I,46): SET(I,47)
80 NEXT I
90 FOR I=0 TO 47 '  draw vertical boundaries
100   SET(0,I): SET(1,I): SET(126,I): SET(127,I)
110 NEXT I
120 SET(X,Y) '   put a point at location X,Y
130 '    change direction of dot if necessary
140 IF POINT(X+DX,Y) OR POINT(X-DX,Y) DX=-DX
150 IF POINT(X,Y+DY) OR POINT(X,Y-DY) DY=-DY
160 X=X+DX: Y=Y+DY: GOTO 120
10000 END
```

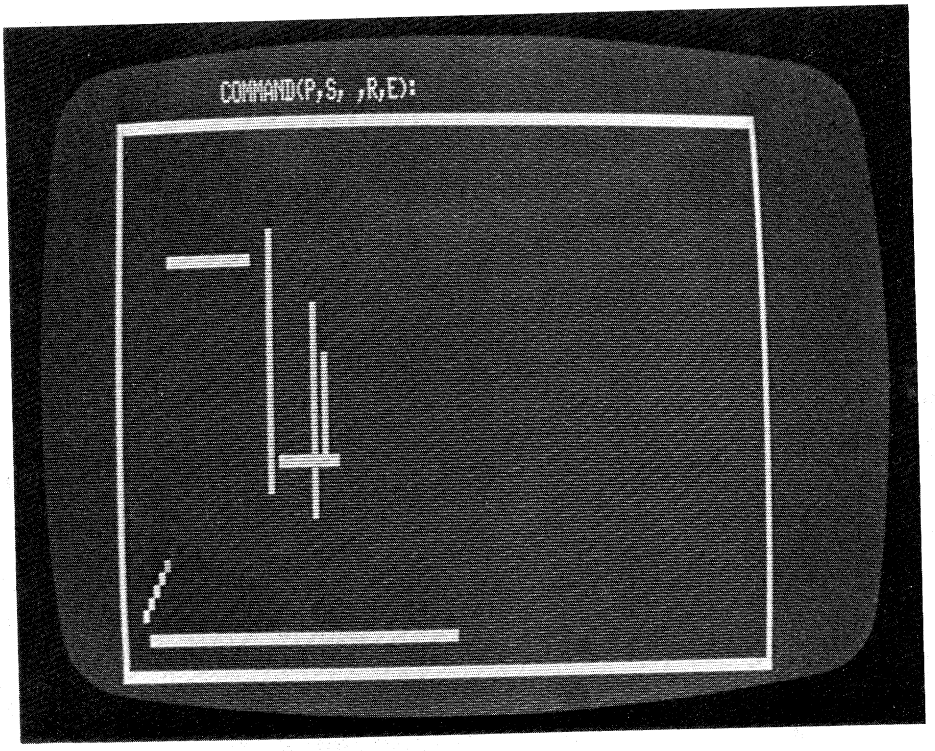


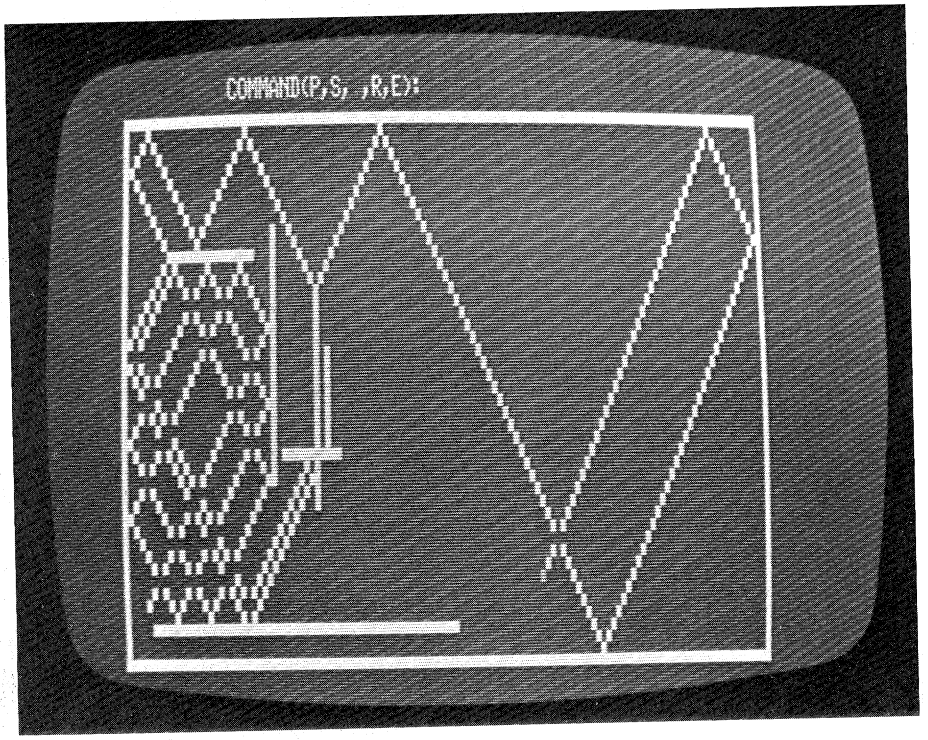


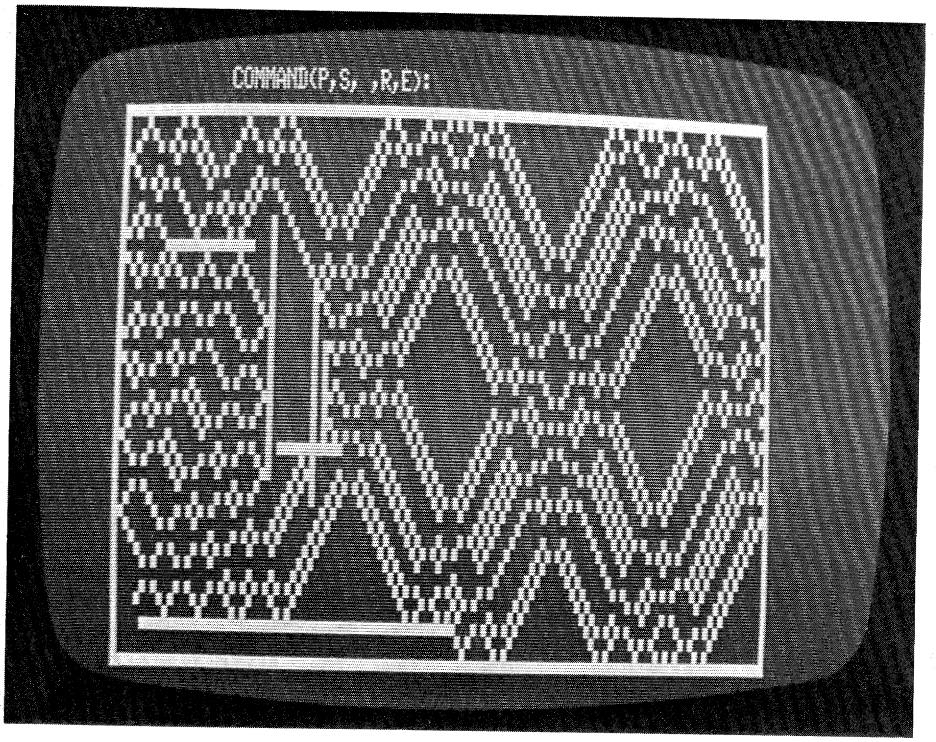
A very nice program that actually served as an inspiration for the discussion above is the one that Steve Grillo wrote as an exercise in visual graphics. He calls it "The Happy Hopi" for obvious reasons.

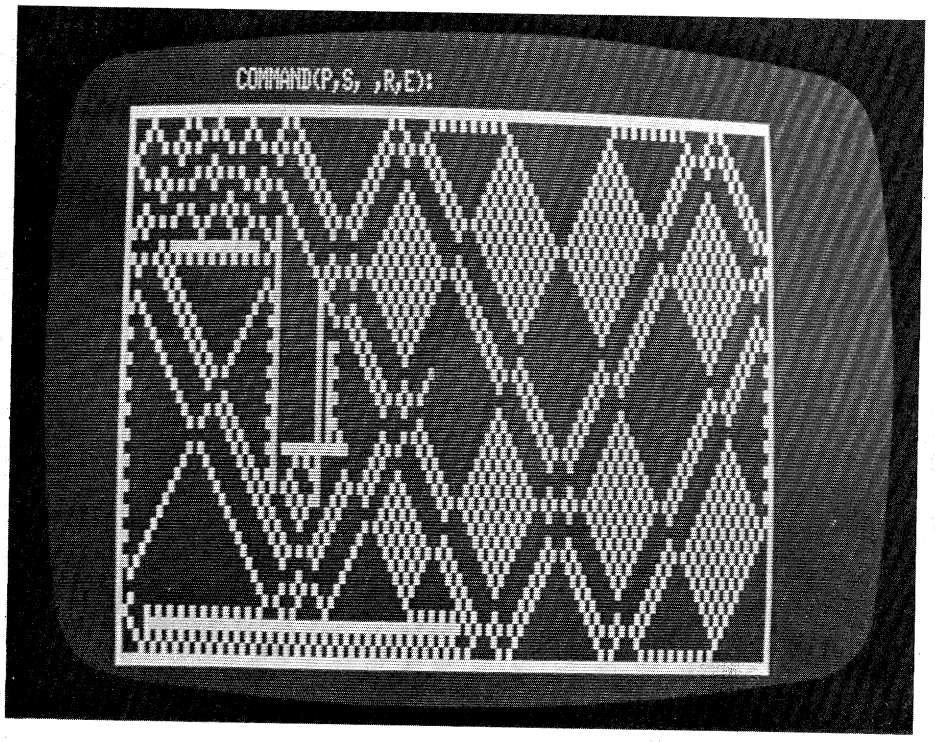
```
10 'FILENAME: "C5P17"
20 'FUNCTION: CREATING COMPUTER ART
30 ' AUTHOR : SPG                    DATE:   2/79
40 '  next line initializes variables; clears screen
50 RANDOM: CLEAR 1000: DEFSTR A: DEFINT B-Z: E=1: P=-1: CLS
60 CLOSE: PRINT "  PRESS:"
70 PRINT "     L-   TO LOOK AT OLD PICTURES"
80 PRINT "     M-   TO CREATE NEW PICTURES"
90 A=INKEY$: IF A="" THEN 90
100 IF A="L" THEN 790
110 '       user wants to create new art
120 CLS
130 '    the next loop draws 6 random lines on the screen
140 '              (3 vertical, 3 horizontal)
150 FOR T=1 TO 3
160 '      set up random variables
170    B0=RND(25)*2+4: B1=B+RND(50-.5*B)*2: B2=RND(21)*2+4
180    C=RND(11)*2+4: C1=C+RND(23-.5*C)*2: C2=RND(60)*2+4
190 '     draw the lines using the variables
200    FOR N=B0 TO B1: SET(N,B2): NEXT
210    FOR N=C TO C1: SET(C2,N): NEXT
220 NEXT T
230 '  the next two lines initialize the direction of the dot
240 S=INT(RND(100)/50): IF S=0 THEN S=-1
250 T=INT(RND(100)/50): IF T=0 THEN T=-1
260 '   the next two lines draw a boundary around the screen
270 FOR N=1 TO 125: SET(N,3): SET(N,47): NEXT
280 FOR N=4 TO 46: SET(1,N): SET(125,N): NEXT N
290 '    next line gives the point a starting spot &
300 '        makes sure it isn't occupied
310 X=RND(61)*2+2: Y=RND(20)*2+5: IF POINT(X,Y) THEN 310
320 '     print the command statement on the screen
330 PRINT@ 10,"COMMAND(P,S, ,R,E):       ";
340 '
350 '
360 '      the rest is the main portion of the program
370 '   next 4 lines change the direction if necessary
380 IF POINT(X+1,Y) THEN S=-1
390 IF POINT(X-1,Y) THEN S=1
400 IF POINT(X,Y+1) THEN T=-1
410 IF POINT(X,Y-1) THEN T=1
420 '    previous point is blanked if P=1
430 IF P=1 THEN RESET(X,Y)
```

```
440 '   next 4 lines actually tell the dot where to move
450 IF S=1 THEN X=X+1
460 IF S=-1 THEN X=X-1
470 IF T=1 THEN Y=Y+1
480 IF T=-1 THEN Y=Y-1
490 ' next line erases point if it was white and E=1
500 IF POINT(X,Y) AND E=1 THEN RESET(X,Y) ELSE SET(X,Y)
510 '  the next line checks to see if a key was pressed-
520 '    if not, then go back to the main program
530 A=INKEY$: IF A="" THEN 380
540 '     print the letter that was pressed in the corner
550 PRINT@ 30,A;
560 '       execute the command
570 '    reverse the dot?
580 IF A="R" THEN T=-T: S=-S: GOTO 330
590 '    erase the dots trail?
600 IF A="P" THEN P=-P: GOTO 330
610 '    erase the dot when it runs over another?
620 IF A="E" THEN E=-E: GOTO 330
630 '    stop the motion of the dot?
640 IF A<>" " THEN 670
650 A=INKEY$: IF A="" THEN 650 ELSE 330
660 '     store the picture on disk?
670 IF A<>"S" THEN 330
680 '   store the file using random access
690 OPEN "R",1,"SCRNGRPH/DAT"'  open the file for access
700 FIELD 1, 255 AS B$: M=LOF(1): I=15359' initialize
710 A=""'    nullify A$
720 '   the next 5 lines copy the screen to disk
730 I=I+1
740 IF LEN(A)<255 THEN 770
750 M=M+1: LSET B$=A: PUT 1,M
760 IF M/4=INT(M/4) THEN 60 ELSE 710
770 A=A+CHR$(PEEK(I)): POKE I,46: GOTO 730
780 '     this portion copies the art on disk to the screen
790 OPEN "R",1,"SCRNGRPH/DAT": FIELD 1, 255 AS B$
800 IF LOF(1)<>0 GOTO 830
810 PRINT "SORRY, BUT THERE IS NO ART ON FILE...   <<CR>>";
820 A=INKEY$: IF A="" THEN 820 ELSE 60
830 CLS
840 FOR C=1 TO LOF(1) STEP 4
850    FOR D=0 TO 3: GET 1,C+D: PRINT B$: NEXT D
860    PRINT @970, "  <<CR>>";
870    A=INKEY$: IF A="" THEN 870 ELSE PRINT
880 NEXT C
10000 END
```


COMMAND(P,S, ,R,E):

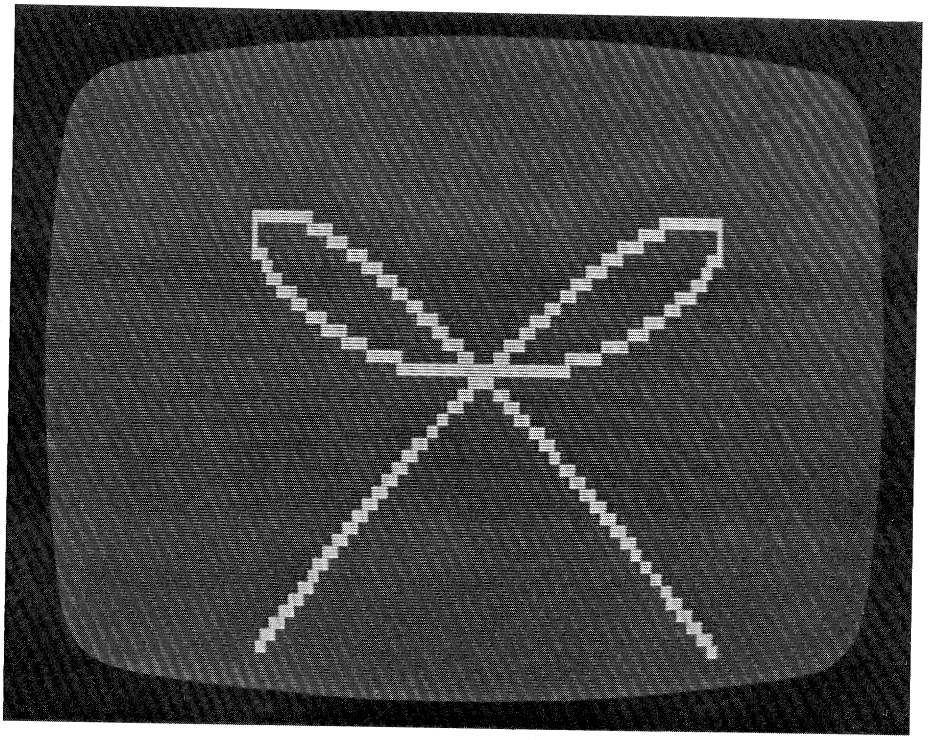

COMMAND(P,S, ,R,E):

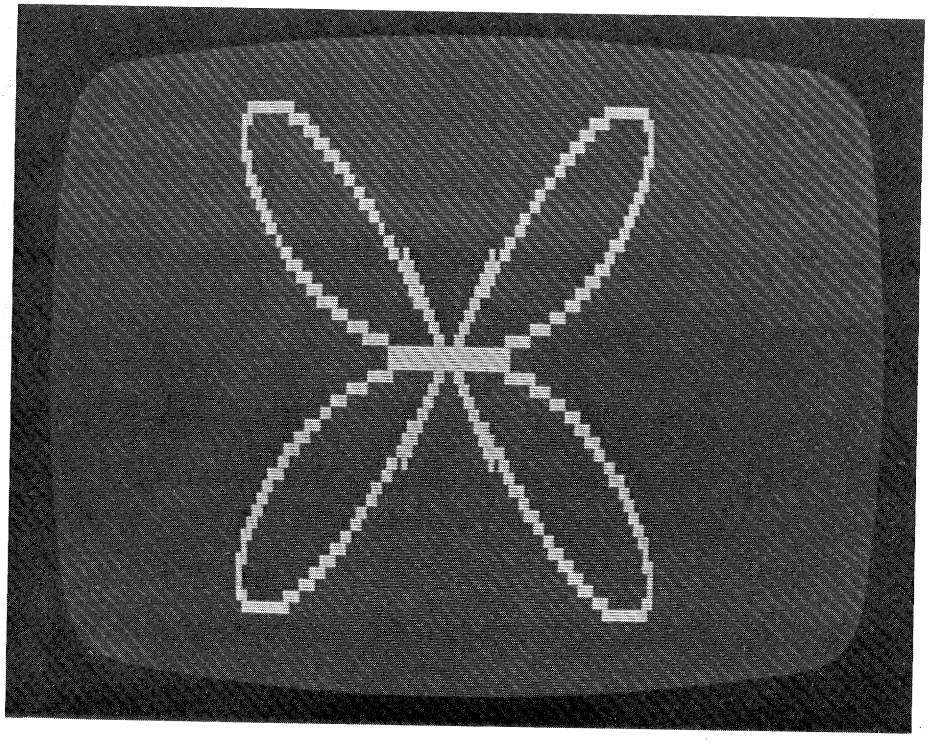

COMMAND(P,S, ,R,E):

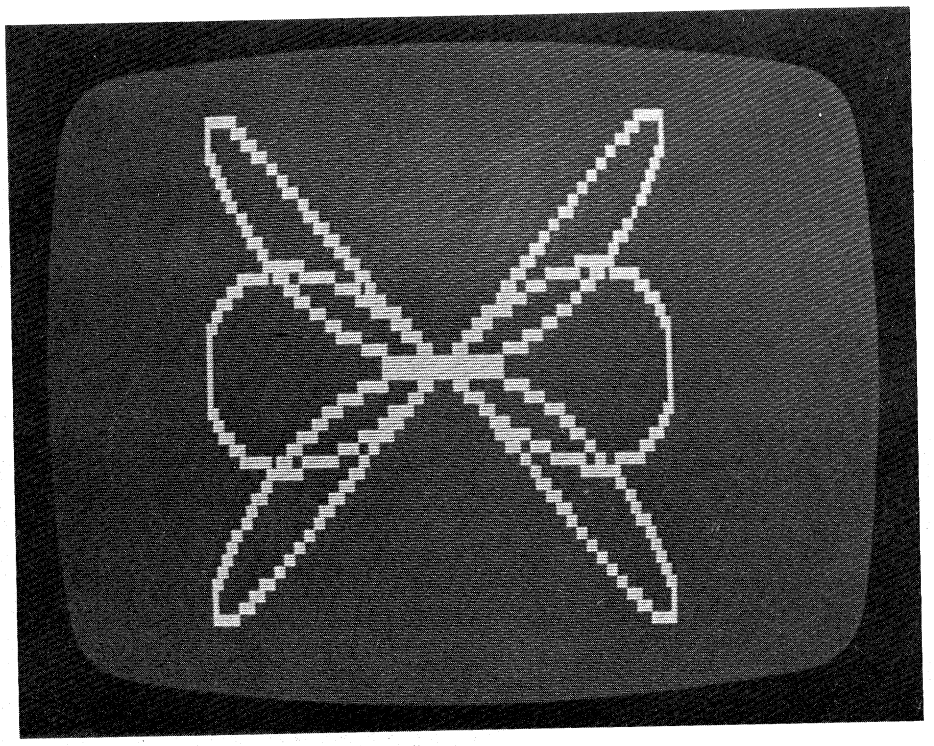

COMMAND(P,S, ,R,E):

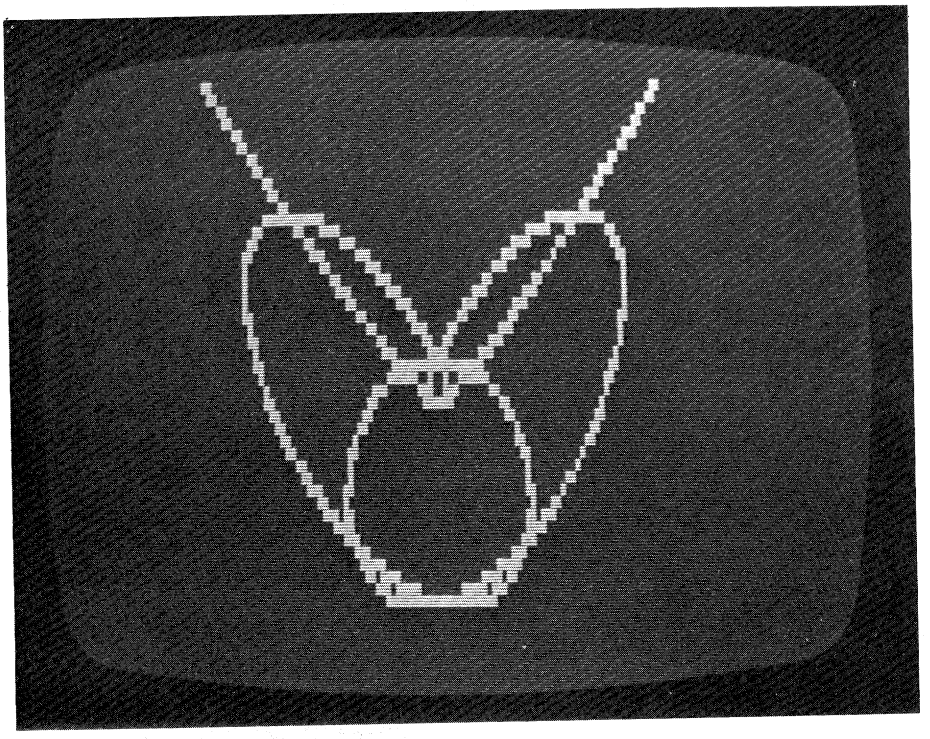
The last program produces graphic designs that are familiar to all mathematicians: These are variously called roses, limacons, lemniscates, or Lissajous figures, and they are not difficult to produce. The following program allows the user to select a pattern, choose its density, then produce the output on either the screen or both the screen and the line printer. We have included a number of examples to show you its flexibility.

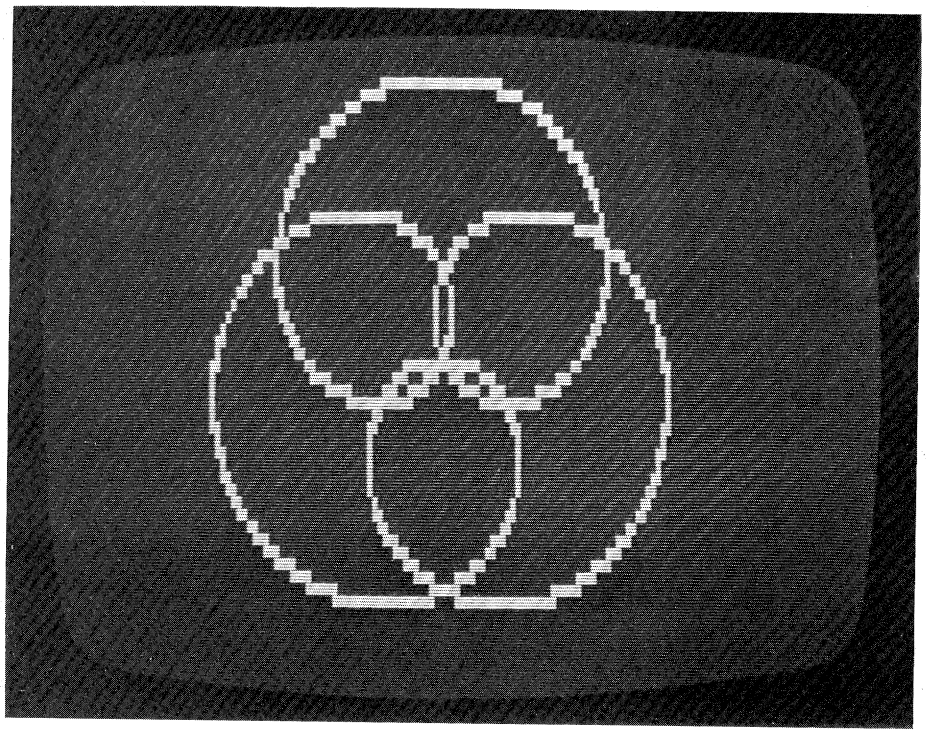
```
10 'FILENAME: "C5P18"
20 'FUNCTION: PRODUCE INTERESTING PATTERNS
30 ' AUTHOR : SPG                    DATE:   4/79
40 DEFINT X,Y,Z,Q,K
50 DEFINT K,Q,X,Y,Z: P2=6.283: Z=23: K=2: C=62: T=3
60 INPUT "FIRST VARIABLE (0<X<10)";A
70 INPUT "SECOND VARIABLE (GREATER THAN THE FIRST VARIABLE)";B
80 INPUT "OUTPUT TO PRINTER (1=YES)";P
90 INPUT "DENSITY  (1=SUPER DENSE, BUT SLOW; 200=SCATTERED)";ST
100 ST=ST/1000'     compute step size
110 CLS
120 '     create the picture
130 FOR TH=0 TO P2 STEP ST
140     R=Z*SIN(TH*T): X=K*R*COS(A*TH)+C: Y=R*SIN(B*TH)+Z
150     SET(X,Y)
160 NEXT TH
170 IF P<>1 THEN 300'  did the user want it copied?
180 '     print values for two main variables
190 LPRINT CHR$(29)'     normal printer density (10 cpi)
200 LPRINT TAB(12)"First Variable=";A;
210 LPRINT TAB(44)"Second Variable=";B
220 LPRINT
230 LPRINT CHR$(31)'  set printer density to 16.5 cpi
240 '    copy the screen to the printer
250 FOR Y=0 TO 47
260   FOR X=0 TO 123
270     IF POINT(X,Y) THEN LPRINT "*"; ELSE LPRINT " ";
280  NEXT X: LPRINT " ": SET(0,Y)
290 NEXT Y
300 GOTO 300'   freeze the screen
10000 END
```



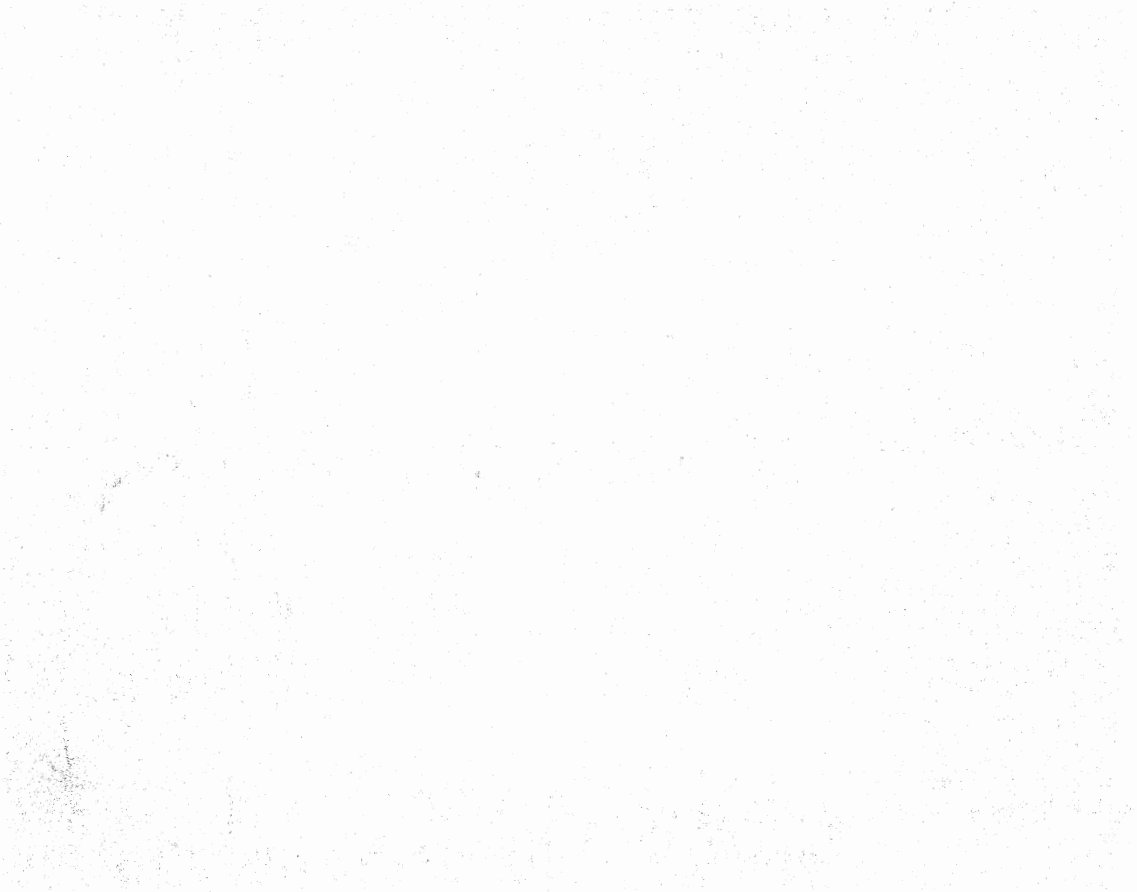


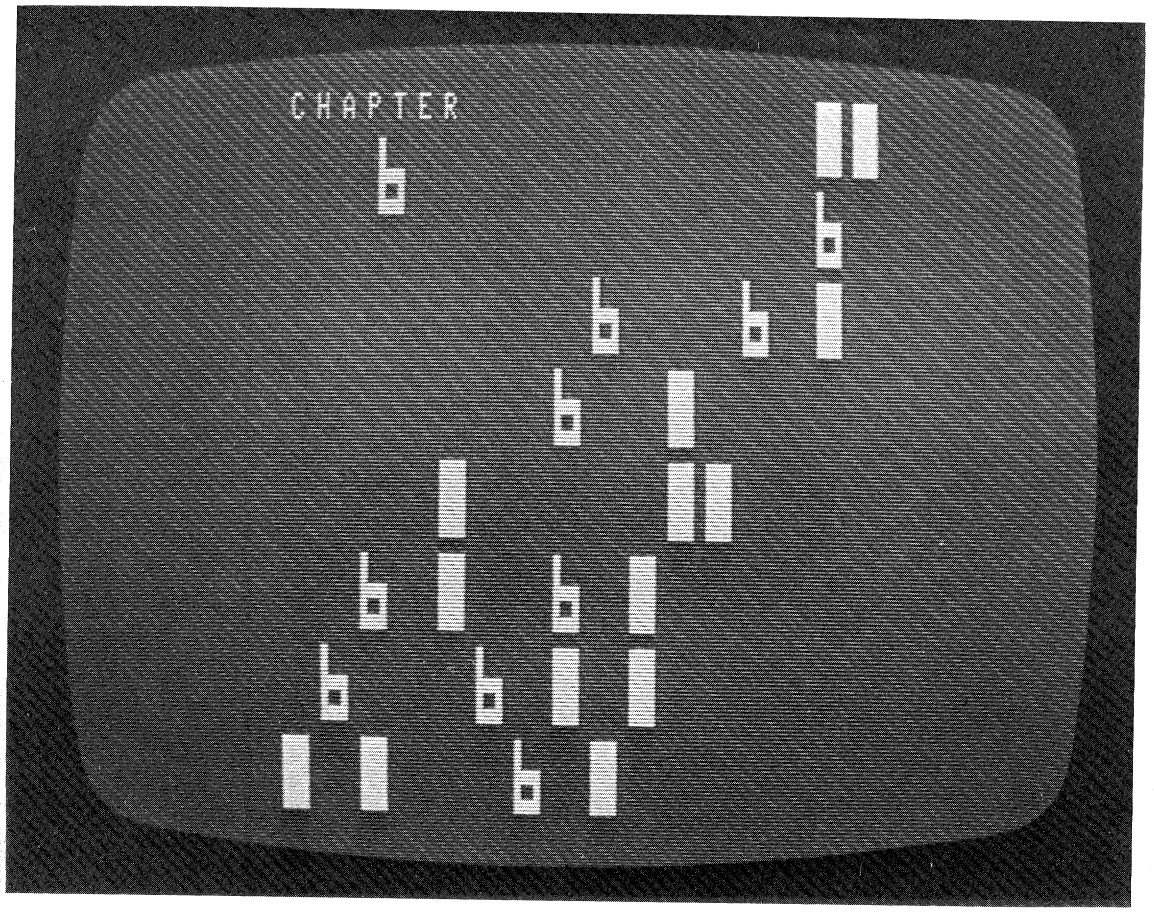






Graphing techniques are supremely rewarding to both the programmer who masters them and the program user who delights in their visual effects. They often transform a dull program into an exciting one, and provide the user with pictures, generally known to be worth a thousand words each. Sometimes, though, words are the stock in trade for the computer, as in the many word processing applications indicate. When this is the case, BASIC's ability to handle strings is of paramount importance, and this capability makes it a language chosen by many. String handling will be discussed in the next chapter.





I/O, Strings, and Disk BASIC

As good as Level II BASIC is with its graphics and enhanced string handling capabilities, the TRS-80 Disk Operating System or TRSDOS, includes some embellishments that further increase the power of the language. These additions to the language are supplemental to the 12K ROM (Read Only Memory) BASIC, and occupy RAM (Random Access Memory). You may refer to the memory map in Appendix C for a visual explanation of the way these enhancements use the RAM.

This chapter discusses the Disk BASIC features that don't deal with tape or disk I/O, since chapters 7 and 8 cover those topics thoroughly.

DEFUSR and USRn

These two language features are very closely related. The DEFUSR corresponds to a DEF FN, except that it refers to a user-defined machine language function, and the USR corresponds to the invocation of that function.

The DEFUSR defines the entry point to a machine language routine as an address, and it also associates that routine with a number from 0 to 9, which allows up to 10 machine language

routines to be called within a program.

```
10 DEFUSR0=32000 ' Each of the four arguments is in
20 DEFUSR1=32100 ' decimal (by default).
30 DEFUSR2=32200
40 DEFUSR3=32300
```

The four lines above could just as well have defined the machine language subroutine entry points in hexadecimal.

```
10 DEFUSR0=&H7D00 '   7D00 HEX= 32000 DECIMAL
20 DEFUSR1=&H7D64 '   7D64 HEX= 32100 DECIMAL
30 DEFUSR2=&H7DC8 '   7DC8 HEX= 32200 DECIMAL
40 DEFUSR3=&H7E2C '   7E2C HEX= 32300 DECIMAL
```

You may wish to define a machine language subroutine's entry point in hex depending on which way it is convenient for you to think of addresses in the computer's memory. In this example we have assumed that the four programs have been loaded into the appropriate places, and that each one is less than 100 bytes long. Note that when BASIC is loaded and asks the question "MEMORY SIZE?" you should respond 31999 (or less) to prevent your BASIC program from using the area above 32000, which is the portion of memory in which the machine language subroutines were loaded.

The USR instruction is really a function, and in TRSDOS BASIC it must have a suffix digit of 0 to 9 corresponding to the DEFUSR statement that specifies the entry point. There is but a single argument to the function, and as with all functions it returns only one value. Both the argument and the returned value are integers.

Examples:

```
500 X=USR3(J+1)
600 Y=USR7(RND(99))
700 Z=USR6(-5)
```

If you are interested in writing an assembly language routine that can be placed in memory for use with USR and DEFUSR, we refer you to your local Radio Shack store, on whose shelves you will find several books dealing with this subject. Below are some examples of functions that could be written in assembly language (for speed), and are therefore suited to the DEFUSR–USR approach.

(1) The argument is the screen address (0 to 1023) for the cursor. The value returned is the length of the nonblank string of characters to the right (or left) of the cursor.

(2) The argument is a distribution specifier for a random variable. The value returned is a random variate of the specified distribution (Poisson, normal, or other).

(3) The argument is a coded musical note. Ten thousands = port number (1, 2, or 3). Thousands = A to G (A = 1000, B = 2000, etc.) Hundreds = Octave (100 = lowest, 800 = highest). Tens = time

(1 = 32nd note, 2 = 16th, 3 = 8th, 4 = 4th, . . .). Ones = sharp (1), natural (2), or flat (3). The value returned is the specified note for the specified time on the specified port.

POS, INSTR, and MID$

Although the POS function can be used with Level II BASIC without having to use TRSDOS BASIC, it is closely related to the other topics of this chapter. It is a function that returns the position of the cursor on the line, a value from 0 to 63. The argument of the POS function is a dummy. It must be there, and can be any numeric expression.

Examples:

```
10 PRINT "A" TAB(POS(0)+10)"B" TAB(POS(0)+10)"C"
```

There will be 10 spaces between the A and the B, and 10 spaces between the B and the C.

```
10 FOR I=1 TO 100: INPUT X$
20    IF POS(0)+LEN(X$)>=63 THEN PRINT
30    PRINT X$+" ";
40 NEXT I
```

This loop reads text one word at a time and prints the words side by side, separated by blanks. A word will not be divided between two lines.

The INSTR function searches one string to see if it contains another. Like the MID$ function, it can have either two or three arguments. The function is written

INSTR(p,string1$,string2$)

where the optional variable p indicates the byte position at which the search is to begin; string1$ is the string expression being searched; and string2$ is the string expression being sought.

The value returned is the starting position of the substring that was found in the target string, or zero if it wasn't found. If the optional starting position for the search is not specified, the search begins at byte 1.

Instruction	Output
`10 A$="NOT WITH A BANG, BUT A WHIMPER"`	
`20 PRINT INSTR(A$," ")`	4
`30 PRINT INSTR(5,A$," ")`	9
`40 PRINT INSTR(A$,"A")`	10
`50 PRINT INSTR(A$,"J")`	0
`60 PRINT INSTR(4,A$,"T ")`	20
`70 PRINT INSTR(A$,"WITH")`	5
`80 PRINT INSTR(2,A$," A ")`	9

The INSTR function can simplify the neat printing of inputted text, as in the example for the POS function.

```
10 'FILENAME: "C6P1"
20 'FUNCTION: DEMONSTRATION OF STRING FUNCTIONS
30 ' AUTHOR : JPG                    DATE:   6/79
40 CLEAR 3000: DIM B$(50): I=1 '     initialize variables
50 INPUT "TYPE TEXT (NO COMMAS)"; A$: LPRINT A$
60 IF A$="$$END" THEN 160 ' check for end of data entries
70 B$(I)=B$(I)+" "+A$'     concatenate old and new strings
80 IF LEN(B$(I))<60 THEN 50 ' back if string short enough
90 ' loop to find first (rightmost) space < 60 characters
100 FOR J=60 TO 1 STEP -1: X=INSTR(J,B$(I)," ")
110 IF X=0 THEN NEXT J '         Keep checking for a space
120 'found one.  adjust strings accordingly, then go back
130 B$(I+1)=MID$(B$(I),X): B$(I)=LEFT$(B$(I),X)
140 I=I+1: GOTO 50
150 '                                    print the final copy
160 FOR J=1 TO I: LPRINT B$(J): NEXT J
10000 END

The program which is responsible
for allowing the user to
modify a record's contents
should be fairly simple yet requires great care in its design.
Consider the fields of the
record and the effect that
a change would have
on each of them.
$$END
 The program which is responsible for allowing the user to modify
 a record's contents should be fairly simple yet requires great
 care in its design. Consider the fields of the record and the
 effect that a change would have on each of them.
```

In the example above, notice how in line 130 the two-argument MID$ function acts like a RIGHT$ function, but instead of the numeric argument being a length, it is a starting position. The INSTR can take advantage of this feature of the MID$ in a name reversal subroutine, such as the one below. Compare the following program to program C2P5 in chapter 2. This example is typical of many applications, such as mailing lists or accounting, in which the name on a record is its key and it is arranged last name, then first but it is used in the order first, then last name. For example, a stored record might contain:

MINDERBINDER MILO

and the output should be:

MILO MINDERBINDER

```
10 'FILENAME: "C6P2"
20 'FUNCTION: REVERSAL OF NAMES
30 ' AUTHOR : JPG                    DATE:   7/79
40 '  the next lines use the subroutine
50 INPUT N$: LPRINT N$
60 IF N$="$$END" THEN STOP
70 GOSUB 80: LPRINT N$:  GOTO 50
80 '  the next line does the reversal
90 X=INSTR(N$," "): N$=MID$(N$,X+1)+" "+LEFT$(N$,X-1)
100 RETURN
10000 END
```

```
Bleaux Jo
Jo Bleaux
Farnsworth Ferdinand
Ferdinand Farnsworth
Badinage Billybob
Billybob Badinage
$$END
```

A one-line user-defined function could do all that this subroutine does, but it leaves much to be desired in clarity.

```
5 DEF FNR$(N$)=MID$(N$,INSTR(N$," ")+1)
              +" "+LEFT$(N$,INSTR(N$," ")-1)
```

INKEY$

The INKEY$ string function is a very powerful part of LEVEL II BASIC. It does not have an argument. You can think of it as a single-character input function that doesn't need an enter keystroke. When the INKEY$ function is executed, it "strobes", or scans, the keyboard checking for a depressed key. If no key is depressed, the string returned is a null string, "". If a key is depressed, the string returned is that keystroke, even if it is a control key such as the backspace or the ENTER key. Because the keyboard scan occurs so fast, when INKEY$ is executed it must be given multiple opportunities to execute in order to find a key that is depressed. For this reason the INKEY$ is always placed inside a loop.

Example:

```
100 X$=INKEY$: IF X$="" THEN 100
110 PRINT @ 960,X$;
```

This program segment strobes the keyboard repeatedly. When a key is depressed, the keystroke isn't null, X$ isn't "", so line 110 is executed and prints that character at the bottom of the screen. Note the ";" punctuation at the end of the PRINT @. Remember that it keeps the carriage/cursor from returning so that what was printed on the bottom line stays on the bottom line without the standard single-line scrolling of the screen.

```
100 X$=INKEY$: IF X$="" THEN 100
110 PRINT @ 0, ASC(X$);X$;: GOTO 100
```

When a key is depressed the ASCII code value for the depressed character is printed at the upper left, immediately followed by the character itself.

```
100 PRINT "TYPE JUST DIGITS"
 110 X$=INKEY$: IF X$="" THEN 110
 120 IF X$<"0" OR X$>"9" THEN PRINT "ERROR": GOTO 100
 130 PRINT @ 0,X$;: GOTO 110
```

The two following subroutines might seed a fertile mind with ideas for other uses of their capabilities.

```
10 'FILENAME: "C6P3"
20 'FUNCTION: USING INKEY$ FOR NUMERIC VARIABLES
30 ' AUTHOR : JPG                    DATE:   7/79
40 '   routine to test the input subroutine
50 CLS: PRINT "TYPE AN INTEGER VALUE (0=STOP)"
60 GOSUB 1000: PRINT: PRINT A
70 IF A=0 THEN 10000
80 GOTO 80
1000 '   subroutine to place an all-digit positive value
1010 '           into a string (A$)
1020 A$="" '     null the string
1030 X$=INKEY$: IF X$="" THEN 1030 '  wait for pressed key
1040 IF X$=CHR$(13) THEN A=VAL(A$): RETURN '  /EN/ KEY?
1050 'make sure key is a digit, then add strings & go back
1060 IF X$>="0" AND X$<="9" THEN
        A$=A$+X$: PRINT @ 512,A$;: GOTO 1030
1070 ' print error message (A$ not legal key) and go back
1080 PRINT @ 576,"***ERROR*** -- SO FAR YOU HAVE -";A$;
1090 GOTO 1030
10000 END
```

```
10 'FILENAME: "C6P4"
20 'FUNCTION: FLASH A PROMPT WHILE KEYING IN A STRING
30 ' AUTHOR : SPG                DATE:   1/80
40 '    test the subroutine
50 P$="NAME: " '   this will be flashed on the screen
60 GOSUB 1000: PRINT: PRINT: PRINT N$
70 STOP
80 '   subroutine to flash a prompt for string input
90 '     Note:
100 '          I is a dummy timing variable which is
110 '     changed as necessary to keep the prompt
120 '     flashing at a fairly constant rate while the
130 '     string is being typed in.
1000 CLS: I=30 '   make sure prompt gets printed soon
1010 X$=INKEY$: IF X$="" THEN 1050 'wait for pressed key
1020 IF ASC(X$)=13 RETURN ' leave routine if done entering
1030 ' add strings and print the string on the screen
1040 N$=N$+X$: PRINT @ 135,N$;: I=I+3
1050 I=I+1 '  increment the timing variable
1060 ' put prompt on screen & reset timer if necessary
1070 IF I>=20 THEN I=0: PRINT @ 128,P$;
1080 '   blank out the prompt if it is time to do so.
1090 IF I>7 AND I<13 PRINT @ 128, STRING$(LEN(P$)," ");
1100 GOTO 1010
10000 END
```

LINE INPUT

The commonly used INPUT command is quite flexible, but it has its limitations. You can't input commas or quotes, and if you try to input a string with leading blanks, BASIC does you the questionable favor of eliminating them to shorten the string. Also, a question mark is printed on the screen every time the INPUT is executed, sometimes leading to awkward-looking displays.

The LINE INPUT command overcomes all of these difficulties at the minor expense of being able to input only one variable at a time, which must be a string. Probably the most useful feature of the LINE INPUT results from the fact that if you input a null string (hit the ENTER key) it is inputted, whereas this action is ignored during the usual INPUT operation.

The following examples should serve to show the power of this very useful command.

Assume this statement is executed:

```
10 LINE INPUT "TYPE A LINE: ";X$: PRINT X$: GOTO 10
```

Study this dialog to see how the LINE INPUT works. (The /EN/ signifies the ENTER stroke).

```
TYPE A LINE: 79,282.44/EN/
79,282.44
TYPE A LINE: HE SAID, "HI!"/EN/
HE SAID, "HI!"
TYPE A LINE:                                  CENTERED TITLE/EN/
                                  CENTERED TITLE
```

The next program segment illustrates a useful feature of LINE INPUT: it accepts a new value as input, even if a null string is entered. Note that the INPUT statement cannot accept a null string. If a null string is INPUT, the previous INPUTted value is used.

```
10 INPUT A$: IF A$="" PRINT "NULL A" '  type ABC (no output)
20 INPUT B$: IF B$="" PRINT "NULL B" '  type DEF (no output)
30 INPUT A$: IF A$="" PRINT "NULL A AGAIN"
               ELSE PRINT LEN(A$);A$   '  type "", output is  3  ABC
40 LINE INPUT X$: IF X$="" PRINT "NULL X"  ' type XYZ (no output)
50 LINE INPUT Y$: IF Y$="" PRINT "NULL Y"  ' type RST (no output)
60 LINE INPUT X$: IF X$="" PRINT "NULL X AGAIN"
            ELSE PRINT LEN(X$);X$  ' type "", output is NULL X AGAIN
```

This feature of the LINE INPUT is quite useful in file editing. In the example below, the array X$ contains the names, street addresses, city, state, zip code, and phone numbers of some data that is on file, and this subroutine allows the user to update the data.

```
10 'FILENAME: "C6P5"
20 'FUNCTION: MORE EXAMPLES OF USING LINE INPUT
30 ' AUTHOR : JPG                    DATE:   9/79
40 CLEAR 1000: DIM X$(20,6)'make lots of room for strings
50 ' X$(I,1) is name  X$(I,2) is street  X$(I,3) is city
60 ' X$(I,4) is zip   X$(I,5) is state   X$(I,6) is phone
70 '  main program begins here
80 N=3: GOSUB 100: GOSUB 100: GOSUB 100
90 STOP
100 '  subroutine for update of records
110 FOR J=1 TO 6: READ P$(J): NEXT J: RESTORE
120 FOR I=1 TO N:CLS: FOR J=1 TO 6
130 PRINT @64*J,P$(J), X$(I,J);
140 LINE INPUT A$: IF A$<>"" THEN X$(I,J)=A$
150 NEXT J,I: RETURN
160 DATA NAME,STREET,CITY,STATE,ZIP,PHONE
10000 END
```

The LINE INPUT in the subroutine uses an ENTER stroke on the keyboard as an indication that there is no change in the filed data. Otherwise, the user types in the new information and it replaces the old information in the array.

MID$ for Replacement

Our previous encounter with MID$ showed it to be a powerful function that would extract one string from another.

For example, if A$ = "ABCDEFG" the statement:

```
X$=MID$(A$,4,2)
```

copies the two characters of A$, starting at the fourth, into X$.

With Disk BASIC, the MID$ function can disobey the usual rule that functions must appear to the right of the equal sign. If MID$ appears to the left of the equal sign, it is a *replacement* function instead of an *extraction* function. It copies any portion of the string to the right of the equal sign into the string argument of MID$ beginning at the position specified by the middle argument for the length specified by the third argument.

```
Suppose X$="ABCDEFGH"
```

Instruction	Output
`10 MID$(X$,4,2)="XY": LPRINT X$`	ABCXYFGH
`20 MID$(X$,1,5)="12345": LPRINT X$`	12345FGH
`30 MID$(X$,6,1)="XYZ": LPRINT X$`	ABCDEXGH

This function is powerful in text editing programs. The example below searches the 64-character string X$ for the specified target string T$. If T$ is found in X$, it is changed to the new version N$. For example, the string might contain a line of text and the user could change an occurrence of the misspelled word "COERTION" to the proper spelling "COERCION".

```
10 'FILENAME: "C6P6"
20 'FUNCTION: TO SUBSTITUTE WITHIN A STRING
30 ' AUTHOR : JPG                DATE: 8/80
40 '
50 CLEAR 200: CLS
60 DIM X$(20)
70 DATA "antelope elund dikdik impala gazelle serenuk"
80 DATA "couser tiger lion leopard jaguar lynx"
90 FOR I=1 TO 2
95   READ X$(I): PRINT X$(I): GOSUB 1000
100 NEXT I: GOTO 10000
```

```
1000 '*****              subroutine to substitute
1010 LINE INPUT "type target string: ";T$
1020 LINE INPUT "type new (replacement) string: ";N$
1030 IF LEN(N$)>LEN(T$) THEN PRINT "too long": GOTO 1020
1050   FOR J=1 TO LEN(X$(I)): P=INSTR(J,X$(I),T$)
1060     IF P<>0 THEN MID$(X$(I),P)=N$
1070   NEXT J: PRINT X$(I)
1080 RETURN
10000 END


antelope elund dikdik impala gazelle serenuk
type target string: elund
type new (replacement) string: eland
antelope eland dikdik impala gazelle serenuk
couser tiger lion leopard jaguar lynx
type target string: couser
type new (replacement) string: cougar
cougar tiger lion leopard jaguar lynx
READY
>
```

TIME$

TIME$ is a variable that is defined by the computer when it is turned on and Disk BASIC is loaded. If this statement is executed:

```
500 PRINT TIME$
```

the output is the system date and time. That is, whatever date and time was typed in when TRSDOS was initiated is displayed as

MM/DD/YY HH:MM:SS

This 17-character string is initialized automatically to

00/00/00 00:00:00

if the TIME and DATE DOS commands were not used to accurately set the internal clock and calendar.

The TIME$ string is useful in two very important application areas. First, the date portion can be used in programs for account ageing, proper billing, inventory reorder, and all the other business applications that require action on particular dates. Second, the time portion can be used by a programmer for timing program loops and user responses. The two programs below illustrate this use.

```
10 'FILENAME: "C6P7"
20 'FUNCTION: TIMING OF ARITHMETIC OPERATIONS
30 'AUTHOR  : JPG                DATE: 12/79
40 '
50 DEFSNG A-Z
60 'add=1, sub=2, mul=3, div=4
70 CLS: F$="%     %  % %  % %  % %  % %"
80      G$="#######  ###  ###  ###  ###"
90 LPRINT USING F$; "NO. OPS","ADD","SUB","MUL","DIV"
100 'iterations are 100 to 1000 in steps of 100.
110 'note that each iteration contains 10 operations.
120 FOR I=10 TO 100 STEP 10
130 S=0: K=1.012321: GOSUB 500: T1=T
140 FOR J=1 TO I: S=S+K: S=S+K: S=S+K: S=S+K: S=S+K
150                S=S+K: S=S+K: S=S+K: S=S+K: S=S+K
160 NEXT J: GOSUB 500: TA=T-T1: T1=T
170 FOR J=1 TO I: S=S-K: S=S-K: S=S-K: S=S-K: S=S-K
180                S=S-K: S=S-K: S=S-K: S=S-K: S=S-K
190 NEXT J: GOSUB 500: TS=T-T1: T1=T
200 FOR J=1 TO I: S=S*K: S=S*K: S=S*K: S=S*K: S=S*K
210                S=S*K: S=S*K: S=S*K: S=S*K: S=S*K
220 NEXT J: GOSUB 500: TM=T-T1: T1=T
230 FOR J=1 TO I: S=S/K: S=S/K: S=S/K: S=S/K: S=S/K
240                S=S/K: S=S/K: S=S/K: S=S/K: S=S/K
250 NEXT J: GOSUB 500: TD=T-T1
260 LPRINT USING G$; I,TA,TS,TM,TD
270 NEXT I
280 STOP
500 'subroutine returns the time in seconds
510 H=VAL(MID$(TIME$,11,1)):'   isolate hours
520 M=VAL(MID$(TIME$,13,2)):'   isolate minutes
530 S1=VAL(MID$(TIME$,16,2)):'  isolate seconds
540 T=S1+M*60+H*3600: RETURN
10000 END
```

```
NO. OPS   ADD   SUB   MUL   DIV
     10     1     0     1     2
     20     1     2     1     3
     30     2     2     3     3
     40     3     3     3     5
     50     3     4     4     5
     60     4     4     5     6
     70     4     5     6     7
     80     5     5     7     9
     90     6     6     7    10
    100     7     7     8    11
```

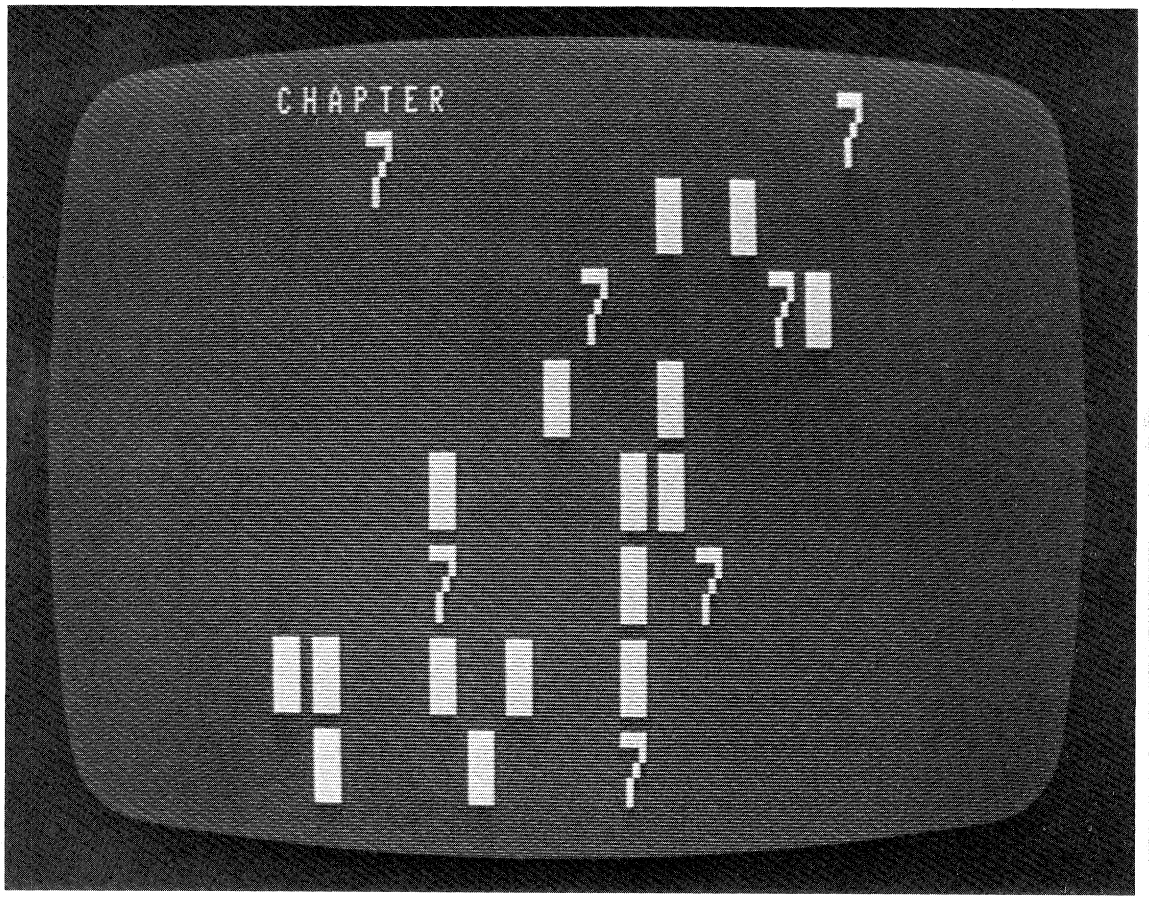
```
10 'FILENAME: "C6P8"
20 'FUNCTION: SIMPLE MATH PROGRAM USING TIMED INPUTS
30 ' AUTHOR : JPG                    DATE:   7/79
40 DEF FNT(A,B)=VAL(MID$(TIME$,A,B))
50 CLS
60 '  time user responses to 10 seconds or less
70 A=RND(100): B=RND(100)
80 PRINT @ 520, A;"+";B;"=";
90 GOSUB 170: T1=T: Y$=""
100 X$=INKEY$: IF X$<>"" THEN 130
110 GOSUB 170: IF T-T1<10 THEN 100
120 PRINT: PRINT "SORRY -- OUT OF TIME": GOSUB 210: GOTO 70
130 IF ASC(X$)<>13 THEN Y$=Y$+X$: PRINT @ 534,Y$;: GOTO 100
140 C=VAL(Y$): PRINT
150 IF C=A+B THEN PRINT "CORRECT!!"
             ELSE PRINT "SORRY, THAT IS INCORRECT."
160 GOSUB 210: CLS: GOTO 70
170 ' TIMING SUBROUTINE
180 H=FNT(11,1): M=FNT(13,2): S=FNT(16,2)
190 T=S+M*60+H*3600: RETURN
200 '   hold screen for a few seconds, then go back
210 FOR I=1 TO 400: NEXT I: RETURN
10000 END
```

```
        32 + 80 =
SORRY -- OUT OF TIME
```

If you need more accurate timing of a routine or of a response from a user, you may access the running clock of the computer from BASIC by PEEKing at (decimal) addresses 16448 through 16450. Try this program to test this feature.

```
10 'FILENAME: "C6P9"
20 'FUNCTION: USE COMPUTER AS STOPWATCH (TO 100THS OF SECONDS)
30 ' AUTHOR : SPG                    DATE:   7/79
40 CLS: M=16450: S=16449: P=16448 'clock locations in memory
50 INPUT "TYPE 'ENTER' TO BEGIN";A$
60 '     set orisinal timins values
70 CLS: M1=PEEK(M): S1=PEEK(S): P1=PEEK(P)
80 INPUT "PRESS /EN/ AGAIN TO STOP THE TIMER";A$
90 '   set final timins values
100 M2=PEEK(M): S2=PEEK(S): P2=PEEK(P)
110 '   compute the elapsed time
120 E2=M2*15300+S2*255+P2: E1=M1*15300+S1*255+P1
130 PRINT:PRINT
140 '   chanse the time to seconds
150 ET = (E2-E1)/255
160 '  print the elapsed time in seconds
170 PRINT USING "ELAPSED TIME WAS ABOUT ###.## SECONDS";ET
10000 END
```

This chapter has explored a number of niceties that Radio Shack's TRS-80 provides the programmer. These features give a flexibility to BASIC that tends to make it more acceptable as a language to be used in most application areas. But it is the ability to deal with files that makes BASIC particularly useful in business, education, scientific, and home applications. The file handling capabilities of the microcomputer will be discussed in depth in the next chapter.



Sequential Access File Processing

The next three chapters will differ from those that have gone before because they cover the BASIC commands, statements, and functions that deal with cassette and disk files. This chapter will cover the subject of sequential files on tape and on disk.

Commands for Program Files

SAVE, CSAVE

Files on tape or disk can be either programs or data, and in most instances both kinds of files will reside on a given medium. When a program is stored on disk or tape, it is copied directly from memory onto the medium as a whole. The command to do this is CSAVE for a tape file and SAVE for a disk file. The name under which the program is stored is written within quotes immediately following the SAVE or CSAVE command. The names of programs stored on tape are limited to a single character, but the names of programs on disk can be from one to eight characters long, as discussed in the TRSDOS manual. Just remember that the SAVE command *does not* store data

files, just program files. The SAVE command can store a program on disk in ASCII character format by placing a comma and the letter A following the file name.

LOAD, CLOAD

The reverse of the program saving process takes programs stored on disk or tape and copies them into the computer's memory. The commands that effect this transfer are LOAD for disk programs and CLOAD for cassette tape programs. The file name must be specified for a disk load. If the file name is not specified for a tape load, the computer will load the first program it encounters on the tape.

KILL, RUN

Disk BASIC has three more commands that manage program files. The KILL command is used to delete a file from the disk. It is the same command that TRSDOS uses (see Appendix E for a list of all TRSDOS commands), except that the file name must be enclosed in double quotes. The RUN command can be used to run the program in memory, or in Disk BASIC it can be used to load a specified program, then run it. In Disk BASIC the LOAD command can also copy the named program into memory and run it if the command ends with a comma and the letter R.

Examples:

```
10 LOAD "PROG1"          Disk BASIC command within a program.
                         After loading the program, the computer
                         returns to the command mode (>READY).

20 RUN "PROG4"           Load the named program from disk and
                         begin its execution.

30 LOAD "PROG4",R        This effect is identical to line 20.

LOAD "PROG2"             Load the program from command mode.

RUN "PROG3"              Load the program, then run it.

LOAD "PROG3",R           This command is identical to RUN
                         "PROG3".

SAVE "PROG5"             Save the program in memory to disk
                         under the name "PROG5".

KILL "PROG6"             Delete the file called "PROG6" from
                         the diskette.

CSAVE "B"                Save the program in memory to cassette
                         under the name "B".
```

```
CLOAD "X"          Load the program "X" from cassette to
                   memory.

CLOAD              Load the next program on the cassette to
                   memory.

RUN                Execute the program in memory.

SAVE "PROG6",A     Save the program in memory onto disk
                   in ASCII (character) format, one record
                   per line.
```

MERGE

This command combines a program in memory with a program on disk. The program on disk must have been SAVEd in ASCII format with a SAVE "filename",A command. The result of the combination is left in memory. The MERGE command meshes together, or merges, the two programs according to their respective line numbers. If the disk program has a line number that is not present in the memory version, that line is added to the memory version. If the line number is present in both versions, the line from the disk version replaces the line in memory.

Examples:

Assume the disk file's name is "TESTSUB", and the command is MERGE "TESTSUB"

Memory	Disk	Result in memory
10A	15B	10A
20A	25B	15B
30A		20A
		25B
		30A

Memory	Disk	Result in memory
10A	30B	10A
20A	40B	20A
		30B
		40B

Memory	Disk	Result in memory
10A	20B	10A
20A	30B	20B
30A	40B	30B
		40B

The MERGE command is nice for adding commonly used routines to various programs. For example, suppose you have a good sorting subroutine that is numbered from 1000-1999. You can

MERGE it into a number of programs without having to reenter it line by line in each program.

Commands for Data Files

Data files are entirely different from program files. For this reason very few commands work for both kinds of files. The KILL command is one of the few exceptions that doesn't discriminate. It can delete both program files and data files.

A data file is a collection of records. Each record is a related set of data. For example, you could have a data file of rainfall for a given year. There might be 12 monthly rainfall records, each one containing between 28 and 31 data values for the daily rainfall. Or maybe there would be 52 weekly records, each with 7 daily data values. You could even arrange the data to be hourly (if needed) so that each of the 365 records contains 24 values. It is the programmer's decision to arrange the data and records. The commands that manipulate data files are of two types: Either they deal with the entire file as a unit, or they manipulate records one at a time. You have already seen the KILL command, which deals with the entire file.

OPEN, CLOSE

The records on a disk data file are inaccessible unless the file is opened for use. Also, some records on a file may be lost if that file is not closed when processing is complete.

The OPEN command has two primary functions:

(1) It *identifies the mode* of the file. That is, the OPEN statement prepares the file for (a) *input*, in which records can be transferred sequentially from file to memory; (b) *output*, in which records can be transferred sequentially from memory to file; and (c) *random*, in which the transfer can occur in either direction.

(2) It *associates the name* of a file with a *file number*, so that commands using the file need only refer to its number once the file has been opened.

The OPEN command has the format:

```
OPEN m$,n,filename$
```

where m$ is either "I" for input mode, "O" for output mode, or "R" for random access mode. The next chapter, which discusses direct access files, will cover the case where m$ is "R". n is a numeric expression from 1 to 15, which is the number that the program uses to refer to the file. filename$ is a standard file name.

Once the records on a file have been read, written,or modified, the file must be closed. The command

```
CLOSE n
```

closes the file numbered n. The CLOSE command can have more

than one argument separated by commas, or it can be just the word CLOSE. In the latter case, all opened files are closed.

When a program deals with tape files, OPEN and CLOSE commands are not used.

Examples:

```
10 OPEN "I",1,"INVEN/DAT"      The file called INVEN/DAT is
                                opened for input.  It is file #1.

20 CLOSE 2                     The file numbered 2 is opened for
30 OPEN "O",2,"MASTER"         output.  If the old file #2 (the
                                one that was closed) was called
                                MASTER, it is no longer accessible
                                for input.

40 CLOSE
50 CLOSE 1,2,3
60 CLOSE: OPEN "I",15,"ACCRCVB/DAT"
70 INPUT "FILE NAME";F$: OPEN "I",2,F$
80 INPUT "MODE";M$: OPEN M$,5,"MASTER2"
```

Sequential File INPUT and PRINT

Sequential files, whether they are on cassette tape or floppy disk, are accessible with only two commands that actually read or write records once the file is opened. They are the INPUT # n. If n is -1, the cassette drive is accessed, otherwise n must be between 1 and 15 inclusive, and it corresponds to the file number that was assigned with the OPEN statement.

Two variations to the INPUT and PRINT are available for record input and output. These are the LINE INPUT # and the PRINT #, USING. The formats of the two statements are:

INPUT # n, v1, v2, . . .

LINE INPUT # n, v1, v2, . . .

PRINT # n, v1, v2, . . .

PRINT # n, USING f$; v1, v2, . . .

With the LINE INPUT # statement, there can be only one string variable name following the command.

Examples:

```
50 INPUT #-1,A,B$,C
60 PRINT #-1,A;B$;C;X
70 PRINT #4,X$;",";M$;",";J
80 LINE INPUT #1,V$
90 PRINT #3, USING "### ####.#";X,Y
```

Command	File medium	File transfer	Variables
INPUT # n, . . .	disk	file to memory	string and numeric
PRINT # n, . . .	disk	memory to file	string and numeric
INPUT # -1, . . .	cassette	file to memory	string and numeric
PRINT # -1, . . .	cassette	memory to file	string and numeric
LINE INPUT # n	disk	file to memory	one string only
PRINT # n, USING . . .	disk	memory to file	any, formatted

Table 7.1 Sequential File Commands

The following program illustrates first the output of numeric variables to cassette tape, then the input from that same tape.

```
 10 'FILENAME: "C7P1"
20 'FUNCTION: STORE A NUMERIC MATRIX ON TAPE
30 ' AUTHOR : JPG                    DATE:   8/79
40 DIM X(30,5) ' define matrix as 30 elements by 5 elements
50 'fill matrix with random values from 1001 to 9999
60 FOR I=1 TO 30: FOR J=1 TO 5
70 X(I,J)=RND(8999)+1000: PRINT X(I,J);: NEXT J,I
80 '  now store the matrix on tape
90 PRINT "Rewind the tape and set past the leader,"
100 INPUT "then press /EN/";A$
110 FOR I=1 TO 30
120    PRINT #-1,X(I,1);X(I,2);X(I,3);X(I,4);X(I,5)
130 NEXT I
140 INPUT "Rewind again, press play, then press /EN/";A$
150 FOR I = 1 TO 30: INPUT #-1, A,B,C,D,E
160    PRINT "REC #"; I,"CONTAINS";A;B;C;D;E
170 NEXT I
10000 END
```

The next program represents a disk-based game that uses a sequential file called "GEOG/DAT". The point of the game is to challenge the computer in a dialog to see if the computer's "memory" contains the name and description of a geographical location that the player knows.

The program uses a data structure that is known as a binary tree. Each record has two elements of string information, the name of the geographical location, and the characteristic that distinguishes it from the previously reached location. Each record also has two pieces of numeric information, YES and NO links to further locations. Thus each record has four fields.

Variable name	Field length	Type	Description
L1	4	Numeric	YES links, 0 to 999
L2	4	Numeric	NO links, 0 to 999
A1$	Not limited	String	Location name
Q$	240-len(A$)	String	Distinguishing characteristic

Table 7.2 GEOGRAPH Record Layout

The program "learns" new geographical locations and their characteristics from the player. It starts the play with the question, "ARE YOU THINKING OF A GEOGRAPHICAL LOCATION?" There are five possible responses:

1. A YES response elicits a very specific location, for example, "IS IT LAKE ERIE?"
2. A NO response is refused, and the computer displays the message, "YES, LIST, SAVE, OR DEBUG". Later on in the game, a NO response elicits a very general characteristic, for example, "IS IT A LAKE?".
3. DEBUG displays a table of links, locations, and characteristics. It is used only for debugging purposes.
4. LIST displays all locations that the computer "knows".
5. SAVE transfers all records, both those that were initially loaded into memory from the file, and those that the computer "learned" during the play.

Consider the sample dialog below:

```
ARE YOU THINKING OF A GEOGRAPHICAL LOCATION? YES
IS IT A LAKE? NO
IS IT A MOUNTAIN? YES
IS IT MT. POPOCATAPETL? NO
IS IT IN AFRICA? YES
IS IT MT. KILIMANJARO? YES
.
.
.
I THOUGHT SO.
ARE YOU THINKING OF A GEOGRAPHICAL LOCATION? YES
IS IT A LAKE? YES
IS IT LAKE ERIE? NO
IS IT IN EUROPE? NO
WHAT WAS THE PLACE YOU WERE THINKING OF? LAKE TITICACA
TYPE A CHARACTERISTIC WHICH WOULD DISTINGUISH LAKE TITICACA
FROM LAKE GENEVA
? IS IT IN SOUTH AMERICA
ARE YOU THINKING OF A GEOGRAPHICAL LOCATION?
```

Table 7.3 shows the contents of a file record by record after one typical session of this game.

Record #	A1$	Q$	L1	L2
1	LAKE ERIE	IS IT A LAKE	4	2
2	MT. SHASTA	IS IT A MOUNTAIN	6	3
3	CINCINNATI	WHAT ABOUT A CITY	7	999
4	LAKE TITICACA	IS IT IN SOUTH AMERICA	4	5
5	LAKE GENEVA	IS IT IN EUROPE	5	9
6	MT. EVEREST	IS IT IN ASIA	8	999
7	WASHINGTON	IS IT A CAPITOL	7	999
8	MT. FUJI	IS IT VOLCANIC	8	999
9	LAKE MEAD	IS IT DAMMED	9	999
10	END	END	0	0

Table 7.3 Example Record for GEOGRAPH

This table can be represented as a *binary tree structure*, which indicates how the two links L1 and L2 tie the array together.

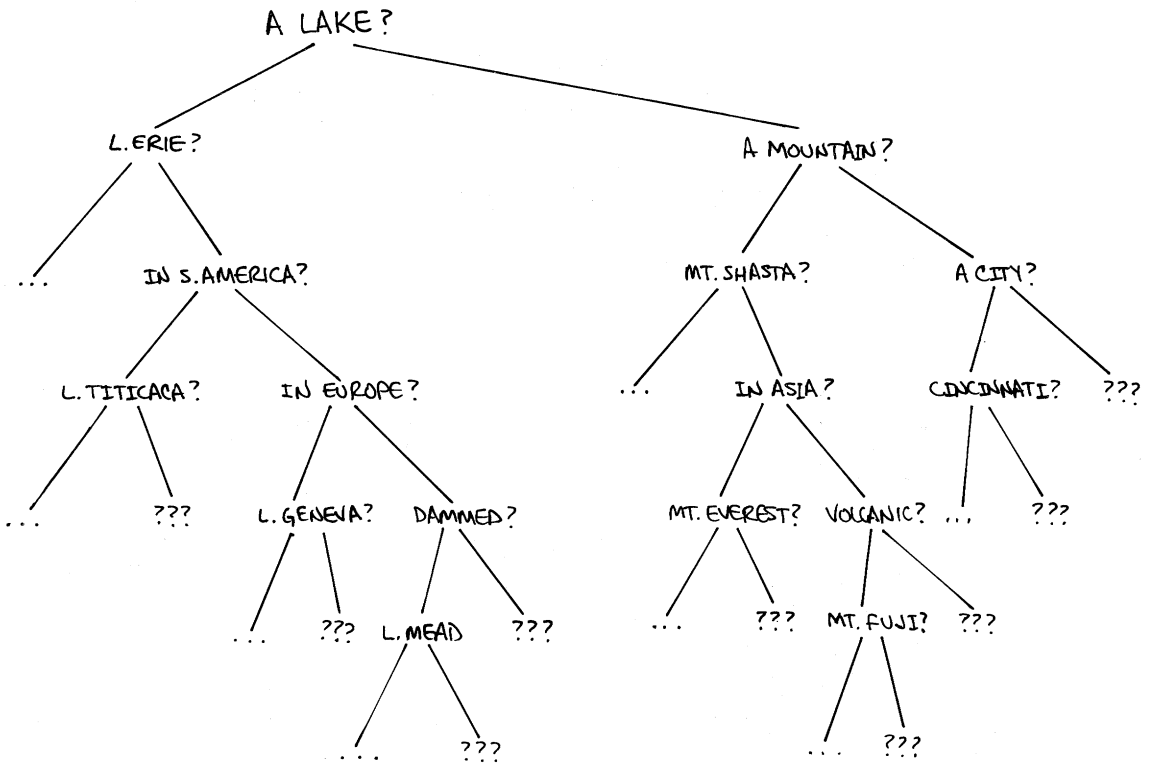


Figure 7.1 GEOGRAPH Binary Tree Structure

```
10 'FILENAME: "C7P2"
20 'FUNCTION: GEOGRAPHY QUIZ GAME
30 ' AUTHOR : JPG                    DATE:   6/79
40 'Q$=characteristic, A1$=location name, A$=temporary strings
50 'L1=left link,      L2 =right link,   N =number of places
60 DEFINT I-N: CLEAR 2000: DIM Q$(50),A1$(50),L1(50),L2(50)
70 '       set up first record if needed
80 INPUT "DOES THE DATA FILE EXIST";A$
90 IF A$<>"YES" THEN Q$(1)="IS IT A LAKE": L1(1)=1:L2(1)=999:
                     N=1: A1$(1)="LAKE ERIE": GOTO 180
100 '            set all filed sites
110 CLOSE 1: OPEN "I",1,"GEOG/DAT": I=0
120 I=I+1: INPUT #1, Q$(I),L1(I),A1$(I),L2(I): PRINT I,\
130 IF Q$(I)<>"END" THEN 120
140 N=I-1: INPUT "/CR/";A$
150   IF N>50 THEN PRINT "C A R E F U L !  MORE THAN 50!"
160 '
170 '             next place
180 CLS: INPUT "ARE YOU THINKING OF A LOCATION";A$
190   IF A$="LIST" THEN K=1
200   IF A$="YES"  THEN K=2
210   IF A$="DEBUG"THEN K=3
220   IF A$="SAVE" THEN K=4
```

```
230 ON K GOSUB 270,300,600,630
240 IF K>0 AND K<5 THEN 180
250 INPUT "YES, LIST, SAVE, OR DEBUG  /CR/";A$: GOTO 180
260 '           "LIST" ROUTINE
270 PRINT "THE PLACES I KNOW ARE ..."
280 FOR I=1 TO N: PRINT A1$(I);: NEXT I
290 INPUT "   /CR/";A$: RETURN
300 '          "YES", the player is thinking of a place
310 I=1
320 PRINT Q$(I);: INPUT A$
330 IF A$="YES" THEN 410
340 IF A$<>"NO" THEN 320
350 ' NO, not the listed characteristic so if right link
360 ' is null, use it -- otherwise get new site's details
370   IF L2(I)<>999 THEN I=L2(I): GOTO 320
                     ELSE GOSUB 540
380 '         set up all links and get a new location
390 L2(I)=N+1: GOSUB 580: RETURN
400 '       ask if this is the correct characteristic
410 PRINT "IS IT ";A1$(I);: INPUT A$
420 IF A$<>"YES" THEN 460
430 '   yes, the computer guessed it (print the message)
440 PRINT: PRINT: PRINT "... I THOUGHT SO."
450 FOR X=1 TO 400: NEXT X: RETURN
460 IF A$<>"NO" THEN 410
470 '         set next location if left link isn't null
480   IF I<>L1(I) THEN I=L1(I): GOTO 320
490 '           set the new location's details
500 GOSUB 540
510 '       set up all new links and get next location
520 L1(I)=N+1: GOSUB 580: RETURN
530 '            DIALOG
540 INPUT "WHAT WAS THE PLACE YOU WERE THINKING OF";A$
550 PRINT "TYPE A CHARACTERISTIC that WOULD DISTINGUISH "
560 PRINT A$;" FROM ";A1$(I): INPUT Q1$: RETURN
570 '           new location links
580 N=N+1:Q$(N)=Q1$:A1$(N)=A$:L1(N)=N:L2(N)=999:RETURN
590 '        "DEBUG" routine -- displays all links, data
600 PRINT"I   Q$(I)   A1$(I)   L1(I)   L2(I)"
610 FOR I=1 TO N:PRINT I;Q$(I),A1$(I),L1(I);L2(I):NEXT I
620 INPUT "/CR/";A$: RETURN
630 '       "SAVE" routine
640 PRINT "REC.#";: CLOSE 1: OPEN "O",1,"GEOG/DAT"
650 FOR I=1 TO N
660 PRINT #1,Q$(I);",";L1(I);",";A1$(I);",";L2(I)
670 PRINT I;: NEXT I
680 PRINT #1,"END,";0;"END,";0
690 INPUT "/CR/";A$: RETURN
10000 END
```

The following program demonstrates two software techniques. One of these, the Shell-Metzner sort, is also used in a later chapter. The other technique, that of merging two sequential files, is the primary reason for including the program here.

The program has three parts. Part 1 creates a number of files, that number determined by the user, and all files are of the same size, again determined by the user. Part 2 sorts each of the files in memory, then rewrites the files in sorted order. Part 3 merges all the files into one which is in sorted order.

Since the program's purpose is to demonstrate a technique rather than to produce output for some more useful reason, the output that follows the listing should be considered in that light. The first part of the output shows three unsorted files. The second part shows those three files, each in sorted order. The last part shows the single file that results from the merge operation. Note that this sorted output could not have been produced nearly as quickly had the sort been conducted any other way.

```
10 'FILENAME: "C7P3"
20 'FUNCTION: SORT-MERGE OF SEQUENTIAL FILE
30 ' AUTHOR : JPG                    DATE:   11/79
40 CLEAR 2000: DEFINT A-Z: CLS: DIM N$(50)
50 J$="### %                      % ": J$=J$+J$+J$
60 INPUT "How many files";NF
70 INPUT "How many entries will be in each file";SIZE
80 INPUT "File group name";F$
90 PRINT "0=Shuffle  1=Create  2=Edit  3=Sort"
95 INPUT "4=Merge    5=Stop"; Y
100 ON Y+1 GOSUB 110,170,230,320,480: GOTO 90
110 '    shuffle the entries in all of the files
120 FOR W=1 TO NF: FL$=F$+RIGHT$(STR$(W),1)
130 K=1: GOSUB 630
140 FOR J=1 TO SIZE: A=RND(SIZE): B=RND(SIZE)
150 T$=N$(A): N$(A)=N$(B): N$(B)=T$: NEXT J
155 LPRINT "File "; FL$; " in original order."
160 GOSUB 670: N=SIZE:  GOSUB 720: NEXT W: RETURN
170 '      create the files
180 FOR W=1 TO NF: FL$=F$+RIGHT$(STR$(W),1)
190 FOR J=1 TO SIZE: PRINT "FILE #";W;", ENTRY #";J;
200 INPUT N$(J): NEXT J: PRINT
210 K=1: GOSUB 670     ' write out names on file
220 NEXT W: RETURN
230 '      edit entries within a file
240 FOR W=1 TO NF: FL$=F$+RIGHT$(STR$(W),1)
250   PRINT "INPUT FILE IS ";FL$: K=1: GOSUB 630' read
260   FOR J=1 TO SIZE
270       PRINT N$(J), TAB(30),":";
280       INPUT NE$: IF LEN(NE$)<>0 THEN N$(J)=NE$
290   NEXT J
300 GOSUB 670     '  write out edited file
310 NEXT W: RETURN
```

```
320 '  single file sort
330 CLS: FOR W=1 TO NF: FL$=F$+RIGHT$(STR$(W),1)
340 K=1: GOSUB 630  '   read the file
370 M=SIZE
380 M=INT(M/2): PRINT M;: IF M=0 THEN PRINT: GOTO 440
390 K=SIZE-M: J=1
400 I=J
410 L=I+M: IF N$(I)<=N$(L) THEN 430
420 T$=N$(I): N$(I)=N$(L): N$(L)=T$: I=I-M
425   IF I>=1 THEN 410
430 J=J+1: IF J<=K THEN 400 ELSE 380
440 LPRINT "File "; FL$; " in sorted order."
450 N=SIZE: GOSUB 720 '   print the sorted file
460 K=1: GOSUB 680 '  write out sorted file onto disk
470 PRINT: PRINT: PRINT: PRINT: NEXT W: RETURN
480  '      merge the files
490 PRINT "Requires";NF+1;"files    /EN/": LINE INPUT A$
500 LPRINT "Merged file 'NEW'.": OPEN "O",NF+1,"NEW" '  open file
510 FOR W=1 TO NF: OPEN "I",W,F$+RIGHT$(STR$(W),1)
520 INPUT #W, N$(W): K(W)=1: NEXT W
530 CN=0: S=0: PLUG$="ZZZZZZ": GOSUB 590
540 PRINT #NF+1,SMALL$;",";:CN=CN+1:PRINTCN;: K(M)=K(M)+1
550 IF K(M)<=SIZE INPUT #M,N$(M):GOSUB 590:GOTO 540
560 N$(M)=PLUG$:S=S+1:IF S<NF THEN GOSUB 590:GOTO 540
570 PRINT:CLOSE:SIZE=SIZE*NF:K=NF+1:FL$="NEW":GOSUB 630
580 N=SIZE: GOSUB 720: SIZE=SIZE/NF: RETURN
590 '  find the smallest of NF strings
600 SMALL$=PLUG$
610 FOR W=1 TO NF: IF N$(W)<SMALL$ THEN SMALL$=N$(W):M=W
620 NEXT W: RETURN
630 '    load a data file from diskette
640 OPEN "I",K,FL$
650 FOR J=1 TO SIZE: INPUT #K,N$(J): NEXT J
660 CLOSE: RETURN
670 '     save a data file on diskette
680 OPEN "O",K,FL$
690 FOR J=1 TO SIZE
700     PRINT #K, N$(J);",";
710 NEXT J: PRINT: CLOSE: RETURN
720 '      print a file from memory
730 N1=INT(N/3): FOR J=1 TO N1
740   LPRINT USING J$;J,N$(J),J+N1,N$(J+N1),J+2*N1,N$(J+2*N1)
750 NEXT J: LPRINT: RETURN
10000 END
```

```
File test1 in orisinal order.
  1 SALIVA TELLY          5 GNASHGNAT NATHANIE    9 MINDERBINDER MILO
  2 FISTULA FELICITY      6 PONTIAC CARLO        10 ANOMALY ANTHONY
  3 BELLER ELLERY         7 SITHLE SIBILANT      11 LAVORIS MORRIS
  4 HEGIRA IRA            8 PARSLEY PELVIS       12 UVULA URSULA

File test2 in orisinal order.
  1 LANCASTER BART        5 GROMMET ANDROMEDA     9 SIDDHARTHA GHAUTAM
  2 SITHLE SIMEON         6 CUERVO CERVESA       10 ALIQUOT ALICE
  3 PARVENU MARVIN        7 MONACO MONICA        11 VERMIFORM VERNON
  4 ASININE ARNOLD        8 POWER CYCLONE        12 LOOSELIPS LINDA

File test3 in orisinal order.
  1 GABLE MARK            5 LANCHESTER ELSIE      9 RUBELLA ELLA
  2 SITHLE SAMANTHA       6 PATELLA PETER        10 HARSHBARGER HERSCH
  3 ENEMA ANOMIE          7 MERGER MARGINAL      11 BANDERSNATCH FRUMI
  4 MUSHMOUTH MUNGO       8 SITHLE SYBIL         12 ANOMALY ANNABELLE

File test1 in sorted order.
  1 ANOMALY ANTHONY       5 HEGIRA IRA            9 PONTIAC CARLO
  2 BELLER ELLERY         6 LAVORIS MORRIS       10 SALIVA TELLY
  3 FISTULA FELICITY      7 MINDERBINDER MILO    11 SITHLE SIBILANT
  4 GNASHGNAT NATHANIE    8 PARSLEY PELVIS       12 UVULA URSULA

File test2 in sorted order.
  1 ALIQUOT ALICE         5 LANCASTER BART        9 POWER CYCLONE
  2 ASININE ARNOLD        6 LOOSELIPS LINDA      10 SIDDHARTHA GHAUTAM
  3 CUERVO CERVESA        7 MONACO MONICA        11 SITHLE SIMEON
  4 GROMMET ANDROMEDA     8 PARVENU MARVIN       12 VERMIFORM VERNON

File test3 in sorted order.
  1 ANOMALY ANNABELLE     5 HARSHBARGER HERSCH    9 PATELLA PETER
  2 BANDERSNATCH FRUMI    6 LANCHESTER ELSIE     10 RUBELLA ELLA
  3 ENEMA ANOMIE          7 MERGER MARGINAL      11 SITHLE SAMANTHA
  4 GABLE MARK            8 MUSHMOUTH MUNGO      12 SITHLE SYBIL

Mersed file 'NEW'.
  1 ALIQUOT ALICE        13 HARSHBARGER HERSCH   25 PATELLA PETER
  2 ANOMALY ANNABELLE    14 HEGIRA IRA           26 PONTIAC CARLO
  3 ANOMALY ANTHONY      15 LANCASTER BART       27 POWER CYCLONE
  4 ASININE ARNOLD       16 LANCHESTER ELSIE     28 RUBELLA ELLA
  5 BANDERSNATCH FRUMI   17 LAVORIS MORRIS       29 SALIVA TELLY
  6 BELLER ELLERY        18 LOOSELIPS LINDA      30 SIDDHARTHA GHAUTAM
  7 CUERVO CERVESA       19 MERGER MARGINAL      31 SITHLE SAMANTHA
  8 ENEMA ANOMIE         20 MINDERBINDER MILO    32 SITHLE SIBILANT
  9 FISTULA FELICITY     21 MONACO MONICA        33 SITHLE SIMEON
 10 GABLE MARK           22 MUSHMOUTH MUNGO      34 SITHLE SYBIL
 11 GNASHGNAT NATHANIE   23 PARSLEY PELVIS       35 UVULA URSULA
 12 GROMMET ANDROMEDA    24 PARVENU MARVIN       36 VERMIFORM VERNON
```

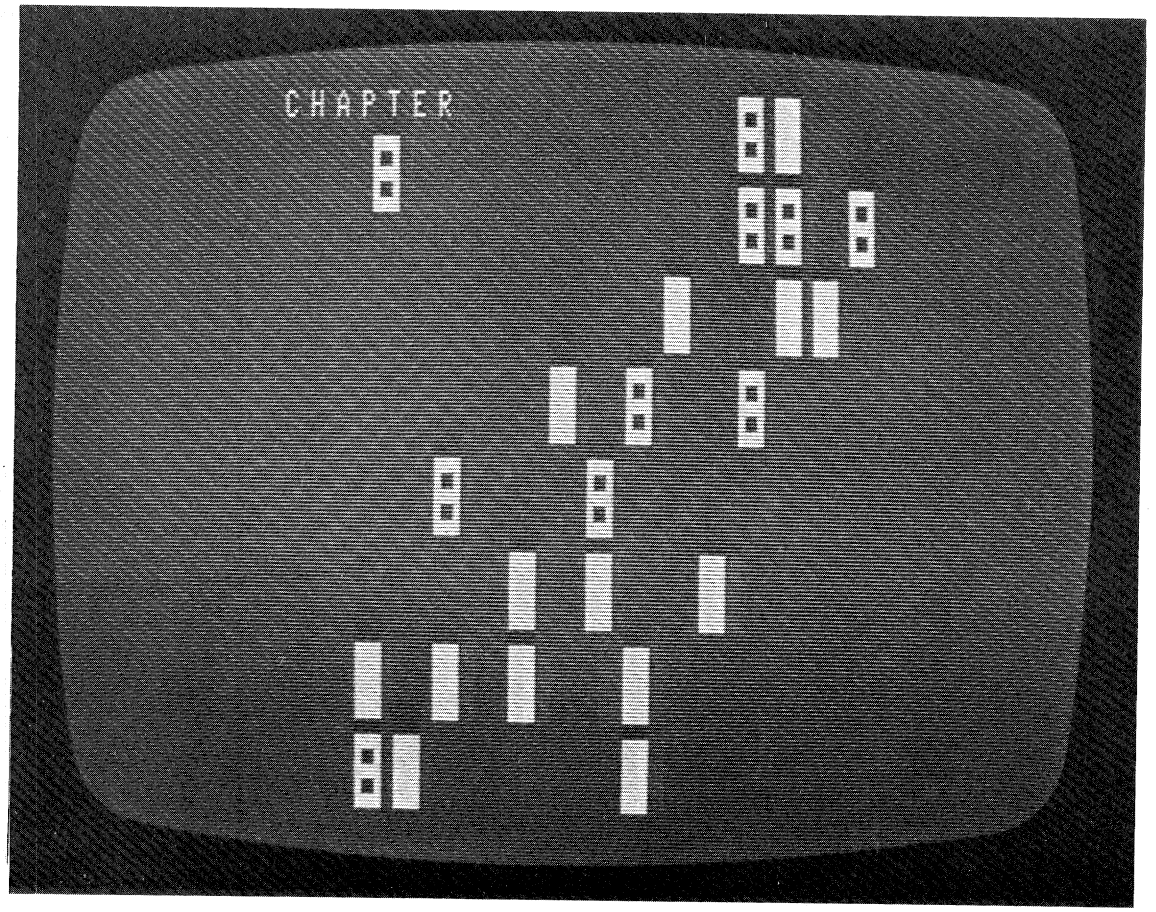
The last program in this chapter illustrates the use of sequential ASCII files in one very common and necessary application, the renumbering of a BASIC program. It contains many advanced BASIC programming features, such as variable dimensioning, input and output mode sequential files, MID$, and INSTR.

```
10 'FILENAME: "C7P4"
20 'FUNCTION: RENUMBER DISK BASIC PROGRAMS FROM BASIC
30 ' AUTHOR : SPG                    DATE:   1/13/79
40 CLEAR 1000: DEFINT A-Z
50 CLS: PRINT: PRINT
60 PRINT "Keep the followins in mind when usins this prosram:"
70 PRINT "   a.  Your prosram must already be saved on"
80 PRINT "       disk in ASCII format."
90 PRINT "   b.  The prosram on file can't contain"
100 PRINT "       the followins type of line:"
110 PRINT "            IF X=Y 100"
120 PRINT "       Change all such lines to:"
130 PRINT "            IF X=Y THEN 100"
140 PRINT "   c.  Any prosram statements within a strins"
150 PRINT "       will be renumbered."
160 PRINT "       (Example:  10 PRINT 'GOTO 240' )"
170 PRINT: PRINT "/EN/";
180 T$=INKEY$: IF T$="" THEN 180
190 CLS: PRINT: PRINT
200 PRINT "    After this prosram is finished renumberins"
210 PRINT "your prosram, the new renumbered version will"
220 PRINT "be under the file name RENUM/DAT"
230 PRINT
240 PRINT "    If at any time, this prosram is stopped before"
250 PRINT "the complete renumberins, your prosram will still"
260 PRINT "be saved under its old file name (this prosram"
270 PRINT "doesn't change your prosram at all; it creates a"
280 PRINT "completely new file)."
290 PRINT: PRINT "/EN/"
300 T$=INKEY$: IF T$="" THEN 300
310 CLS: PRINT: PRINT
320 PRINT "    How many lines are in the prosram that you"
330 INPUT "want renumbered (approximately)";M
340 M=M+20: DIM B1(M), B2(M)
350 INPUT "What should the new line increment be";X
360 INPUT "What should the first line number be";Y: Y=Y-X
370 IF Y<0 THEN PRINT ">>> Error <<<   Try asain": GOTO 350
380 INPUT " What is the name of the prosram";B$
390 OPEN "I",1,B$
```

```
400 ' set up arrays B1 (old line #s) and B2 (new line #s)
410 K=K+1: PRINT K;
420 IF EOF(1) THEN CLOSE: PRINT: GOTO 460
430 LINE INPUT #1,A$
440 B1(K)=VAL(LEFT$(A$,INSTR(A$," ")-1)): B2(K)=K*X+Y: GOTO 410
450 '   open both files
460 OPEN "I",1,B$: OPEN "O",2,"RENUM/DAT"
470 '  main routine immediately below
480 D=D+1
490 POKE 14305,1 '  keep disk drive on during renumbering
500 IF EOF(1) THEN 10000
510 LINE INPUT #1,A$: GOSUB 580 '  read and change a line
520 '    save & print the line, then go back for another
530 PRINT #2,A$: PRINT A$: GOTO 480
540 '* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
550 '*      routine to change the line number of a line      *
560 '*  and change all reference numbers following commands  *
570 '* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
580 H=1: GOSUB 830'  change the line number of this line
590 '  skip the remark lines
600 IF INSTR(A$,"'")>0 AND INSTR(A$,"'")<7 THEN RETURN
610 IF INSTR(A$,"REM")>0 AND INSTR(A$,"REM")<7 THEN RETURN
620 RESTORE '  reset the data pointer
630 'data of the list of commands to check for
640 DATA GOTO,GOSUB,THEN,ELSE,DONE
650 '  get next command
660 '   if done, check for ON...GOTO's and ON...GOSUB's
670 READ N$: H=1: IF N$="DONE" THEN 760
680 H=INSTR(H,A$,N$) ' is the command in the line?
690 IF H=0 THEN 670'if not, then go back to get next command
700 '         N$ was found in the line!
710 H=H+LEN(N$)'H is where the next possible line number is
720 IF VAL(MID$(A$,H))=0 THEN 680 ' not a number after N$
730 GOSUB 830: GOTO 680'  change the number after N$
740 '   subroutine to change the numbers in all
750 '   "ON...GOTO...'s  and   ON...GOSUB...'s
760 H=INSTR(A$,"ON")
770 IF H=0 THEN RETURN '   ON not found, so go back
780 '   if no commas, then changes already done, return
790 H=INSTR(H,A$,",")+1: IF H=1 THEN RETURN
800 ' change the line number after the comma, and check more
810 GOSUB 830: GOTO 790
820 '   routine to change first line number after H in A$
830 FOR R=H TO LEN(A$): IF MID$(A$,R,1)=" " THEN NEXT R
840 H1=R
850 FOR R=H1 TO LEN(A$): T1=ASC(MID$(A$,R))
860 IF T1>47 AND T1<58 THEN NEXT R
870 V=VAL(MID$(A$,H1,R-H1)) ' V is the line number
```

```
880 '   check to see if V is a valid line number
890 FOR T=1 TO M
900     IF V=B1(T) THEN 940
910 NEXT T
920 GOSUB 950: RETURN '  not a valid line #
930 ' yes, a good #,  change the number and return
940 A$=LEFT$(A$,H1-1)+MID$(STR$(B2(T)),2)+MID$(A$,R): RETURN
950 '    error trapping subroutine, not found line number
960 PRINT: PRINT: PRINT
970 PRINT "   I'm sorry, but I can't find"
980 PRINT "line #";V;"anywhere in the program.
990 PRINT
1000 PRINT "It is referenced in this line:"
1010 PRINT A$: PRINT
1020 PRINT "These are you choices at this point:"
1030 PRINT "    1.  Continue, leaving this line as it is   (or)
1040 PRINT "    2.  Stop execution of this renumbering program"
1050 PRINT
1060 INPUT " Which do you want me to do";J
1070 IF J=0 THEN RETURN
1080 PRINT "O.K.,   your program is unchanged under the old"
1090 PRINT "file name of ";B$;"."
1100 PRINT
1110 PRINT "Press /EN/ and I will delete the other unneeded"
1120 INPUT "file";T$
1130 KILL "RENUM/DAT"
10000 END
```

Sequential files are useful in applications that require the processing of most or all of the records on the file. Also, sequential files must be processed one record at a time in their physical order, and that severely limits the use of this method of file management. In the next chapter, you will discover the advantages that direct access file processing can offer.



Direct Access File Processing

A computer without the ability to manage files is just a glorified calculator. Can you imagine the tedium of retyping the same long program every time you want to run it? Programming and files go together, because the importance of saving them, then having the computer be able to load them directly into memory, increases as the programs get larger and more sophisticated. But sequential files are at times rather cumbersome. Their structure dictates that on the average half of the file must be read to find a desired record, even if its position in the file relative to all of the others is known.

Direct access files allow the programmer to read from or write to any one of the records on the file without any intermediate read or write operations. This is the reason for the access method having the name "direct access". If one specific record is desired and its position is known, you can issue a command in BASIC that fetches the record without any unnecessary inputs of other records. The savings in time alone make direct access file processing a tempting method of data management. This chapter will review the programming techniques used for input and output of data on a direct access storage device, specifically the TRS-80 mini-floppy disk.

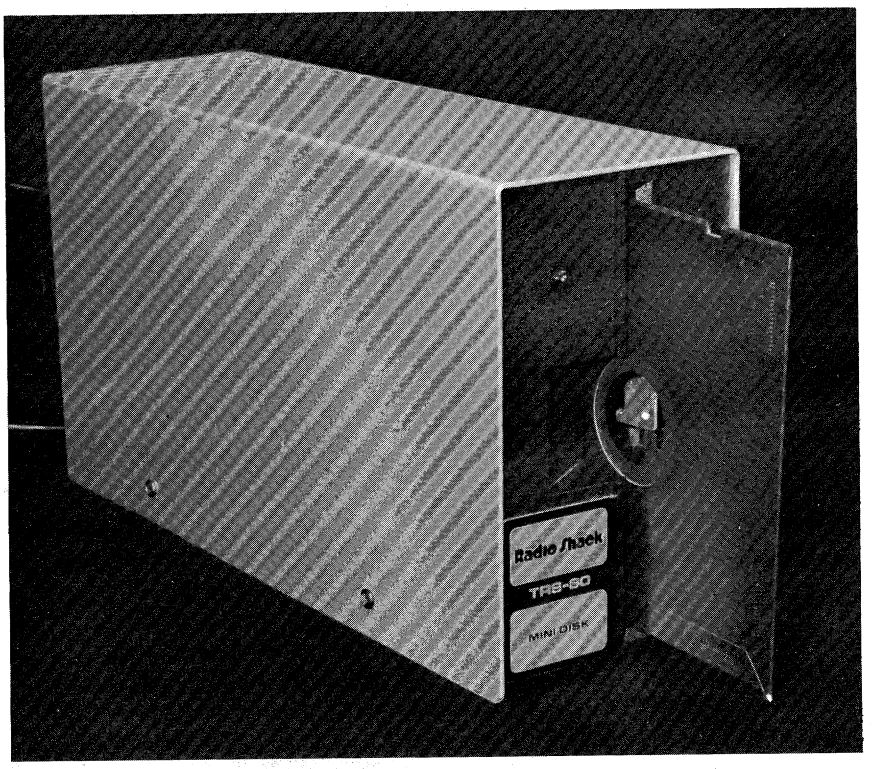


Figure 8.1 TRS-80 Disk Drive

FIELD

A programmer using sequential access techniques on a disk file is not concerned with the length of the record in a PRINT # statement. Put another way, the *logical record* (the set of related data items in the PRINT #) can vary in length; when the system calculates that over 256 bytes of records are ready to be stored, it writes out a *physical record* (one complete sector of bytes) to the disk.

When a program uses direct access (also called random access) files, the output statement PUT does the actual transfer of information from memory to the disk. Contrary to the system doing the transfer as in sequential processing, the programmer is responsible for this operation. The record that is PUT onto the disk is again a 256-byte physical record. However, the logical record size is also determined by the programmer and it can be as large as 255 bytes.

The purpose of the FIELD statement is to define the layout of the 255-byte record to TRSDOS, the operating system. Not all 255 bytes need to be defined, but in every case the FIELD statement starts in the first byte of the 255-byte physical record.

With direct access file processing, the program prepares a special area of memory called the *sector buffer*, or just *buffer*, 255 bytes at a time. Data transfer to the disk is a three step affair: First, the buffer is described with the FIELD statement, then the variables in memory are placed in this buffer by using the LSET and RSET statements, and

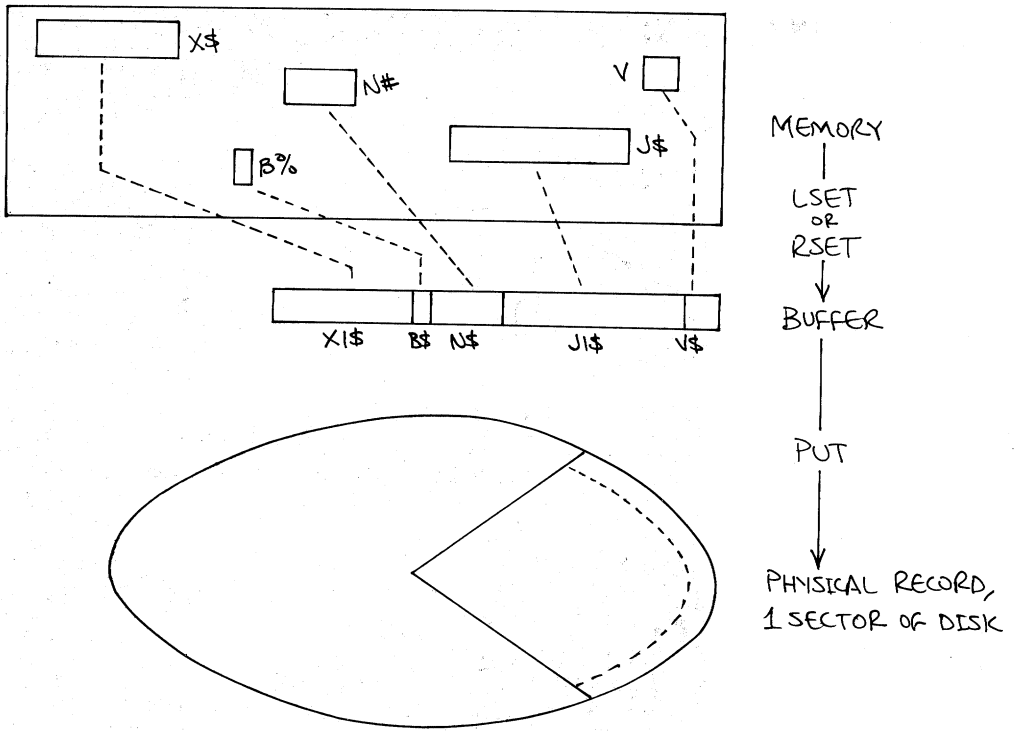


Figure 8.2 Schematic of Data Transfer to Disk

last, the entire buffer is copied onto the disk with the PUT command. Figure 8.2 illustrates this three-step transfer.

Note that the variable names in memory and the buffer are different in figure 8.2. This is because they are physically located in two different places.

The format of the FIELD statement is:

FIELD fn, len1 AS var1, len2 AS var2, . . .

The FIELD describes the buffer in all of these ways:

1. (fn) File number or specifier, as defined in the OPEN statement, from 1 to 15. fn is any numeric expression, such as 1, 2, . . . , 15, A, X, I*2, . . .
2. (len1) Length of the first field, in bytes.
3. (var1) Name of the first field, any legal Level II BASIC variable name.
4. (len2, var2, len3, var3, . . .) Lengths and names of all subsequent fields in the 255-byte buffer. The sum of all lengths (len1 + len2 + . . .) cannot exceed 255, but it can be less.

Examples:

```
500 FIELD 1, 255 AS N$
```

The entire buffer of File #1 is just one field, a string 255 bytes long. Suppose the program has OPENed file #1, then the FIELD statement above is executed, then a specific record is read into the buffer with a GET (to be discussed later). Then the program has available for inspection and processing the 255-character string N$.

```
500 FIELD 2, 40 AS X$, 8 AS D$, 100 AS J$
```

The buffer for File #2 is partially used, with only 148 bytes of the 255 bytes being fielded as three variables X$, D$, and J$. D$ might be an 8-byte double precision numeric variable stored as a string to conserve space. See the discussion on the conversion functions in this chapter. Note that *all* data fields that are described in the form of strings: Each integer value is fielded as a 2-byte string, each single precision value as a 4-byte string, and each double precision value as an 8-byte string, compared to sequential files, in which the value 1.23456789012345 takes up 18 bytes, including the leading and trailing blanks.

```
500 FIELD K,                  32 AS N$(1), 20 AS S$(1), 12 AS C$(1)
510 FIELD K, 64 AS B2$,       32 AS N$(2), 20 AS S$(2), 12 AS C$(2)
520 FIELD K, 128 AS B3$,      32 AS N$(3), 20 AS S$(3), 12 AS C$(3)
530 FIELD K, 192 AS B4$,      32 AS N$(4), 20 AS S$(4), 11 AS C$(4)
```

This series of statements describes a single 255-byte buffer, but each statement is responsible for one fourth of the record. Since 255 is one less than 64*4, the last FIELD statement cannot describe 192+64, but only 192+63. This layout is shown in figure 8.3.

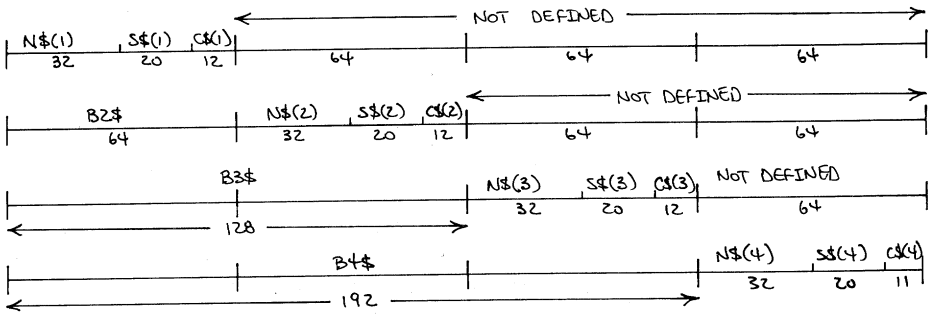


Figure 8.3 255-Byte Buffer Description by Four FIELD Statements

The fields B2$, B3$, and B4$ are dummy fields that will not be used. Their purpose is to reposition the location of the first fields N$(2), N$(3), and N$(4) further down the buffer. The four statements could also have been written as a single statement:

```
500 FIELD K, 32 AS N$(1), 20 AS S$(1), 12 AS C$(1),
             32 AS N$(2), 20 AS S$(2), 12 AS C$(2),
             32 AS N$(3), 20 AS S$(3), 12 AS C$(3),
             32 AS N$(4), 20 AS S$(4), 11 AS C$(4)
```

The dummy field must be used to describe some buffers whose item-by-item descriptions would exceed the 256 byte limit on the length of BASIC statements. For example, suppose you wish to store 125 two-byte integers on each record. You have to field the array so that it would look like this:

X$(1)	X$(2)	X$(3)	X$(4)	.....	X$(125)	(unused)
2	2	2	2		2	5

You cannot write this as a single statement:

```
500 FIELD 1, 2 AS X$(1), 2 AS X$(2), ...
```

because that statement would be more than 1000 characters long without any blanks! But this little program segment can do it:

```
100 FOR I=1 TO 125
110 FIELD 1, (I-1)*2 AS D$, 2 AS X$(I)
120 NEXT I
```

This program segment describes the 125 different fields for 2-byte integers in the buffer. The variable D$ is a dummy variable, whose sole purpose is to displace the location of the FIELDed integer two bytes further down the record.

GET

The GET statement transfers the information from a record on disk to the buffer in memory. It uses just two arguments, and one is optional. The form of the GET statement is:

```
GET fn
```
or
```
GET fn, rn
```

The fn is a numeric expression that corresponds to a value from 1 to 15, indicating the file number. The rn is an optional numeric expression that corresponds to the record number, from 1 to the last 255-byte physical record number. If the record number is not specified in the GET, the next record on the file is read and its

contents are transferred to the buffer.

Examples:

```
100 GET 1, 2
```

Copy the contents of the second record on File #1 into the corresponding buffer.

```
100 GET 2, J
```

Copy the Jth record to the buffer for File #2.

```
100 GET K, L*3
```

The record number is L*3 and its contents are copied to the buffer of File #K.

```
100 GET 3, LOF(3)
```

Get the last on file (LOF) record from File #3. See the discussion following.

LOF

The LOF function returns the number of the last physical record of the file specified by its argument.

Examples:

```
20 IF X<LOF(1) THEN GET 1,X
```

If the value of X has not reached the file's limit, read the Xth record.

```
50 PRINT "FILE";J;"HAS";LOF(J);"RECORDS"
```

LSET and RSET

The variables that are specified in a FIELD statement are in the sector buffer, which is a location different from the other variables in the program. They are not alterable in the same way that ordinary string variables in memory can be modified by BASIC. For example, in the sequence of statements below, the record that is written back onto the disk is completely unaltered, because line 130 affected some strings in ordinary memory whose names were the same as those in the buffer.

```
100 OPEN "R",1,"TEST"
110 FIELD 1, 10 AS X$, 20 AS Y$, 30 AS Z$
120 GET 1,1
130 S$=X$+"ABC": Y$=Y$+"DEF": Z$=Z$+"GHI"
140 PUT 1,1
```

To transfer information to the buffer from memory you must use one of two instructions, the LSET or the RSET. These commands transfer strings to the fielded buffer. LSET left justifies a string in memory into a fielded string variable, and RSET right justifies a string in memory into a fielded string variable.

Examples:

```
10 FIELD 1, 5 AS A$, 6 AS B$, 7 AS C$
20 LSET A$="ABC"
30 J$="XYZ": LSET B$=J$
40 LSET C$="ABCDEFGHIJ"
```

The result is:

ABC..XYZ...ABCDEFG
A$ B$ C$

where the periods represent blanks.

Note that all strings are left justified in their fields, and that in the case where C$ was fielded, the string being placed in the buffer was truncated to the right.

PUT

The output statement for direct access files is the PUT. Like the GET, it has two arguments, the file number and the physical record number, or sector number, with the last being optional. If the record number is omitted, the record in the buffer corresponding to the file number is written at the current record number position.

Examples:

```
100 PUT 1, 4
```

Write a record into the fourth position of File #1.

```
100 PUT L, J-1
```

Write a record into the J-1 position of File #L.

```
100 PUT 5, LOF(5)+1
```

Write a record on File #5 just past the end of the file.

```
100 PUT 5
```

Write a record into the current record position of File #5.

```
100 FIELD 1, 255 AS A$
110 GET 1, N: A1$=A$: GET 1,P
120 PUT 1,N: LSET A$=A1$: PUT 1,P
```

Switch the contents of the Nth and Pth records of File #1.

MKI$, MKS$, and MKD$

When numeric values (constants, variables, or expressions) are fielded into a direct access file buffer, they must be converted to strings. Integer values are converted to two-byte-long strings with the MKI$ function, single precision values are converted to four-byte strings with the MKS$ function, and double precision values are converted to eight-byte strings with the MKD$ function. It may help to think of these functions as "*MaKe Integer* String", "*MaKe Single* precision String", "*MaKe Double* precision String".

Examples:

```
10 FIELD 1, 2 AS XI$, 4 AS XS$, 8 AS XD$
20 X$=MKI$(X): LSET XI$=X$
30 LSET XS$=MKS$(3.27)
40 LSET XD$=MKD$(PI#)
```

Line 10 fields the File #1 buffer to store three strings of two, four, and eight bytes.

Line 20 first converts the value X to a two-byte string X$, assuming the value of X lies between -32768 and +32767 inclusive, else an error would occur. Note that if X is not an integer, its fractional value is dropped, in effect converting X to an integer. Then line 20 transfers the string X$ to the two-byte field XI$. After this conversion to string, either the LSET or RSET instruction can be used, since the two-byte string fits exactly into its two-byte buffer space.

Line 30 converts the single precision value 3.27 to a four-byte string and transfers it directly into the file buffer.

Line 40 converts the double precision value at PI# into an eight-byte string directly into the file buffer.

If this statement were executed

```
50 LSET XD$=MKI$(5)
```

the two-byte string equivalent to the integer value 5 would be transferred into the leftmost two bytes of the eight-byte buffer field called XD$. Not only would this waste space, but it would make the subsequent proper reconversion of XD$ to its original value of 5 extremely difficult.

To show you the advantage of these three conversion functions in saving space, compare the storage space used in two possible cases:

(1) three values, -32767, -2.71828E-14, and -3.141592653589793 are stored as strings of characters, each character representing a single digit;

(2) the same three values are converted with the MKI$, MKS$, and MKD$ functions to two, four, and eight byte strings.

Case 1: Conversion to character strings, then storage in the buffer.

```
10 FIELD 1, 6 AS I$, 12 AS S$, 18 AS D$
20 LSET I$=STR$(-32767)
30 LSET S$=STR$(-2.71828E-14)
40 LSET D$=STR$(-3.141592653589793)
```

The result is:

```
-32767-2.71828E-14-3.14159263589793
  6bytes   12bytes          18bytes
```

In case 1, the total amount of record space used to store these three values is 36 bytes.

Case 2: Representation of values as direct copies of memory storage.

```
10 FIELD 1, 2 AS I$, 4 AS S$, 8 AS D$
20 LSET I$=MKI$(-32767)
30 LSET S$=MKS$(-2.71828E-14)
40 LSET D$=MKD$(-3.141592653589793)
```

The result is:

```
I$  S$  D$
2   4   8
```

In case 2, the total amount of record space used to store the same three variables *to the same accuracy* is just 14 bytes, a saving of more than 150%.

CVI, CVS, and CVD

When a direct access record is read into a buffer with a GET, the numeric values are most likely in 2-, 4-, or 8-byte string form as a result of MKI$, MKS$, or MKD$ functions. These strings are not printable, and they are not convertible with a VAL function. The functions that are used to reverse the process of the MKI$, MKS$, and MKD$ functions are CVI, CVS, and CVD, the *Con*V*ert* to *I*nteger, *Con*V*ert* to *S*ingle precision, and *Con*V*ert* to *D*ouble precision. They take two, four, and eight byte buffer fields and convert the strings to numeric values.

Examples:

```
510 FIELD 1, 2 AS N$, 8 AS X$, 4 AS A$
520 GET 1,L
530 A%=CVI(N$)
540 B=CVS(A$)
550 C#=CVD(X$)
560 PRINT A%,B,C#
```

In the program segment above, the buffer is described as having three strings, N$, X$, and A$. Line 520 fills that buffer with the first 14 bytes of the Lth record. Then lines 530-550 produce three numeric values A%, B, and C# that correspond to the strings N$, A$, and X$.

Direct Access File Creation

The steps involved in creating a direct access file are more involved than those used in creating sequential access files because the programmer is responsible for opening the file, converting values to strings, filling the buffer, and finally writing the record. The order of these steps is important, so they are listed below, in the order in which they must appear in the program.

1. *Open* an existing file or open a new file, entering its name in the TRSDOS directory (in effect *creating* it).
   OPEN "R",n,"filename"
2. *Describe* the records of the file with the FIELD statement.
   FIELD n, len1 AS var1$, len2 AS var2$, . . .
3. *Load* the buffer with just strings.
   LSET var1$=exp
   or
   RSET var1$=exp
4. *Write* the record at the desired location.
   PUT nfile, nrec

A complete program may best serve as an example of the process. Suppose the problem is to create a direct access file called MASTER containing this information: amount owed (AM$), date of last purchase (PU$), date of last payment (PA$), and amount of last payment (AL$). Table 8.1 supplies the buffer field information.

Since each one of these logical records is less than half the length (256 bytes) of each physical record, it would be economical to include two logical records for each physical record. To allow for possible growth of these records to include more fields, we can start the second logical record half way down at byte 129 of the 255 byte buffer.

Once the file has been opened and its buffer described, the user can enter information starting where the file ends.

Length	Name	Description
24	NA$	Customer name
20	SA$	Street address
26	CZ$	City-state-zip
12	PH$	Phone No. (ARC-EXC-DDDD)
4	AM$	Amount owed (single precision)
2	PU$	Date of last purchase (integer, coded MMDDY)
2	PA$	Date of last payment (integer, coded MMDDY)
4	AL$	Amount of last payment (single precision)

94 bytes total

Table 8.1 Buffer Field Information

```
10 'FILENAME: "C8P1"
20 'FUNCTION: DIRECT ACCESS ACCOUNTS RECEIVABLE FILE BUILDER
30 ' AUTHOR: JPG                    DATE:   2/80
40 CLEAR 200: CLS: N=0
50 '    open the file for random access input/output
60 OPEN "R",1,"MASTER"
70 '   define the use of the buffer area
80 FIELD 1,              24 AS NA$(1), 20 AS SA$(1),
                         26 AS CZ$(1), 12 AS PH$(1),
                          4 AS AM$(1),  2 AS PU$(1),
                          2 AS PA$(1),  4 AS AL$(1)
90 FIELD 1, 128 AS D$, 24 AS NA$(2), 20 AS SA$(2),
                         26 AS CZ$(2), 12 AS PH$(2),
                          4 AS AM$(2),  2 AS PU$(2),
                          2 AS PA$(2),  4 AS AL$(2)
100 GOSUB 600'  clear the disk buffer
110 '  skip the next section if the file doesn't exist yet
120 IF LOF(1)=0 THEN 200
130 '   determine what the value of N should be
140 N=(LOF(1)-1)*2+1: GET 1,LOF(1)
150 '   increment N and clear the buffer if
160 '  the second sub-record is not blank
170 IF ASC(NA$(2))<>32 THEN N=N+1: GOSUB 600
180 '  increment N (logical record number)
190 '  and find  I (the sub-record pointer)
200 N=N+1: I=INT(N/2)*2-N+2
210 LINE INPUT "NAME (TYPE 'END' TO STOP): ";N$
220 '  if that's all, save this record, then end
230 IF N$<>"END" THEN 290
```

```
240 ' if this record hasn't been written yet, do so now
250 IF I=2 THEN GOSUB 500
260 '    close up the file, then stop
270 CLOSE: GOTO 10000
280 '   otherwise, begin filling the buffer with text
290 LSET NA$(I)=N$
300 LINE INPUT "STREET ADDRESS: ";A$: LSET SA$(I)=A$
310 LINE INPUT "CITY-STATE-ZIP: ";A$: LSET CZ$(I)=A$
320 LINE INPUT "PHONE NO.: ";      A$: LSET PH$(I)=A$
330       INPUT "AMOUNT OWED";    A : LSET AM$(I)=MKS$(A)
340       INPUT "DATE LAST PURCHASE (MMDDY)";A: LSET PU$(I)=MKI$(A)
350       INPUT "DATE LAST PAYMENT  (MMDDY)";A: LSET PA$(I)=MKI$(A)
360       INPUT "AMT LAST PAYMENT";A: LSET AL$(I)=MKS$(A)
370 '   if 2nd sub-record, save that record,
380 '   then clear the disk buffer and return
390 IF I=2 THEN GOSUB 500: GOSUB 600
400 '   so back for another record from the user
410 PRINT: PRINT: PRINT: GOTO 200
420 '   save the buffer onto disk
500 PUT 1,INT((N-1)/2)+1: RETURN
510 '  this loop clears the buffer to keep things straight
600 FOR J=1 TO 2
610  LSET NA$(J)="": LSET SA$(J)="": LSET CZ$(J)=""
620  LSET PH$(J)="": LSET AM$(J)="": LSET PU$(J)=""
630  LSET PA$(J)="": LSET AL$(J)=""
640 NEXT J: RETURN
10000 END
```

Direct Access File Processing

The creation of a direct access file must precede any other activity that deals with the file, so it is important to understand this technique. However, the reason for a direct access file's existence in the first place is to provide a way to input any one of its records into memory for processing. This is done with the GET instruction, along with a number of other instructions and functions, just as the PUT instruction cannot operate alone. The usual order of execution for the instructions that load variables from a direct access disk record to various memory locations is as follows:

1. *Open* the file.
   OPEN "R", n, "filename"
2. *Describe* the records of the file with a FIELD statement.
   FIELD n, len1 AS var1$, len2 AS var2$, . . .
3. *Read* the record into the buffer
   GET nfile, nrec
4. *Transfer* the buffer's contents into memory, converting numeric variables if necessary.

Note that for writing a record, the order is:

FIELD . . .
LSET . . . = MKt$(. . .)
PUT . . .

where the t in MKt$ indicates type, for example S, D, or I.
For reading a record, the order is:

FIELD . . .
GET . . .
. . . = CVt(. . .)

where the t in CVt indicates type.

The following program fetches the Nth record from the file MASTER that was created in the previous program C8P1, then allows a change in that record, then rewrites it in updated form. This program exemplifies what is meant by the term "direct access". Note that the user of the program enters the actual entry number (logical record number), and the computer determines which physical record to GET and where in that record the entry is located.

```
10 'FILENAME: "C8P2"
20 'FUNCTION: DIRECT ACCESS FILE UPDATE
30 ' AUTHOR : JPG                DATE:   4/80
40 CLEAR 200: CLS
50 OPEN "R",1,"MASTER": GOSUB 220
70 INPUT "LOGICAL RECORD NUMBER (0=END)";N
80 IF N=0 THEN CLOSE: GOTO 10000
90 GET 1,INT((N-1)/2)+1
100 I=INT(N/2)*2-N+2'I=logical record #
110 PRINT "ENTER NEW INFO. OR /EN/ = NO CHANGE"
120 PRINT "NAME: ";NA$(I);
130 LINE INPUT A$: IF A$<>"" THEN LSET NA$(I)=A$
140 PRINT "STREET ADDRESS: "; SA$(I);
150 LINE INPUT A$: IF A$<>"" THEN LSET SA$(I)=A$
160 PRINT "CITY-STATE-ZIP: "; CZ$(I);
170 LINE INPUT A$: IF A$<>"" THEN LSET CZ$(I)=A$
180 PRINT "PHONE          : "; PH$(I);
190 LINE INPUT A$: IF A$<>"" THEN LSET PH$(I)=A$
200 PRINT: PRINT "/EN/": LINE INPUT A$: CLS: PRINT: GOTO 70
220 FIELD 1,             24 AS NA$(1), 20 AS SA$(1),
                        26 AS CZ$(1), 12 AS PH$(1),
                         4 AS AM$(1),  2 AS PU$(1),
                         2 AS PA$(1),  4 AS AL$(1)
230 FIELD 1, 128 AS D$, 24 AS NA$(2), 20 AS SA$(2),
                        26 AS CZ$(2), 12 AS PH$(2),
                         4 AS AM$(2),  2 AS PU$(2),
                         2 AS PA$(2),  4 AS AL$(2)
240 RETURN
10000 END
```

Program C8P2 is noteworthy because it uses the LINE INPUT to advantage by allowing the user to enter a /EN/ (ENTER keystroke) as a null response if no change is necessary. This speeds up the response time considerably over having to enter a "NO", a digit, or even a blank, as a null response. Another possible technique could have been used to update this file, and that is to display the entire record in menu format down the screen, then allowing the user to select a field for change. For example, the list below could be displayed:

```
1.  NAME              HARSHBARGER HERSHEL

2.  ADDRESS           99 KNOWNOTHING ACRES

3.  CITY-STATE-ZIP    SHOWME MO 72787

4.  PHONE             123-456-7890

5.  NO FURTHER CHANGE

    ENTER SELECTION BY DIGIT
```

The user could type the digit, immediately followed by the new information, for example:

```
2451 INANITY AVE.
3DULLARD CT 42179
5
```

The screen would show the entire record again.

```
1.  NAME              HARSHBARGER HERSHEL

2.  ADDRESS           451 INANITY AVE.

3.  CITY-STATE-ZIP    DULLARD CT 42179

4.  PHONE             123-456-7890

5.  NO FURTHER CHANGE

    ENTER SELECTION BY DIGIT
```

The menu could be modified by allowing the user to enter a 6, which would ask for a new record number to give the user an opportunity to change another record.

As a last example, we include a program that might brighten a morning by selecting some random "Message of the Day" from a

file that is easy to run and expand. The idea is simple: Have the user come in and type

RUN "MESSAGE"

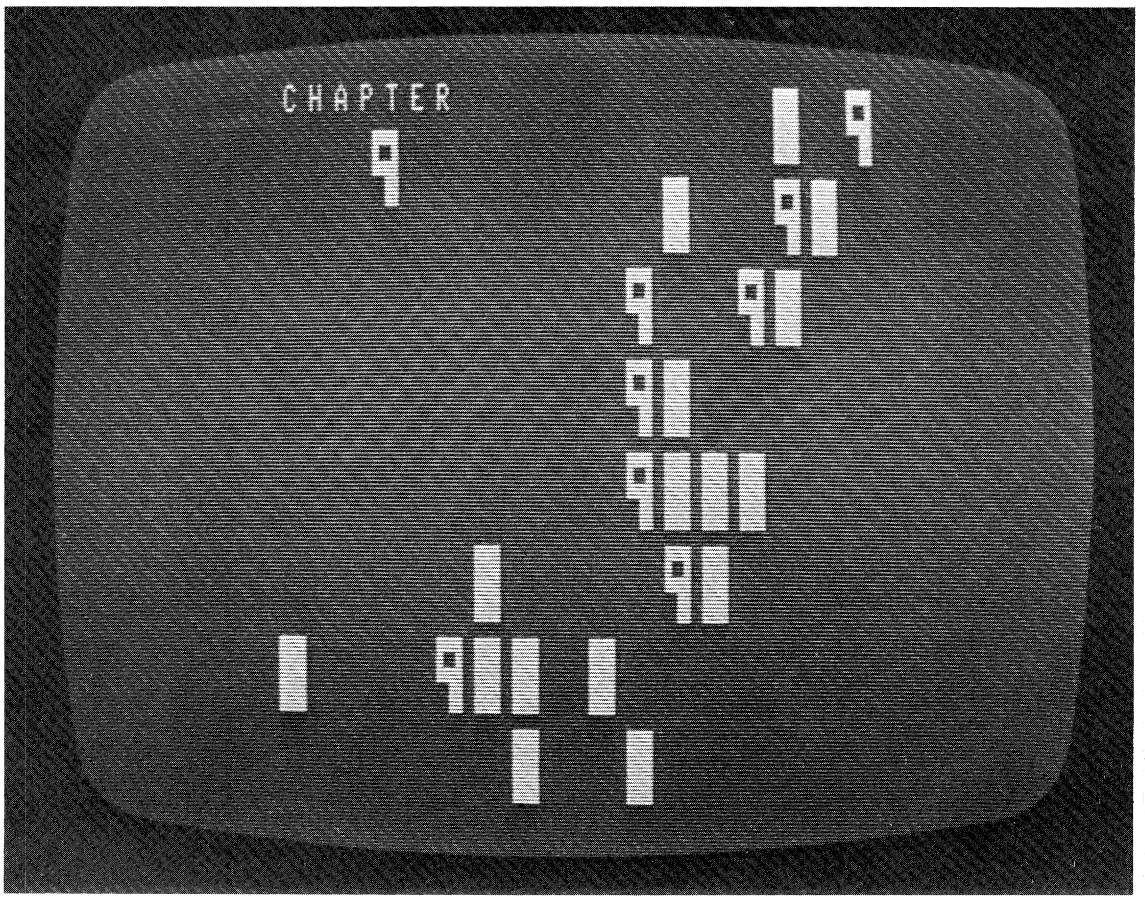
from BASIC. The computer randomly accesses a single record from a file of jokes, sayings, greetings, and potpourri of other messages. It displays the record on the screen, gives the user a chance to add a new message, then branches off to a master menu of activities for the day.

```
10 'FILENAME: "C8P3"
20 'FUNCTION: MESSAGE FILE BUILDER AND ACCESSER
30 ' AUTHOR : JPG                    DATE: 4/80
40 CLEAR 300: DEFINT A-Z: CLS
50 OPEN "R", 1, "MESSAGE/DAT": FIELD 1, 255 AS A$
60 LAST=LOF(1)
70 IF LAST=0 THEN LSET A$="THE FUNGO BAT IS IN TOLEDO.":
                  PUT 1, 1
80 GET 1, RND(LAST): PRINT @ 520, A$
90 LINE INPUT "Your turn: ";B$
100 IF B$<>"" THEN LSET A$=B$: PUT 1, LAST+1: GOTO 60
110 CLOSE: RUN "MENU" '  some other master menu display
10000 END
```

A typical dialog with this program could go something like this, with the user's entries followed by /EN/.

```
RUN/EN/

     THERE ARE 336 DIMPLES IN THE STANDARD GOLF BALL.

     DARTH VADER IS ALIVE AND WELL IN URUGUAY./EN/

     THE FUNGO BAT IS IN TOLEDO.

     OCT 9:  CABBAGE & LIME JELLO SIT DOWN DINNER./EN/

     /EN/
```

This chapter has introduced the techniques that programmers use to read and write records using direct access files. The techniques for file management—those that deal with overall file design and access algorithms—will be introduced in the next section.



Conversational Programming

When computers did their jobs at a distance through batch processing with punched cards, no one really cared about the quality or quantity of messages that the programs wrote to the operator at the console typewriter. After all, one of the skills that the operator learned was to decipher the terse and arcane jargon of the operating system.

When timesharing became popular in the early 1970's, programmers realized that a user at the terminal was more likely to enjoy the session if a dialog could be established to "humanize" the computer. The users would accept mild rebuke from the computer. Since that time programmers have developed many techniques to promote proper responses from the users. This chapter discusses some techniques that have been found effective.

User Prompts and Menus

It is up to the programmer to let the user know what a proper response should be. This is the reason for BASIC allowing a string to be printed along with the question mark with an INPUT statement. The prompt should also be used whenever a list of values or strings is to be inputted from the terminal. For example, suppose

a dialog has proceeded to the point shown below, with the user's responses underlined:

```
HOW MANY DATA VALUES ARE THERE? 15

     ENTER EACH DATA VALUE

? 140

? 220

? .
  .
  .
```

Notice that although the first response, the 15, was cued well with the question

```
HOW MANY DATA VALUES ARE THERE?
```

all the other entries are just cued once. Then the user must keep track of the number of entries that are typed. If a double carriage return is entered (the ENTER key was hit twice, or the keyboard suffers from bounce) that's just too bad, because as chapter 6 pointed out, that just enters the previous entry again. Also, notice that the question mark prompt is artificial; it is the result of an INPUT being executed, not a question being asked.

A much better dialog would be:

```
HOW MANY DATA VALUES ARE THERE? 15

TYPE DATA VALUE 1? 140

TYPE DATA VALUE 2? 220
  .
  .
  .
```

This dialog is produced with the statement:

```
100 PRINT "TYPE DATA VALUE";I;: INPUT X(I)
```

At least the dialog reminds the user of which value to enter. However, it still suffers from the question mark being printed, and from an accidental double /EN/ causing false input.

The next sample dialog is made possible by using the LINE INPUT statement.

```
HOW MANY DATA VALUES ARE THERE? 15

TYPE DATA VALUE 1: 140

TYPE DATA VALUE 2: 220
  .
  .
  .
```

If the user accidentally double strokes the /EN/, a good program should be able to discover the mistake. Also, since the input variable is a string, the program must be able to check it for illegal characters, because the computer won't give any automatic second chances for input with a ? REDO message, as is normally the case with numeric input.

Here is a subroutine that acts like the line

```
100 PRINT "TYPE DATA VALUE";I;: INPUT X(I)
```

except that it uses LINE INPUT and translates the string to a value after checking to see if it is an integer between 0 and 200 inclusive.

```
800 '  input subroutine
810 PRINT "TYPE DATA VALUE";I;": ";
820 LINE INPUT A$: N=LEN(A$): V=VAL(A$)
830 FOR J=1 TO N: C$=MID$(A$,J,1)
840   IF C$<"0" OR C$>"9" THEN 870
850 NEXT J
860   IF V<=200 THEN X(I)=V: GOTO 880
870 PRINT "*** ERROR ON INPUT ***"
880 RETURN
```

Another useful technique that should accompany data input is echo checking, which involves the display of all data entered in one session. This may be sectioned in such a way as to simplify the process of updating any value that was entered in error.

The INKEY$ is useful when the responses are limited to single characters, because the user is saved the trouble of pressing the ENTER key after every stroke. But be careful about mixing INKEY$, LINE INPUT, and INPUT. As the programmer, you are responsible for making data input and interactive dialog as easy to understand and use as possible, and that includes being consistent.

A *menu* is the display of a list of commands with a convenient way to access the commands. Some examples of menus are shown below.

(1)

```
ACCOUNTS RECEIVABLE PACKAGE:

        1.  ADD ACCOUNTS

        2.  DELETE ACCOUNTS

        3.  POST PAYMENTS

        4.  POST CHARGES

        5.  AGE ACCOUNTS
```

```
        6.  BILLING RUN

        7.  SUMMARY REPORTS

ENTER THE ACTIVITY OF YOUR CHOICE BY NUMBER:
```

(2)

```
Statistics Package:

       1.  Build data file

       2.  Sort data file

       3.  Create test data file

       4.  Edit data file

       5.  Chi-Square

       6.  T-Test

       7.  Descriptive

       8.  Correlation

       9.  Anova

      10.  Probabilities

      11.  Curvilinear regression

Enter choice of activity:
```

(3)

```
G A M E S :

       C      Chess

       A      Animal

       B      Breakout

       I      Ivanhoe
```

```
N       Nim

E       Enemy Below

T       Tic-Tac-Toe

W       Hunt the Wumpus

O       Othello

R       Rocket

K       KingKong

S       Startrek

Enter your choice by its first letter:
```

ERR, ERL, ON ERROR GOTO, and RESUME

Not many versions of BASIC, even the better versions found on the newest microcomputers, have *user-defined* error trapping. This is one of the TRS-80 Level II BASIC's outstanding features. In general terms, it allows the programmer to branch to specific lines of code when an error during the execution of a program.

The ON ERROR GOTO statement enables you to branch to a segment of code that checks for possible program recovery in case of an execution time error. This statement must be executed before the error occurs, otherwise the computer generates the usual error message and terminates execution. If the line number specified is 0 (ON ERROR GOTO 0) the error trapping feature is disabled and BASIC will print an error message as it usually does.

The RESUME command returns control of the program to any line after an execution time error has occurred. The RESUME can be written in any one of three ways:

(1) RESUME (no line number) or RESUME 0	returns control to the statement that caused the error.
(2) RESUME line-no	returns control to the specified line number.
(3) RESUME NEXT	returns control to the line following the one that caused the error.

Example:

```
10 ON ERROR GOTO 9000
.
.
.
50 X=Y/N
60 PRINT "X=";X
.
.
.
9000 '  N must not be zero -- if it is, change it
9010 '  into a very small number
9020 N=1E-20
9030 PRINT "N CHANGED FROM 0 TO 1E-20"
9040 RESUME
```

The effect of this program is to prevent termination of the program's execution due to a division by zero in line 50. A message is printed to indicate the change that was necessary to keep the program running. Notice that line 9040 could have been written as either of the following:

```
9040 RESUME 0
9040 RESUME 50
```

There is a particular advantage to the RESUME or RESUME 0, and this is that it allows a single error handling routine, like lines 9000-9040 above, to manage the possible occurrence of errors in many places within the program. For example, the example above could have contained the line

```
200 A(R)=2+V/N
```

If this line were executed with N=0, the error trap would function just as well, and the RESUME statement would return execution to the next line.

The ERR and ERL functions allow the programmer to use some very specific error traps. ERL returns the line number in which the error occurred, and the ERR returns a value related to the error code. Both functions need no argument.

The value that ERR returns is related to the error code this way:

ERR/2+1=Code value

or

ERR=2*Code value - 2

Appendix F lists all of the error codes that Level II BASIC and Disk BASIC can generate.

The following example shows various ways that the ERR and ERL functions can serve to isolate specific errors.

```
10 ON ERROR GOTO 9000
20 INPUT "ARRAY SIZE (ROW, COLUMN)";R,C
30 DIM X(R,C)
40 FOR I=1 TO R
50 INPUT "ENTER";C;"VALUES FOR ROW";I
60 FOR J=1 TO C: INPUT X(I,J): NEXT J,I
70 PRINT "ANY VALUES TO CHANGE (0,0=NONE)"
80 PRINT "ENTER ROW, COLUMN";
90 INPUT R1,C1
.
.
.
9000 '   error code 7 = out of memory
9010   IF ERL<>30 OR ERR/2+1<>7 THEN 9050
9020 PRINT "*** OUT OF MEMORY ***"
9030 PRINT "RERUN PROGRAM AND REDUCE ARRAY SIZE"
9040 STOP
9050 '   error code 10 = subscript out of range
9060   IF ERL<>90 OR ERR/2+1<>10 THEN 9100
9070 PRINT "*** SUBSCRIPT OUT OF BOUNDS ***"
9080 PRINT "TRY AGAIN -- THAT POSITION IS NOT LEGAL"
9090 RESUME 80
9100 '   no idea what's going on -- better
9110 '   let BASIC handle the problem
9120 ON ERROR GOTO 0
```

Anticipating User Responses

It is always a pleasure to interact with a computer through a well thought out program, one that was written with the user in mind. Many users, especially novices, regard any such interaction with trepidation because they know the computer is a machine. Either through fear of doing damage or through anxiety caused by the unfamiliar nature of this form of communication, these users dread the sessions and commit blunders. Good programming practice includes anticipating user errors and if possible correcting them or certainly allowing the user a second chance. Whatever approach is taken, you should always try to create a friendly and informative dialog with the user.

Consider these two dialogs:

```
.
.
.
NAME? KUMQUAT, KATHY

NAME? 14 PUNCTILIOUS PLACE

NAME? PERSIMMON, ALASKA 98765

NAME? 878-787-8/break/
```

The user has realized that the computer ignored all of these entries. The clue was the repeated prompt NAME? which the user remembers (too late) should change to

```
STREET?

CITY STATE ZIP?

PHONE?
```

The user's action of interrupting the program's execution with a /BREAK/ is rather excessive, because now the program may have to be restarted. The dialog could have been:

```
.
.
.
NAME? MOMERATH, POMEROY

NAME? MOMERATH POMEROY

STREET? 291 OUTGRABE ALLEY

CITY STATE ZIP? BRILLIG, VERMONT 24938

CITY STATE ZIP? BRILLIG VERMONT 24938
.
.
.
```

At least the program was kept running, and the user realized that the repeated prompt meant that something was wrong with that entry. But how did the user know? The dialog didn't say that commas weren't allowed, yet obviously that's what was wrong. As soon as commas were removed, the program proceeded nicely.

The example above shows rat maze behavior on the part of the user (try, try again—when you get it right, maybe you'll remember it) and sadistic behavior on the part of the programmer. After all, if the program was clever enough to determine that there was a comma in the entered data, certainly it should be clever enough to remove the comma, and kind enough to remind the user to omit commas between entries. The following dialog is nicer in all respects.

```
.
.
.
NAME? BACILLUS, VACILLA

    PLEASE OMIT COMMAS IN DATA INPUT

NAME? BACILLUS VACILLA

STREET ADDRESS? 66 SEPSIS ST

CITY? WOUND

STATE ZIP? WEST VIRGINIA, 49228

    NO COMMAS, PLEASE.  YOU KNOW I DON'T LIKE THEM!

STATE ZIP? WEST VIRGINIA 49228

PHONE? 499-994-4949
.
.
.
```

This dialog is polite (note all the PLEASEs) and varies even if the same input error is committed.

Check Digit Calculations

When specific numbers are used in a record as a code and their accuracy is imperative, such as serial numbers, account numbers, and other unique identification numbers, a check digit is often employed to verify the number. A check digit is a single digit added to the code number to make that code number self-checking. It has a unique relation to the rest of the code number and the way it is calculated determines the types of errors that can be detected by it.

The four kinds of errors that a check digit can detect are:

(1) Transcription: A wrong number is written, such as a 1 for a 7 or a 5 for a 0.

(2) Transposition: The correct numbers are written but their positions are reversed between neighboring columns, such 41582 for 41852.

(3) Double transposition: Numbers are interchanged between columns other than neighboring columns, such as 41582 for 48512.

(4) Random: A combination of two or more of the above, or any other error not listed.

The use of check digits involves the computer's initial calculation and storage of the digit with its corresponding code number as the number gets filed initially. For example, as a file of accounts is built, the account numbers become the access codes to

to the file which will incorporate the check digits. As each new account is entered, its access code is generated from the account number followed by the calculated check digit. This access code becomes a permanent part of the record. When an access operation is performed on the file, it is done by access code, including the check digit. If there is any error in either the account number portion or the check digit portion of the access code, the user is warned that the code doesn't exist, and should try again.

Check digits are related to their code numbers through any one of various methods of calculation. Some methods of calculating check digits are rather poor, detecting only some of the possible errors that occur. We will show you one of the best, and for that reason the most popular: the modulus 11 procedure. The method of calculation is the same as that which is used for detection. That is, when an access code is checked, its check digit is recalculated and compared to the one that was originally attached. If it is different, the user is warned of an error.

Modulus Eleven Check Digit Calculation Method

1. Multiply each digit in turn with its corresponding weight. The weight is simply the digit's position in the code number from right to left, plus one. For example, for the code 32604:
   3*6 2*5 6*4 0*3 4*2
   18 10 24 0 8
2. Add the resultant products.
   18 + 10 + 24 + 0 + 8 = 60
3. Divide the sum by the modulus (in this method it is 11) and keep the remainder. 60 / 11 = 5 with a remainder of 5.
4. Subtract the remainder from the modulus, and the result is check digit. 11 - 5 = 6. If the remainder is 0 the check digit is 0. If the remainder is 1, the check digit is 1. If the remainder is 10, that account number must be rejected.

Therefore the access code is 326046.

Examples:

1. Code number 421865
   4*7+2*6+1*5+8*4+6*3+5*2
   28+12+5+18+10=105
   105/11=9, remainder=6
   11-6=5
   Access code=4218655

2. Code number 2493
   2*5+4*4+9*3+3*2=59
   59/11=5, remainder 4
   11-4=7
   Access code=24937

3. Code number 2653
   2*5+6*4+5*3+3*2
   10+24+15+6=55
   55/11=5, remainder 0
   11-0=11: special case: checkdigit=0
   Access code=26530

4. Code number 4653
   4*5+6*4+5*3+3*2=65
   65/11=5, remainder 10
   Reject this account number. Do not allow it in the system, and choose the next code number, 4654. It's check digit is 0, making the access code 46540.

5. The access code that the user gave is 26214. Is this a proper access code? (Does the check digit of 4 correspond to that calculated for code number 2621?)
   2*5+6*4+2*3+1*2=42
   42/11=3, remainder 9
   11-9=2
   The access code 26214 is improper since 2621 has check digit 2, not 4. An error message should be issued to the user.

Instead of recalculating the check digit, the computer program that verifies the accuracy of the input access code can multiply all digits of the access code by their weights, using a weight of 1 for the check digit itself. When the sum of all of these products is divided by the modulus, the result should be zero.

The modulus 11 check digit calculation procedure can detect all of the user's transcription and transposition errors, which account for 95% of all the errors that users commit. The method also detects 90% of the random errors. Thus the modulus 11 method can detect 99.5% of all errors. If even more accuracy is needed, the method can be modified by using prime numbers larger than 11 for a modulus. For example, the modulus 37 method detects 99.987% of all user errors.

Check digit calculations impose a penalty in calculation time, but that penalty is not severe when the application involves a single user on the computer.

The following program attaches a check digit to a six digit account number.

```
10 'FILENAME: "C9P1"
20 'FUNCTION: CHECK DIGIT CALCULATOR
30 ' AUTHOR : JPG                DATE:   4/80
40 '
50 PRINT "To stop, press /break/, otherwise enter"
60 PRINT "an account number (any number of digits)"
70 PRINT: LINE INPUT "Please enter the number: ";N$
80 S=0 '    reset the sum to zero
90 '  the followins is a loop for addins the values and
100 '     multiplyins the Ith digit by its weisht
110 '  while keepins a runnins total of the sums so far
120 FOR I=1 TO LEN(N$)
130   S=S+VAL(MID$(N$,I,1))*(LEN(N$)+2-I)
140 NEXT
150 '  set modulo eleven from the sum
160 M=S-INT(S/11)*11
170 '  add the proper check digit to the number
180 IF M>0 THEN N$=N$+RIGHT$(STR$(11-M),1)
           ELSE N$=N$+"0"
190 '   if acct. # is sood, print it, otherwise
200 '   print reject messase
210 IF M=10 THEN PRINT "Acct. # rejected"
            ELSE PRINT "Acct. # is ";N$
10000 END
```

Praise and Chastisement

One of the most irritating forms of dialog is that which is made up of uniform words of praise. Consider this dialog:

```
WHAT IS 2+2, JOHNNY? 4

GOOD!  WHAT IS 5+7, JOHNNY? 12

GOOD!  WHAT IS 6+19, JOHNNY? 25

GOOD!  ...

GOOD!  ...
```

Johnny is by now well beyond reading the computer's dialog, and merely looks at the digits to make the proper response. If that's the case, the program should do away with the words entirely. Praise does work, though, and if handled properly it can act as a strong stimulus for proper user behavior whether the user is a child being drilled in arithmetic skills, a bookkeeper posting payments, or an executive studying stock portfolios. The secret of using praise with the computer is to randomize it. Consider this dialog:

```
WHAT IS 5+7, JOHNNY? 12

GOOD WORK.

WHAT ABOUT 8+17, JOHNNY? 23

YOU MISSED THAT ONE.  BETTER LUCK NEXT TIME.

CAN YOU COMPUTE 8+17? 25

SUPER!

TRY ADDING 22+17? 39

VERY GOOD!
.
.
.
```

These kinds of messages are fun to create in a program, fun for the user to receive, and significantly improve the user's performance in whatever task is being tried. The technique relies on generating entire sentences from randomly chosen phrases in two phrase pools: one pool is for praise and the other for chastisement, or scolding. The following program illustrates the technique.

```
10 'FILENAME: "C9P2"
20 'FUNCTION: ILLUSTRATE PRAISE AND SCOLDING
30 ' AUTHOR : JPG                    DATE:   6/80
40 CLEAR 1000
50 R=20 '  numbers will initially be between 1 and 20
60 CLS: INPUT "What is your name";N$
80 X=RND(R): Y=RND(R): S=X+Y  'Get the random numbers
90 PRINT "What is";X;"+";Y;", ";N$;
100 INPUT S1
110   IF S1<>S THEN GOSUB 150: GOTO 90
            ELSE R=R+1: GOSUB 120: GOTO 80
120 FOR I=1 TO 10: READ Y$: NEXT I
130 GOSUB 150: RETURN
140 '    subroutine to print the messages
150 GOSUB 170: A$=X$: GOSUB 170: B$=X$
160 PRINT A$+", "+B$: RESTORE: RETURN
170 J=RND(5): FOR I=1 TO J: READ X$: NEXT I
180 IF J<5 THEN FOR I=J+1 TO 5: READ Y$: NEXT I
190 RETURN
200 DATA You missed, Bad news, Ugh, Too bad, Yuch, Klutz!
210 DATA nerd!, turkey!, buffoon!, dolt!, Good, Excellent
220 DATA Fine, Nice, Great, You got it, You're right
230 DATA Just fine, Keep it up, Super
10000 END
```

Informing the User During Processing

Communication with the user is more than just asking for data and giving answers. It also involves informing the user about how to perform a task, such as running a program or managing a file, and informing the user that something is happening during some process in a program's execution.

Suppose, for instance, the dialog proceeds in this fashion:

```
Do you want to:

      (1) sort   (2) update   (3) rename a file? 1

What is the name of the file? MAYACCTS
```

(A pause while the file is opened)

```
On what key do you wish to sort?

    (1)  name

    (2)  account number

    (3)  date of last service

    (4)  amount past due

Enter your choice by digit: 2

Output on (1) screen, (2) printer, (3) file? 3
```

(A very long pause while the file is being sorted. There may be much hissing and clacking of drives, to the possible dismay of the user)

```
Done.
```

Consider the concern of the novice user during the sorting process. Is there something wrong because of all the noise? Why isn't anything happening? Is it sorting the file, and creating a new one, as it should be? How long does this go on? Should I turn off the computer before it breaks?

The programmer can do a lot to ease the user's anxiety by printing simple intermediate messages during processing to keep the user informed about what is going on. The last program in chapter 7 shows how to use a simple PRINT statement during a sort to let the user know how it is progressing. When a sort takes hours, as it may if the number of elements to be sorted is in the thousands, a PRINT strategically placed within the sort can help the user greatly. It can tell:

(1) Whether the sort is working.

(2) Approximately how long it takes.
(3) What stage of the sort is being performed.

Another use of the process-time message is to keep the user from doing something during critical times, or to inform the user of abnormal events. For example, a process may turn the disk drive on and off with long pauses when internal processing takes over and the drive is off. The user should be given some message, such as:

```
*** STILL USING THE DISK DRIVE! ***

***  DO NOT OPEN THE DRIVE DOOR! ***
```

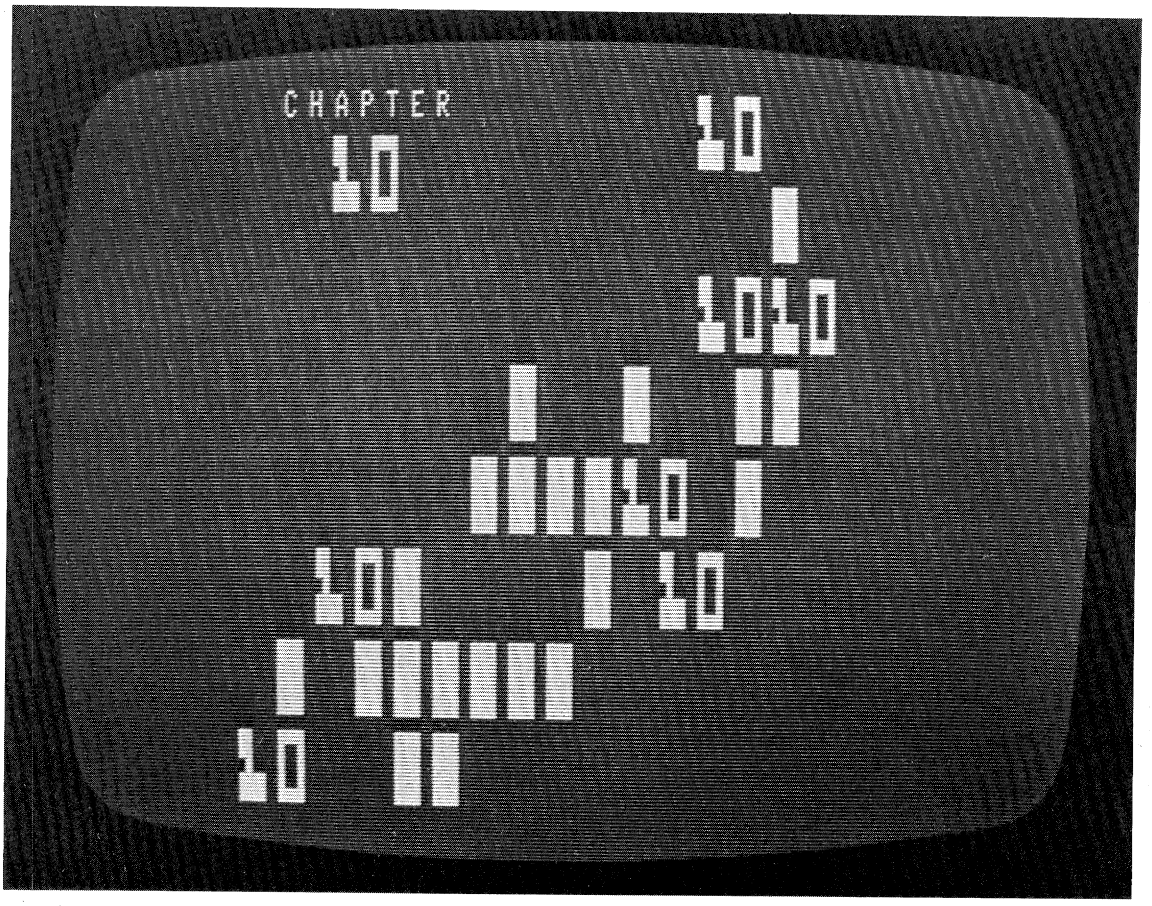
Another case in point is in the instance when one drive gets full, the program senses it, and opens a new file on another drive. The user may be caught by surprise, and absolutely mustn't disturb the process. A helpful message might be:

```
DON'T WORRY ABOUT ACTIVITY ON OTHER DRIVES.

   YOUR FILE WAS TOO LARGE FOR ONE DRIVE.

    *** DO NOT OPEN ANY DRIVE DOORS ***
```

Such messages as these often spell the difference between a happy user who is pleased to work with the system, and an anxious user who can't wait to get away from the machine.

Obviously, the programmer must be aware of the user's point of view when the program is still in its initial stages of composition. This technique of user-proofing a program is not trivial. It requires both the knowledge of programming techniques and considerable imagination in trying to anticipate user responses.

The technique that follows in the next chapter is an aid to the programmer rather than to the user. As you will see, it involves the overall design of programs to make them more readable and easier to alter.



Structured Programming

The number of programmers that use BASIC will increase greatly during the 1980s, primarily due to the rising popularity and falling price of microcomputers. Many of these new programmers will commit the same fundamental errors in technique that were committed ten and twenty years ago by today's professionals. Unless these new ranks of programmers learn the proper ways of programming, their level of frustration will increase in proportion to their level of productivity.

By the late 1960's the speed and memory capacity of computers had increased to such a point that it was economically unwise for a programmer to "fine-tune" his or her product to run a little bit faster or to fit into a slightly smaller area of memory. The programmer's time was worth more than any possible gain in machine time or memory use. Aside from this economic consideration, the number of programmers was rising, and the ability to communicate the contents of a program to new members of the staff became a very valuable asset.

Programs got more complex as they dealt with more file storage hardware. With some of these programs, as their complexity grew, their reliability dwindled, sometimes to the point of complete failure.

Some of these aging and dying programs represented large investments in time and money, yet because of their great complexity, their slightest alteration risked severe and unexpected complications.

During the 1970's, programmers developed various techniques to design their product from the beginning as a set of independent modules, each of which could be altered without affecting any other. Also, new and independent modules could be added to the program without the risk of creating a bug in a previously tested and debugged module.

These techniques yielded three immediate benefits:

(1) *Programmer productivity* increased due to a better understanding of the fundamental problems that were being solved.

(2) *Program debugging* and testing was simplified. A new module could be tested independently of all others.

(3) *Program maintenance* was simplified as a result of the ease of module addition, modification, and deletion.

These three factors made programs more flexible and extended their useful life. As the environment and the computer hardware changed, these modular programs had a much better chance of surviving and being productive than their predecessors.

Program Planning

Students in most introductory programming classes are taught that a program is written in five stages:

(1) Understand the problem.

(2) Formulate and flow chart an algorithm, or method, for its solution.

(3) Code the program.

(4) Test it and debug it.

(5) Document the program and its output.

The first stage is intuitively obvious. How can any problem be solved without a thorough understanding of its nature? A programmer usually understands the problem when he or she understands the four major phases of input, processing, storage, and output, clearly. This overall view usually determines the format of most of the program's outputs.

The second stage normally involves a rather detailed design of all records: input, transaction, storage, and output. The algorithm is usually flowcharted using some variation of the standard ANSI (American National Standards Institute) symbols.

The third and fourth stages are often performed simultaneously. The key ingredient in the testing phase is imagination. The programmer must be able to anticipate as many of the user's responses as possible. This phase was discussed in some detail in the last chapter.

Documentation is the fifth phase, and this subject is covered in the next chapter.

In this chapter we will consider program planning to include both the first and the second phase.

Phrase Flowcharts

When one thinks of flow charts, what usually comes to mind is a map-like drawing full of variously shaped symbols and arrows. This is not necessarily the case. A flowchart is a step-by-step representation of a problem's method of solution (an algorithm), and it is often just a set of English phrases and notes in some semblance of the order of solution. This is a *phrase flowchart* . It is easy to understand, and it is a more natural product of the programmer than a symbolic flowchart, particularly during the initial stages of the program's creation. As an example of a phrase flowchart, we have included here the original phrase flowchart for the Sort-Merge program in chapter 7.

(1) Create 4 files, fill with random alphanumeric data,
    50 records with 30 characters per record,
    list them.
(2) Sort each file.
    Repeat the next four steps four times.
    (a) Load file into memory.
    (b) Sort.
    (c) Close: open.
    (d) Write out.
(3) Merge.
    (a) Read 1 record into each of four buffers.
        Start each of 4 counters at 1.
    (b) Scan top of stacks, write out smallest to 1 file,
        read new one, add 1 to proper counter.

The usual approach to making phrase flowcharts is to state in phrase form what is to be done in a series of steps, and to number each step. The final product vaguely resembles a set of broadly worded instructions. Its chief advantages are simplicity and familiarity to the program's creator.

ANSI Flowcharts

Since the 1960's, the computer industry has made efforts to establish some standards that would cover the wide variety of products of the technology. The organization that has had the most impact has been the American National Standards Institute (ANSI). This body has established industry-wide standards for computer languages, methods and codes for machine-to-machine communication, and even the symbols that should be used for flowcharting.

ANSI symbolic flowcharts are what comes to mind when flowcharts are mentioned. They are symbolic; that is, they are made up of connected diagrams that represent various segments and processes of the program. The five basic ANSI flowchart symbols are shown in figure 10.1. These five symbols are interconnected with straight lines. The chart is read from top down and to the right

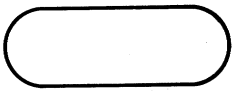
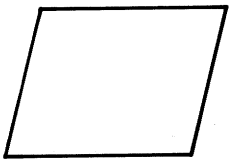
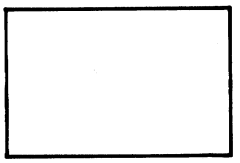
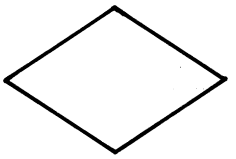
Symbol	Name	Meaning
	Terminal	Start or end of a sequence of operations
	Input/Output	I/O operation
	Process	Any processing function
	Decision	Any kind of branching operation
	Connector	Connection between parts of a flowchart

Figure 10.1 Basic ANSI Flowchart Symbols

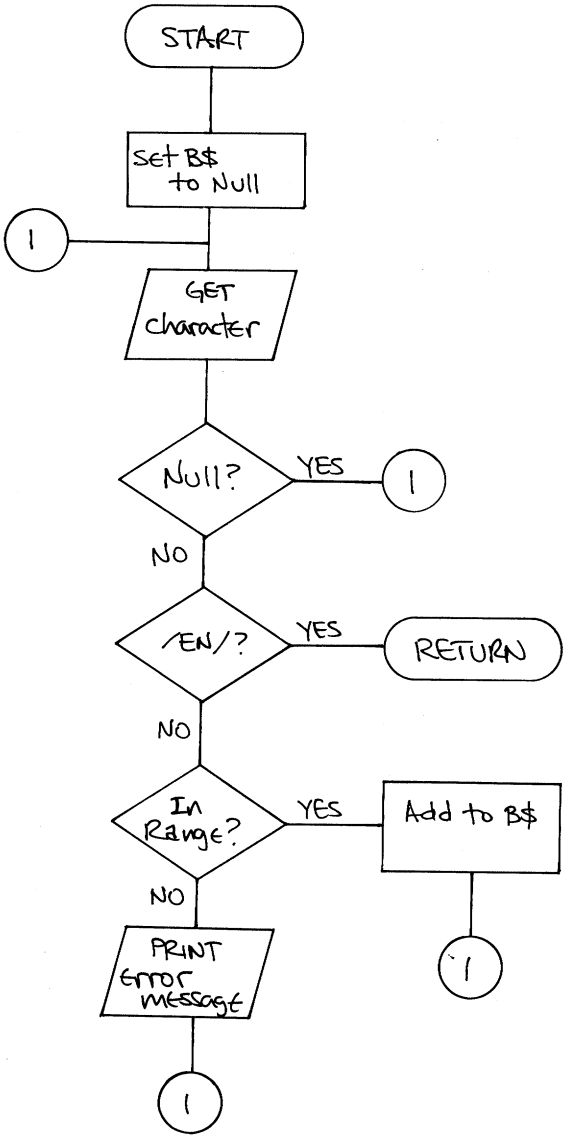
unless an arrow indicates a different direction of logic flow.

Consider the problem of inputting an all-digit answer with the INKEY$ function, as programmed in chapter 6. First, we will show you a phrase flowchart of the problem's solution, then an ANSI symbolic flowchart.

Phrase Flowchart for Digit Input Subroutine

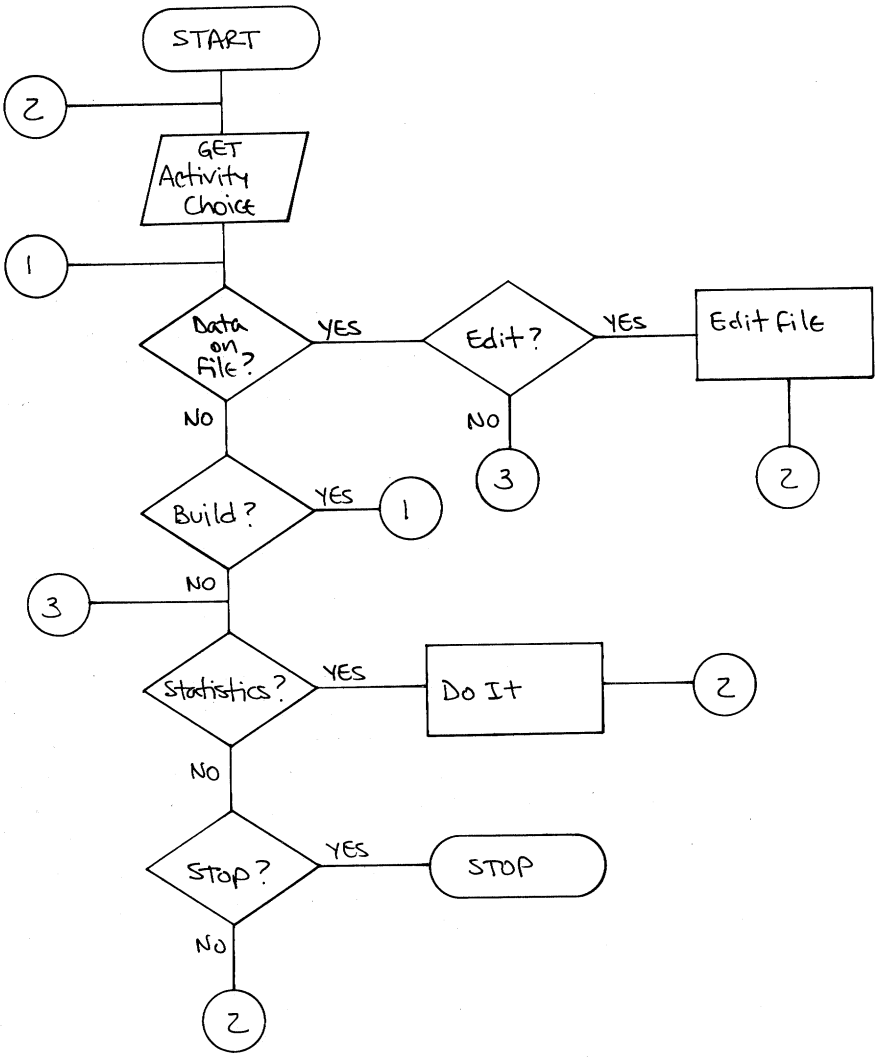
(1) Null the input string.
(2) Input a character with INKEY$.
    (a) Return if /EN/ stroke.
    (b) Print error message if out of range, then get new character (step 2).
    (c) Add to input string if OK, then get new character (step 2).

Symbolic Flowchart for Digit Input Subroutine



Symbolic flowcharts can be very specific; that is, every symbol can represent a single instruction, such as the flowchart above. Or they can be very broad; that is, every symbol represents a large set of instructions. The following example is a broad symbolic flowchart.

Symbolic Flowchart for a Statistics Package



Programming Structures

All programs and their included parts can be broken down into one or more of three basic structures.

1. The *Sequence Structure* consists of a set of imperative program statements that are executed in sequence. For example, this program segment is a sequence structure.

```
.
.
.
200 OPEN "R", 1, "RANKEN"
210 FIELD 1, 255 AS X$
220 LAST=LOF(1)
230 GET 1, LAST
```

2. The *Selection Structure*, or the *IF–THEN–ELSE Structure*, represents a choice between two and only two actions based on a condition. If the condition is true, one action is performed; if it is false, the other action is performed. Consider the following sequence of code as an example.

```
.
.
.
100 IF A<NUM THEN SUM=SUM+X: K=K+1:
                  S2=S2+X*X: GOTO 50
            ELSE S3=SUM*SUM
```

3. The *Iteration Structure,* or *FOR–NEXT Structure*, provides for executing a function as long as a condition is true. When the condition is no longer true, the program performs the next function in sequence. A common alternate form of the iteration structure allows the function to be executed *until* the condition becomes true. In BASIC the FOR–NEXT is the standard iteration structure, in some cases the IF–THEN–GOSUB can act as an alternate form. Here are some examples of the logic of iteration structures.

```
.
.
.
100 FOR I=1 TO 100 STEP 5
    - - - - -
150 NEXT I
.
.
.
```

```
.
.
.
200 IF A THEN GOSUB 500: GOTO 200
.
.
.
```

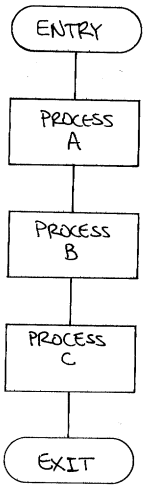
```
.
.
.
300 IF X<Y THEN GOSUB 500: GOTO 300
.
.
.
```

```
.
.
.
200 IF NOT A THEN GOSUB 500: GOTO 200
.
.
.
```

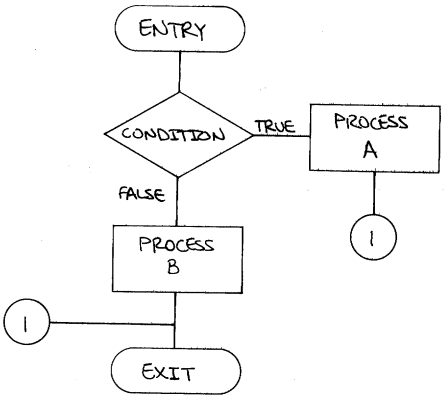
```
.
.
.
300 IF X=Y THEN GOSUB 500: GOTO 300
.
.
.
```

Figure 10.2 shows the symbolic flowcharts for each of these structures and will help clarify the differences in their actions.

Sequence Structure



Selection Structure



Iteration Structure

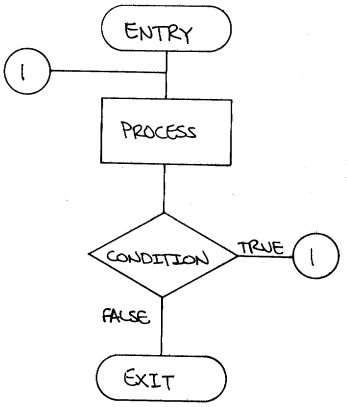


Figure 10.2 Symbolic Flowcharts of Programming Structures

GOTO–Less Programming

The term GOTO–less programming has been associated with Edsger Dijkstra of the University of Eindhoven, who in 1965 suggested that any program could be written without using GOTO statements. He said that GOTOs in a program decreased a program's clarity and one's ability to test and maintain it.

In practice the GOTO is permitted in structured programs so long as it is used as an exit from a module. When BASIC allows multiple statements per line, the form:

ln1 IF condition THEN GOSUB ln2 : GOTO ln1

is well suited as an iteration structure, even though it contains a GOTO. Many texts would have you overcomplicate code in order to maintain conformity to a standard that was established for COBOL. We feel that BASIC's concise coding can clarify rather than complicate the logic of a program.

The practice that should be avoided in BASIC is the use of GOTOs followed by line numbers to the point where the reader loses the program's continuity. If you need to get back to a previous statement, you might find a way to GOSUB to the set of statements above the GOTO, and replace it with a RETURN. For example, consider this little program that calculates the mean and standard deviation of a sample.

```
10 'FILENAME: "C10P1"
20 'FUNCTION: DEMONSTRATION OF POOR STRUCTURE
30 ' AUTHOR : JPG                    DATE:   9/79
40 I=1: S=0: S2=0
50 PRINT "NUMBER";I;: INPUT X
60 IF X<>0 THEN 80
70 GOTO 90
80 I=I+1: S=S+X: S2=S2+X*X: GOTO 50
90 M=S/I: PRINT "MEAN=";M
100 PRINT "STD. DEV.=";SQR((S2-S*M)/(I-1))
110 INPUT "ANOTHER SAMPLE";A$
120 IF A$<>"YES" THEN 10000
130 GOTO 40
10000 END
```

This program is a trivial example that is quite easy to follow, even if it has been written with as many GOTOs as possible. The first and most obvious step is to get rid of all simple GOTOs immediately after the IFs. The following program is an improvement.

```
10 'FILENAME: "C10P2"
20 'FUNCTION: BETTER STRUCTURE THAN PREVIOUS PROGRAM
30 ' AUTHOR : JPG                    DATE:   9/79
40 I=1: S=0: S2=0
50 PRINT "NUMBER";I;: INPUT X
60 IF X<>0 THEN I=I+1: S=S+X: S2=S2+X*X: GOTO 50
           ELSE I=I-1: M=S/I: PRINT "MEAN=";M:
                PRINT "STD. DEV.=";SQR((S2-S*M)/(I-1))
70 INPUT "ANOTHER SAMPLE";A$
80 IF A$="YES" THEN 40
10000 END
```

This is a much better program. It is clearer, even though its selection structures, the IF–THEN–ELSE statements, are rather complex.

Now consider the following program. Its single GOTO is a part of the UNTIL iteration structure. The only other references to line numbers are in the GOSUBs. Subroutine 60 is clearly the main subroutine, and subroutines 200, 300, and 400 perform the initialization, input, final calculations, and print-out functions.

```
10 'FILENAME: "C10P3"
20 'FUNCTION: SAME PROGRAM WITH GOOD STRUCTURE
30 ' AUTHOR : JPG                    DATE:   9/79
40 INPUT "DO YOU WANT STATISTICS (YES OR NO)";A$
50 IF A$="NO" THEN STOP
              ELSE GOSUB 60: GOTO 40
60 GOSUB 200       '  initialize variables
70 GOSUB 300       '  input data
80 GOSUB 400       '  calculate mean and std. dev.
90 GOSUB 500       '  print std. dev. and mean
100 RETURN
200 S=0: S2=0: M=0: RETURN ' initializing routine
210 '       subroutine for data input
300 INPUT "HOW MANY OBSERVATIONS";N
310 FOR I=1 TO N
320   PRINT "TYPE VALUE NO.";I;: INPUT X
330   S=S+X: S2=S2+X*X
340 NEXT I
350 RETURN
360 '     subroutine for calculations
400 M=S/N: SD=SQR((S2-S*M)/(N-1)): RETURN
410 '     subroutine to print results
500 PRINT "MEAN=";M
510 PRINT "STD. DEV.=";SD
520 RETURN
10000 END
```

At this point you may be wondering what advantage this program could possibly have over its simpler-looking predecessors. The key is in its ease of modification and growth potential. For example, if you wanted to calculate the range of the sample, you could alter the input subroutine to keep track of the lowest-seen and highest-seen values, and do so knowing that whatever you do there cannot affect the other modules. If you wanted to add a new feature, you could just add a new subroutine.

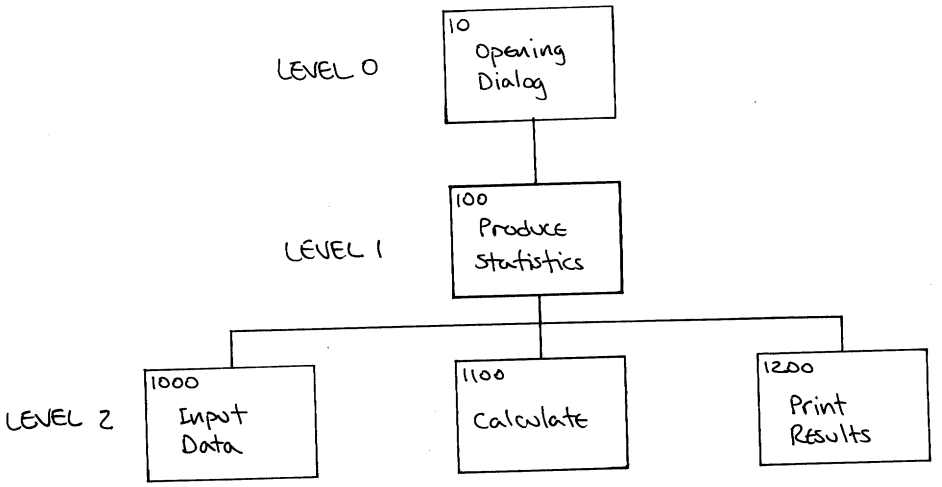
Top-down Programming

The entire subject of GOTO–less programming is but a facet of of the area of structured programming. However, it leads naturally into the area of program design, top-down programming, and modular programming. It is the segmentation of the logic of a program into its parts.

Top-down programming involves the overall design of a program into a series of modules arranged in a hierarchical order. That is, the primary objective of the program is incorporated into its first module or section, the secondary objectives are next, and the subordinate objectives are below those. The previous program is an example of top-down programming. There is a clearly identifiable main module and a number of subordinate modules, the subroutines.

A good way to visualize the design of a structured program is with a *hierarchy chart* or *structure chart*. This is a structure chart of program C10P3.

Structure Chart for Program C10P3

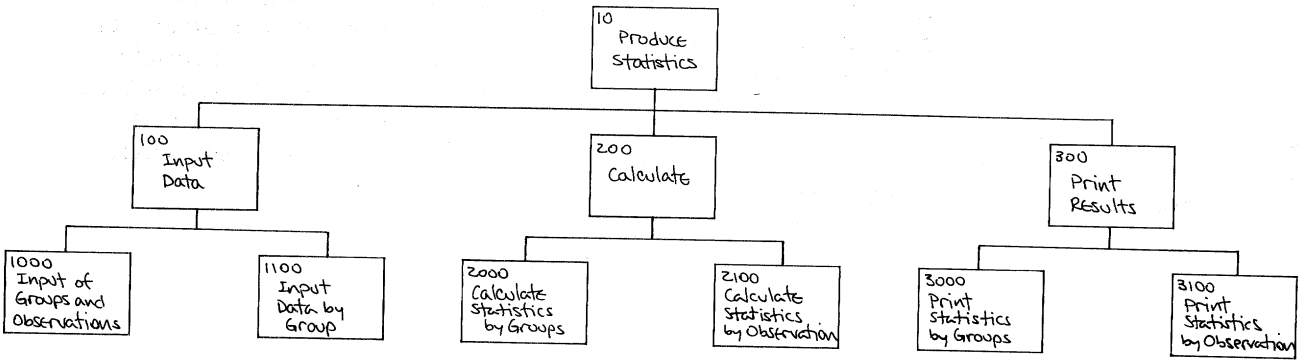


The structure chart shows all of the modules in the program and the relationships between these modules. Input and output functions are distinguished from processing functions. It is easily generated before any actual coding is done. In contrast to the symbolic flowchart, which is more of a representation of an existing program, the structure chart serves best as a design aid.

A major advantage of the structure chart over a programmer's informal phrase flow chart as a program design tool is the structure chart's capacity for modification and growth. Let us consider the previous statistics program as a kernel for a much larger system of programs for statistical analysis. Suppose you want to add these features:

1. Input of data by groups.
2. Each element of a group can have more than one observation.

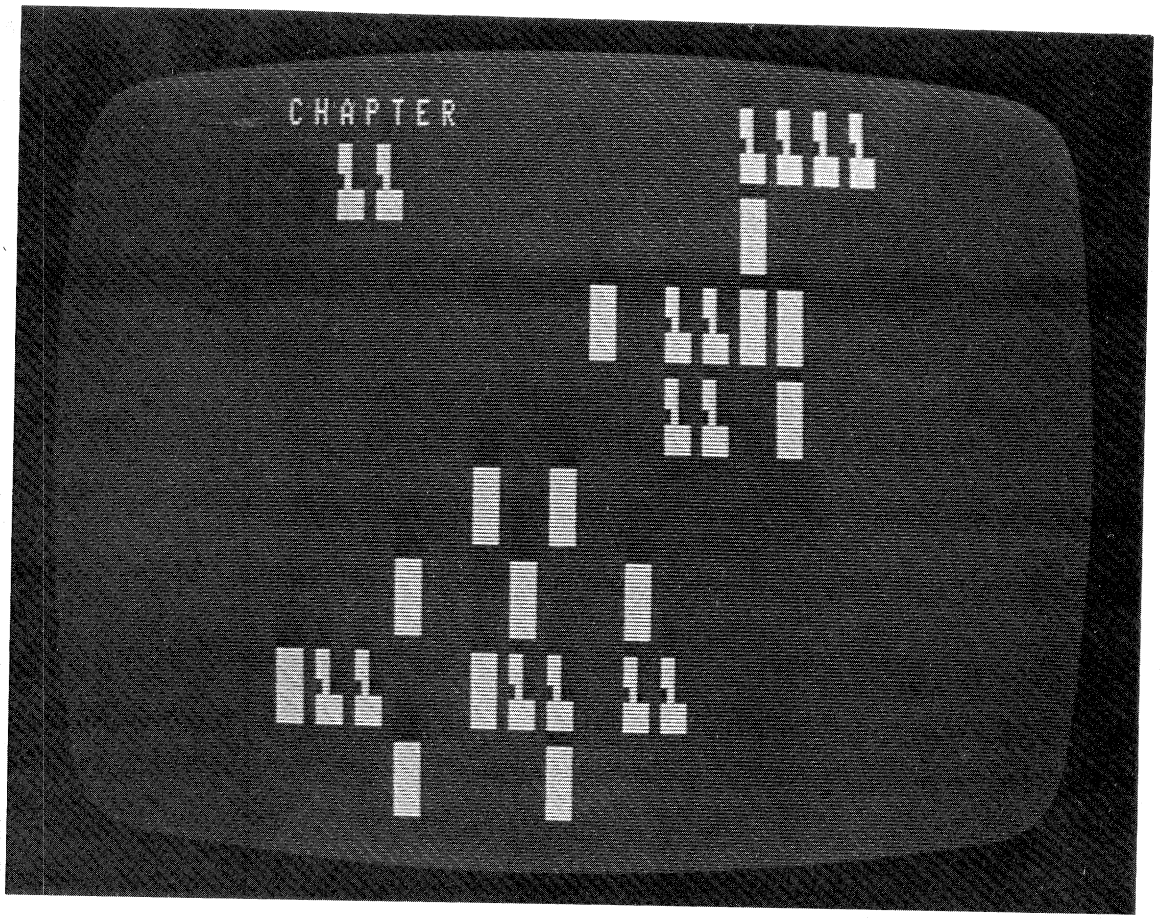
The new structure chart could be:



This major change was incorporated by adding another level to the existing structure chart. This added level describes the new modules that will be necessary to implement the changes. The programmer could write new subroutines, numbered in the 1000's, with no more change to the existing modules than some new GOSUBs in the level 1 modules.

The program planning and design aids we have shown in this chapter can greatly increase programmer productivity and enhance

program clarity. You can rewrite existing programs in a more structured style, and in so doing, very likely breathe new life into those programs. The chapter that follows will show you how to prepare associated documentation with your programs. It is the documentation that usually spells the difference between an easily usable program and marketable program, and one that no matter how fine it is technically, just doesn't seem to be popular with its users.



Documentation

Documentation is often considered to be the programmer's bane and burden, the cross to bear for writing good programs. It is unfortunate that this attitude exists, because it tends to reduce the amount of documentation that needs to be created in many programs.

In fact, documentation should be treated as the chef treats his or her pots, pans, and cookbooks. Accumulating the proper ingredients is only part of the preparation of a fine dish, just as coding the proper algorithms is only part of the production of a good program. A program must display its author's pride and skill in both its execution and its documentation.

In this chapter, we will discuss a wide variety of techniques that fall into either one of two broad categories: internal or external documentation.

Internal Documentation

As a class of techniques, internal documentation includes all methods of explaining a program's workings from within the program itself. You will see that it means more than a liberal dose of REM statements, although that is an important consideration.

Program Comments

The REM statement provides a method for the programmer to write a comment about a statement or program structure without altering the program's execution. The first few lines of any sizeable program should include the following information in the form of remarks:

1. Program ID—file name and brief statement of function.
2. Author ID—name and date of program creation.
3. Revision ID—name, date, and number of revision.

In many microcomputers, including the TRS-80, REM statements suffer the disadvantage of occupying valuable memory space. Many programmers keep two versions of their working programs: One is rich with comments and highly readable, and it is generally kept on file; the other is stripped of all REM statements, and used for execution only. When a change must be made in the program, the programmer lists the well documented program as an aid in locating the area of change. The appropriate changes are made in *both* programs, and when the undocumented version runs properly, the documented version is restored on file.

REM statements come in a variety of forms. The most familiar one is the single line that is dedicated solely to a remark.

```
10  REM       PROGRAM TO COUNT VIRUSES
.
.
.
500 '         SUBROUTINE TO CONVERT SPEED OF LIGHT
510 '             TO FURLONGS PER FORTNIGHT
.
.
.
3000 '        REMOVE UNWANTED RECORDS
.
.
.
```

Other REMs appear as last statements in multistatement lines.

```
.
.
.
50 A=A+1: REM          COUNT THIS VIRUS
.
.
.
550 V=2*X+R '          CALCULATE VENOUS FLOW
.
.
.
2250 X(I)=X(I)-N '     REDUCE VOLUME CALCULATED
```

A word of caution concerning remarks in a multistatement line: Take great care to avoid inserting a remark *between* two statements on the same line. It is easy to do this in Level II BASIC because the line's length can be up to 255 bytes, and the down-arrow (↓) provides a neat way to return the carriage and provide a line-feed without generating the /EN/ (hex 0D) character that marks the end of the line. But a REM between two statements on the same line makes all statements after the remark a part of the remark. For example, *don't* do this:

```
10 X=X+1: ' INCREMENT X: Y=Y+1: ' INCREMENT Y
```

The statement Y=Y+1 is seen by BASIC as a part of the remark following the statement

```
X=X+1
```

Right-justified or indented REM statements are often more easily distinguishable from the active BASIC statements. Consider the two following examples:

```
10 DIM X(100): CLS
20 INPUT "# OF GROUPS";NG              ' INPUT THE LIMITING
30 INPUT "GROUP SIZE";N                ' SIZES FOR THE SAMPLE
.
.
.
500 '                          SUBROUTINE FOR ANOVA CALCULATIONS
510 FOR I=1 TO NG
520 '                          TREAT EACH GROUP SEPARATELY
530 S(I)=0: S2(I)=0
.
.
.
```

If your printer prints lower case as well as upper case characters, you can use lower case for remarks. This technique is used in many examples in this book.

Windowboxes

A very distinctive form for remarks is called the *windowbox*. This form visually isolates titles, and is particularly effective for major program headings and structure and subroutine titles.

```
10 '          ********************************
20 '          *          FILE EDITING        *
30 '          ********************************
40 '
```

```
1000 '       ------------------------------
1010 '             SORT THE ARRAY
1020 '       ------------------------------
1030 '
```

```
2000 '                    *
2010 '                  *******
2020 '                ***********
2030 '              ****** INPUT ******
2040 '                ***********
2050 '                  *******
2060 '                    *
2070 '
```

```
1 ' ****************************************
2 ' *          G-E-N-E-A-L-O-G-Y           *
3 ' *   A MANAGEMENT SYSTEM FOR RECORDS    *
4 ' *         OF ANCESTRY AND KIN          *
5 ' *                  BY                  *
6 ' *            ALICE X. HOLEY            *
7 ' *                7/4/76                *
8 ' ****************************************
9 '
```

Blank Lines and Text Formatting

A lot of information about a program's structure and logic can be conveyed without words. In some cases, the insertion of a blank line (actually there's an apostrophe as a first character) is enough to demarcate one program structure from another. In other instances, a programmer can tell a lot about the program's actions by indenting certain lines.

Examples:

```
1000 ' ******************
1010 ' * FIELD RECORDS *
1020 ' ******************
1030 '
1040 FIELD 1, 255 AS X$
1050 '
1060 FIELD 2, 4 AS F1$, 4 AS F2$, 4 AS F3$,
              4 AS F4$, 4 AS F5$, 4 AS F6$,
              4 AS M$
1070 '
```

```
100 FOR I=1 TO NG                ' DO BY GROUPS
110    FOR J=1 TO N(I)           ' DO BY OBJECTS
120       -------
130       -------
140       -------
150    NEXT J
160    -------
170    -------
180 NEXT I
```

```
50 IF P<Q THEN X=X+1: L=SQR(M)
          ELSE J(I)=V
```

```
300 IF A<B THEN IF B<C THEN Q=V
                       ELSE Z=V
           ELSE X=V
```

Grouped Line Numbers

One of the most obvious signs of a programmer's attention to the clarity of the program's logic is the proper selection of line numbers to be grouped within structures. For example, during the initial stages of program design you should try to group the Level 0 structure, the main or driver portion of the program, within lines 1 to 999. Then you can reserve blocks of line numbers for various other structures: Lines numbered 1000 to 1999 for the first module of Level 1, 2000 to 2999 for the second module, and so on. Then the second level can be blocked into the ten thousands. This makes any modification to the program fairly simple.

A number of commercially available line renumbering programs can be very useful for grouping line numbers. It is possible to selectively renumber lines past a certain value with these programs. This allows you to modify one structure at a time, starting with a low level and working up.

External Documentation

Structure Charts

The first documentation of a program is a set of written notes, possibly a phrase flowchart. Its purpose is to organize on paper some of the structural requirements of a program. This should be replaced immediately with a general *structure chart* for the program. The structure chart can be modified as the program is composed and tested, but its overall design should remain essentially constant.

Symbolic Flowcharts

During the coding and testing phase of a program, it is often necessary to unravel the intricacies of an algorithm, or to create a new symbolic flowchart as the algorithm is coded. This form of external documentation is useful during coding to record visually the logic of a specific module, so that it can be modified without destroying its logic. Sometimes the lucky programmer has symbolic flowcharts already prepared from some other source. If not, the algorithm may be coded first from a phrase flowchart; then the symbolic flowchart is drawn to record the algorithm in some language-free form for future use.

Run Books

Once the program or system of programs is written and tested, it needs to be released to the user, who is most often someone totally unfamiliar with the code itself. The programmer must inform the user how to run the program, how to answer its questions, and what to do in case of specific anticipated failures. Some of these directions can be included in user prompts within the program. Some overall guiding directions can be given in a set of PRINT statements when the program begins to execute. But the most important directions, those that tell the user how to load the program and its ancillary files and how to get it running, cannot have internal directions. They must be written into an external document called a *run book*, or user manual. Run books range in size from a three-by-five printed card that can be pasted onto an obvious place near the computer to a full set of books that detail the programs' actions for an entire system.

Run books usually start with a section that tells the purpose of the program, followed with a section written in cookbook fashion on how to start execution of the program. Then various other sections can be included, such as how each module works, what the range of answers should be for certain inquiries, and what to do in case of a program or hardware failure. Programs that are produced for commercial purposes are often sold primarily on the basis of the quality of their run books.

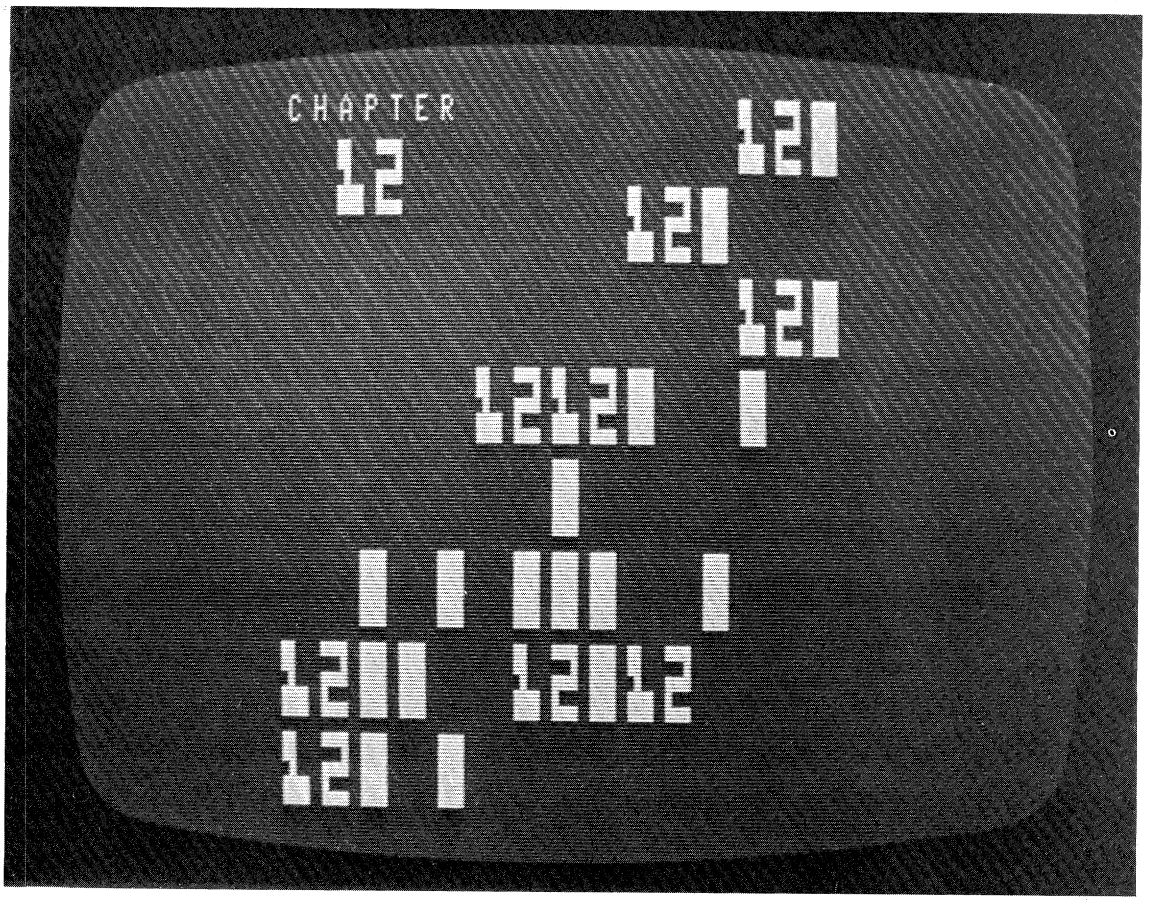
Help Files

Aside from charts and run books, there exists a third kind of external documentation which has gained increasing acceptance as timesharing systems have become more popular. This form of documentation is called the *help file*. A help file is a set of records accessible from within a program which gives information about a certain module of a system or porgram. This file is usually opened on demand from the user with a response of "HELP" instead of the normal response to a program's question. For example, suppose the user is presented with this menu and doesn't understand it.

```
        WORD PROCESSING SYSTEM

1.  GENERATE     RAW DOCUMENT FILE

2.  EDIT         RAW DOCUMENT FILE

3.  LIST         RAW DOCUMENT FILE

4.  RUN        FINAL DOCUMENT GENERATOR

5.  EDIT       FINAL DOCUMENT

6.  PRODUCE N FINAL DOCUMENTS

TYPE DIGIT OR 'HELP' OR 'HELP N':
```

If the user is not sure of the overall purpose and actions of the system, entry of the word "HELP" as an answer opens a file of text that is displayed on the screen. If there is doubt about how to run any one of the six listed modules, the user can type the word HELP immediately followed by the digit corresponding to the desired activity. For example, the response HELP 5 would open a file of text that explains how to list a raw document file.

Documentation and program structuring techniques, as discussed in this and previous chapters, represent an important aspect of the programmer's craft. Equally important is a working knowledge of the techniques that a programmer uses to structure and manage the various files that the programs access.



File Manipulation Techniques

Programs that deal with files must be thought out very carefully because they use data that is stored in some fairly permanent fashion. The files are not just an extension of the computer's high-speed memory. They provide a method of computer-accessible storage for masses of data that sometimes represent months or years of accumulated work.

The records in a computerized file are managed much as records in a manually maintained file in some office's steel cabinet. This chapter will discuss some of the various techniques of building, accessing, modifying, deleting, and sorting sequential and direct access files.

Building

The first phase in any dealings with computerized files is building them. They have to exist on the disk before they can be accessed in any way for any purpose.

Building Sequential Files

Sequential disk files, as you remember from the discussion in chapters 6 and 7, require special handling. String variables that are printed next to each other must be separated by a comma.

You may use the PRINT # statement either as part of a subroutine or within the main flow of the program. As an alternative method to the usual PRINT # statement followed by a series of variables names, you may build a single long string variable to be decomposed into its component variables on input.

A typical file building subroutine is shown in program C12P1, starting at line 1000.

```
10 'FILENAME: "C12P1"
20 'FUNCTION: DEMONSTRATE HOW TO BUILD A SEQUENTIAL FILE
30 ' AUTHOR : JPG                DATE: 6/80
40 '
50 'user will supply N sets of names, each set up to 50 names
60 CLEAR 2000: DEFINT A-Z: CLS: DIM N$(50)
70 INPUT "HOW MANY SETS OF NAMES";N
80 INPUT "HOW MANY NAMES IN EACH SET";S
90 'input array of names  into memory one at a time
100 FOR K=1 TO N
110   FOR I=1 TO S: PRINT "NAME";I;: INPUT N$(I): NEXT I
120   INPUT "WHAT IS FILE NAME";F$
130   GOSUB 1000                 'FILE ARRAY ONTO DISK
140 NEXT K: GOTO 1070
1000 'subroutine for buildins sequential file
1010 CLOSE: OPEN "O",1,F$              'F$ is file name
1020 'let user see what sets stored
1030 FOR I=1 TO S
1040   PRINT I; N$(I)
1050   PRINT #1, CHR$(34);N$(I);CHR$(34);
1060 NEXT I: CLOSE: RETURN
1070 END
```

Sometimes it is desirable to prepare the sequential output variables as a string separate from the actual file building subroutine. Program C12P2 below shows how a string array N$ can be used to hold logical records that contain both numeric and string variables. It uses the identical builder subroutine starting at line 1000 as in the previous program.

```
10 'FILENAME: "C12P2"
20 'FUNCTION: ANOTHER WAY TO BUILD SEQUENTIAL FILES
30 '  AUTHOR : JPG                    DATE: 6/80
40 '
50 CLEAR 2000: DEFINT A-Z: CLS: DIM N$(50)
60 'input array of composite strings
70 FOR I=1 TO 50: PRINT "NAME ('END' TO STOP)";I;
80   INPUT A$
90     IF A$="END" THEN N=I-1: GOSUB 1000: GOTO 1080
100          'accept rest of data only if not end
110   INPUT "AGE";A: INPUT "WEIGHT";W
120   INPUT "SEX (M OR F)";S$: INPUT "PT SCORE";PT
130         'build composite string
140   N$(I)=A$+";"+S$+";"+STR$(A)+STR$(W)+STR$(PT)
150 NEXT I
160 PRINT "ONLY 50 STRINGS ALLOWED IN THIS BUILDER"
170 PRINT "YOUR 50 ENTRIES WILL BE SAVED."
180 N=50: GOSUB 1000: GOTO 1080
1000 'sequential file building subroutine
1010 'N = number of logical records
1020 CLOSE: OPEN "O",1,F$      'F$ is name of file
1030 'let user see what is stored
1040 FOR I=1 TO N
1050   PRINT I;N$(I)
1060   PRINT #1, CHR$(34); N$(I); CHR$(34);
1070 NEXT I: CLOSE: RETURN
1080 END
```

Building Direct Access Files with Hash Addressing

The TRS-80 disk system allows two basically different methods to build direct access files. The first method is universally permissible on any computer system with direct access files: You PUT the records to the file one at a time in sequence, with the first PUT placing information on record #1, and the Nth PUT placing information on record #N. The second method of direct access file building is not always available on a given computer. It allows you to PUT a record into the Ith position of the file without having written any of records 1 through I-1. This latter method makes *hash addressing* easy to implement. A hash address is a calculated record number that is generated from a record's *key*, that part of the record which uniquely identifies it. Since the first method was amply demonstrated in chapter 8, we will include here an example of the second technique.

Suppose you want to build a file of names in which the names are the unique keys. The following are the assumptions and conditions for the program:

1. The number of names is exactly 45. They are to be found in a sequential file generated by the merge operation described in chapter 7.
2. There is room for a direct access file of up to 75 records.

3. The *key* of the record is the name itself. That is, once the direct access file is built, the name will be used to access the proper record directly.
4. The record number (the record's position on the file) will be calculated by the program as a unique number between 1 and 45 inclusive. If one name happens to generate the same record number as another name, the second record will be placed in the *overflow area*, which consists of positions 46 through 75 in the file.

Figure 12.1 shows a structure chart for this direct file builder program C12P3.

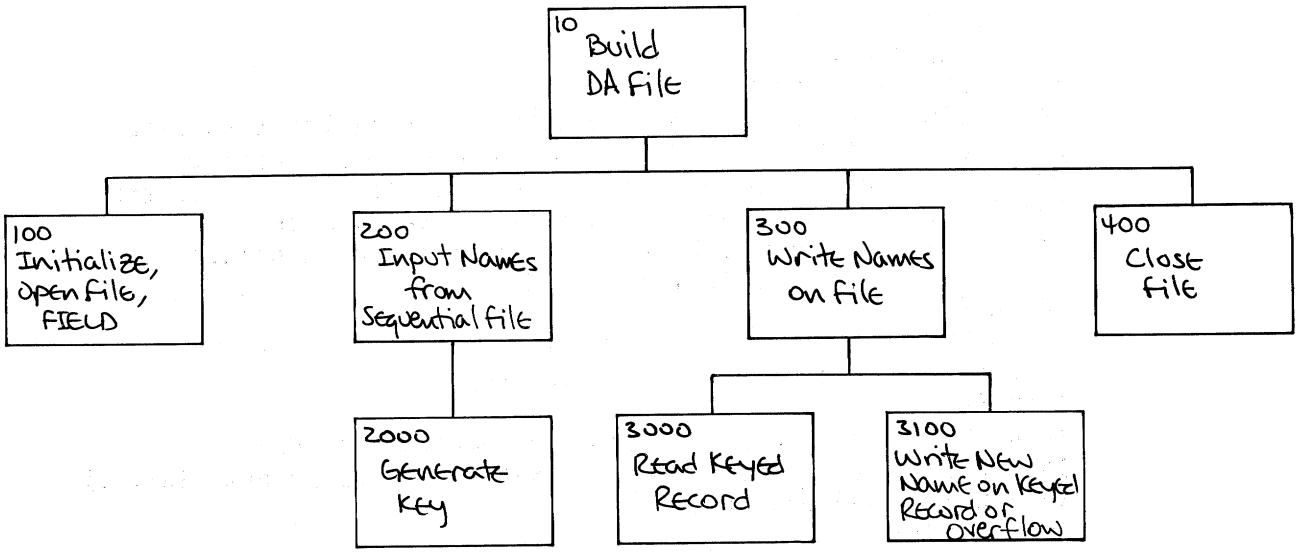


Figure 12.1 Structure Chart for Program C12P3

```
10 'FILENAME: "C12P3"
20 'FUNCTION: DIRECT ACCESS FILE BUILDER
30 ' AUTHOR : JPG                DATE: 6/80
40 '
50 CLEAR 2000: CLS
60 'set name of sequential file used for input
70 INPUT "what is input (sequential) file name";F1$
80 OPEN "I",1,F1$
90 'set name of direct access file used for output
100 INPUT "what is output (direct access) file name";F2$
110 OPEN "R",2,F2$: FIELD 2, 2 AS NN$, 30 AS NA$
120 'NN$ is record number, NA$ is stored name
130 'set up loop to read all 45 records from input file
140 LPRINT "SA$  name";
150 LPRINT TAB(30); "hash   rec. contents";
160 LPRINT TAB(52); "written on"
170 FOR I=1 TO 45
180   INPUT #1,N$
190 LPRINT I; TAB(5); N$;
200   GOSUB 1000        'generate hash address based on name
210   GOSUB 2000        'fetch the hashed address record
220 '   if record not on file (R<>N) write rec. in prime area
230 '   if on file (R=N) write rec. in overflow area
240     IF R=N THEN GOSUB 4000   'set overflow address"
250   GOSUB 3000      'write record
260 NEXT I
270 GOTO 10000
1000 '          generate a hash address
1010 S=0
1020 FOR J=1 TO LEN(N$): S=S+ASC(MID$(N$,J,1)): NEXT J
1030 K=S/45: N=(K-INT(K))*45: N=INT(N+1.1) 'to nearest int. +1
1040 LPRINT TAB(30); N;
1050 RETURN
2000 '          set hashed record
2010 GET 2,N: R=CVI(NN$)
2020 IF R<>N THEN NB$="garbage"
             ELSE NB$=LEFT$(NA$,10)
2030 LPRINT TAB(35); R; TAB(40); NB$;
2040 RETURN
3000 '          write record
3010 LSET NA$=N$: LSET NN$=MKI$(N): PUT 2,N
3020 LPRINT TAB(59);N
3030 RETURN
4000 '          generate overflow area address
4010 FOR J=46 TO 75
4020   GET 2,J: JJ=CVI(NN$)
4030     IF JJ=J THEN NEXT J   'this record is occupied
4040   N=J                     'this is the one
4050 LPRINT TAB(53); "oflow";
4060 RETURN
10000 CLOSE: END
```

SA#	name
1	ALIQUOT ALICE
2	ANOMALY ANNABELLE
3	ANOMALY ANTHONY
4	ASININE ARNOLD
5	BANDERSNATCH FRUMIOUS
6	BELLER ELLERY
7	CUERVO CERVESA
8	ENEMA ANOMIE
9	FISTULA FELICITY
10	GABLE MARK
11	GNASHGNAT NATHANIEL
12	GROMMET ANDROMEDA
13	HARSHBARGER HERSCHEL
14	HEGIRA IRA
15	LANCASTER BART
16	LANCHESTER ELSIE
17	LAVORIS MORRIS
18	LOOSELIPS LINDA
19	MERGER MARGINAL
20	MINDERBINDER MILO
21	MONACO MONICA
22	MUSHMOUTH MUNGO
23	PARSLEY PELVIS
24	PARVENU MARVIN
25	PATELLA PETER
26	PONTIAC CARLO
27	POWER CYCLONE
28	RUBELLA ELLA
29	SALIVA TELLY
30	SIDDHARTHA GHAUTAMA
31	SITHLE SAMANTHA
32	SITHLE SIBILANT
33	SITHLE SIMEON
34	SITHLE SYBIL
35	UVULA URSULA
36	VERMIFORM VERNON
37	WEEDEATER WANDA
38	WHEREWITHALL WILLY
39	WOMBAT WILLY
40	WOMBAT WILLY
41	ZOTZOT XAVIER
42	ZOTZOT ZELDA
43	ZOTZOT ZEPPO
44	ZOTZOT ZIGGY
45	ZOTZOT ZIPPO

hash	rec. contents		written on
26	0 garbage		26
34	0 garbage		34
27	8370 garbage		27
10	-6683 garbage		10
14	10981 garbage		14
32	13088 garbage		32
32	32 BELLER ELL	oflow	46
22	13618 garbage		22
45	-6683 garbage		45
4	-6683 garbage		4
10	10 ASININE AR	oflow	47
8	8250 garbage		8
37	-24031 garbage		37
10	10 ASININE AR	oflow	48
9	-6683 garbage		9
23	-6683 garbage		23
18	12628 garbage		18
11	-13569 garbage		11
35	4215 garbage		35
6	-13569 garbage		6
17	17744 garbage		17
12	-14533 garbage		12
9	9 LANCASTER	oflow	49
4	4 GABLE MARK	oflow	50
32	32 BELLER ELL	oflow	51
28	8224 garbage		28
10	10 ASININE AR	oflow	52
28	28 PONTIAC CA	oflow	53
20	-6683 garbage		20
44	-6683 garbage		44
44	44 SIDDHARTHA	oflow	54
8	8 GROMMET AN	oflow	55
4	4 GABLE MARK	oflow	56
22	22 ENEMA ANOM	oflow	57
6	6 MINDERBIND	oflow	58
30	8250 garbage		30
23	23 LANCHESTER	oflow	59
41	-6683 garbage		41
37	37 HARSHBARGE	oflow	60
37	37 HARSHBARGE	oflow	61
12	12 MUSHMOUTH	oflow	62
7	14896 garbage		7
37	37 HARSHBARGE	oflow	63
33	14936 garbage		33
41	41 WHEREWITHA	oflow	64

Accessing

The process of accessing a record involves reading the record from the direct access or sequential access file, and storing the information on the record into memory.

Accessing Sequential Files

Sequential access files can be loaded into memory one record at a time, or they can be copied into an array in their entirety. The GEOGRAPH program uses the latter technique to fill the record pointers, location names, and location descriptors.

Accessing Direct Access Files

Direct access files may be read either sequentially one record at a time from the first to some desired key, or directly by accessing a specific record according to its calculated hash address. The two programs that follow show both techniques.

```
10 'FILENAME: "C12P4"
20 'FUNCTION: DIRECT ACCESS IN SEQUENTIAL ORDER
30 ' AUTHOR : JPG                DATE: 6/80
40 '
50 CLEAR 2000: CLS
60 INPUT "Existing direct access file name";F2$
70 OPEN "R",2,F2$: FIELD 2, 2 AS NN$, 30 AS NA$
80 'read all records in order, print blank line when no record
90 LPRINT "All records on file": LPRINT
100 FOR I=1 TO 45
110   GET 2,I: N=CVI(NN$)
120 '  if N is not I then record position is not used
130   IF N=I THEN LPRINT I;NA$
            ELSE LPRINT I;" "
140 NEXT I
150 'read overflow area
160 FOR I=46 TO 75
170   GET 2,I: N=CVI(NN$)
180     IF N=I THEN LPRINT I;NA$
                ELSE LPRINT I;" "
190 NEXT I
10000 CLOSE: END
```

```
All records on file

 1
 2
 3
 4 GABLE MARK
 5
 6 MINDERBINDER MILO
 7 ZOTZOT ZELDA
 8 GROMMET ANDROMEDA
 9 LANCASTER BART
 10 ASININE ARNOLD
 11 LOOSELIPS LINDA
 12 MUSHMOUTH MUNGO
 13
 14 BANDERSNATCH FRUMIOUS
 15
 16
 17 MONACO MONICA
 18 LAVORIS MORRIS
 19
 20 SALIVA TELLY
 21
 22 ENEMA ANOMIE
 23 LANCHESTER ELSIE
 24
 25
 26 ALIQUOT ALICE
 27 ANOMALY ANTHONY
 28 PONTIAC CARLO
 29
 30 VERMIFORM VERNON
 31
 32 BELLER ELLERY
 33 ZOTZOT ZIGGY
 34 ANOMALY ANNABELLE
 35 MERGER MARGINAL
 36
 37 HARSHBARGER HERSCHEL
 38
 39
 40
 41 WHEREWITHALL WILLY
 42
 43
 44 SIDDHARTHA GHAUTAMA
 45 FISTULA FELICITY
```

```
46 CUERVO CERVESA
47 GNASHGNAT NATHANIEL
48 HEGIRA IRA
49 PARSLEY PELVIS
50 PARVENU MARVIN
51 PATELLA PETER
52 POWER CYCLONE
53 RUBELLA ELLA
54 SITHLE SAMANTHA
55 SITHLE SIBILANT
56 SITHLE SIMEON
57 SITHLE SYBIL
58 UVULA URSULA
59 WEEDEATER WANDA
60 WOMBAT WILLY
61 WOMBAT WILLY
62 ZOTZOT XAVIER
63 ZOTZOT ZEPPO
64 ZOTZOT ZIPPO
65
66
67
68
69
70
71
72
73
74
75
```

```
10 'FILENAME: "C12P5"
20 'FUNCTION: DIRECT ACCESS BY HASH ADDRESS CALCULATION
30 ' AUTHOR : JPG                DATE: 6/80
40 '
45 CLEAR 2000: CLS
50 INPUT "Existing direct access file name";F2$
60 OPEN "R",2,F2$: FIELD 2, 2 AS NN$, 30 AS NA$
70 'set name from user
80 INPUT "name ('end' to stop)";N$
85 LPRINT "name sought is ";N$;
90   IF N$="end" THEN 10000
                 ELSE GOSUB 1000  'set hash address
100 GET 2,N: L=LEN(N$)
120   IF N$=LEFT$(NA$,L) THEN LPRINT TAB(40); "on record #";N:
                                GOTO 70
                              ELSE GOSUB 2000   'read oflow area
140   IF R<>0 THEN LPRINT TAB(40); " found on overflow #";R
              ELSE LPRINT TAB(40); "not on file"
150 GOTO 70
1000 '                  generate hash address
1010 S=0
1020 FOR I=1 TO LEN(N$)
1030   S=S+ASC(MID$(N$,I,1))
1040 NEXT I
1050 K=S/45: N=(K-INT(K))*45: N=INT(N+1.1)
1060 RETURN
2000 '                  read overflow area
2010 FOR J=46 TO 75
2020   GET 2,J
2030     IF N$=LEFT$(NA$,L) THEN R=J: GOTO 2100
                    ELSE NEXT J
2040 R=0       'not on file
2100 RETURN
10000 CLOSE: END
```

```
name sought is MINDERBINDER MILO          on record # 6
name sought is SITHLE SIBILANT             found on overflow # 55
name sought is MERGER MARGINAL            on record # 35
name sought is SUBLIMINAL SAMUEL          not on file
name sought is UVULA URSULA                found on overflow # 58
name sought is end
```

Modifying

Direct access files have a major advantage over sequential access files when you have to modify just one record. Consider the process that is required when modifying a sequential file:

1. Open old file for input. Open new file for output.
2. Read all records on old file and write out on new file until proper record is found.
3. Change the record that needs modification.
4. Write out changed record on new file.
5. Read the rest of the records from the old file, and write them out onto the new file.

When one record of a direct access file needs modification, it can be done without reading any other records, and without having to create a new file.

Deleting

Deletion is the file operation that results in an unwanted record being culled or purged, and it is normally performed in two steps. The first step is to mark a record for future deletion. Numerous records can be marked, perhaps 10 or 20 percent of a file's contents are records marked for deletion. The second step is to rebuild the file excluding the marked records.

Marking records for deletion can be done in one of two ways:

1. A specific numeric value that appears in every record, such as record number or age, can be assigned a negative value or an unlikely value, such as -32767.
2. A string can be replaced with a flag or marker, signifying the fact that it has been marked for deletion.

Deleting the records is accomplished by rewriting the file one record at a time, omitting the flagged records.

Sorting

Of all the operations that can be done on a file, its arrangement into some predetermined ascending or descending order seems to be one of the most common and troublesome. You can sort a file using any of three basic approaches:

1. In-memory sort
   a. Read the file into an array in memory.
   b. Sort the array in memory.
   c. Write the memory array back out onto the file.
2. Record-by-record sort
   Rearrange the records on file by reading them two at a time, and switching their positions when necessary.
3. Sort-merge
   a. Read the file a portion at a time into an array in memory, sort each portion in memory, and create a series of new sub-files.
   b. Merge the sub-files into a single large file.

In all three approaches, the key operation is the sorting operation itself, which rearranges the records into the desired order. The choice of sorting algorithm to do this is critical to the overall

speed of the operation. We will discuss this choice of algorithms shortly, but first you must consider which type of program you should use: In-memory, record-by-record, or sort-merge.

The in-memory sort is by far the fastest, *if you have the memory space.* The array in memory must be able to hold the entire file, and in the case of 255-byte logical records that occupy the entire physical record, this could mean that the file would have to be quite small. Consider the case where the sorting program itself, not including the array, occupies 2000 bytes. In a 16K disk system, the memory that is left for the sort is only about 3800 bytes. In a 32K system, the space is a more respectable 20,000 bytes. And in a maximum system of 48K, the total space free for the array is roughly 36,400 bytes. Study of the table 12.1 will give you a feel for the effect of memory size on the computer's capacity to sort arrays of records. Table 12.1 points out a rather disturbing fact: If you have a 16K system, you cannot store more than 14 physical records of 255 bytes each in the computer's free memory. However, all is not lost. If your file is longer than the 14 maximum records, it can still be sorted. This is because it is a direct access file and it can be sorted record-by-record. However, this second technique is between 100 and 1000 times slower than an in-memory sort!

We suggest that whenever the file is too large to fit into memory to be sorted there, you should use a combination of the in-memory and record-by-record sorts, which is the third technique, the sort-merge. We have discussed and demonstrated the sort-merge in chapter 7 when it was shown operating on a sequential file.

Logical Record Size in Bytes	System Size (free memory) 16K (3800)	32K (20,000)	48K (36,400)
255	14	78	142
128	29	156	284
64	59	312	568
32	118	625	1137
16	237	1250	2275
8	475	2500	4550
4	950	5000	9100

Table 12.1 Number of Records That Can Be Sorted in Memory

Sorting Algorithms

Aside from the medium on which the array to be sorted is located, which is either memory or disk, the single most important factor that determines the speed of the overall operation is your choice of sorting algorithm. The choice is rather large and varied, but there are some algorithms that are clearly better than others.

Exchange Sorts

The class of sorting algorithms known as exchange sorts includes the bubble sort, delayed exchange sort, and direct exchange sort. They are all very slow and unfortunately are often found in many otherwise good systems of programs.

Binary Sorts

The better sorting programs all use binary sorts of one form or another. These algorithms rely on the fact that one can sort two sets of 50 elements with subsequent merging faster than one sort of 100 elements; one can sort twenty sets of five elements with subsequent merging faster than five sorts of twenty elements; and one can sort 50 sets of two elements with subsequent merging faster than 25 sets of 4 elements. It is the comparison and subsequent rearrangement, or switching, of the elements that takes the time, and not the merging.

We will show you two recommended methods for sorting: the Shell (really, a variation called the Shell-Metzner) and the Quicksort (one word). The Shell-Metzner sort was used previously in the Sequential File Sort-Merge program in chapter 7. We will not explain how these two sorts work, but we will show each of the sorts in two forms: Symbolic flowchart and BASIC subroutine. For those of you who are curious about the techniques of sorting and wish to study how they work, we suggest that you investigate Knuth's Volume 3 of the *Art of Computer Programming: Sorting and Searching.*

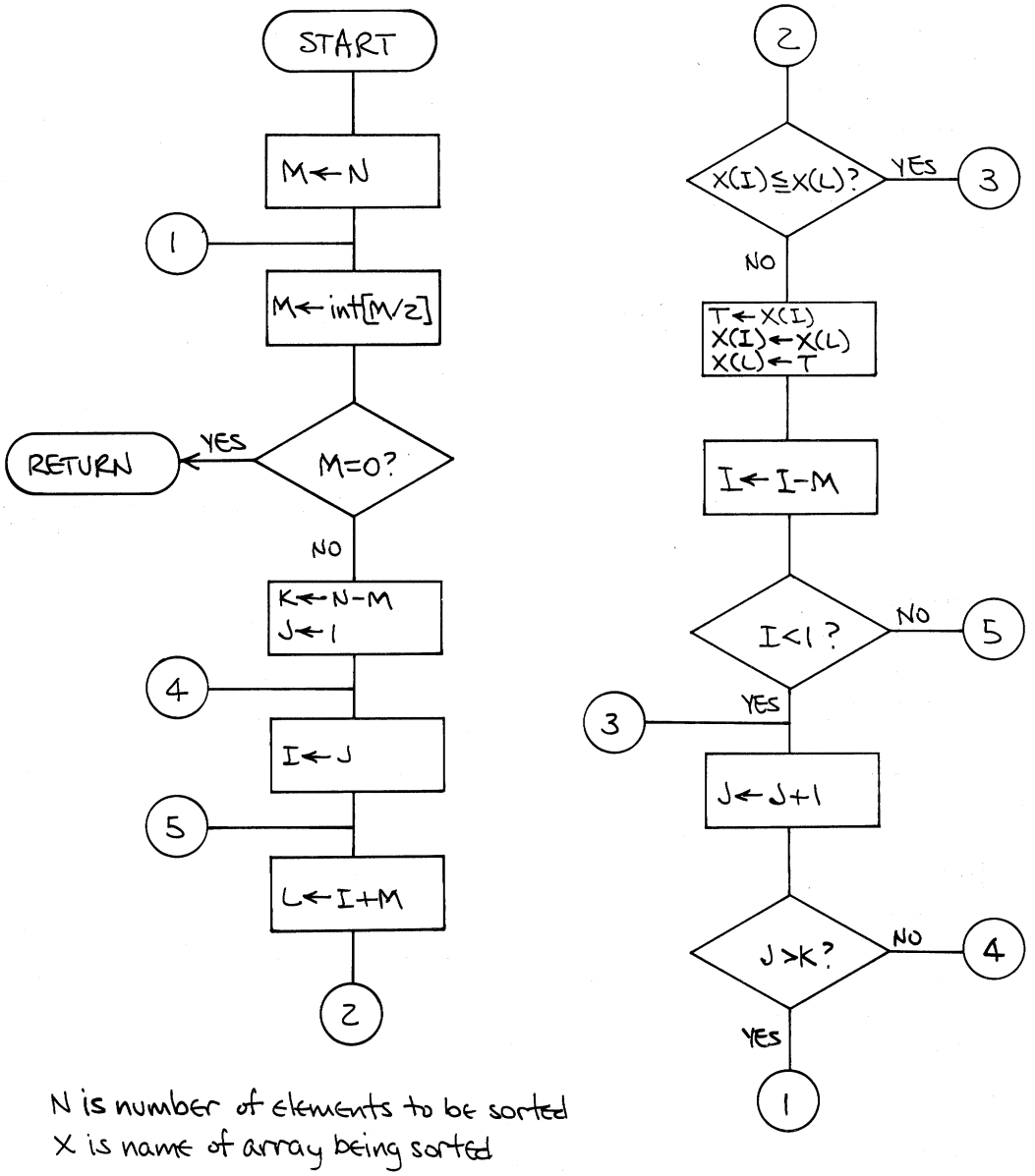


Figure 12.2 Shell-Metzner Flowchart

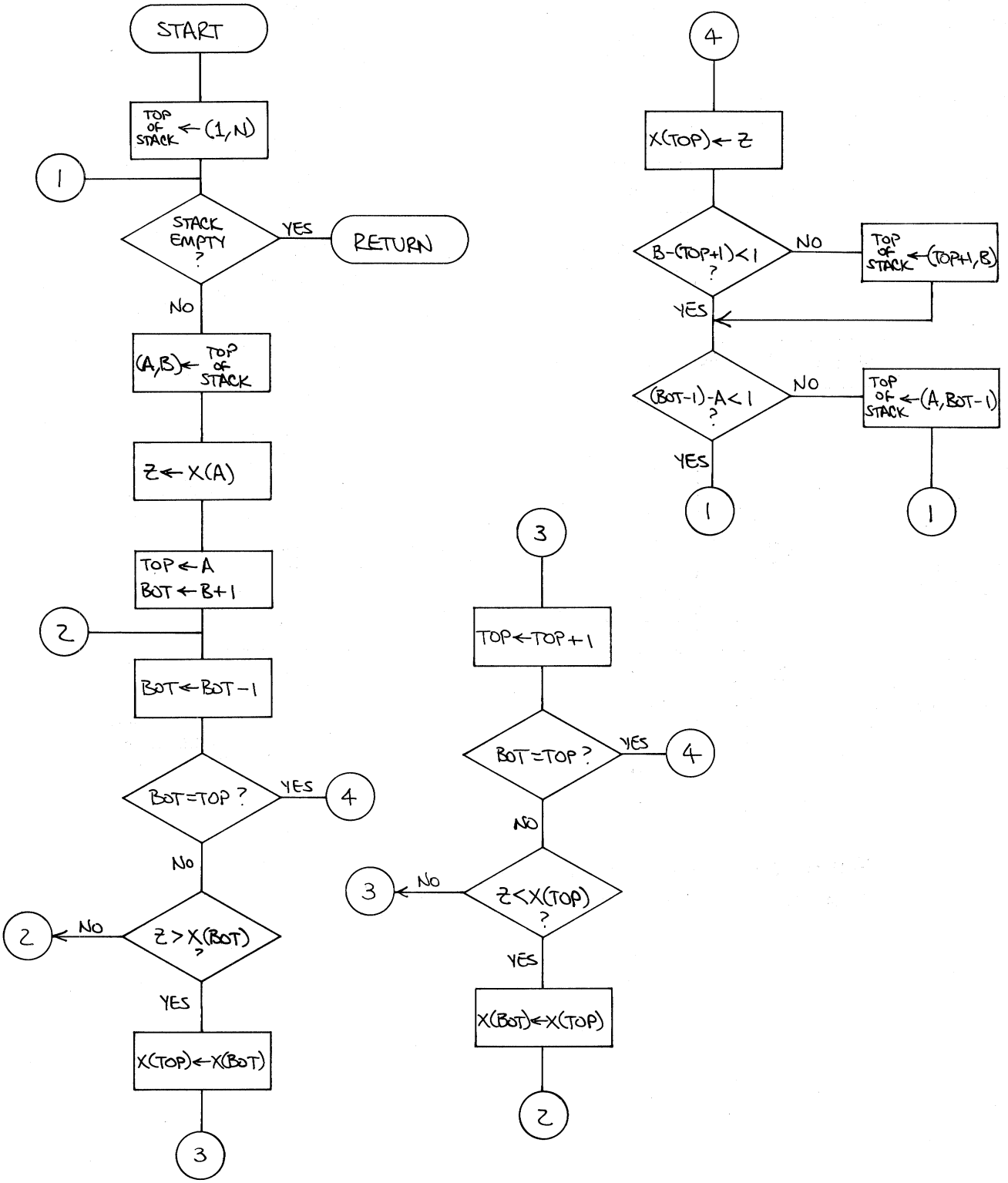


Figure 12.3 Quicksort Flowchart

Program C12P6 demonstrates three sorts: exchange, Shell-Metzner, and Quicksort. The program times the sorts, so you can get a sense of their relative efficiencies.

```
10 'FILENAME: "C12P6"
20 'FUNCTION: TO DEMONSTRATE SORTING ROUTINES
30 ' AUTHOR : JPG                    DATE: 7/80
40 '
50 CLEAR 300
60 DEFINT A-Z: DIM X(1000), STK(700)
70 INPUT "how many numbers to sort";N
80 FOR I=1 TO N: X(I)=RND(900)+99: NEXT I
90 PRINT "choose sort": PRINT "1  exchange"
100 PRINT "2  Shell-Metzner": PRINT "3  Quicksort"
110 INPUT "which one";C
120 GOSUB 160: B$=TIME$
130 ON C GOSUB 1000, 2000, 3000
140 C$=TIME$: GOSUB 160
150 PRINT C$: PRINT B$: GOTO 70
160 FOR I=1 TO N: PRINT X(I);: NEXT I: PRINT: RETURN
1000 '                                  **exchange sort**
1010 FOR I=1 TO N-1
1020   FOR J=I+1 TO N
1030     IF X(I)>X(J) THEN T=X(I): X(I)=X(J): X(J)=T
1040   NEXT J
1050 NEXT I
1060 RETURN
2000 '                              **Shell-Metzner sort**
2010 M=N
2020  M=INT(M/2)
2030    IF M=0 THEN 2120
2040    K=N-M: J=1
2050    I=J
2060    L=I+M
2070      IF X(I)<=X(L) THEN 2100
2080      T=X(I): X(I)=X(L): X(L)=T: I=I-M
2090        IF I>=1 THEN 2060
2100    J=J+1
2110      IF J<=K THEN 2050
                  ELSE 2020
2120 RETURN
```

```
3000                                              **Quicksort**
3010 P=0: R=P+P: STK(R+1)=1: STK(R+2)=N: P=P+1
3020   IF P=0 THEN 3150
3030 P=P-1: R=P+P: A=STK(R+1): B=STK(R+2)
3040 Z=X(A): TP=A: BT=B+2   'TP is top, BT is bottom
3050 BT=BT-1
3060   IF BT=TP THEN 3110
3070     IF Z<=X(BT) THEN 3050
                     ELSE X(TP)=X(BT)
3080     TP=TP+1
3090     IF BT=TP THEN 3110
3100       IF Z>=X(TP) THEN 3080
                       ELSE X(BT)=X(TP): GOTO 3050
3110 X(TP)=Z
3120   IF B-TP>=2 THEN R=P+P: STK(R+1)=TP+1:
                       P=P+1: STK(R+2)=B
3130   IF BT-A>=2 THEN R=P+P: STK(R+1)=A:
                       P=P+1: STK(R+2)=BT-1
3140 GOTO 3020
3150 RETURN
10000 END
```

Table 12.2 below summarizes a few observed sorting times using program C12P6.

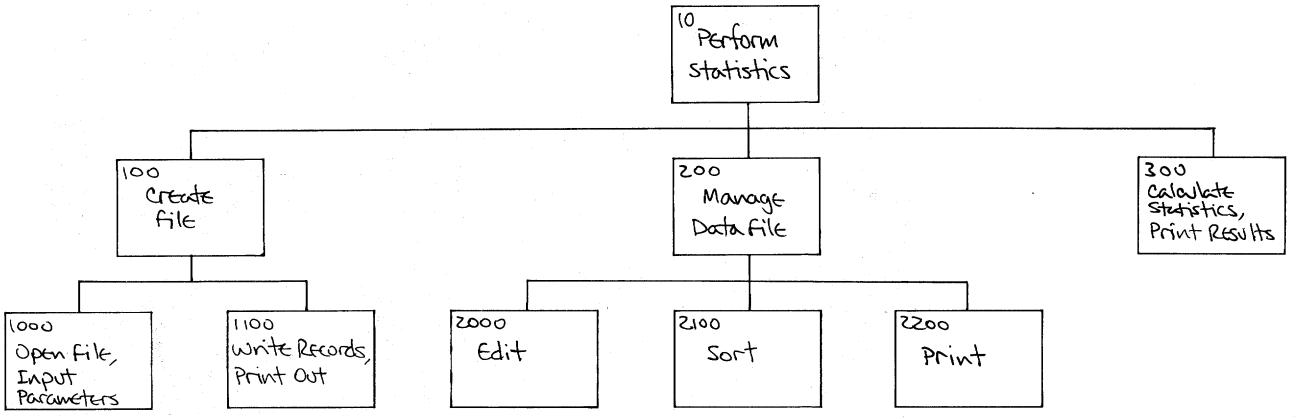
N	"	Exch.	S-M	Q
20	"	5	5	5
50	"	24	16	16
100	"	94	33	29
200	"	382	93	73
500	"	2217	303	184

Table 12.2 Sort Times in Seconds

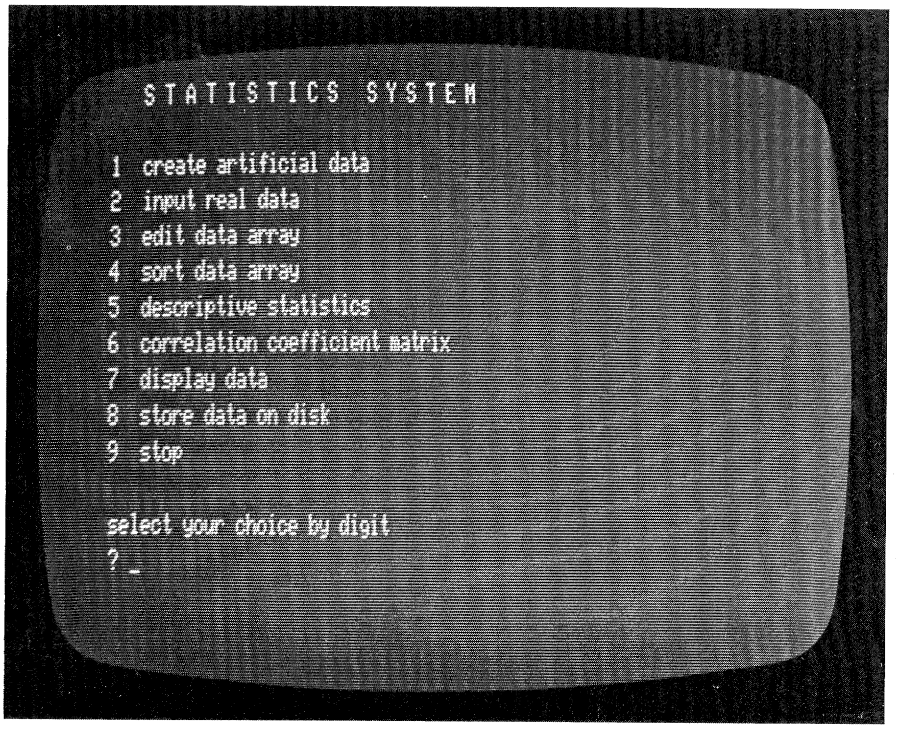
Direct Access File Statistics Program

Program C12P7 is included here to show the techniques that have been explained in this chapter. We have selected statistics as an area for data management because its records are often created, edited, deleted, and sorted, as well as having various statistical operations performed on them.

Structure Chart for Program C12P7, Statistics System



The user of this program selects an activity from a menu that is displayed by the program's first module. The main activities are independent of each other, but subordinate to the main module. The menu display looks like this:



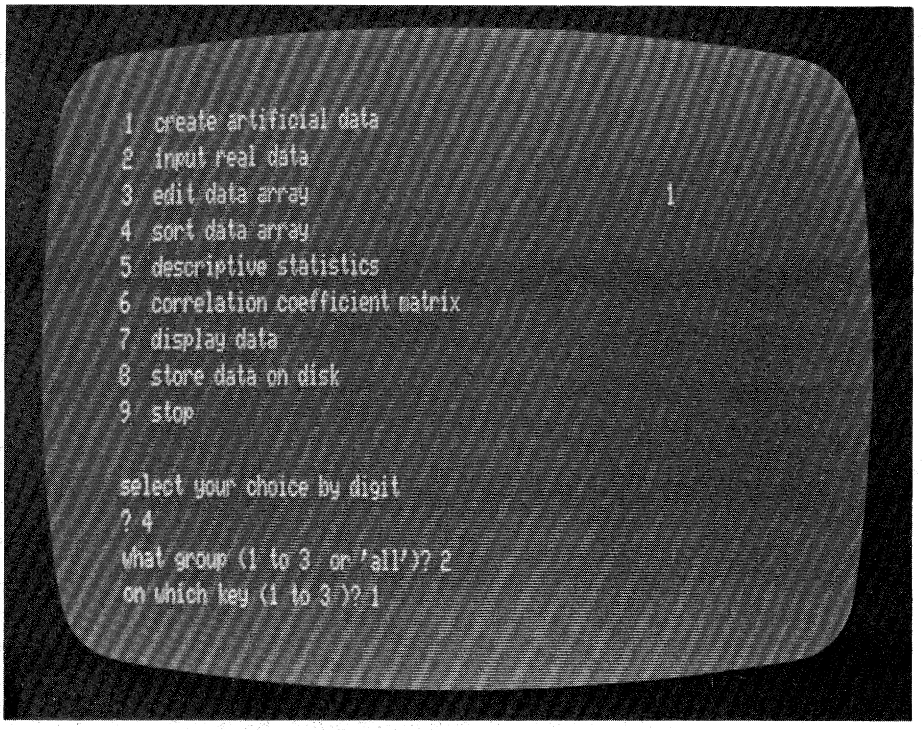

```
10 'FILENAME: "C12P7"
20 'FUNCTION: GENERAL PURPOSE STATISTICS PACKAGE
30 ' AUTHOR : JPG                DATE: 6/80
40 '
50 CLEAR 20000: DEFINT I-L
60 'maximum of 3 groups of observations,
70 '           4 variables per observation
80 '           20 observations, or subjects per group
90 ' values  , sums , sums of sqs,, sqd, sums, corrs,
100 DIM X(3,4,20),S(3,4),S2(3,4),S3(4),R(3,4,4)
110 CLS
120 '                    display menu
130 PRINT "   S T A T I S T I C S   S Y S T E M": PRINT
140 PRINT "1  create artificial data"
150 PRINT "2  input real data"
160 PRINT "3  edit data array"
```

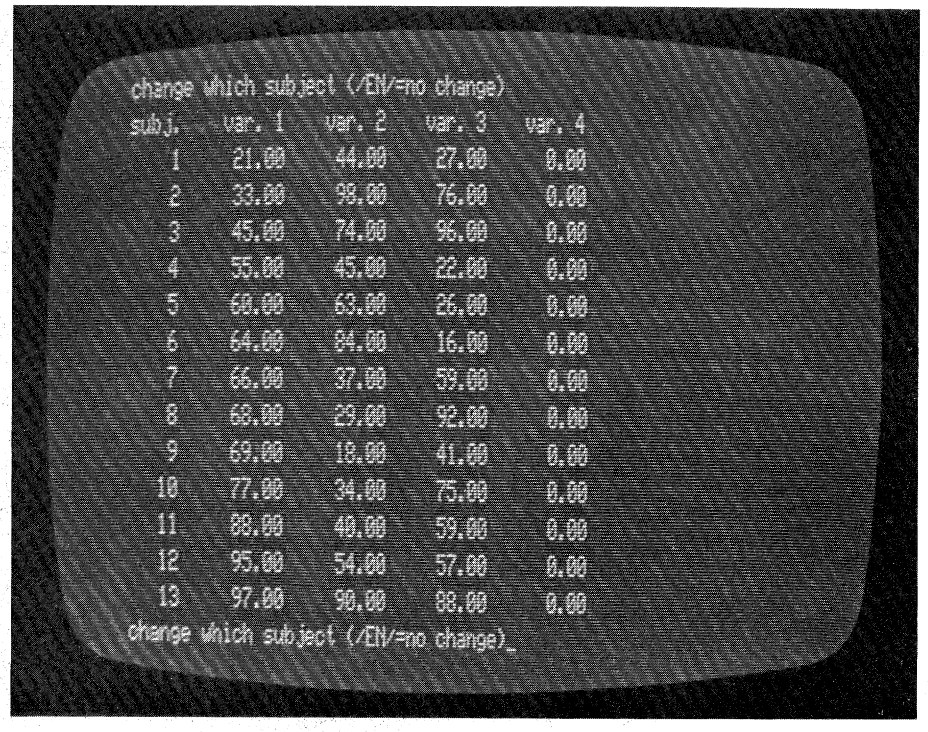
```
170 PRINT "4  sort data array"
180 PRINT "5  descriptive statistics"
190 PRINT "6  correlation coefficient matrix"
200 PRINT "7  display data"
210 PRINT "8  store data on disk"
220 PRINT "9  stop"
230 PRINT: PRINT "select your choice by digit"
240 INPUT A$: SP=VAL(A$) 'SP for subroutine pointer
250 IF SP<1 OR SP>9 THEN 130
260 ON SP GOSUB 1000,2000,3000,4000,5000,
                6000,7000,8000,9000
270 GOTO 110
280 '    u t i l i t y   s u b r o u t i n e s
290 '                    change all LPRINTs to PRINTs
300 POKE 16422,88: POKE 16423,4: RETURN
310 '                    restore all LPRINTs to LPRINTs
320 POKE 16422,141: POKE 16423,5: RETURN
330 '                    change all PRINTs to LPRINTs
340 POKE 16414,141: POKE 16415,5: RETURN
350 '                    restore all PRINTs to PRINTs
360 POKE 16414,88: POKE 16415,4: RETURN
370 '                             prompt and screen hold
380 INPUT "/EN/ to continue"; A$: RETURN
390 '                          Load data file from disk
400 INPUT "file name";F$
410 '       set number of groups and variables first
420 OPEN "I",1,F$: INPUT #1, NG, NV
430 '          set number of subjects per group next
440 FOR K=1 TO NG: INPUT #1, NM(K): NEXT K '# per gp
450 FOR K=1 TO NG          'first by group,
460   FOR I=1 TO NM(K)     'then by subject in group,
470     FOR J=1 TO NV      'and last by variable
480       INPUT #1, X(K,J,I)
490 NEXT J,I,K: CLOSE: RETURN
500 '                        Store data array onto disk
510 '  save number of groups and number of variables
520 OPEN "O",1,F$: PRINT #1, NG,NV
530 '         then save number of subjects per group
540 FOR K=1 TO NG: PRINT #1, NM(K): NEXT K
550 FOR K=1 TO NG
560 '          and last, the observations themselves
570   FOR I=1 TO NM(K)
580     FOR J=1 TO NV
590       PRINT #1, X(K,J,I)
600 NEXT J,I,K
610 CLOSE: RETURN
```

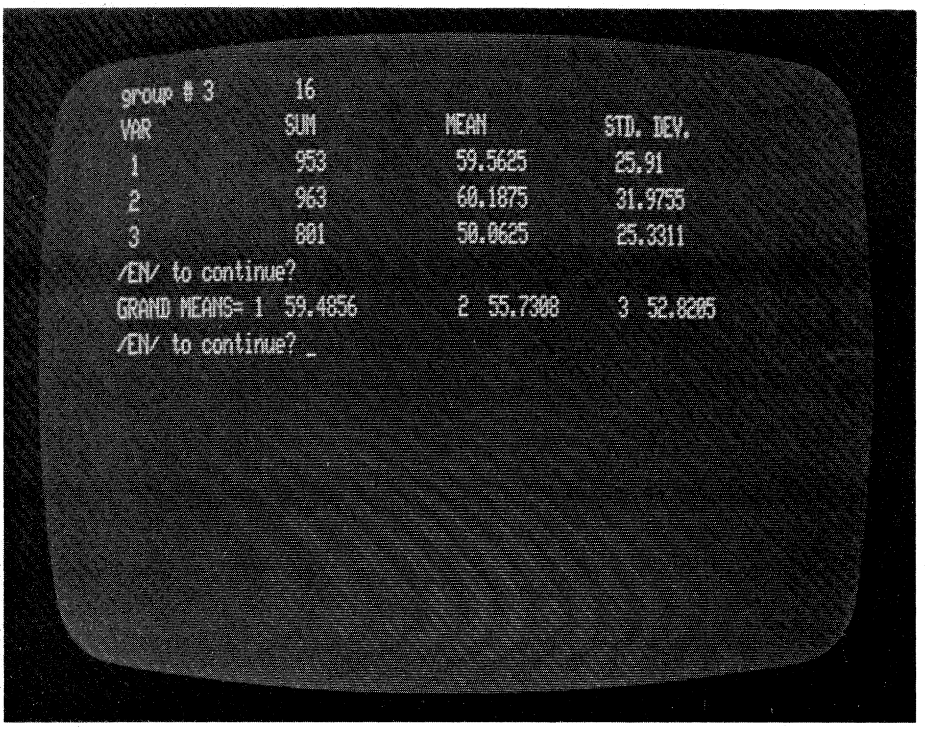
```
1000 '                              Create artificial data
1010 CLS: INPUT "how many groups (to 3)";NG
1020 INPUT "range of subjects per group
          (1 to 20 -- type low & high)";N1,N2
1030 INPUT "how many variables per subject (to 4)";NV
1040 FOR K=1 TO NG                  'K is group index
1050   NM(K)=RND(N2-N1)+N1
1060   PRINT "group";K, NM(K)"subjects"
1070   FOR I=1 TO NM(K)             'I is subject index
1080     FOR J=1 TO NV            'J is variables index
1090       X(K,J,I)=RND(90)+9: PRINT X(K,J,I);
1100     NEXT J: PRINT,
1110   NEXT I: PRINT: GOSUB 370
1120 NEXT K: RETURN
2000 '                                          Data input
2010 CLS
2020 INPUT "data on file (yes or no)";A$
2030   IF A$="yes" THEN GOSUB 390: RETURN
2040 INPUT "how many groups (to 3)";NG
2050 INPUT "how many observations per subject (to 4)";NV
2060 FOR K=1 TO NG
2070   PRINT "how many in group";K;: INPUT NM(K)
2080     FOR I=1 TO NM(K)
2090       PRINT "enter"; NV; "values for subject #"; I
2100     IF NV=1 THEN INPUT X(K,1,I): GOTO 2140
2110    IF NV=2 THEN INPUT X(K,1,I),X(K,2,I): GOTO 2140
2120   IF NV=3 THEN INPUT X(K,1,I),X(K,2,I),X(K,3,I):
                    GOTO 2140
2130  INPUT X(K,1,I),X(K,2,I),X(K,3,I),X(K,4,I)
2140 NEXT I,K
2150 INPUT "save on file (yes or no)";A$
2160   IF A$="yes" THEN GOSUB 500
                   ELSE RETURN
2170 RETURN
3000 '                                  Edit data file
3010 FT$="  ###    ####.##    ####.##    ####.##    ####.##"
3020 IF NG=0 THEN PRINT "no data!": GOTO 3130  'return
3030 PRINT NG "groups"
3040 FOR K=1 TO NG
3050   PRINT
"subj.     var. 1     var. 2     var. 3     var. 4"
3060     FOR I=1 TO NM(K)
3070     PRINT USING FT$;
              I, X(K,1,I), X(K,2,I), X(K,3,I), X(K,4,I)
3080     NEXT I
3090     LINE INPUT
         "change which subject (/EN/=no change)";A$
3100     IF A$<>"" THEN INPUT "which var. to what";J1,X2:
                        X(K,J1,VAL(A$))=X2: GOTO 3090
                 ELSE 3110
```

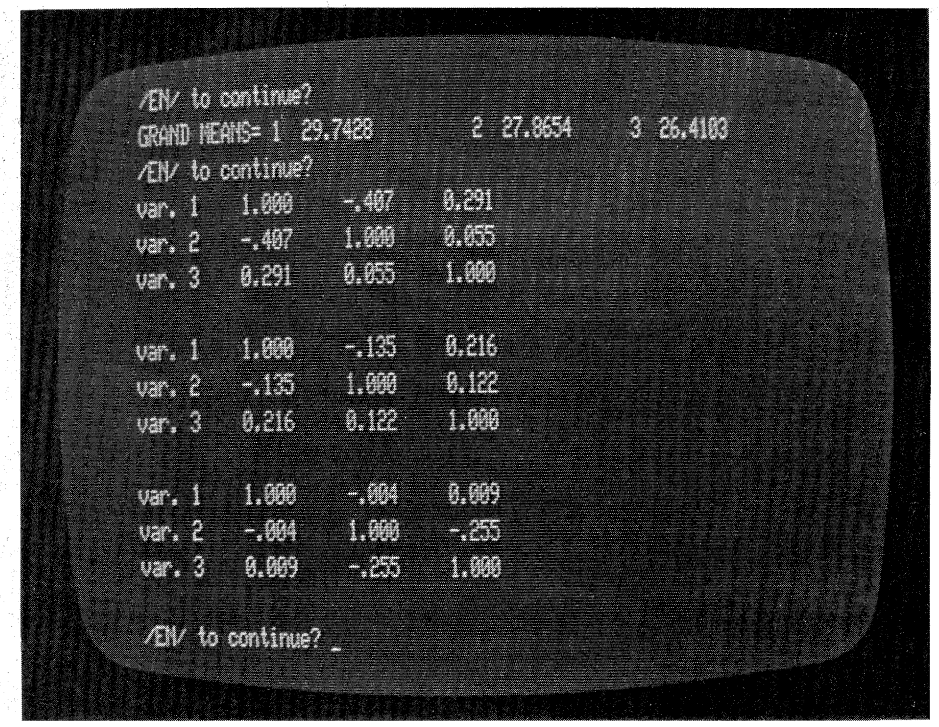
```
3110 NEXT K: INPUT "save this version (yes or no)";A$
3120 IF A$="YES" THEN GOSUB 500    'go to SAVE subr.
3130 RETURN
4000 '            sort data using Shell-Metzner sort
4010 PRINT "what group (1 to";NG;" or 'all')";:
          INPUT A$
4020 IF A$="all" THEN N1=1: N2=NG: GOTO 4040
                 ELSE NK=VAL(A$)
4030 IF NK=0 THEN 4200
           ELSE N1=NK: N2=NK
4040 PRINT "on which key (1 to";NV;")";: INPUT Q
4050 FOR K1=N1 TO N2: M=NM(K1)
4060   M=INT(M/2): PRINT @ 245,M;
4070     IF M=0 THEN 4190 ELSE 4080
4080   K=NM(K1)-M: J=1
4090   I=J
4100   L=I+M
4110     IF X(K1,Q,I)<=X(K1,Q,L) THEN 4170
4120     FOR J1=1 TO NV
4130       T=X(K1,J1,I): X(K1,J1,I)=X(K1,J1,L):
           X(K1,J1,L)=T
4140     NEXT J1
4150     I=I-M
4160       IF I>=1 THEN 4100
4170     J=J+1
4180     IF J<=K THEN 4090
                 ELSE 4060
4190 NEXT K1
4200 RETURN
5000 '                          Descriptive statistics
5010 INPUT "on screen (1) or printer (2)";A
5020   IF A=2 THEN GOSUB 330  'set for printer
5030 FOR J=1 TO NV: S3(J)=0: NEXT J   'grand sums
5040 FOR K=1 TO NG      'group by group calculations
5050   CLS: PRINT "group #";K, NM(K)
5060   PRINT "VAR","SUM","MEAN","STD. DEV."
5070   FOR J=1 TO NV: S(K,J)=0: S2(K,J)=0
5080     FOR JJ=1 TO NV: R(K,J,JJ)=0: NEXT JJ
5090     FOR I=1 TO NM(K)
5100       S(K,J)=S(K,J)+X(K,J,I)      'sums
5110       S2(K,J)=S2(K,J)+X(K,J,I)*X(K,J,I)
5120       FOR JJ=1 TO NV   'crossproducts
5130         R(K,J,JJ)=R(K,J,JJ)+X(K,J,I)*X(K,JJ,I)
5140       NEXT JJ
5150     NEXT I
5160     M(K,J)=S(K,J)/NM(K)     'means
5170     V=(S2(K,J)-S(K,J)*M(K,J))/(NM(K)-1)  'vars.
5180     D(K,J)=SQR(V)          'standard deviations
5190     PRINT J, S(K,J), M(K,J), D(K,J)
```

```
5200      S3(J)=S3(J)+S(K,J)                'grand sums
5210    NEXT J: N=N+NM(K)
5220    IF A<>2 THEN GOSUB 370
5230  NEXT K
5240  PRINT "GRAND MEANS=";
5250  FOR J=1 TO NV: PRINT J;S3(J)/N,: NEXT J
5260  IF A=2 THEN GOSUB 350           'reset for screen
5270  IF A<>2 THEN GOSUB 370        'prompt to continue
5280  RETURN
6000  '                          Correlation coefficients
6010  INPUT "on screen (1) or printer (2)";A
6020    IF A=2 THEN GOSUB 330         'set for printer
6030  GOSUB 5030              'set descriptive statistics
6040  FOR K=1 TO NG
6050    FOR J=1 TO NV
6060      PRINT "var.";  J;
6070      FOR JJ=1 TO NV
6080      IF J=JJ THEN R(K,J,JJ)=1: GOTO 6120   'diag.
6090        T=R(K,J,JJ)-S(K,J)*S(K,JJ)/NM(K)
6100        B=NM(K)*D(K,J)*D(K,JJ)
6110        R(K,J,JJ)=T/B
6120        PRINT TAB(JJ*10) USING "#.###";R(K,J,JJ);
6130      NEXT JJ: PRINT
6140    NEXT J: PRINT
6150  NEXT K
6160    IF A=2 THEN GOSUB 350         'reset for screen
6170    IF A<>2 THEN GOSUB 370      'prompt to continue
6180  RETURN
7000  '                                  list data array
7010  INPUT "on screen (1) or printer (2)";A
7020    IF A=2 THEN GOSUB 330       'set for printer
7030  CLS: PRINT NG "groups",
                NV "variables per subject"
7040  FOR K=1 TO NG
7050    PRINT NM(K) "subjects in group #" K
7060    FOR I=1 TO NM(K) STEP 2
7070      FOR J=1 TO NV
7080        PRINT X(K,J,I);
7090      NEXT J: PRINT,
7100      FOR J=1 TO NV
7110        PRINT X(K,J,I+1);
7120      NEXT J: PRINT
7130    NEXT I: PRINT
7140    IF A=2 THEN GOSUB 350         'reset for screen
7150    IF A<>2 THEN GOSUB 370      'prompt to continue
7160  RETURN
8000  '                                 store data on disk
8010  INPUT "filename";F$: GOSUB 500: RETURN
9000  '                                    closing routine
9010  CLOSE
9020  END
```


1 create artificial data
2 input real data
3 edit data array
4 sort data array
5 descriptive statistics
6 correlation coefficient matrix
7 display data
8 store data on disk
9 stop
select your choice by digit
? 4
what group (1 to 3 or 'all')? 2
on which key (1 to 3)? 1


change which subject (/EN/=no change)
subj. var. 1 var. 2 var. 3 var. 4
1 21.00 44.00 27.00 0.00
2 33.00 98.00 76.00 0.00
3 45.00 74.00 96.00 0.00
4 55.00 45.00 22.00 0.00
5 60.00 63.00 26.00 0.00
6 64.00 84.00 16.00 0.00
7 66.00 37.00 59.00 0.00
8 68.00 29.00 92.00 0.00
9 69.00 18.00 41.00 0.00
10 77.00 34.00 75.00 0.00
11 88.00 40.00 59.00 0.00
12 95.00 54.00 57.00 0.00
13 97.00 90.00 88.00 0.00
change which subject (/EN/=no change)_

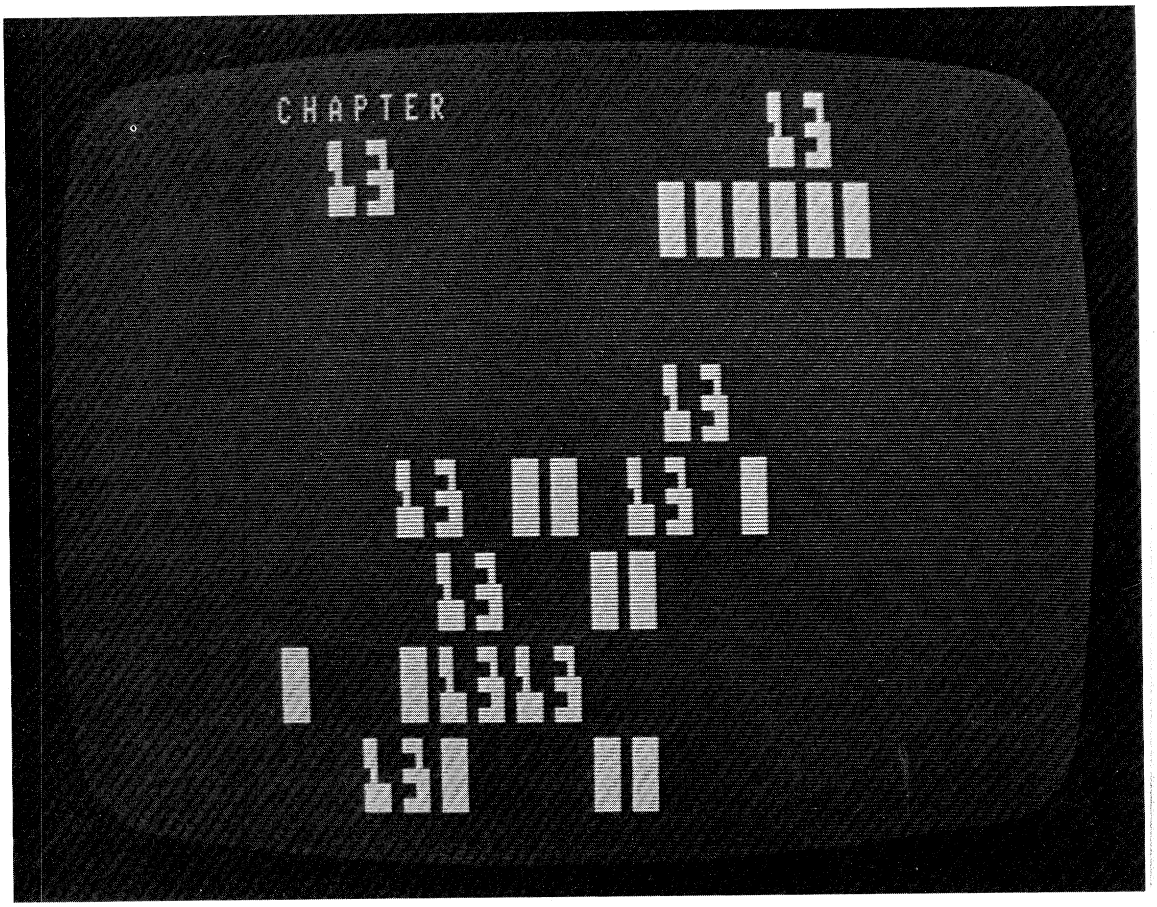

group # 3 16
VAR SUM MEAN STD. DEV.
1 953 59.5625 25.91
2 963 60.1875 31.9755
3 801 50.0625 25.3311
/EN/ to continue?
GRAND MEANS= 1 59.4856 2 55.7308 3 52.8205
/EN/ to continue? _


/EN/ to continue?
GRAND MEANS= 1 29.7428 2 27.8654 3 26.4103
/EN/ to continue?
var. 1 1.000 -.407 0.291
var. 2 -.407 1.000 0.055
var. 3 0.291 0.055 1.000
var. 1 1.000 -.135 0.216
var. 2 -.135 1.000 0.122
var. 3 0.216 0.122 1.000
var. 1 1.000 -.004 0.009
var. 2 -.004 1.000 -.255
var. 3 0.009 -.255 1.000
/EN/ to continue? _

This chapter has shown by example several techniques that are commonly used to manage, or manipulate, the records that exist on sequential or direct access files. So far, we have assumed that the information of the records is independent of the file's structure. That is, no record has any information that relates to its position on the file, or to the position of any other record on the file. We have introduced one exception to that scheme, and that was the direct access file structure generated by hash addressing, each record of which had its record number embedded within the record as a data entry.

In the next chapter, we will expand on the idea of a record containing some information about its position on the file. You will discover fast accessing methods, sorts that don't rearrange keys, and a process for arranging the data file so that these techniques and others are a part of the overall package.

Through development of a parts inventory system we will use most of the ideas discussed in the previous chapters of this book. The system is based on the Binary Sequence Search Tree structure which has been thoroughly explained in *Microcomputer Power: Guide to Systems Applications.*



Inventory System Application

The discussion and program listings that follow are all part of an existing, working system that uses Binary Sequence Search Tree file organization and structure. It is a generalized inventory system that maintains the data file's entries, whether they relate to shoes, ships, or sealing wax. The programs are written to be non-specific so that the reader may adapt them to whatever record-keeping functions are desired.

Each record of the file occupies 63 bytes of a 256-byte physical record, or sector, on a direct access disk file. The records consist of 13 fields and are detailed in table 13.1.

The last record on the file has only three fields:

Field 1 (integer) is the root record number for the tree structure.

Field 2 (integer) is the total number of records.

Field 3 (string) is a file description.

No.	Type	Size (bytes)	Variable Name	Description
1	Int.	2	LL	Left link of BSST
2	Int.	2	RL	Right link of BSST
3	Str.	12	PT$	Part number or name
4	Str.	25	Des$	Description of part
5	Sns.	4	DP!	Dealer price, per unit
6	Sns.	4	LP!	List price, per unit
7	Int.	2	OH	Units on hand
8	Int.	2	OO	Units on order
9	Int.	2	SB	Minimum stock quantity
10	Int.	2	M	Units sold in past month
11	Int.	2	N	Units sold in past 3 months
12	Int.	2	O	Units sold in past 12 months
13	Int.	2	P	Units sold in past 24 months

Table 13.1 Inventory System Record Description

#	Name	Purpose	Bytes	Lines
0	INVMAIN	Main driver program	700	20
1	INVPR1	Add a part to a file	4300	110
2	INVPR2	List parts (sorted)	1800	50
3	INVPR3	Delete a part from a file	3300	90
4	INVPR4	Change a part description or information about a part	3600	110
5	INVPR5	List the records in entered order	1200	40
6	INVPR6	List deleted parts in a file	1100	30
7	INVPR7	Balance BSST tree links	4500	120
8	INVPR8	List files in use	700	20
9	INVPR9	Delete a file	1700	30
10	INVPR10	Show all information on a selected part	3600	80

Table 13.2 Index to Programs in Inventory System

INVMAIN Program

The entire system of programs runs from one central menu driver program that both displays the list of activities and branches to the program that performs the selected activity.

There are ten programs subordinate to INVMAIN in the inventory system. Each one was designed to perform a different task. Table 13.2 is an index to those programs, including their names, purpose, and approximate size in bytes (to the nearest hundred), and length in number of lines (to the nearest ten).

```
10 'FILENAME: "INVMAIN"
20 'FUNCTION: DRIVER PROGRAM FOR BSST INVENTORY SYSTEM
30 ' AUTHOR : SPG                DATE: 1/12/79    REV: 7/80
40 '
50 '**                  d i s p l a y   m e n u
60 CLS: PRINT: PRINT: PRINT
70 PRINT "O N - L I N E   I N V E N T O R Y   S Y S T E M"
80 PRINT:PRINT
90 PRINT  "1  add parts              7   balance BSST tree file"
100 PRINT "2  list parts (sorted)    8   list files in use"
110 PRINT "3  delete a part          9   delete a file"
120 PRINT "4  change part desc.      10  get info. on a part"
130 PRINT "5  list all records       11  stop"
140 PRINT "6  list deleted parts"
150 PRINT
160 LINEINPUT "Select your activity by number: "; A$
170   IF A$="" THEN 50
180   IF A$="11" THEN STOP
190   IF VAL(A$)<1 OR VAL(A$)>10 THEN 50
200 P$="INVPR"+A$: RUN P$
210 END
```

The output of the INVMAIN program is the menu display. Here is what it looks like.

```
O N - L I N E   I N V E N T O R Y   S Y S T E M


1  add parts              7   balance BSST tree file
2  list parts (sorted)    8   list files in use
3  delete a part          9   delete a file
4  change part desc.      10  get info. on a part
5  list all records       11  stop
6  list deleted parts

Select your activity by number: 1
```

INVPR1 The INVPR1 program adds items to an inventory file.

```
1000 'FILENAME: "INVPR1"          (CALLED BY "INVMAIN")
1005 'FUNCTION: ADD AN ITEM TO THE INVENTORY
1010 ' AUTHOR : SPG              DATE: 1/19/79  REV. 7/80
1015 '
1020 DEFINT A-Z: CLEAR 1000: FIRST=0
1025 CLS
1030 '**                                  set file name - FL$.
1035 FL$=""
1040 PRINT "Please type either:"
1045 PRINT "         the file name of an old file"
1050 PRINT "                            or"
1055 PRINT "    'help'  for a list of the files in use"
1060 PRINT "                            or"
1065 LINE INPUT "'0'  for a new file: "; FL$
1070   IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
1075   IF FL$<>"0" AND FL$<>"" THEN 1175
1080   IF VAL(FL$)>0 THEN 1030          '          try again.
1085 '                                 new file.  set its name.
1090 INPUT "What is this new file's name (to 8 chars.)";FL$
1095   IF FL$="" THEN 1030
1100 '            store new filename into file "FILENAME".
1105 CLOSE: PRINT "stand by ...": OPEN "R",1,FL$
1110   IF LOF(1)<>0 THEN PRINT "File ";FL$;" already exists":
                         GOTO 1030
1115 CLOSE: D$=LEFT$(FL$+"        ",8)   'trim to 8 chars.
1120 OPEN "R",1,"FILENAME": FIELD 1, 255 AS A$
1122 '**                                  search for right file.
1125 S=S+1
1130   IF LOF(1)=S-1 THEN B$=D$: L=1: GOTO 1145
                     ELSE GET 1,S
1135 FOR L=1 TO 255 STEP 8
1140   IF MID$(A$,L,1)<>" " THEN NEXT L: GOTO 1122
                           ELSE B$=LEFT$(A$,L-1)+D$
1145 '**                                  store file description.
1148   D1$=MID$(A$,L,1+LEN(D$)-1)
1150     IF D$<>D1$ THEN LSET A$=B$: PUT 1,S: CLOSE
1155 CLOSE: OPEN "R",1,FL$
1160 GOSUB 1800
1165   IF IEN=0 THEN IEN=1: GOTO 1225
1170 '                                  set last record on file.
1175 '**                               open the file labeled FL$.
1180 CLOSE: OPEN "R",1,FL$
1185 '                           check to see if file exists --
1190 '                            if not, go back to beginning.
1195   IF LOF(1)=0 THEN PRINT " no such file ...": GOTO 1030
1200 '                       set record that has the root, etc.
1205   IF IEN=0 THEN IEN=1
```

```
1210 GOSUB 1700
1215 GET 1,LOF(1)
1220 R=CVI(ROOT$): NM=CVI(NM$): FL$=DESC$: IEN=LOF(1)
1225 CLS: PRINT "Diskette file description is   ";FL$
1230 '**                              set new part number  P1$.
1235 I=R: P1$=""
1240 LINEINPUT "Part number (/EN/=return): ";P1$
1262 '**                                 redefine part number.
1265 P1$=RIGHT$("              "+P1$,12)
1270 '                                                 is P1$ null?
1275   IF RIGHT$(P1$,2)<>" 0" AND RIGHT$(P1$,1)<>" " THEN 1285
1280   IF LOF(1)=1 AND FIRST=0 THEN 1422
                               ELSE 1392
1285 '**                                          set links to 0.
1290   IF I=0 THEN R=1: RL=0: LL=0: GOTO 1327
1295 '**                                        read data record.
1300 FIRST=1:  GOSUB 1600
1305   IF P1$>P2$ THEN 1317
1310   IF P1$=P2$ THEN PRINT "already on file": GOTO 1225
1315   IF LL=0 THEN LSET LL$=MKI$(IEN): PUT 1,I: GOTO 1327
              ELSE I=LL: GOTO 1295
1317 '**                                       field data record.
1320 GOSUB 1800
1325   IF RL=0 THEN LSET RL$=MKI$(IEN): PUT 1,I
              ELSE I=RL: GOTO 1295
1327 '**                                     set new links to 0.
1330 RSET PT$=P1$: LSET LL$=MKI$(0): LSET RL$=MKI$(0)
1335 '                            set other details on the part.
1340 INPUT "Short part description";DS$: LSET DES$=DS$
1345 INPUT "Dealer price";SP!: LSET DP$=MKS$(SP!)
1350 INPUT "List price";XP!: LSET LP$=MKS$(XP!)
1355 INPUT "How many on hand";OH: LSET OH$=MKI$(OH)
1360 INPUT "What is minimum stock";SB: LSET SB$=MKI$(SB)
1365 LSET N$=MKI$(0): LSET OO$=N$: LSET M$=N$: LSET O$=N$
1370 LSET P$=N$
1375 NM=NM+1   '                 increase no. of active records.
1380   IF R=0 THEN R=1  '                 if root is 0 make it 1.
1385 '                               put the new record on file.
1390 PUT 1, IEN: IEN= LOF(1)+1: GOTO 1225
1392 '**                           field data on the last record.
1395 GOSUB 1700
1400 '               put the new data into the record buffer.
1405 LSET DESC$=FL$
1410 LSET NM$=MKI$(NM)
1415 LSET ROOT$=MKI$(R)
1420 PUT 1, LOF(1)+1   '               put last record onto file.
1422 '**                                 return to menu display.
1425 INPUT "/EN/"; A$: RUN "INVMAIN"
1600 '**                              subroutine to read a record.
1610 GOSUB 1800
```

```
1620   GET 1,I: LL=CVI(LL$): RL=CVI(RL$): P2$=PT$
1630 RETURN
1700 '**                          routine to field root record .
1705 FIELD 1, 2 AS ROOT$, 2 AS NM$, 25 AS DESC$
1710 RETURN
1800 '**                          routine to field data record.
1805 FIELD 1, 2 AS LL$, 2 AS RL$, 12 AS PT$, 25 AS DES$,
             4 AS DP$, 4 AS LP$,  2 AS OH$,  2 AS OO$,
             2 AS SB$, 2 AS M$,   2 AS N$,   2 AS O$,
             2 AS P$
1810 RETURN
1820 END
```

This is the dialog that INVPR1 produces as the user executes it.

```
Please type either:
        the file name of an old file
                    or
    'help'  for a list of the files in use
                    or
'0'  for a new file: parts←

Diskette file description is   parts
Part number (/EN/=return): 1984
Short part description? unflanged grommet
Dealer price? .13
List price? .39
How many on hand? 6999
What is minimum stock? 1000←
```

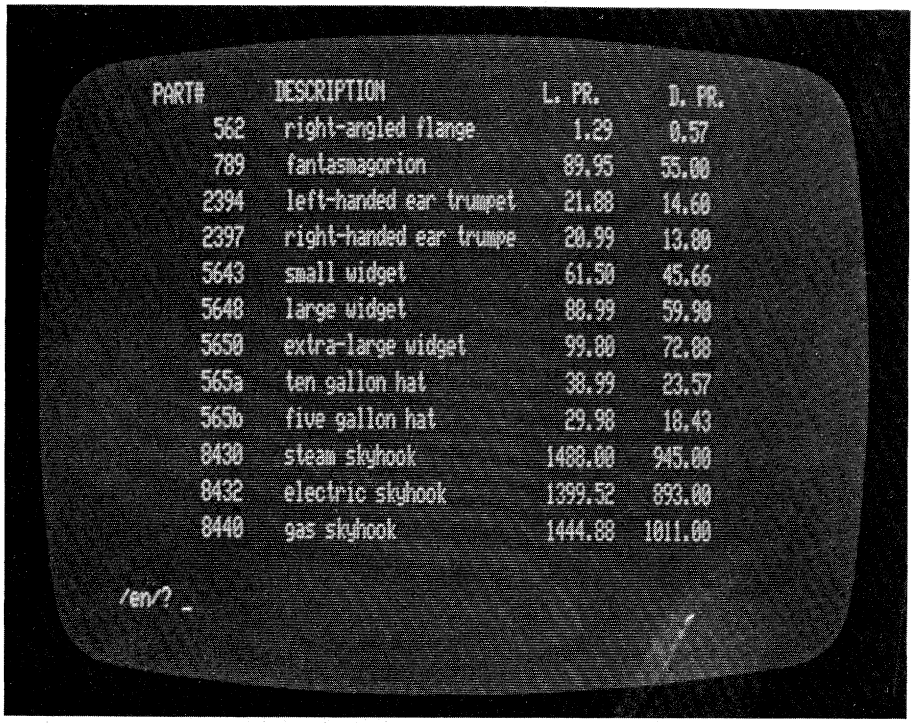
INVPR2

The INVPR2 program performs a traversal of the binary sequence search tree structure which allows the listing of all parts in sorted order.

```
2000 'FILENAME: "INVPR2"          (CALLED BY "INVMAIN")
2010 'FUNCTION: LIST PARTS IN SORTED ORDER
2020 ' AUTHOR : SPG               DATE: 1/19/79  REV: 7/80
2030 '
2040 DEFINT A-Z
2050 CLEAR 500: DIM STK(100): CLS
2060 PRINT "S O R T E D   O R D E R   T R A V E R S A L"
2070 B$=
"%          %    %                      %   ####.##   ####.##"
2080 '
2090 '**                                open appropriate file.
2100 CLOSE
2130 INPUT "Name of file ('help'=list of files)";FL$
2135   IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
2140   IF VAL(FL$)>0 THEN PRINT "invalid entry.": GOTO 2090
2150 OPEN "R",1,FL$
2160   IF LOF(1)=0 THEN PRINT "no such file.": GOTO 2090
2170 '                             now field last (root) record.
2180 FIELD 1, 2 AS ROOT$, 2 AS NM$, 25 AS DESC$
2190 GET 1,LOF(1): R=CVI(ROOT$)
2200 '                             initialize ordered traversal.
2210 P=R: T=0
2220 GOSUB 2500
2230 FIELD 1, 2 AS LL$, 2 AS RL$, 12 AS PT$, 25 AS DES$,
           4 AS DP$, 4 AS LP$
2240 '**                                                traverse.
2250 T=T+1: STK(T)=P
2260   IF P<>0 THEN GOSUB 2600: P=LL: GOTO 2240
2270 T=T-1
2280   IF T=0 THEN 2350
2290 P=STK(T)
2300 GOSUB 2600  '                 set Pth record and its links.
2310 LOOP=LOOP+1: PRINT USING B$; PT$; DS$; LP!; DP!: T=T-1
2320   IF LOOP>11 THEN PRINT: INPUT "/en/";A$: GOSUB 2500
2330 P=RL: GOTO 2240
2340 '
2350 '*****                         closing routine to set root.
2360 FIELD 1, 2 AS ROOT$, 2 AS NM$, 25 AS DESC$
2370   PRINT "file is of the class  ";
2380   GET 1, LOF(1): F$=DESC$
2390   PRINT F$
2400   INPUT "/EN/"; A$
2405 '*****                                     return to menu.
2410 RUN "INVMAIN"
```

```
2500 '*****                          print heading for table.
2510 CLS: PRINT
"   PART#       DESCRIPTION                  L. PR.       D. PR."
2515 LOOP=0    '              zero the counter variable LOOP
2520 RETURN
2600 '*****                              get data from buffer.
2605 GET 1,P: LL=CVI(LL$): RL=CVI(RL$)
2610   DS$=DES$: DP!=CVS(DP$): LP!=CVS(LP$)
2620 RETURN
2630 END
```

This is what the output looks like from a typical run of the traversal program.



INVPR3 The INVPR3 program performs record deletion.

```
3000 'FILENAME: "INVPR3"          (CALLED BY "INVMAIN")
3010 'FUNCTION: TO DELETE ITEMS FROM THE FILE
3020 ' AUTHOR : SPG              DATE: 1/19/79  REV. 7/80
3030 '
3040 DEFINT A-Z: CLEAR 500: CLS
3050 '**                    set root (R) from last record.
3060 CLOSE
3070 PRINT "What is the file's name"
3080 INPUT "(type 'help' for the list of files)"; FL$
3090   IF FL$="" OR VAL(FL$)>0
           THEN PRINT "invalid entry.": GOTO 3050
3100   IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
3110 OPEN "R",1,FL$
3120   IF LOF(1)=0 THEN PRINT "no such file.": GOTO 3050
3130 GOSUB 3850: GET 1,LOF(1): R=CVI(ROOT$): N=CVI(NM$)
3140 '
3150 '**       set new name, N1$.    set W to latest root.
3160 '
3170 INPUT "Part to be deleted -- '0'=return"; N1$: W=R
3180 N1$=LEFT$(N1$,12): N1$=RIGHT$("            "+N1$,12)
3190 '     if no more deletions, replace R on last record.
3200 IF RIGHT$(N1$,5)="    0" OR RIGHT$(N1$,1)=" " THEN 3700
3210 '      set root.  if deletion is not root, search on.
3220 I=R: GOSUB 3800: LL=CVI(LL$): RL=CVI(RL$)
3230   IF N1$<>PT$ THEN 3320
3240 '          if no right branch, new root is left link.
3250 IF RL=0 THEN R=LL: GOTO 3580
3260 '          if no left branch, new root is right link.
3270 IF LL=0 THEN R=RL: GOTO 3580
3280 '      neither link is null.  new root is right link.
3290 F=R: R=RL: S=0
3295 '**                                  set Ith. record.
3300 IF W=0 THEN INPUT "Part not on file. /EN/";A$: GOTO 3150
3310 '
3320 '**           this name is not root.  look some more.
3330 I=W: GOSUB 3800: LW=CVI(LL$): RW=CVI(RL$)
3340   IF N1$=PT$ THEN 3390
3350 '                  name still not found.  look again.
3360 F=W
3370   IF N1$<PT$ THEN S=1: W=LW: GOTO 3295
                  ELSE S=0: W=RW: GOTO 3295
3380 '              name is found!  check links LW and RW.
```

```
3390 '**      if at least the left link is null, check RW.
3400   IF LW=0 THEN 3500
3410 '       left link is not null.  what about right link?
3420 IF RW=0 THEN 3550
3430 '            neither link is null.  set father record.
3440 I=F: GOSUB 3800
3445   IF S=0 THEN LSET RL$=MKI$(RW)
               ELSE LSET LL$=MKI$(RW)
3450 '    once new right link is attached, rewrite on file.
3460 PUT 1,I:  Z=RW
3470 '**                                    reset new null link.
3480 I=Z:  GOSUB 3800:  LZ=CVI(LL$)
3490   IF LZ=0 THEN LSET LL$=MKI$(LW): GOTO 3580
               ELSE Z=LZ: GOTO 3470
3500 '**        left link is null.  what about right link?
3510 IF RW=0 THEN 3550
3520 '        set father record, write with new right link.
3530 I=F: GOSUB 3800
3540   IF S=0 THEN LSET RL$=MKI$(RW): GOTO 3580
               ELSE LSET LL$=MKI$(RW): GOTO 3580
3550 '**     set father, reset left link with this right.
3560 I=F: GOSUB 3800
3570   IF S=0 THEN LSET RL$=MKI$(LW)
               ELSE LSET LL$=MKI$(LW)
3580 '**       rewrite both new records, return to search.
3590 N=N-1: PUT 1,I: I=W
3600 GOSUB 3800
3610 LSET DP$=MKS$(-DP!): PUT 1,I
3620 GOTO 3150
3700 '*****             routine to rewrite new root record.
3710 GOSUB 3850
3720 GET 1,LOF(1): LSET ROOT$=MKI$(R): LSET NM$=MKI$(N)
3730 PUT 1,LOF(1)
3740 RUN "INVMAIN"
3800 '*****               routine to fetch the Ith. record.
3810 '                   set Ith. record.
3820 GOSUB 3900
3830 GET 1,I: PT$=P1$: DP!=CVS(DP$)
3840 RETURN
3850 '*****           routine to describe the root record.
3860 FIELD 1, 2 AS ROOT$, 2 AS NM$,25 AS DESC$
3870 RETURN
3900 '*****           routine to describe the data record.
3910 FIELD 1,  2 AS LL$, 2 AS RL$, 12 AS P1$, 25 AS DES$,
               4 AS DP$, 4 AS LP$,  2 AS OH$,  2 AS OO$,
               2 AS SB$, 2 AS M$,   2 AS N$,   2 AS O$,
               2 AS P$
3920 RETURN
3930 END
```

The dialog between the user of INVPR3 and the computer is shown below.

```
What is the file's name
(type 'help' for the list of files)? parts
Part to be deleted -- '0'=return? 7890
part not on file. /EN/?
Part to be deleted -- '0'=return? 789
Part to be deleted -- '0'=return? 565z
part not on file. /EN/?
Part to be deleted -- '0'=return? 565b
Part to be deleted -- '0'=return? ←
```

INVPR4

The INVPR4 program performs an access to the file in order to update the contents of a record. It is essentially a modification of the INVPR1 program that added records to the file.

```
4000 'FILENAME: "INVPR4"          (CALLED BY "INVMAIN")
4005 'FUNCTION: TO CHANGE AN INVENTORY RECORD
4010 ' AUTHOR : SPG              DATE: 1/19/79   REV. 7/80
4015 CLEAR 500: DEFINT A-Z: CLS
4020 '**                                        open a data file.
4025 INPUT "File name -- ('help'=list of files)";FL$
4030   IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
4035   IF FL$="" THEN PRINT "bad file name": GOTO 4020
4040 OPEN "R",1,FL$
4045   IF LOF(1)=0 THEN PRINT "no such file": GOTO 4020
4050 FIELD 1,2 AS ROOT$, 2 AS NM$, 25 AS DES$
4055 GET 1,LOF(1)
4060 R=CVI(ROOT$)
4065 PRINT "   file description is  ";DES$
4070 FIELD 1, 2 AS LL$, 2 AS RL$, 12 AS P1$, 25 AS DES$,
             4 AS DP$, 4 AS LP$,  2 AS OH$,  2 AS OO$,
             2 AS SB$, 2 AS M$,   2 AS N$,   2 AS O$,
             2 AS P$
4075 '
4080 '         m a i n   r e c o r d   c h a n g e r
4085 '
4090 N1$=""
4095 INPUT "Part number -- /EN/=no further changes";N1$
4100 N1$=LEFT$(N1$,12): N1$=STRING$(12-LEN(N1$)," ")+N1$
4105   IF RIGHT$(N1$,2)=" 0"
       OR RIGHT$(N1$,1)=" "
          THEN RUN "INVMAIN"
4110 '                                  search for part number N1$.
4115 I=R: GOSUB 4770
4120 '**                                      start access of tree.
4125   IF N1$>PT$ THEN 4140
4130   IF N1$=PT$ GOTO 4160
4135   IF LL<>0 THEN I=LL: GOSUB 4770: GOTO 4120
4140 '**                                      check right link.
4145   IF RL<>0 THEN I=RL: GOSUB 4770: GOTO 4120
4150 I=0
4155   IF I=0 THEN PRINT "part no. ";N1$;" was not found.":
                   INPUT "/EN/";A$: GOTO 4080
4160 '**                                 ***record was found!***
4165 '  set the rest of the information out of the buffer.
4170 DP!=CVS(DP$): LP!=CVS(LP$): OH=CVI(OH$): OO=CVI(OO$):
     SB=CVI(SB$): M=CVI(M$): N=CVI(N$): O=CVI(O$): P=CVI(P$)
4175 PRINT "RECORD  #";I
4180 '
```

```
4185 '**   m a i n   ' m i n i d r i v e r'   r o u t i n e
4190 PRINT "1  description      5  no. on order"
4195 PRINT "2  dealer price     6  no. there should be"
4200 PRINT "3  list price       7  sales history info."
4205 PRINT "4  no. on hand      8  no more changes"
4210 INPUT "Which activity";A$
4215   IF A$="" THEN PUT 1,I: GOTO 4080
4220   IF VAL(A$)<1 OR VAL(A$)>8 THEN 4185
4225   IF VAL(A$)=8 THEN A$="": PUT 1,I: GOTO 4080
4230 ON VAL(A$) GOSUB 4300,4350,4400,4450,4500,4550,4600
4235 GOTO 4185
4300 '*****           subroutine to change description.
4310 PRINT "description=";DES$: A$=""
4320 INPUT A$
4330   IF LEN(A$)<>0 THEN LSET DES$=A$
4340 RETURN
4350 '*****             subroutine to change dealer price.
4360 PRINT "dealer price=";DP!
4370 INPUT DP!
4380   IF DP!<>0 THEN LSET DP$=MKS$(DP!)
4390 RETURN
4400 '*****               subroutine to change list price.
4410 PRINT "list   price=";LP!
4420 INPUT LP!
4430   IF LP<> THEN LSET LP$=MKS$(LP!)
4440 RETURN
4450 '*****       subroutine to change # of units on hand.
4460 PRINT "no.  on hand=";OH
4470 INPUT OH
4480   IF OH<>0 THEN LSET OH$=MKI$(OH)
4490 RETURN
4500 '*****      subroutine to change # of units on order.
4510 PRINT "no.  on order=";OO
4520 INPUT OO
4530   IF OO<>0 THEN LSET OO$=MKI$(OO)
4540 RETURN
4550 '*****              subroutine to change stock limit.
4560 PRINT "no.  there should be (stock limit)=";SB
4570 INPUT SB
4580   IF SB<>0 THEN LSET SB$=MKI$(SB)
4590 RETURN
4600 '*****        subroutine to change the sales history.
4610 PRINT "sales history information:"
4620 PRINT 1 TAB(5) M "units sold in the past 2 months."
4630 PRINT 2 TAB(5) N "units sold in the past 6 months."
4640 PRINT 3 TAB(5) O "units sold in the past 12 months."
4650 PRINT 4 TAB(5) P "units sold in the past 24 months."
4660 A$="": PRINT
```

```
4670 INPUT "which do you want to change (/EN/=none)";A$
4680 IF VAL(A$)<1 OR VAL(A$)>4 THEN RETURN
4690 C$="": PRINT
4700 INPUT "What is the new number"; C$
4710 ON VAL(A$) GOSUB 4730,4740,4750,4760
4720 RETURN
4730 M=VAL(C$): LSET M$=MKI$(M): RETURN
4740 N=VAL(C$): LSET N$=MKI$(N): RETURN
4750 O=VAL(C$): LSET O$=MKI$(O): RETURN
4760 P=VAL(C$): LSET P$=MKI$(P): RETURN
4770 '*****        set Ith record, links, and part number.
4780 GET 1,I: LL=CVI(LL$): RL=CVI(RL$): PT$=P1$
4790 RETURN
4800 END
```

User interaction with INVPR4 looks like this.

```
File name -- ('help'=list of files)? parts
   file description is  parts
Part number -- /EN/=no further changes? 1984
RECORD  # 22
1  description      5  no. on order
2  dealer price     6  no. there should be
3  list price       7  sales history info.
4  no. on hand      8  no more changes
Which activity? 1
description=unflanged grommet
? reflanged grommet
1  description      5  no. on order
2  dealer price     6  no. there should be
3  list price       7  sales history info.
4  no. on hand      8  no more changes
Which activity? ←
```

INVPR5

The INVPR5 program lists all records in their physical order, ignoring the BSST's logically sorted order.

```
5000 'FILENAME: "INVPR5"          (CALLED FROM "INVMAIN")
5010 'FUNCTION: TO LIST ALL RECORDS ON A DATA FILE
5020 ' AUTHOR : SPG              DATE: 1/19/79    REV. 7/80
5030 CLEAR 500: DEFINT A-Z
5040 '**                                  set file name from user.
5050 CLOSE: CLS
5060 INPUT "file name ('help'=list of files)";FL$
5070   IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
                                         ELSE OPEN "R",1,FL$
5080     IF LOF(1)=0 THEN PRINT "no such file.": GOTO 5040
5090 FIELD 1, 2 AS ROOT$, 2 AS NM$,15 AS DESC$
5100 GET 1, LOF(1)
5110 NM=CVI(NM$)
5120 CLS
5130 PRINT "file description is "; DESC$
5140 PRINT TAB(10); NM; "active items."
5150 B$=
"## ###  ###  %          %  %           %   ####.##   ####.##"
5160 GOSUB 5600
5170 FOR I=1 TO LOF(1)-1
5180   GOSUB 5500
5190   LOOP=LOOP+1
5200     IF LOOP>11 THEN GOSUB 5600
5210   PRINT USING B$; I, LL, RL, PT$, DES$, DP!, LP!
5220 NEXT I
5230 INPUT "/EN/"; A$: RUN "INVMAIN"
5235 '
5500 '*****                          field and get data record.
5510 PT$=""
5520 FIELD 1, 2 AS LL$,2 AS RL$,12 AS PT$,25 AS DES$,
           4 AS DP$,4 AS LP$
5530 GET 1,I
5540 LL =CVI(LL$): RL =CVI(RL$): PT$=PT$
5550 DP!=CVS(DP$): LP!=CVS(LP$)
5560 RETURN
5600 '*****                      subroutine to print headings.
5610 LOOP=0
5620 INPUT "/EN/";A$: CLS
5630 PRINT
"    ll   rl      pt         description       dp       lp"
5640 RETURN
5650 END
```

The output from the INVPR5 program looks like this.

```
    ll    rl      pt          description        dp        lp
 1   2     3          8430   steam skyhook      945.00   1488.00
 2   8     5          5643   small widget        45.66     61.50
 3   6     7         2456y   yellow wazoo         6.75      7.89
 4  18     8           789   fantasmagorion     -55.00     89.95
 5   9     0          565a   ten gallon hat      23.57     38.99
 6  20    10          8440   gas skyhook       1011.00   1444.88
 7  11    12         62651   older folder-h       0.69      0.89
 8  13    21          2397   right-handed e      13.80     20.99
 9   0    14          5648   large widget        59.90     88.99
10  15     0         2456s   striped wazoo        8.50     12.99
11   0    16         36433   funnelscope         67.80     76.89
/EN/? ←

    ll    rl      pt          description        dp        lp
12  17     0         62659   plain folder-h       0.79      0.99
13  22     0          2394   left-handed ea      14.60     21.88
14   0     0          5650   extra-large wi      72.88     99.80
15   0     0         2456b   blue wazoo           6.25      9.99
16   0     0         46559   towhook             42.84     49.95
17   0     0         62656   newer folder-h       1.99      2.39
18   0     0           562   right-angled f       0.57      1.29
19   0     0          565b   five gallon ha     -18.43     29.98
20   0     0          8432   electric skyho     893.00   1399.52
21   0     0          5440   flanged gromme       0.16      0.39
22  18     0          1984   reflanged grom       0.13      0.39
/EN/? l←
```

INVPR6

The INVPR6 program provides a list of the deleted parts by searching the desired file in sequence until it finds a negative dealer price, which is how the deletion program INVPR3 marks a record to be deleted. All the records with negative dealer prices are listed.

```
6000 'FILENAME: "INVPR6"       (CALLED FROM "INVMAIN")
6010 'FUNCTION: TO LIST DELETED PARTS
6020 ' AUTHOR : SPG             DATE: 1/19/79    REV. 7/80
6030 DEFINT A-Z
6040 B$=
" %          %  %                        %   ####.##    ####.##"
6050 '**                              set file name from user.
6060 CLOSE
6070 INPUT "file name ('help'=list of files)";FL$
6080   IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
6090   IF VAL(FL$)>0 THEN PRINT "no such file.": GOTO 6050
6095 OPEN "R",1,FL$
6097   IF LOF(1)=0 THEN PRINT "file is empty.": GOTO 6050
6100 FIELD 1, 4 AS F$, 12 AS PT$, 25 AS DES$,
             4 AS DP$, 4 AS LP$
6110 GOSUB 6220
6120 FOR I=1 TO LOF(1)-1
6130   GET 1,I:  DP!=CVS(DP$)
6140     IF DP!>0 THEN NEXT I: GOTO 6200
6150   LP!=CVS(LP$)
6160   LOOP=LOOP+1
6170     IF LOOP>=12 THEN INPUT "/EN/"; A$: GOSUB 6220
6180   PRINT USING B$; PT$, DES$, ABS(DP!), LP!
6190 NEXT I
6200 '**                                 return to menu driver.
6210 INPUT "/EN/"; A$: RUN "INVMAIN"
6220 '*****                                      print headins.
6230 CLS: LOOP=0
6235 PRINT "   L I S T   O F   D E L E T E D   I T E M S"
6237 PRINT
6240 PRINT
"   part #        description                dlr pr    list pr"
6250 RETURN
6260 END
```

Output from INVPR6 would look like this.

```
    L I S T   O F   D E L E T E D   I T E M S

    part #        description                dlr pr    list pr
          789  fantasmasorion                 55.00      89.95
         565b  five gallon hat                18.43      29.98
 /EN/? ←
```

INVPR7

The INVPR7 program performs the tree balancing necessary to maximize the efficiency of the BSST file structure. Table 13.3 shows a typical tree before and after balancing. Program INVPR7 causes this transformation.

	Before			After		
Record	Part #	LL	RL	Part #	LL	RL
1	7	2	3	8	2	3
2	2	4	6	4	4	5
3	8	0	5	12	6	7
4	1	0	0	2	8	9
5	11	9	10	6	10	11
6	4	15	7	10	12	13
7	5	0	8	14	14	15
8	6	0	0	1	0	0
9	9	0	14	3	0	0
10	15	11	0	5	0	0
11	14	12	0	7	0	0
12	12	0	13	9	0	0
13	13	0	0	11	0	0
14	10	0	0	13	0	0
15	3	0	0	15	0	0

Table 13.3 A Typical Tree Before and After Balancing

```
7000 'FILENAME: "INVPR7"
7010 'FUNCTION: TO BALANCE THE BINARY SEARCH TREE
7015 ' AUTHOR : SPG              DATE: 1/19/79    REV. 7/80
7020 '  after a method originally described by JDR
7025 CLS: PRINT: PRINT
7030 PRINT "     T H I S   P R O G R A M   B A L A N C E S
   T H E   B I N A R Y   S E A R C H   T R E E"
7035 PRINT: PRINT
7040 CLEAR 300: DEFINT A-Z: DIM MSK(200), KLN(200)
7045 '**                              get file name from user.
7050 CLOSE
7055 INPUT "file name ('help'=list of files)";FL$
7060   IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
7065   IF VAL(FL$)>0 THEN PRINT "no such file.": GOTO 7045
7070 OPEN "R",1,FL$
7075   IF LOF(1)=0 THEN PRINT "no such file.": GOTO 7045
7080 GOSUB 7650
7085 FOR S=1 TO LOF(1)-1
7090   GET 1,S
7095   LL=CVI(LJ$): RL=CVI(LK$): DP!=CVS(DP$)
7100     IF DP!<0 THEN 7110
```

```
7105   IF LL>0 AND RL=0 OR RL>0 AND LL=0 THEN Y=Y+1
7110 NEXT S
7115 BAL=100*(Y/(LOF(1)-2))
7120 PRINT "The tree is already "; 100-BAL; "% balanced."
7125 S=7*(LOF(1)-1): H=INT(S/3600): MI=INT(S/60)
7130 MI=MI-H*60: S=S-(H*3600+MI*60)
7135 PRINT "   Balancing the file could take as long as --"
7140 PRINT "    (h:m:s),   "; H; ":"; MI; ":"; S
7145 INPUT "Do you want it balanced further (y or n)";S$
7150   IF LEFT$(S$,1)<>"y" THEN RUN "INVMAIN"
7155 GOSUB 7600
7160 GET 1,LOF(1)
7165 N=CVI(NM$): R=CVI(ROOT$): D$=DESC$
7170 PRINT "N=";N,  "R=";R,  "DS$=";DESC$:  N1=N
7175 '              fill kln array with balanced pointers.
7180 K=1: M=INT((N1+1)/2)
7185 KLN(K)=M: MSK(M)=1: I=1: Q=I
7190 '**
7195 M=INT((M+1)/2): U=Q: Q=Q+Q
7200 FOR C=U TO Q-1
7205   L=KLN(C)-M
7210     IF L<1 OR L>N1 THEN 7220
7215     IF MSK(L)=0 THEN K=K+1: KLN(K)=L: MSK(L)=1
7220 '**
7225   L=KLN(C)+M
7230     IF L<1 OR L>N1 THEN 7240
7235     IF MSK(L)=0 THEN K=K+1: KLN(K)=L: MSK(L)=1
7240 '**
7245 NEXT C: I=I+1
7250   IF M>1 THEN 7190
7255 FOR I=1 TO N1:
       IF MSK(I) <> 1 THEN K=K+1: KLN(K)=I
7260 NEXT I
7265 PRINT "    N O W    C O P Y I N G"
7270 '  traverse, copying into "NEATFILE" in sorted order.
7275 CLOSE 2:  OPEN "R",2,"NEATFILE"
7280 I=0: T=0: P=R
7285 '**                          fetch and stack until p=0.
7290   IF P<>0 THEN T=T+1: STK(T)=P: GOSUB 7650: GET 1,P:
                    P=CVI(LJ$): GOTO 7285
7295 '      if t<>0 pop the stack, transfer to "NEATFILE".
7300   IF T<>0 THEN P=STK(T): T=T-1: I=I+1: GOSUB 7800:
                    P=CVI(LK$): GOTO 7285
7305 '               print out the positions to be filled.
7310 FOR I=1 TO N
7315   PRINT KLN(I),
7320 NEXT I: PRINT
```

```
7325 '        build tree, simply traverse and rewrite FL$.
7330 '  note that KLN contains pointers for next new file.
7335 R=0: T=0: CLOSE 1: KILL FL$: OPEN "R",1,FL$
7340 '**      set record from "NEATFILE" accordins to KLN.
7345 I=R: T=T+1: L=KLN(T)
7350 GOSUB 7900: GET 2,L
7355   IF I=0 THEN R=T: GOSUB 7650: GOTO 7400
7360 '**                          set new record.
7365 GOSUB 7700
7370   IF P2$=PT$ THEN PRINT P2$; "duplicate": GOTO 7435
7375 '                  rebuild file in BSST order onto FL$.
7380   IF P2$>PT$ THEN 7390
7385 IF LL=0 THEN LSET LJ$=MKI$(T): PUT 1,I: GOTO 7400
           ELSE I=LL: GOTO 7360
7390 '**
7395 IF LR=0 THEN LSET LK$=MKI$(T): PUT 1,I
           ELSE I=LR: GOTO 7360
7400 '**                                  put variables in field.
7405 LSET LJ$=MKI$(0): LSET LK$=MKI$(0): RSET PT$=P2$
7410 LSET DES$=D$: LSET DP$=D2$: LSET LP$=L2$
7415 LSET OH$=O2$: LSET OO$=O3$: LSET SB$=S2$: LSET M$=M2$
7420 LSET N$=N2$: LSET O$=O4$: LSET P$=P4$
7425 '      PRINT THE PART NUMBER
7430 PRINT USING "%           %"; PT$;: PUT 1,T
7435 '**                          check for end of loop.
7440   IF T<N THEN 7340
7445 '         rewrite new root record on new "INVENDAT".
7450 GOSUB 7600
7455 LSET ROOT$=MKI$(1): LSET NM$=MKI$(N): LSET DESC$=D$
7460 PUT 1, LOF(1)+1
7465 '                retrun to INVMAIN after cleanins up.
7470 KILL "NEATFILE": PRINT: PRINT
7475 PRINT "b a l a n c i n g   i s   c o m p l e t e."
7480 INPUT "/EN/"; A$: RUN "INVMAIN"
7485 '
7600 '*****                           field the root record.
7605 FIELD 1, 2 AS ROOT$, 2 AS NM$, 25 AS DESC$
7610 RETURN
7650 '*****                     routine to field data record.
7655 FIELD 1, 2 AS LJ$, 2 AS LK$, 12 AS PT$, 25 AS DES$,
      4 AS DP$, 4 AS LP$, 2 AS OH$, 2 AS OO$, 2 AS SB$,
      2 AS M$, 2 AS N$, 2 AS O$, 2 AS P$
7660 RETURN
7700 '*****       routine to fetch a data record from FL$.
7705 GOSUB 7655: GET 1,I: LL=CVI(LJ$):
     LR=CVI(LK$):  RETURN
```

```
7800 '*****  routine to set from FL$, write on "NEATFILE".
7805 GOSUB 7650: GET 1,P
7810 GOSUB 7900: L3$=LJ$: L4$=LK$
7815 LSET P2$=PT$: LSET DS$=DES$: LSET D2$=DP$
7820 LSET L2$=LP$: LSET O2$=OH$: LSET O4$=OO$: LSET S2$=SR$
7825 LSET M2$=M$: LSET N2$=N$: LSET O4$=O$: LSET P4$=P$
7830   IF I>=1 THEN
                 I1$=PT$: PRINT I; I1$;" description=  ";DES$
7835 PUT 2,I: RETURN
7840 RETURN
7900 '*****        routine to describe data on "NEATFILE".
7905 FIELD 2, 2 AS L3$, 2 AS L4$, 12 AS P2$, 25 AS DS$,
               4 AS D2$, 4 AS L2$,  2 AS O2$,  2 AS O3$,
               2 AS S2$, 2 AS M2$,  2 AS N2$,  2 AS O4$,
               2 AS P4$
7910 RETURN
7920 END
```

The balancing program is an example of a program communicating to the user its state of execution. It not only calculates how well balanced the tree is (by counting unbalanced terminal nodes) but it prints the records that it is copying as it performs the balancing.

```
 T H I S   P R O G R A M   B A L A N C E S
T H E   B I N A R Y   S E A R C H   T R E E


file name ('help'=list of files)? parts
the tree is already  84 % balanced.
   balancing the file could take as long as --
    (h:m:s),    0 : 0 : 49

do you want it balanced further (y or n)? y
N= 7            R= 1             DS$=parts
    N O W    C O P Y I N G
 1         2923 description=  towhook
 2        36932 description=  ear trumpet
 3       245623 description=  blue wazoo
 4       564322 description=  widget
 5       843234 description=  skyhook
 6      3643325 description=  funnelscope
 4              2               6              1
 3              5               7
      564322         36932       3643325         2923        245623
       843234

b a l a n c i n g   i s   c o m p l e t e.
/EN/? 1←
```

INVPR8

The INVPR8 program accesses the file called FILENAME, which is a file whose sole purpose is to record the names of all files used in the system.

```
8000 'FILENAME: "INVPR8"    (CALLED FROM MOST PROGRAMS)
8010 'FUNCTION: LISTS ALL OF THE FILES IN USE
8020 ' AUTHOR : SPG              DATE: 1/19/79     REV. 7/80
8030 CLEAR 1000: DEFINT A-Z: CLS: S=0
8040 PRINT "        T H I S   P R O G R A M   L I S T S
       A L L  O F  T H E  F I L E S  I N  U S E:"
8050 PRINT: PRINT
8060 C$="%      %      %      %      %      %     %      %"
8070 OPEN "R",1,"FILENAME": FIELD 1, 255 AS A$
8080 S=S+1
8090   IF LOF(1)=S-1 THEN 8160
                     ELSE GET 1,S
8100 FOR L=1 TO 223 STEP 32: IF MID$(A$,L,1)=" " THEN 8160
8110   IF MID$(A$,L,1)=" " THEN 8160
8120     FOR I=L TO L+24 STEP 8
8130       PRINT USING C$; MID$(A$,I,I+7);
8140     NEXT I
8150 NEXT L: GOTO 8080
8160 INPUT "/EN/";A$: RUN "INVMAIN"
8170 END
```

The output produced by INVPR8 looks like this.

```
        T H I S   P R O G R A M   L I S T S
        A L L  O F  T H E  F I L E S  I N  U S E:


parts         moparts                                  /EN/? ←
```

INVPR9

The INVPR9 program removes the name of a file from the index file "FILENAME" and kills that file if the user wishes.

```
9000 'FILENAME: "INVPR9"
9010 'FUNCTION: TO DELETE ANY SELECTED FILE
9020 ' AUTHOR : SPG                    DATE: 1/19/79    REV. 7/80
9030 CLS
9040 PRINT "  F I L E   D E L E T I O N   P R O G R A M"
9050 PRINT: PRINT
9060 INPUT "name of file to be deleted ('help'=list)"; FL$
9070 IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
9080 OPEN "R",1,FL$
9090 IF LOF(1)=0 THEN PRINT "file contained no data.":
                      PRINT "It has been deleted.":
                      PRINT "Will be deleted from 'help'":
                      GOTO 9140
9100 PRINT "file contains"; LOF(1); "data items."
9110 PRINT "Are you sure you want it deleted (Y=yes)"
9120 INPUT B$
9130   IF B$<>"Y" THEN 9260     'note that Y is capitalized.
9140 CLOSE: KILL FL$           'delete the data file itself.
9150 OPEN "R",1,"FILENAME"     'then delete from "FILENAME".
9160 FL$=FL$+STRING$(8-LEN(FL$)," ")
9170 FIELD 1, 255 AS N$
9180 M=M+1
9190 GET 1,M
9200   FOR J=1 TO 255 STEP 8
9210     IF MID$(N$,J,1)=" " THEN 9250
9220     IF MID$(N$,J,8)<>FL$ THEN NEXT J: GOTO 9180
9230 N$=LEFT$(N$,J-1)+MID$(N$,J+8,LEN(N$))
9240 PUT 1,M
9250 PRINT "Deletion completed.  ";
9260 PRINT "Returning to main menu."
9270 RUN "INVMAIN"
9280 END
```

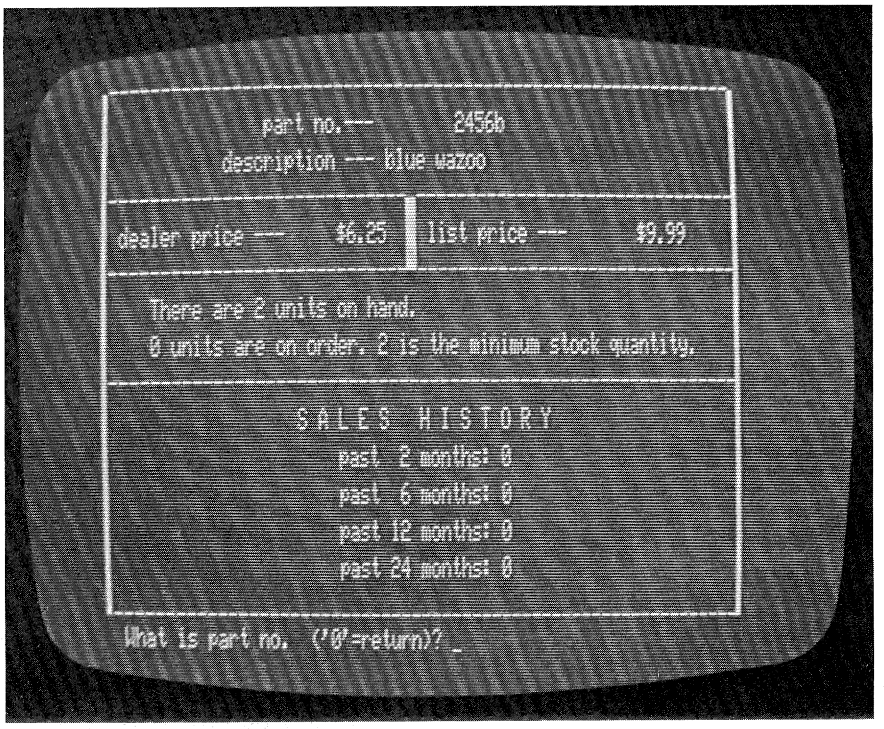
INVPR10

The INVPR10 program uses the BSST access algorithm to get a specific record and then it displays all of the information on that record. The display is designed specifically for the record contents of this system.

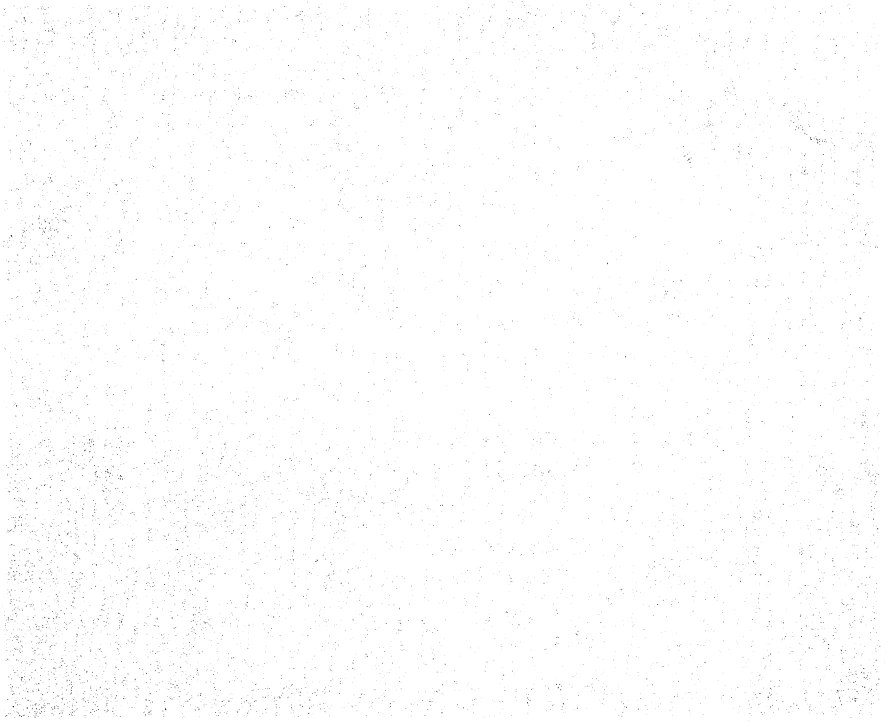
```
10000 'FILENAME: "INVPR10"          (CALLED FROM "INVMAIN")
10010 'FUNCTION: TO PRINT THE CONTENTS OF ANY RECORD.
10020 ' AUTHOR : SPG              DATE: 1/19/79    REV. 7/80
10030 DEFINT A-Z:  CLEAR 500
10040 '**                set file name from user.
10050 CLOSE
10060 INPUT "What is file name ('help'=list)"; FL$
10070   IF FL$="" THEN 10040
10080   IF FL$="help" OR FL$="HELP" THEN RUN "INVPR8"
10090 OPEN "R",1,FL$
10100   IF LOF(1)=0 THEN PRINT "no such file.": GOTO 10040
10110 GOSUB 10800
10120 GET 1,LOF(1): R=CVI(ROOT$): N=CVI(NM$)
10130 '
10140 '
10150 '**   set new part no., N1$: set W to latest root.
10160 N1$=""
10170 INPUT "  What is part no.  ('0'=return)";N1$
10180 W=R
10190 'pad part number to 12 characters with blanks.
10200 N1$=RIGHT$("              "+N1$,12)
10210 R$=RIGHT$(N1$,1): IF R$=" " THEN 10660
10220 '         set root.  if part is not root, search on.
10230 I=R: GOSUB 10700: LL=CVI(LL$): RL=CVI(RL$)
10240   IF N1$<>PT$ THEN 10340
10250 '         if no right branch, new root is left link.
10260   IF RL=0 THEN R=LL:  GOTO 10150
10270 '         if no left branch, new root is right link.
10280   IF LL=0 THEN R=RL:  GOTO 10150
10290 '    neither link is null.  new root is right link.
10300 F=R:  R=RL:  S=0
10310 '**               part number not found.
10320 IF W=0 THEN INPUT "part not on file. /EN/"; A$:
                  GOTO 10150
10330 '
10340 '**          name is not root.  look some more.
10350 I=W:  GOSUB 10700:  LW=CVI(LL$):  RW=CVI(RL$):
      IF N1$=PT$ THEN 10380
10360 '           name still not found.  look again.
10370 F=W:
      IF N1$<PT$ THEN S=1:  W=LW:  GOTO 10310
                 ELSE S=0:  W=RW:  GOTO 10310
```

```
10380 '**                                   name is found.
10390 GOSUB 10900
10400 DP!=CVS(DP$): OH=CVI(OH$): M=CVI(M$): N=CVI(N$)
10410 LP!=CVS(LP$): OO=CVI(OO$): O=CVI(O$): P=CVI(P$)
10420 SB=CVI(SB$)
10430 CLS
10440 PRINT STRING$(63,"-")
10450 PRINT "                    part no.--- "; PT$
10460 PRINT "               description --- "; DES$
10470 PRINT STRING$(63,"-")
10480 PRINT "  dealer price --- ";
10490 PRINT USING "$$,###.##"; DP!;
10500 PRINT "    list price ---   ";
10510 PRINT USING "$$,###.##"; LP!
10520 PRINT STRING$(63,"-")
10530 PRINT "     There are"; OH; "units on hand."
10540 PRINT TAB(4); OO; "units are on order.";
10550 PRINT SB; "is the minimum stock quantity."
10560 PRINT STRING$(63,"-")
10570 PRINT TAB(19); "S A L E S   H I S T O R Y"
10580 PRINT TAB(23); "past  2 months:"; M
10590 PRINT TAB(23); "past  6 months:"; N
10600 PRINT TAB(23); "past 12 months:"; O
10610 PRINT TAB(23); "past 24 months:";P
10620 PRINT STRING$(63,CHR$(95))
10630 FOR Y=10 TO 15: SET (60,Y): SET (61,Y): NEXT Y
10640 FOR Y=1 TO 43: SET (1,Y): SET (126,Y): NEXT Y
10650 GOTO 10150
10660 '**                          return to menu driver.
10670 INPUT "/EN/";A$: RUN "INVMAIN"
10700 '*****           routine to fetch the Ith record.
10710 GOSUB 10900:  GET 1,I: PT$=P1$: DP!=CVS(DP$)
10720 RETURN
10800 '*****       routine to describe the root record.
10810 FIELD 1, 2 AS ROOT$, 2 AS NM$, 25 AS DESC$
10820 RETURN
10900 '*****       routine to describe the data record.
10910 FIELD 1, 2 AS LL$, 2 AS RL$, 12 AS P1$, 25 AS DES$,
              4 AS DP$, 4 AS LP$,  2 AS OH$,  2 AS OO$,
              2 AS SB$, 2 AS  M$,  2 AS  N$,  2 AS  O$,
              2 AS  P$
10920 RETURN
10930 END
```

This is what the screen looks like when it displays the information on an item on file.



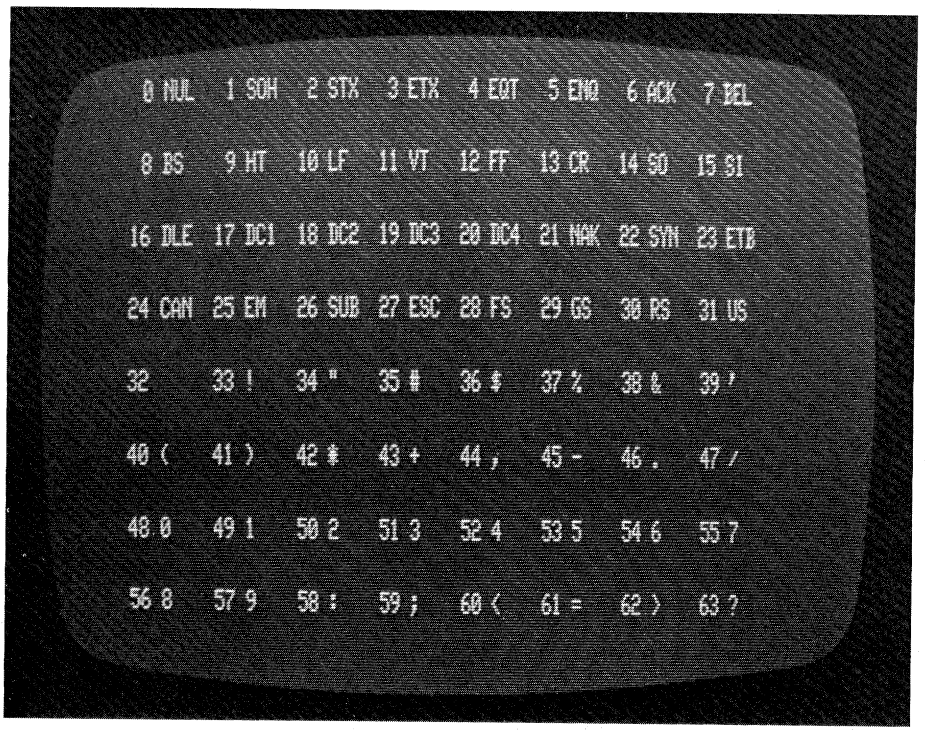
Well, there it is. This last system represents a high degree of sophistication on a microcomputer. It contains examples of some of the best building, accessing, updating, and sorting techniques that are available today, on any computer. When you have mastered these techniques, you will have at your disposal the variety of tools that will provide you with both breadth and depth in the application of your craft. We can only suggest that you keep using these tools, and others as you discover them, to further sharpen your expertise.

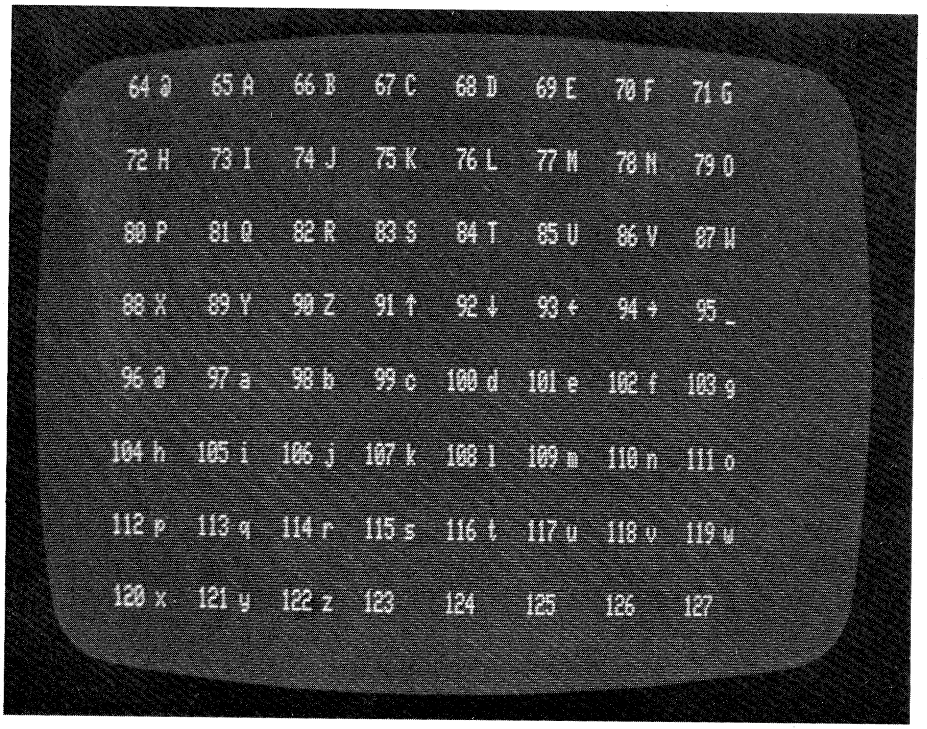


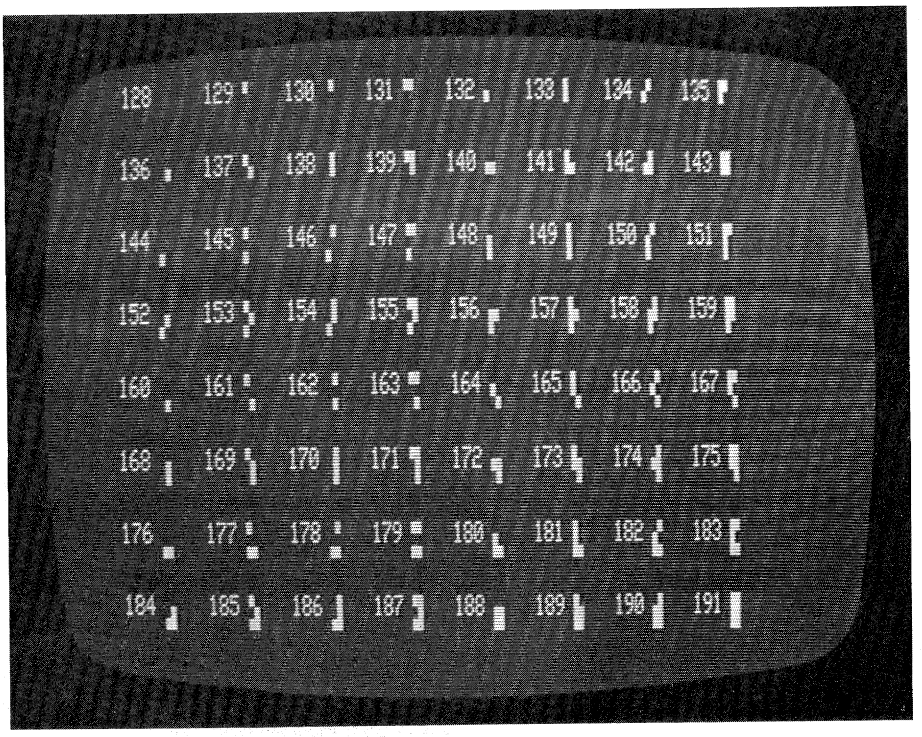
Appendix

The table that follows this brief discussion compares three popular versons of BASIC. The three have been abbreviated M, E, and D: M stands for Microsoft's BASIC as implemented on the TRS-80, including the enhancements in TRSDOS; E stands for BASIC-E Version 2.0, a somewhat dated (late 1978) version of one of the most popular compiler-type BASICs; and D stands for DEC BASIC PLUS, Digital Equipment Company's most popular minicomputer BASIC. BASIC-PLUS is included here in order to compare an "old" (early '70s) version of the language formed on much more expensive hardware to the newer BASICs developed strictly for microcomputers. The reader should note also that some highly advanced versions of BASIC-E exist, such as CBASIC or CBASIC-2.

Type of BASIC			BASIC Features
M	E	D	
N	Y	Y	Compiler that produces storable object code
Y	N	N	Direct, or calculator, mode of operation
Y	N	N	Built-in editor
N	N	Y	RENUM or RESEQ
Y	N	N	TRACE to help debussing
Y	Y	N	Long variable names (more than letter-disit)
Y	N	N	Double precision variables
Y	N	N	Inteser variables
Y	N	N	Hex and octal variables
Y	N	N	VARPTR
N	N	Y	Multiline user-definable functions
Y	N	N	Error trapping with ON ERROR - GOTO
Y	N	Y	Text error messages
Y	N	Y	Implicit dimensioning; Default size is 10
N	N	Y	Matrix instructions, e.g. MAT X=INV(Y)
N	N	Y	Matrix I/O
Y	N	N	? substitutes for PRINT
Y	N	Y	' substitutes for REM
Y	Y	Y	IF - THEN - ELSE
Y	N	Y	PRINT USING
Y	N	Y	INSTR finds the position of a character
Y	N	N	STRING$ prints a series of characters
Y	N	N	PEEK & POKE
Y	N	Y	TIME$ accesses system clock


0 NUL 1 SOH 2 STX 3 ETX 4 EOT 5 ENQ 6 ACK 7 BEL
8 BS 9 HT 10 LF 11 VT 12 FF 13 CR 14 SO 15 SI
16 DLE 17 DC1 18 DC2 19 DC3 20 DC4 21 NAK 22 SYN 23 ETB
24 CAN 25 EM 26 SUB 27 ESC 28 FS 29 GS 30 RS 31 US
32 33 ! 34 " 35 # 36 $ 37 % 38 & 39 '
40 (41) 42 * 43 + 44 , 45 - 46 . 47 /
48 0 49 1 50 2 51 3 52 4 53 5 54 6 55 7
56 8 57 9 58 : 59 ; 60 < 61 = 62 > 63 ?


64 @ 65 A 66 B 67 C 68 D 69 E 70 F 71 G
72 H 73 I 74 J 75 K 76 L 77 M 78 N 79 O
80 P 81 Q 82 R 83 S 84 T 85 U 86 V 87 W
88 X 89 Y 90 Z 91 ↑ 92 ↓ 93 ← 94 → 95 _
96 ` 97 a 98 b 99 c 100 d 101 e 102 f 103 g
104 h 105 i 106 j 107 k 108 l 109 m 110 n 111 o
112 p 113 q 114 r 115 s 116 t 117 u 118 v 119 w
120 x 121 y 122 z 123 124 125 126 127



ASCII codes 192 through 255 are called space compression codes because printing one of these characters causes tabbing for 0 to 63 spaces.

```
10 'FILENAME: "APPENDIX"
20 'FUNCTION: PRODUCE ASCII CODES & CHARACTERS IN 3 SCREENS
30 '  AUTHOR : JDR                              DATE 8/80
40 CLEAR 1000 : CLS : DIM A$(191)
50 FOR I=0 TO 31 : READ X$ : A$(I)=LEFT$(X$+" ",3) : NEXT I
60 DATA NUL,SOH,STX,ETX,EOT,ENQ,ACK,BEL,BS,HT,LF
70 DATA VT,FF,CR,SO,SI,DLE,DC1,DC2,DC3,DC4,NAK
80 DATA SYN,ETB,CAN,EM,SUB,ESC,FS,GS,RS,US
90 FOR I=32 TO 191 : A$(I)=CHR$(I) : NEXT I
100 FOR M=0 TO 128 STEP 64 : N=M+63
110     GOSUB 140    ' print screen of ASCII codes
120     FOR K=1 TO 1000 : NEXT K : CLS    ' delay
130     NEXT M : STOP
140 ' subroutine to display 64 codes and ASCII equivalents
150 FOR I=M TO N
160     IF I=INT(I/8)*8 THEN PRINT
170     IF I<32 THEN PRINT RIGHT$("  "+STR$(I),3);" ";A$(I);" ";
              ELSE PRINT RIGHT$(STR$(I),3);" ";A$(I);"    ";
180 NEXT I : RETURN
190 END
```

Address		Hardware	Memory Contents
Hex	Decimal		
0000	0	1K ROM	I/O Drivers and Bootstrap
0400	1024	11K ROM	Level II BASIC, Disk BASIC
3000	12288	2K ROM	I/O addresses
3800	14336	1K RAM	Keyboard memory
3C00	15360	1K RAM	CRT screen memory
4000	16384	5K RAM	BASIC vectors
4200		4K RAM	TRSDOS
5200		6.5K RAM	Disk BASIC TRSDOS Utilities
6C00		1K RAM	User Memory
7000		4K RAM	User Memory (to 16K system)
8000	32768	16K RAM	User Memory (to 32K system)
C000		16K RAM	User Memory (to 48K system)
FFFF	65535		

@ (Lower Case)
ABS
AND
ASC
ATN
CDBL
CHR$
CINT
CLEAR
* CLOSE
CLS
* CMD
CONT
COS
CSNG
* CVD
* CVI
* CVS
DATA
DEFDBL
* DEFFN
DEFINT
DEFSNG
* DEFUSR
DEFSTR
DELETE
DIM
EDIT
ELSE
END
EOF
ERL
ERR
ERROR
EXP
* FIELD
FIX
FOR
FRE
* GET
GOSUB
GOTO
IF
INKEY$
INP
INPUT
* INSTR
INT
* KILL
LEFT$
LET
LEN
LINE
LIST
* LOAD
* LOC
* LOF
LOG
* LSET
MEM
* MERGE
MID$
* MKD$
* MKI$
* MKS$
NEW
NEXT
NOT
ON
* OPEN
OR
OUT
PEEK
POINT
POKE
POS
PRINT
* PUT
RANDOM
READ
REM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
* RSET
* SAVE
SET
SGN
SIN
SQR
STEP
STOP
STRING$
STR$
TAB
TAN
THEN
* TIME$
TROFF
TRON
USING
* USR
VAL
VARPTR

* indicates Level II Disk BASIC

```
Extended Utility Commands (may use all of memory)
  BACKUP -- duplicate an entire diskette
  FORMAT -- prepare a data diskette

Auxilliary Utility Programs (may use all of memory)
  TAPEDISK     -- copy a tape file to a disk file
  DISKDUMP/BAS -- display a disk file

System Commands (load below 5200 Hex)
  BASIC2 -- Jump to Level II BASIC
  DEBUG  -- examine and alter resisters and ROM
  TRACE  -- display the PC resister

Library Commands
  AUTO   -- modify the power-up sequence
  ATTRIB -- set the protection level on a file
  CLOCK  -- display the system time on screen
  COPY   -- duplicate a file
  DATE   -- set the system date
  DUMP   -- copy memory to a disk file
  KILL   -- delete a file
  FREE   -- display the amount of free space on all drives
  LIB    -- display all library commands
  LIST   -- display a text file
  LOAD   -- load a machine language file
  PRINT  -- list a text file on the line printer
  PROT   -- change the protection of all non-system files
  RENAME -- change a file's name
  TIME   -- set the system clock
  VERIFY -- have TRSDOS verify all user disk writes
```

Level II Error Codes

Code	Abbreviation	Error
1	NF	NEXT without FOR
2	SN	Syntax error
3	RG	RETURN without GOSUB
4	OD	Out of data
5	FC	Illegal function call
6	OV	Overflow
7	OM	Out of memory
8	UL	Undefined line
9	BS	Subscript out of range
10	DD	Redimensioned array
11	/0	Division by zero
12	ID	Illegal use of a direct command
13	TM	Type mismatch
14	OS	Out of string space
15	LS	String longer than 255 characters
16	ST	String formula too complex
17	CN	Can't CONT after END or EDIT
18	NR	No RESUME before end of program
19	RW	RESUME without error
20	UE	Unprintable error after using ERROR
21	MO	Missing operand
22	FD	Bad file data
23	L3	Disk BASIC only
24-49	not used	

Disk BASIC Error Codes

Code	Message	Error
50	FIELD OVERFLOW	More than 255 bytes for a buffer
51	INTERNAL ERROR	DOS or disk I/O fault
52	BAD FILE NUMBER	File number used without OPEN
53	FILE NOT FOUND	Read attempted on nonexistent file
54	BAD FILE MODE	Disk I/O conflicts with OPEN mode
55,56	not used	
57	DISK I/O ERROR	Error during data transfer
58-60	not used	
61	DISK FULL	No more room
62	INPUT PAST END	End of file reached during seq. input
63	BAD RECORD NUMBER	Record number <1 or >340
64	BAD FILENAME	Invalid file specification
65	not used	
66	DIRECT STATEMENT IN FILE	Attempt to LOAD, RUN, or MERGE a data file
67	TOO MANY FILES	Tried 49th. file on diskette
68	DISK WRITE-PROTECTED	Write-protect notch was covered
69	FILE ACCESS DENIED	Wrong password

Index

Introducing . . .
An exciting new breed of COMPUTER GUIDEBOOKS designed to help you work SMARTE HARDER:

THE MICROCOMPUTER POWER SERIES

GUIDE TO SYSTEMS APPLICATIONS
John P. Grillo
J. D. Robertson
288 pages/**61 programs**
$17.95

This book gives an amazingly clear presentation of the complete microcomputer system. It begins at an elementary level and builds your knowledge of hardware and software to the level necessary to understand the complexities of microcomputer systems and their applications. Its unique APPLICATIONS approach stresses the programs (rather than the hardware) and how they are created to build applications systems.

This book includes **61 programs** that have been tested on a TRS-80. With a minimum of conversion all programs can be used on other systems such as PET, Apple II, etc. Some of the programs are menu display manager, mailing list system, Sheli-Metzner sorts, animal guessing game and many, many more.

TECHNIQUES OF BASIC
John P. Grillo
J. D. Robertson
272 pages/**61 programs**
$18.95

This book will free you from most of the restrictions of standard BASIC by letting you progress step by step from very elementary to more sophisticated features of BASIC. This book demonstrates techniques that are as USABLE as they are fascinating.

This book includes **61 different programs** in TRS-80 Level II BASIC covering such diverse topics as simple data analysis, conversational statistics, isotherm graphing, text processing, line renumbering, banner printing, message managing, error trapping, realtime stopwatch, Shell sorts, Quicksort, Monte Carlo techniques, binary search trees and more.

INTRODUCTION TO GRAPHICS
John P. Grillo
J. D. Robertson
144 pages/**38 programs**
$15.95

This Guidebook was written for three different audiences: 1) The novice who has just bought or been introduced to the personal computer and who wants to learn how to control what is being done on the screen . . . 2) The more experienced programmer who is just getting into computer graphics . . . and 3) The individual, possibly an educator, who knows one of the best ways to introduce YOUNGSTERS to computers is via the fascinating GRAPHICS route.

The **38 graphic generation programs** range from extremely simple to a moderately advanced level of sophistication. There's a goldmine of ideas for demonstrating the many exciting graphic capabilities of the TRS-80 and the illustrations are truly ingenious.

The programs are short enough to key in quickly and are excellent examples of the techniques being discussed. Altogether a great source book of ideas for fun things that will arouse and maintain interest and inspire the desire to progress.

DATA MANAGEMENT TECHNIQUES
John P. Grillo
J. D. Robertson
208 pages/**48 programs**
$16.95

Designed for the intermediate to advance programmer, this is the Guidebook that really SHOWS you how to use your computer more effectively. The authors' overall goal is to demonstrate and explain the many ways you can manage data . . . both in memory and on disk or tape files. Included are some of the most popular methods (such as list and array processing) and some less well known, but very powerful methods (such as queue, stack and tree processing). Every technique is illustrated in a simple, straightforward manner with BASIC programs.

Included are full listings and output for **48 programs.** These programs illustrate techniques such as sorting, ISAM file creation and upkeep, linked list file management, generalized information system with a binary sequence search tree, automated code generation, master file maintenance with transaction file merging and multilist file management system.

TAKE YOUR PICK . . . OR TAKE THEM ALL.
Yours to examine **RISK-FREE** for 15 days.

Send your order for the PERSONAL COMPUTER BOOKS to. . . .

Direct Marketing Division
Wm. C. Brown Company Publishers
2460 Kerper Blvd.
Dubuque, Iowa 52001

(See order card on last page of this book)

wcb Personal Computer Series

ISBN 0-697-09951-2