FEATURING ORIGINAL PROGRAMS AND EXERCISES

# UNDERSTANDING APPLE® BASIC

Step-by-step Instruction for the Beginner

**An Alfred Handy Guide**


by Richard G. Peddicord

# UNDERSTANDING APPLE® BASIC

by Richard G. Peddicord

AN ALFRED HANDY GUIDE

Computer Series Editor:
George Ledin Jr.

ALFRED PUBLISHING CO., INC.
SHERMAN OAKS, CA 91403

This Handy Guide is not a publication of Apple Computer Inc. and should not be used in lieu of the instruction manuals that accompany their products. All information regarding Apple computers may not be accurate or completely up to date.

Editorial Supervision: Joseph W. Cellini

Cover Design: Paula Bingham Goldstein



Alfred Publishing Co., Inc.
15335 Morrison St.
PO Box 5964
Sherman Oaks, CA 91413

**Library of Congress Cataloging in Publication Data**

Peddicord, Richard G.
Understanding Apple BASIC.

(An Alfred Handy Guide)
Bibliography: p.
Includes index.
1. Apple // (Computer)—Programming. 2. Basic (Computer program language). I. Title.
QA76.8.A662P43 1983 001.64'2 83-15554
ISBN 0-88284-246-3

# CONTENTS

# 1. INTRODUCTION

The Apple // computer continues to be one of the most popular microcomputers of all time. The version of BASIC that has become standard on these machines is called Applesoft BASIC, and it is one reason why the Apple is so popular.

Applesoft BASIC lets you do things that other versions of BASIC do not, such as put many statements on the same line, use long variable names, and omit certain parameters. Kids like it because it permits lots of variation, and because it is user-friendly.

If you want to learn Applesoft BASIC, this Handy Guide is a good place to start. It is very important that you work the examples on an Apple computer and that you try some of the exercises. Otherwise you may forget, or never understand, some of the concepts.

Many schools and libraries have one or more Apples. Phone around and ask if you can use one. Many computer stores have Apples, and perhaps even some of your friends have them. And of course you can always buy one. The important thing is to be able to use an Apple when you want to.

# 2. ABOUT YOUR APPLE

A great variety of software and hardware are available for the Apple // computer, and its Applesoft BASIC has become one of the more flexible BASICs in operation today.

## THE SCREEN

The Apple // features a high-resolution color screen of 280 columns and 196 rows, with six different colors (black, white, orange, pink, blue, and green) available for each *pixel*, or picture element. This is about the same resolution and color range as the personal game computers on the market today. It follows that you can program games on your Apple //. It is not possible to attain the same speed as these commercial programs, however, because you will be working in Applesoft BASIC, and the commercial programs are written in *assembly language*, which is faster because it's closer to the machine language itself.

The low-resolution screen on the Apple // has 40 columns and 40 rows, and each pixel can be one of 16 different colors from black to white. The reason why you have more choice of color with the lower resolution has to do with the way the color TV signal is generated and how the color image is stored.

## HOOKING UP YOUR APPLE

The configuration that has proven to be most cost effective and useful consists of an Apple // computer with one disk drive connected to a TV set via an RF modulator. The RF modulator converts the Apple's TV signal into the same kind of signal that the television receiver is expecting. Most RF modulators for computers generate an output signal that is picked up on Channel 3 or 4.

The first thing you must do before turning on the power is to check the connections on your system. Your Apple // computer has a power cord that plugs into the back (see Figure 2.1). Plug this into a *grounded* 115-volt AC circuit. If you try to use an ungrounded circuit, the manufacturer's warranty is void, and for good reason.

Coming out of the Apple will be what looks like a hi-fi cable, and indeed it has the same kind of plug on the end, a so-called RCA phono plug. This plugs into a little metal box that attaches to your TV set antenna, and to your outside TV antenna. There will be a sliding switch on the box that switches power between the computer and the antenna. This arrangement permits the family color TV set to be used as your Apple's output device.

You may or may not have a disk drive connected to your Apple. If one is connected, so much the better. If one is not connected, do not attempt to connect one using only this Handy Guide. You must use the instructions supplied with the drive. You can, however, hook up your Apple // to a TV set and turn on the power to

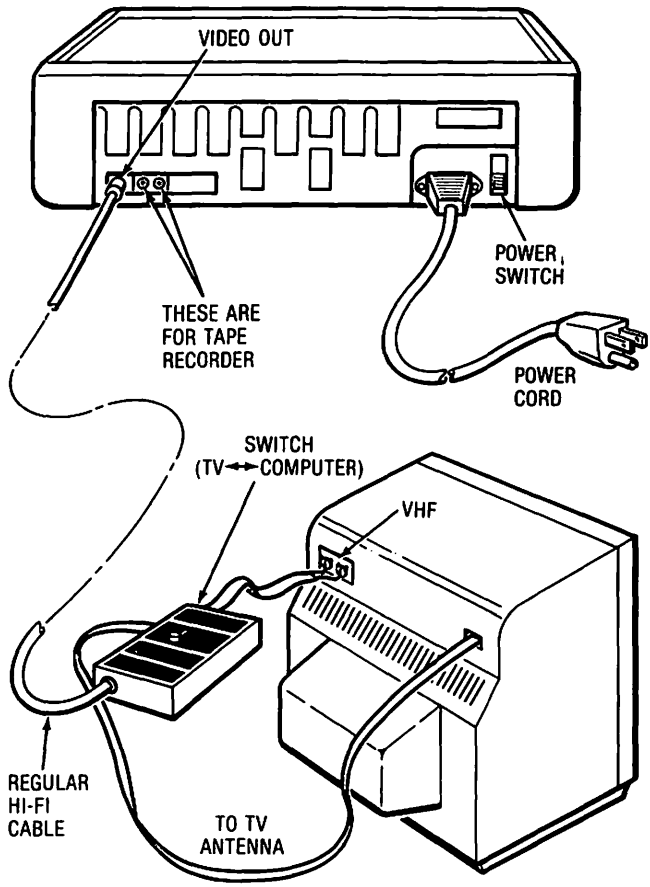



**Figure 2.1** Hooking up your Apple //.

both. The Apple's power switch is located in the back on the far left. It is a rocker switch; push the top part in to turn the power on. The power light in the lower left corner of the keyboard will come on.

## THE KEYBOARD

A diagram of the Apple // keyboard is shown in Figure 2.2. Your first step is to try out a few of the essential keys. The RESET key is most useful. Try it. You will hear a bleep from the Apple's speaker and the flashing square cursor will be immediately to the right of the Applesoft prompt character, ] . Whatever program you were working on will still be in memory, and whatever shapes you had entered will still be there. In other words, no real harm is done if you press RESET.

The SHIFT key works the same as in a typewriter: hold it down while you strike a key; the uppercase version of the character will be sent to the central processing unit.

In Applesoft, you enter characters one at a time and then press RETURN. Until you press RETURN, the machine has no idea what you have typed in. Therefore you can make all the corrections you want to a line before sending it centrally. The *arrow* or *cursor* keys (on the right side of the keyboard) are extremely useful

Figure 2.2 The Apple // keyboard.

in making corrections, since they move the cursor across text on the screen without changing it. Only when you press another key will the indicated change be made. You will use these keys often.

If you want to move the cursor over many characters, you can hold down the repeat key (REPT) while you hold down one of the arrow keys. This same REPT key will cause any character to be repeated.

Some commands that you should be aware of are listed below:

| | |
|---|---|
| HOME | Clears text and moves cursor to top left. |
| DEL | Put beginning line number after DEL, then a comma, then the ending line number. Every line from begin to end will be DELeted from program. |
| LIST | Lists your program. |
| RUN | Runs your program. |
| PR#1 | Enables printer in slot 1. |
| PR#6 | Enables disk drive in slot 6. |
| PR#0 | Disables printer. |

## EXERCISES

1. Play follow the leader with us. Simply do each thing listed below as many times as indicated:
   a. Press RESET 3 times
   b. Press SPACE BAR 2 times
   c. Press RETURN 1 time
   d. Press your nose 1 time
   e. Press letter M 10 times
   f. Press left arrow (←) 10 times
   g. Type PRINT ''GOTCHA'' 1 time
   h. Press RETURN 10 times

# 3. A PRINTING CALCULATOR

Applesoft BASIC has a feature that is quite useful if you have some arithmetic calculations to perform. Basically all you have to do is type the word PRINT followed by an arithmetic expression that the machine can understand, hit RETURN, and the machine will give you the correct answer.

## ADDITION AND SUBTRACTION

Try this one:

```
PRINT 5+6 (hit RETURN)
11
```

You can put several additions or subtractions in one expression. If you don't use parentheses they will be performed from left to right. For example,

```
PRINT 5+6-7-3+4
5
```

The machine adds 5 and 6 to get 11, then subtracts 7, then subtracts 3, then adds 4.

If you put in parentheses, you can change the meaning, as in:

```
PRINT 5+6-(7-3)+4
11
```

Because the parentheses are there, the machine must subtract the 3 from the 7 before it does anything else.

## MULTIPLICATION

Multiplication is indicated by the asterisk *, as in

```
PRINT "13 TIMES 14 IS "13*14
13 TIMES 14 IS 182
```

Notice that if you put a message in quotes it will print exactly what you wrote. This allows you to identify what the numbers are.

If you need to multiply a sum of items by a given number, as in a sales tax calculation, you would use parentheses:

```
PRINT (1.26+0.23+4.99)*0.6
.3888
```

## DIVISION

Division is indicated with the slash sign (/). As with multiplication, if you want to divide a sum of numbers by a certain number, be sure to put all the terms of the sum within parentheses. This forces the machine

to add them together before it divides. Say you want the average of 5, 43, and 67. This will give it to you:

```
PRINT "AVERAGE IS ";(5+43+67)/3
AVERAGE IS 38.3333333
```

If you have an expression with only multiplication and division, the operations will be performed from left to right. Thus:

```
PRINT 1/2*3
1.5
```

But:

```
PRINT 1/(2*3)
.166666667
```

We say that multiplication and division are on the same level of the arithmetic hierarchy. A string of mixed multiplications and divisions will be performed from left to right unless parentheses make it otherwise. Multiplication and division take precedence over addition and subtraction. This means that they will be done first.

For example,

```
PRINT 3+2*6
15
```

If addition took precedence over multiplication it would add, then multiply, as in:

```
PRINT (3+2)*6
30
```

## EXPONENTIATION

The last arithmetic operation is exponentiation, the power operation. Recall that "$a$ to the power $b$" means $a$ muliplied by itself $b$ times. On the Apple keyboard the hat symbol (^) means exponentiation. Thus:

```
PRINT 3^2
9
AND
PRINT 2^3
8
```

Exponentiation is at the top of the hierarchy, in that it will be done before multiplication and division. A string of exponentiations will be done from left to right.

## EXERCISES

1. Perform the following arithmetic calculations using a print statement:

   a. 457 + 32 + 94

   b. 34 times 237

c. 64 divided by 16

2. Find the average of 3.208, −56.007, 3412.76 and −.004.

3. How much sales tax must you pay (at 6%) on the total of the three purchases: $23,689; $108,544; $14,320.

# 4. LINE NUMBERS AND STATEMENTS

There are two different ways your Apple can execute the commands that you give it. One of these, as you have seen, is to immediately execute each statement as soon as you press RETURN. You tell the system to do this by not putting a number in front of the statement.

If you put a number in front of the PRINT command, or any other command for that matter, nothing happens when you press return except the line you just entered jumps up one line on your monitor, to make room for the next line. The system is not even looking at your statements, except to check that they have a line number.

Every time you enter a statement with a line number, the machine stores it in order of increasing line number. If you just type the number and press return, it will delete any statement with that number that was stored. If you type a statement number, then a statement, when you press return the new statement will replace any old one with the same line number.

Whenever you want to see what your program looks like so far, type LIST. The program lines will be listed in increasing line number order.

In Applesoft BASIC you can have as many separate BASIC statements on a line as you want. This is a rather unique feature that proves very useful. Statements on the same line are separated by a colon (:). This is especially useful when you want to make a remark (REM) on the same line.

Enter and RUN this program:

```
10 REM          FIRST PROGRAM
20 PRINT  "**********************"
30 PRINT  "* APPLESAUCE IS BOSS *"
40 PRINT  "**********************"
```

When the Applesoft prompt character appears after the RETURN key is pressed in line 40 you type in the three letters R-U-N and press RETURN. You should get

```
RUN
**********************
* APPLESAUCE IS BOSS *
**********************
```

If you don't get this output type LIST and press return. Your four-line program will appear on the screen, and what was on the screen before gets scrolled out of sight one line at a time. Check the LISTing against the above listing for errors.

As the machine runs your program, line by line, in increasing order of line number, it checks the syntax of each statement as it executes it. If something goes wrong, a ?SYNTAX ERROR IN (statement number) will appear. That will tell you the first statement the machine had difficulty with. Correct that statement and RUN your program again.

To correct a program line type its line number and the correct version. When you press RETURN it will replace the old incorrect one. Later we will show you how to use the editor to correct your errors.

## EXERCISES

1. Make up a 4-line program with 3 statements per line. Remember to use a colon (:) between each statement. RUN your program.
2. List your 4 line program from Exercise 1. Find one statement that you can delete and still have your program run. Delete that statement by retyping the line number followed by the revised line. List your program to make sure everything looks OK, then run it.
3. Enter a few PRINT calculations with the line numbers 10, 20, 30, etc. in front of them. Run your program.
4. Write a program that prints your name and address, including ZIP code, after printing 3 blank lines.
5. Write a program that prints "I WILL NOT PLAY WITH COMPUTERS" 100 times. You may have to look ahead in this book, or ask someone who knows BASIC, to solve this problem.

# 5. VARIABLES AND ASSIGNMENT STATEMENTS

To get any real work done by your program you will have to use *variables* in place of actual numbers, because you will want to do the same thing to different sets of numbers or words. A variable is just a *name* for a memory location that holds the data you want your program to work with.

Your Applesoft BASIC recognizes three kinds of variables, corresponding to three different types of data. Except for special occasions, when working with numbers you will be using the *floating point numbers* that carry nine digits precision. These have *variable names* consisting of letters and digits *only* and the first character must be a letter.

Try this program:

```
10 REM    FIRST PROGRAM WITH VARIABLES
20 V=3 : X=5 : W=-9
30 PRINT V+X-W, "GOTCHA"
```

You should get

```
17              "GOTCHA"
```

If you didn't get this output, LIST your program and locate the errors. Those are colons (:) in line 20, because we are putting more than one statement on a line.

You can use variables to store strings of characters, as in:

```
10 REM   STRING VARIABLES
20 A$="GOTCHA"
30 PRINT "***";A$;"***";A$;"***"
```

In line 20 there are a couple of things to notice. For one, notice that the name for a string variable ends in a dollar sign. This is always true. For another, notice that the string itself, GOTCHA, is put between double quotes. Whenever you assign a string of characters to a string variable you should put double quotes around the string.

Only the first two characters of a variable name are used by the machine to uniquely identify the variable. You can use longer names to help yourself remember what they stand for, if you want, but the machine will only see the first two characters. Also your name cannot contain as a subword any of the reserved words used by Applesoft.

It is probably best to use one or two character names whenever possible.

Consider this program:

```
10 REM     ARITHMETIC ASSIGNMENTS
20 X=(5+6)/3 : Y = 2/3
30 A=X+Y
40 A$="THE ANSWER IS "
50 PRINT A$;A
```

Whenever a statement begins with a variable name followed by an equal sign it is called an *assignment*

*statement.* Its function is to take whatever appears on the right side of the equals sign and put it in the variable location specified on the left side. In Applesoft BASIC you can preface the variable on the left with the word LET. For this reason these assignment statements are also called LET statements.

## EXERCISES

1. Assign the values 3.14159, 181156, and .007 to the variables P, C, and E respectively. Then print the product of all three numbers. Put one statement per line, four lines in all. List, then run, your program.
2. Assign the value ZORO to the string Z$. Then print it five times across the line, three spaces between each printing.
3. (For advanced students only.) Make a large Z across the screen using the word ZORO from Exercise 2.

*Hint:* Use about 24 print statements.

# 6. INPUTTING DATA FROM THE KEYBOARD

The programs you have tried so far have not required that you do anything once your program starts running. If there was any special data in it, you had to put it there before you ran it.

It happens a lot that the programmer does not know what numbers are going to be used when the program is run. Therefore some means must be used to enter the proper values while the program is running. BASIC uses the INPUT statement for this purpose.

Just so that you can get used to it, enter and RUN this program:

```
10 REM     FIRST INPUT PROGRAM
20 INPUT "GIVE US A NUMBER ";N
30 PRINT "YOUR NUMBER IS ";N
```

When you run it you will see the line GIVE US A NUMBER printed, and then the cursor will wait in a flashing mode for your number N. If you type in a nonvalid number it will generate a ?REENTER message or sometimes an ?EXTRA IGNORED message. Experiment around with this program using different input errors. See what happens.

You can INPUT string variables, too, as in:

```
10 REM  STRING VARIABLE INPUT
20 INPUT "WHAT IS YOUR FIRST NAME? ";F$
30 INPUT "WHAT IS YOUR LAST NAME? ";L$
40 PRINT "YOUR FULL NAME IS ";F$;" ";L$
```

You can input more than one data line at a time, as in:

```
10 REM   INPUT OF SEVERAL ITEMS
20 INPUT A, A$, B
```

When you RUN this you will see a question mark appear on the far left, and the cursor will be waiting immediately after it. It is waiting for a number to put into A. If you don't give it one (try it) it will respond with a ?REENTER and when you do finally get it right, that number will go into A. The machine will prompt you for the additional input with a double question mark (??) if you have not yet satisfied the INPUT statement.

In order to satisfy an INPUT statement with one press of RETURN, you must separate your input items with commas and include all string variable data inside double quotes. If you type

```
?5,"GOTCHA",6 (RETURN)
```

you can satisfy line 20 above with only one return.

In general it is best to INPUT only one variable at a time and also have some kind of a prompt message.

# EXERCISES

1. Write a program to test someone's knowledge of multiplication. Ask them to do these multiplications: $2 \times 3$, $5 \times 9$, $43 \times 17$, $856 \times 222$. Ask them one at a time and compare the input with the correct value. If they agree print VERY GOOD and go on to the next question. If the input is wrong print NO, THE CORRECT ANSWER IS followed by the correct answer. Then go on to the next question. When all questions have been asked and answered inform the person how many they got right.

*Hints:* Begin a simple program with just the first question. (When you get that much to work you can go on.) Repeat the structure you had for the first question three more times. You might want to use subroutines for the messages to the user, because these are the same for each of the questions. For example, if you have in your program the lines

```
1000 PRINT "VERY GOOD"
1010 RETURN
```

then every time you need to print VERY GOOD all you have to do is use the command

```
130 GOSUB 1000
140 REM   PROGRAM RETURNS TO HERE
```

and the program will jump to line 1000 and begin executing whatever instructions it encounters. When it finds a RETURN statement, it jumps back to the statement immediately following the GOSUB.

2. Ask someone if they have heard of black holes. Take their answer A$ and test it. If it is Y or YES print MY, MY, YOU KNOW SO MUCH and stop. If A$ is N or NO print WELL THEY ARE QUITE FASCINATING and stop. For any other answer print PLEASE REPEAT and ask the question again.

# 7. READING DATA FROM YOUR PROGRAM

There is another way you can furnish your program with data while it is running. In the method outlined in this chapter, you put the data *in the program* using special DATA statements. For example, we might have four names, each with an age and a height in inches. Go ahead and enter this data:

```
10 REM    FIRST PROGRAM WITH DATA STATEMENTS
20 DATA "GEORGE JONES", 23,71
30 DATA "MONIR AFGAJAN", 8,55
40 DATA "QUAN TRAN", 67,63
50 DATA "MARCY WU", 34,67
```

The first thing we want to do is to READ these DATA statements and print the results. So as not to have to write four separate print statements, we are sending the program back to the same READ statement over and over again. When there is no more data the program will stop. Try it:

```
60 READ N$,A,H
70 PRINT N$,A,H
80 GOTO 60
```

You will get the message ?OUT OF DATA ERROR IN 60, and the cursor will wait for your next instruction.

Notice that by using commas between the print elements in line 70 the different elements are aligned with the tab settings on the screen.

You could have put all the data in one long DATA statement or you could have used a separate DATA statement for each item. READ and DATA work as a team, and they follow these simple rules:

1. All the data elements in all the DATA statements in your program are collected together as one string of data elements. They are collected in increasing order of line number.
2. Every time a READ statement anywhere in your program has to read a data element, it goes and reads the first available element that has not already been read before in the same program. The first element read is the first element in the DATA statement with the lowest line number.

Another way to read all four DATA statements without writing four INPUT statements and without getting into a loop which triggers an error message is to use a FOR . . . NEXT loop, which we discuss in the next section. For now, overlay the lines 60, 70, and 80 that you have on your screen with these lines:

```
55 FOR I=1 TO 4
80 NEXT I
```

and RUN the program. This will keep all the existing lines the same except for line 80, which will be replaced

by the new line 80. Line 55 will be added. Go ahead and LIST, then RUN your program. You should get this result:

```
]LIST

10  REM    FIRST PROGRAM WITH DAT
    A STATEMENTS
20  DATA  "GEORGE JONES",23,71
30  DATA  "MONIR AFGAJAN",8,55
40  DATA  "QUAN TRAN",67,63
50  DATA  "MARCY WU",34,67
55  FOR I = 1 TO 4
60  READ N$,A,H
70  PRINT N$,A,H
80  NEXT I

]RUN
GEORGE JONES     23          71
MONIR AFGAJAN    8           55
QUAN TRAN        67          63
MARCY WU         34          67
```

# 8. A COMPUTERIZED SHOPPING LIST

You may have seen on occasion someone in a supermarket with a hand-held calculator, entering prices, tallying up what everything will cost long before they get to the checkstand. Creating such a computerized shopping list can be quite helpful, especially if you can see the name of the item next to its cost. In the program we develop here, you will enter an item name, how many items, the cost per item, and a tax code (0 = no tax, 1 = tax). Use one data statement per item, as in:

```
100 DATA "MILK",3,1.11,0
```

This statement tells the program that we are buying 3 units of milk at $1.11 per unit, and that there is no tax. Your data statements should start with a number not less than 100, so as to keep room for the program. The last data statement should have a null item name and zeros for each of the remaining items. When the program detects such a statement it will print the totals and stop.

What do we want the program to accomplish? For one, as each item is read we would want to see it printed, and the quantity multiplied by the unit cost to give an item cost. If the item is to be taxed, we would want a separate figure indicating the amount of tax for that item (we will use 6% sales tax).

When all items have been read and printed, we would want to see a total for the item costs without tax, a total for the tax items only, and a grand total, the actual amount you will pay.

If you think you can write such a program, now is the time to try to do it, before you see someone else's solution. If not, the program below is one solution.

```
]LIST

10  REM  ***SHOPPING LIST***
20  READ N$,Q,P,T
30  IF N$ = "" THEN  GOTO 110
40  C = Q * P
50  TC = TC + C
60  IF T = 1 THEN TX = C * .06
70  IF T = 0 THEN TX = 0
80  TT = TT + TX
90  PRINT Q,N$,C,TX
100  GOTO 20
110  PRINT "TOTAL COST= ";TC
112  PRINT "TOTAL TAX= ";TT
114  PRINT "PAY THIS AMOUNT: ";TC
      + TT
120  DATA  "BREAD",2,1.56,0
130  DATA  "BEER",2,2.54,1
140  DATA  "SHAMPOO",1,3.66,1
150  DATA  "CAT FOOD",14,.33,0
200  DATA  "MILK",3,1,11,0
210  DATA  "EGGS",1,.75,0
300  DATA  "",0,0,0
```

```
]RUN

2               BREAD                   3.12
0
2               BEER                    5.08
.3048
1               SHAMPOO                 3.66
.2196
14              CAT FOOD
4.62            0
3               MILK                    3.33
0
1               EGGS                    .75
0
TOTAL COST= 20.56
TOTAL TAX= .5244
PAY THIS AMOUNT: 21.0844
```

# 9. ARITHMETIC EXPRESSIONS

Recall from our earlier discussion that there is a hierarchy among the various arithmetic operations that goes like this:

1. exponentiation
2. multiplication and division
3. addition and subtraction
4. left to right

If you plan to do much financial and/or scientific programming you will need to translate some fairly complex equations into Applesoft BASIC. Here is an example of the kind of thing you might encounter:

$$P1 = \frac{\frac{R}{1200} \cdot \left(1 + \frac{R}{1200}\right)^{M} \cdot L}{\left(1 + \frac{R}{1200}\right)^{M} - 1}$$

The first thing to do is to separate the original expression into a numerator N1 and a denominator D1:

$$N1 = \frac{R}{1200} \cdot \left(1 + \frac{R}{1200}\right)^{M} \cdot L$$

$$D1 = \left(1 + \frac{R}{1200}\right)^{M} - 1$$

This way we can tackle each expression separately. The denominator looks easiest, and indeed it is. We get

```
D1=(1+R/1200)^M-1
```

In order to work out this expression, we needed to remember the hierarchy. Pretend you are the machine for a moment; what would you do with the above arithmetic expression? Because exponentiation is at the top you will raise everything in parentheses to the *m*th power and then subtract 1. The expression inside the parentheses involves first dividing *R* by 1200 and then adding the result to 1. Again we make use of the hierarchy: division is done before addition.

The numerator demands a slightly different treatment. Since the factor R/1200 appears twice this would suggest using a separate expression

```
F=R/1200
```

would save programmer time. The required expression then becomes

```
N1=F*(1+F)^M*L
```

The complete program segment required to compute P1 then becomes

```
100 F=R/1200
110 D1=(1+F)^M-1
120 N1=F*(1+F)^M*L
130 P1=N1/D1
```

## EXERCISES

1. Assuming these values, A = 3.28, B = −6.43, C = 1.087, write arithmetic expressions to evaluate the following formulas. Test your expression by running it.

   a. $\dfrac{A + B}{C}$ b. $\dfrac{A^C - B^2}{A - B}$

   c. $\dfrac{A^{(B^2 - 4C)} + 2AB + B^5}{C^2 - 9A + 1}$

2. Rewrite lines 100 to 130 so as to have the machine do less total work. Notice that the same calculation is being done twice.

3. The warp factor, W, of the spaceship *Astrologos*, is the product of the shield energy, E, the spacecraft velocity, V, and the gravity coefficient, G. The gravity coefficient is the cube of *pi* (*pi* = 3.14159) plus the Hamburg constant .06351. Write a single arithmetic expression to compute the warp factor. Test your expression with these values: E = 4.238, V = 115.233 (kilometers per second). You should get a warp of ----------.

# 10. LOGICAL EXPRESSIONS

We have seen in the preceding section that the value of an arithmetic expression is a number, and that the machine obtains this number by plugging in the current values in the expression.

Another type of expression, equally useful and similarly constructed, is the *logical expression.* These expressions are made up of logical variables (or constants) and logical operations between these variables or constants. Every logical expression, including constants and variables, can take on only the values 0 (false) and 1 (true). At a particular time a given expression is either true (value = 1) or false (value = 0).

One of the most often used logical expressions is a comparison between two numbers. There are six types:

| BASIC expression | MATH wording |
|---|---|
| A = B | A is equal to B. |
| A <> B | A is not equal to B. |
| A > B | A is greater than B. |
| A >= B | A is greater than or equal to B. |
| A < B | A is less than B. |
| A <= B | A is less than or equal to B. |

Armed with this knowledge, you can try some things out on your machine. For example, you know that 3 is less than 5. Test the machine; ask it to:

```
PRINT 3<5
```

Practice writing different expressions of this type that you know are true or false. Check yourself with the machine.

The next step is to become familiar with the logical functions AND, OR, and NOT. For example, the statement

```
10 IF 3<2 OR 4>3 THEN PRINT "HI"
```

will print HI because the second term in the expression 4>3 is true. The OR condition requires that at least one of the arguments be true. Thus

```
20 IF 3<2 OR 4>5000 PRINT "HI"
```

will not cause anything to print.

The other logical operation, AND, requires that both arguments be true in order for its output value to be true.

Can you tell what the following expression will evaluate to?

```
PRINT ( 3<4 AND 4<=3 ) OR 3^2 <>8
```

Enter it and press return. See if you were right.

# EXERCISES

1. Write a program that asks the user for values for $x$, $y$, and $z$. Then have your program evaluate the logical expression

```
x^2<=y+1 or x<-47 or x^2+y^2<=z^2
```

   and print TRUE if it is true, FALSE if it is false.

2. Write a logical expression that is true if 3 times $n$ squared is greater than or equal to 57 minus $x$.

3. The user has just input the answer to a yes/no question. Design an IF . . . THEN statement that branches to statement 340 if the inputted string A$ is Y or YES.

4. Write a program to input the interest rate in percent and the principal in dollars for an intended loan. If the interest rate is above 15 or below 10, or if the principal is above 25000, print the message PLEASE REVIEW LOAN PARAMETERS and stop. Otherwise print LOAN LOOKS OK and stop.

# 11. IF . . . THEN

Every computer program needs to make decisions: if such and such is true do this, otherwise do something else. Without this ability a program will either not do anything or else it will get stuck in a repetitive loop.

In BASIC the most common form of a decision statement is the IF . . . THEN statement. An IF . . . THEN statement consists of the word IF followed by a logical expression, followed by the word THEN, followed by one or more statements on that same line. Here is an example:

```
IF 3<5 THEN PRINT "GOTCHA"
```

If the logical expression is true at the time the IF statement is executed, the program will begin executing the statement immediately following the word THEN. If there are several statements on the same line as the IF statement, these will be executed one after the other.

If the logical expression is false, however, the program jumps to the next numbered statement. Thus the output of the program:

```
10 IF 3<>3 THEN PRINT "HIP":PRINT "HIP"
20 PRINT "HURRAY"
```

is simply ''HURRAY.''

One very common use for IF . . . THEN statements is in testing to see if a loop has completed its work. In the following program we use an IF . . . THEN statement to make it stop after printing 10.

```
10 N=N+1
20 PRINT N
30 IF N<10 THEN GOTO 10
40 STOP
```

This program uses the fact that all variables are set equal to zero before the program starts. Thus the first value of N that is printed is N = 1.

The logical expression following the word IF can be as complicated as you can handle, but usually quite simple expressions take care of most tasks.

## EXERCISES

1. Write a short program to ask the user for his or her favorite number. If it is less than 47 print SORRY, IT'S TOO LOW and stop. If the number they input is 47 or more print SORRY, IT'S TOO HIGH. Make sure your program only prints one message.

*Solution:*

```
10 input "whats your favorite number?";f
```

```
20 if f<47 print "sorry, its too low":
stop
30 print "sorry, its too high"
```

2. Ask the user for their street address number. Print EVEN if it is even or ODD if it is odd.

*Hint:* to find out if a certain number, N, is even, divide it by 2 and see if you get a whole number. Try using this logical expression:

```
N/2=INT(N/2)
```

3. Make the machine count from 10 to 300 in increments of 10, that is, have it print 10, 20, 30, 40, and so forth until it gets to 300.

*Solution:*

```
10 N=N+10
20 PRINT N
30 IF N<300 THEN GOTO 10
40 STOP
```

4. Ask the user for two numbers. If the square of the first number is greater than twice the second number print YES and stop. Otherwise print NO and stop.

# 12. FOR . . . NEXT

The task of doing something over and over again is at the very basis of getting work done, whether it be taking a sum, building a protein, or mowing a lawn. In fact the very notion of a *machine* is bound up with this notion of doing something over and over again.

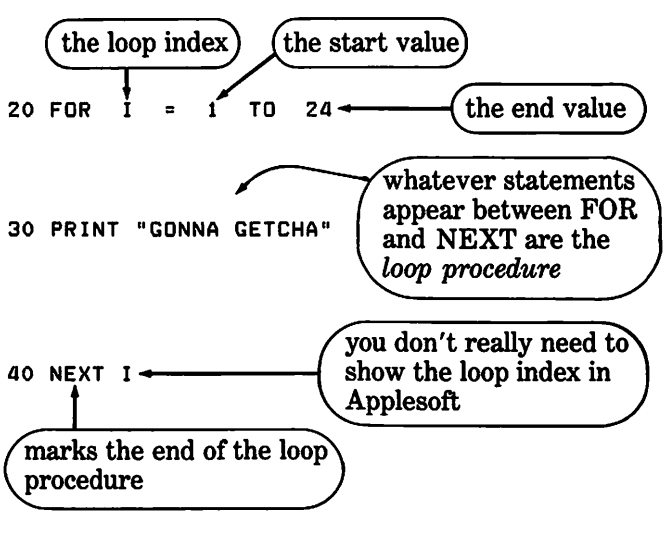
In programming, doing a thing over and over is called the *loop procedure*. In taking a sum of many numbers, the loop procedure usually consists of adding one number to a running total. In building a protein, it could be adding one more amino acid to the growing molecule. In cutting a lawn, it could be one straight path plus a turn.

For our purposes a loop procedure will be a block of BASIC statements that fits between a FOR . . . statement on top and a NEXT statement on the bottom. The entire block, including the FOR and the NEXT, is called the *loop*. Try this one:

```
10 REM    FIRST FOR . . . NEXT LOOP
20 FOR N=1 to 24
30 PRINT "GONNA GETCHA"
40 NEXT N
RUN
```

If you look at line 40 you will see a numeric variable following the word NEXT. In this case it is *N*. Notice that it matches the numeric variable right after FOR in line 20. Thus *N* is the *loop index*. Its value is being controlled and tested throughout the running of the loop procedure.

Let us reprint that last program with some annotation so that you can learn the words that are associated with the use of FOR . . . NEXT loops.




In Applesoft BASIC, the loop index is first set equal to the initial value, and then the loop procedure is immediately executed while that assigned value of the loop index remains constant. At the completion of the loop procedure, during the execution of the NEXT statement, the value of the loop index is tested against the *end value*, which is placed immediately following

the word TO. If the loop index has reached or exceeded the final value, the loop itself has been completed, and program control goes to the statement following the NEXT statement. If, however, the value of the loop index has not reached the final value, then an *increment* is added to the loop index, and program control goes to the statement immediately following the FOR statement, that is, the first statement of the loop procedure. If we do not mention the increment value, as in line 20 above, it is assumed to be one. Otherwise we write STEP following the final value and give a value for the increment. The increment can be positive or negative.

The general form of a FOR . . . NEXT loop is therefore this:

FOR <LOOP INDEX> = <INITIAL VALUE> TO <FINAL VALUE> STEP <increment>

<LOOP PROCEDURE>
NEXT <LOOP INDEX>

The loop index following the NEXT can be deleted in Applesoft BASIC. It will be assumed to belong to the nearest FOR statement towards the top of the program.

## EXERCISES

1. Use a FOR . . . NEXT loop to print the message THE JEDI ARE COMING 1000 times. Use control-S (hold down CTRL key and press S) to stop the output, control-S to resume it. Use RESET to stop the program when you want to move on.
2. Consider the following reggae stanza:

   ```
   HAVE YOU HEARD THE OCEAN ROAR?
        LET IT ROAR (repeat 7 times)
   ```

   Print this stanza five times. Use two FOR . . . NEXT loops, the outer one going from 1 to 5, and the inner one going from 1 to 7. Each choral response LET IT ROAR should be on a separate line.
3. Of the first 1000 integers (that is, the numbers 1, 2, 3, 4, . . .) how many are divisible by 2, 5, or 17?

*Hint:* Inside a FOR . . . NEXT loop that goes from 1 to 1000 you can put a long IF statement that increments a counter by one if it is true. The logical expression following the IF would have three expressions separated by two ORs. The first expression would be true if the number was divisible by 2. An expression like N/2 = INT(N/2) will do the trick.

4. How many of the first 1000 integers are perfect squares, that is, a number whose square root is also an integer?

> *Hint:* Use the square root function, SQR(N) to take the square root of N, and then use the integer function to see if it is a whole number. The answer is ------.

5. Print the multiplication table up to $9 \times 9$, using two FOR . . . NEXT loops each going from 1 to 9, one inside the other. The trick here is to use a PRINT statement which ends in a semicolon, so that the next PRINT statement will resume on the same line where the last one left off. After the inner FOR . . . NEXT loop is complete (when you have just printed the row number times nine) you can issue a PRINT statement with nothing else in it. This will make any PRINT statement following it start on a new line.

# 13. LOAN AMORTIZATION

Just about everybody borrows money from time to time, and there are all kinds of loans. A few concepts, however, apply to almost all loans.

Every loan has an amount that was borrowed. This is called the *principal*, and we shall use the letter B to represent it. Its value is obtained by this statement:

```
20 INPUT "HOW MUCH WILL YOU BORROW?";B
```

The next question, obviously, is at what interest rate:

```
30 INPUT "HOW MUCH ANNUAL INTEREST?";I
```

If we know the annual interest rate I in percent, like 12.6 percent, how do we convert that into a number that we multiply the principal by in order to get the interest? We divide by 100. For example, 57% of something means to multiply the amount of that something by .57. We have compounded the interest daily because most banks do it that way. In line 30, as soon as we know the annual interest rate, we might as well convert it into a daily interest factor:

```
40 D = (I/365)*.01
```

This value of D times the balance due at the beginning of the day gives the interest that must be paid to borrow the money for that one day. If it is not paid it is added to the balance due, and the next day starts with a larger balance due.

The program next needs to determine the repayment schedule. This takes a couple of lines:

```
50 INPUT "HOW MANY PAYMENTS PER YEAR?";N
60 INPUT "HOW MUCH EACH PAYMENT?";P
```

At this point the program has all the information it needs to print the amortization table, which shows how the balance due goes down with time, assuming all the payments are made on time.

We will compound the interest daily between payments:

```
70 FOR I=1 TO 365/N
80 B=B+B*D
90 NEXT
```

Now the payment arrives and we apply it to the balance:

```
100 B=B-P
110 K=K+1
120 PRINT K,B
```

Statement 120 will print one line on our amortization table showing the payment number and balance after payment.

Next we check to see if the balance is still greater than zero. If it is we enter another payment period.

```
130 IF B>0 THEN GOTO 200
140 PRINT "LOAN IS PAID OFF"
```

Enter and run the program. Below is a sample run. Keep working at the program until it does everything it is supposed to.

```
]RUN
HOW MUCH WILL YOU BORROW? 1000
HOW MUCH ANNUAL INTEREST? 15.5
HOW MANY PAYMENTS PER YEAR? 12
HOW MUCH EACH PAYMENT? 145
1              867.818483
2              733.9426
3              598.350632
4              461.020579
5              321.930164
6              181.05682
7              38.3776936
8              -106.130363
LOAN IS PAID OFF
```

## EXERCISES

1. Replace the variable B in line 120 with the expression INT(B*100)/100. This will round B off to the nearest cent. The integer function gives you the largest integer (whole number) not greater than the number inside the parentheses. For example,

   ```
   INT(2.3456*100)/100
   =INT(234.56)/100
   =234/100
   =2.34
   ```

2. Add two more items to the detail report line printed in line 120: the INTEREST portion of the monthly payment and the PRINCIPAL portion of the monthly payment.

3. Add a feature to your program of Exercise 2: have it print a total of all the interest and also the sum of the original amount borrowed plus the total interest.

4. Make the last part of the loan amortization program better for the user. Indicate that they will get a refund, and show the amount (multiply the balance by $-1$).

5. Modify the loan amortization program so that it asks the user for the amount of the payment, P, each time a payment is due. Your new INPUT statement should go between statements 90 and 100. Delete line 60. Include the features in the above exercises.

# 14. MAGIC SUMS

In this section you can practice the use of FOR . . . NEXT loops and review a little mathematics, all just for fun. The problems will all be of the same type, something like:

$$S = 3 + 5 + 7 + \ldots + 33$$

where you have to direct the machine to total up the missing numbers, indicated with + . . . +.

In this case you write the short program:

```
10 FOR N = 3 TO 33 STEP 2
20 S = S + N
30 NEXT N : PRINT "THE SUM IS ";S
```

Go ahead and RUN it. You should get THE SUM IS 288.

Here's another one:

$$S = 10 + 15 + 20 + \ldots + 55$$

The answer is 1495. Don't go on until you get it.

Suppose you see something like this:

$$S = 10^2 + 20^2 + 30^2 + \ldots 100^2$$

The trick to programming this is to make the loop index move from 10 to 20 to 30 and so forth, that is, up by ten each time, and to do the squaring of the loop index before adding it to the sum. Try this solution:

```
10 FOR I = 10 TO 100 STEP 10
20 S = S + I^2 : NEXT I : PRINT "THE SUM IS
";S
```

The program design becomes more complicated when you have magic sums of the form:

$$S = 4^2 + 6^3 + 8^4 + \ldots + 12$$

Notice that the power each term is raised to goes up by one with each new term. The question mark indicates that we don't need to know what the value of the power will be when the last term is reached. Again, we want to make the loop index move from 4 to 6 to 8 and so forth up to 12. Here is one solution:

```
10 FOR I = 4 TO 12 STEP 2
20 S = S + I^(P+2)
30 P=P+1
40 NEXT I : PRINT "THE SUM IS ";S
```

Notice that we are taking advantage of the fact that all variables are reset to zero before your program is run. We thus know that S and P start at zero.

# 15. STRING VARIABLES

In addition to working with numbers, BASIC works with words formed from the different characters on the keyboard. These words are called *strings* to emphasize the fact that they contain characters connected together one after the other, and they need not form an understandable word. You can imagine that each letter is connected only to the next one in the sequence, and that you can pick up the entire string by the first character and swing it around a little.

The name of the area in memory where the string is stored is called a *string variable*. In Applesoft this name must have a dollar sign ($) as its rightmost character, as in

```
50 A$ = "GOTCHA"
```

This command will put the string GOTCHA in the memory area called A$. The double quotes ('') tell where the string starts and stops. They are not part of the string.

The number of characters in a string is called its *length*. The string GOTCHA has length 6. When you manipulate a string it is often necessary to know its length, and in Applesoft BASIC this is accomplished by writing, in this case,

```
60 L = LEN(A$)
```

In general, you put the name of the string inside the parentheses, and the LEN function returns the number of characters in the string.

Run the following program:

```
10  REM   THE NULL STRING
20  N$ = ""
30  PRINT  LEN (N$);N$; LEN ("GOT
     CHA")

]RUN
06
```

This tells you several things about how Applesoft handles the string with no characters, or *null string*. Notice it can be defined by putting *no* characters between the double quotes. The machine automatically sets every string variable equal to the null string before it begins execution of your program.

Most string manipulation requires that you work with substrings of a string. Applesoft has three commands which allow you to pull out the lefthand, middle, or righthand part of a word, as the following program shows:

```
10  REM   PROGRAM TO DEMONSTRATE
     LEFT, MID, & RIGHT
20  A$ = "MISSISSIPPI"
30  L$ = LEFT$ (A$,3)
40  M$ = MID$ (A$,4,4)
50  R$ = RIGHT$ (A$,4)
60  PRINT L$: PRINT M$: PRINT R$
70  PRINT R$;M$;L$
```

```
]RUN
MIS
SISS
IPPI
IPPISISSMIS
```

In line 30, 3 is the number of characters to pull off from the left, and in line 50, 4 is the number of characters to pull off from the right. In 40, however, 4 is the character at which to start the string out of the middle, and 4 is the number of characters it is to contain.

Enter and run this program.

String manipulation using these word separators is required when you are processing names. You may have to strip off first and middle names in order to do an alphabetic sort. See Exercise 3 below.

## EXERCISES

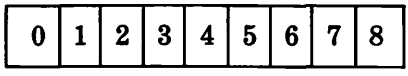
1. Have the user input his or her first, middle, and last names separately. Then print the entire name.
2. Have the user input their entire name, as on a diploma, into a single-string variable. Find out how many characters are in the first name, by counting until you come to a blank character. Print that number. Test your program with several names.
3. Have the user input their entire name, as in Exercise 2 above. Then print their last name. Use the LEN function to make sure you print their last name, in case they entered no middle name.
4. Ask the user for their favorite song. Then print the number of times the letter *a* occurs in their answer.

# 16. ONE-DIMENSIONAL ARRAYS

Sooner or later you will be working with a set of numbers which represent measurements on a single item, such as the temperature outside your house. If you have a lot of measurements you will not want to give each number a separate name that you have to remember, for you would soon saturate your memory as a programmer.

The solution to this problem, and the one that most computer languages have adopted, is to name the individual occurrences *automatically*, by using a two-part name. The first part is called the *array name*, and it is shared by all the values in the array, and the second part is called the *subscript portion*, and it must be a non-negative integer, that is, a whole number.

The individual numbers in a one-dimensional array are usually thought of as being stored in a nice long row of cells—sort of a motel for numbers. We can give the array name to the entire row of cells, the motel name, so to speak, and then simply number the individual cells 0, 1, 2, 3, and so forth.




In all versions of BASIC that we know of, array names and subscripts are written like this:

```
565 X = A(34)
```

This statement would take the floating point number in cell 34 of the array A and write it on top of the number that was in *X*.

Usually the subscript is a variable or an expression, as in

```
45 BAL = RECEIPTS(CUSTOMER)
```

which is valid in Applesoft but not in most versions of BASIC. The machine only looks at the first two characters, however, so you have to make sure the first two characters are unique.

Let's say you have 30 numbers that you want to put into the machine, and they are all of the same type, perhaps a month's worth of daily temperature readings. You might as well call them $T_1$, $T_2$, $T_3$, . . . , $T_{30}$ even before you put them into the machine, say to check their values with a friend. It has become customary in mathematics printing to write the cell number slightly below the single letter that is the array name so that the typesetter needn't set parentheses. It was called the subscript, because it was below the line.

This routine will input your 30 numbers into an array called T:

```
10 REM  ENTER 30 NUMBERS IN ARRAY T
20 DIM T(30)
30 PRINT "ENTER A NUMBER AFTER EACH QUESTION
MARK"
```

```
40 FOR I = 1 TO 30
50 INPUT T(I)
60 NEXT I
70
```

## EXERCISES

1. Ask the user for 12 numbers. Put these into an array N starting in location 1. Make sure to include a dimension statement at the top of your program. When all the numbers have been entered, print the contents of N, one value per line.

2. Have your program find the largest number in the array N of Exercise 1 above. One way to do this is to load $n(1)$ into a temporary variable T and then use a FOR . . . NEXT loop to look at the other locations 2 through 12. If you discover an element larger than the one in T, put the larger value in T. When the loop is done T will contain the largest element in the array N.

3. Continue the program in Exercise 2 above by adding a routine that finds the average of all 12 values.

4. Construct a one-dimensional string array by asking the user for their seven favorite movie stars. Then write program segments to accomplish the following tasks:

   a. Print all 7 names.
   b. Print the longest name.
   c. Print all names that start with 'a.'
   d. (advanced students only) How many times does the letter 'E' occur among all the names?

# 17. ACCOUNTS RECEIVABLE

If you are in business and take in money from several sources, those sources are called your *accounts receivable*. Each one of them has a balance due you at any particular time, and you must keep track of this balance by processing the various bills and payments for that one account.

If you wish to use your Apple computer to calculate your accounts, when a *payment* arrives you must enter its account number, date, and amount. The program will update the balance for that account by subtracting the amount of the payment. We have *credited* their account.

When an *invoice* is processed you will enter its account number, date, type, and amount. The program will add the amount of the invoice to the balance. We have *debited* their account.

In this system you can have up to 10 accounts and 100 transactions. The program starts like this:

```
10 REM   ***ACCOUNTS RECEIVABLE***
20 DIM B(10) :REM  ACCOUNT BALANCE
30 DIM T(100) :REM  TRANSACTION TYPE
40 DIM D(100) :REM  TRANSACTION DATE
50 DIM A(100) :REM  TRANSACTION AMOUNT
```

Notice that we are keeping a record of each transaction, using three one-dimensional arrays.

For testing purposes we will include some initial data:

```
60 DATA 245,873,54,4442,1733
70 DATA -76,0,531,1208,320
80 FOR I=1 TO 10:READ B(I):NEXT
```

So at this point the 10 account balances have been initialized with the above values. We are ready to process transactions.

```
90 K=1 :REM  K IS TRANSACTION NUMBER
100 INPUT "TRANSACTION TYPE? ";T(K)
110 IF T(K)=0 THEN GOTO 300
120 INPUT "ACCOUNT (1 TO 10):";C(K)
130 INPUT "AMOUNT ";A(K)
```

At this point the transaction has been stored. Next we update the balance:

```
140 IF T(K) = 1 THEN B(C)=B(C)+A(K)
150 IF T(K) = 2 THEN B(C)=B(C)-A(K)
```

and then we go on to the next transaction

```
160 K=K+1 : GOTO 100
300 M = K - 1
```

When all transactions have been entered and processed we end up at statement 300 above. The value of M is the number of transactions. It is K − 1 because the *k*-th transaction is not a real one. Our report will consist of a list of transactions for each account. See if you can understand the code.

```
310 FOR I=1 TO 10
315 PRINT "TRANSACTIONS FOR ACCOUNT ";I
320 FOR K=1 TO M
330 IF C(K)=I THEN PRINT C(K),A(K)
340 NEXT
350 NEXT

]LIST

10  REM   ***ACCOUNTS RECEIVABLE*
     **
20  DIM B(10): REM   ACCOUNT BALANCE
30  DIM T(100): REM   TRANSACTION TYPE
40  DIM C(100): REM   TRANSACTION ACCOUNT
50  DIM A(100): REM   TRANSACTION AMOUNT
60  DATA  245,873,54,4442,1733
70  DATA  -76,0,531,1208,320
80  FOR I = 1 TO 10: READ B(I): NEXT
90  K = 1: REM   K IS TRANSACTION NUMBER
100  INPUT "TRANSACTION TYPE:";T(K)
110  IF T(K) = 0 THEN  GOTO 300
120  INPUT "ACCOUNT (1 TO 10):";C(K)
130  INPUT "AMOUNT:";A(K)
140  IF T(K) = 1 THEN B(C) = B(C) + A(K)
150  IF T(K) = 2 THEN B(C) = B(C) - A(K)
160  K = K + 1: GOTO 100
300  M = K - 1
310  FOR I = 1 TO 10
315  PRINT "TRANSACTIONS FOR ACCOUNT ";I
320  FOR K = 1 TO M
330  IF C(K) = I THEN  PRINT C(K),A(K)
340  NEXT
350  NEXT

]RUN
TRANSACTION TYPE:1
ACCOUNT (1 TO 10):1
AMOUNT:25
TRANSACTION TYPE:1
ACCOUNT (1 TO 10):1
AMOUNT:456
TRANSACTION TYPE:2
ACCOUNT (1 TO 10):6
AMOUNT:322
TRANSACTION TYPE:2
ACCOUNT (1 TO 10):6
AMOUNT:763
TRANSACTION TYPE:1
ACCOUNT (1 TO 10):3
AMOUNT:456
TRANSACTION TYPE:1
ACCOUNT (1 TO 10):7
AMOUNT:6789
TRANSACTION TYPE:2
ACCOUNT (1 TO 10):5
AMOUNT:653
TRANSACTION TYPE:2
ACCOUNT (1 TO 10):7
AMOUNT:23
TRANSACTION TYPE:0
TRANSACTIONS FOR ACCOUNT 1
1             25
1             456
TRANSACTIONS FOR ACCOUNT 2
TRANSACTIONS FOR ACCOUNT 3
3             456
TRANSACTIONS FOR ACCOUNT 4
TRANSACTIONS FOR ACCOUNT 5
5             653
TRANSACTIONS FOR ACCOUNT 6
6             322
6             763
TRANSACTIONS FOR ACCOUNT 7
7             6789
7             23
TRANSACTIONS FOR ACCOUNT 8
TRANSACTIONS FOR ACCOUNT 9
TRANSACTIONS FOR ACCOUNT 10
```

## EXERCISES

1. Modify the program above so that it prints out the current balance for each account before it lists the transactions for that account. You could do this by changing one line, number 315.
2. Modify the detail line of the above report (line 330) so that it prints the transaction type T(K) instead of the account C(K).
3. Add a DATE feature to your developing accounts receivable program. Add an extra array and an extra input statement. Add the date on the detail line. Use the format MMDD so that you can later sort on the date. You will probably want to use the TAB (N) command to tab you to column N, especially if you have a printer. One approach is to put only one data element per print line, with a TAB in front of it and a semicolon after it. The last print item for that line will not have a semicolon. If you do this you can modify the output with minimum effort. Once your detail line gets more than a few items long you will need a header line above the columns of data. This can be done either in one PRINT statement or in several.

# 18. BUBBLE SORTING

Now that you have some experience with one-dimensional arrays, it is time you learned a standard sorting technique, the so-called BUBBLE SORT. There are several variations on the same basic idea, which is to examine pairs of array values, and if a particular pair is out of order, you switch the values, then go on to examine another pair. When you have examined and switched enough pairs, your array will be sorted.

First let's get the numbers into an array called A:

```
10 REM  ***BUBBLESORT***
20 DIM A(20)
30 INPUT "HOW MANY NUMBERS?";N
40 IF N>20 THEN PRINT "TOO BIG":GOTO 30
50 FOR I=1 TO N
60 INPUT "NUMBER PLEASE:";A(I)
70 NEXT
```

So at this point N contains the number of values to be sorted, and the unsorted values are in A(1), A(2), . . . A(N).

The method we will use finds the largest value in the array and puts it in the first position. This is accomplished by comparing the first position with each of the other positions in turn (2, . . . , *N*). Whenever the first value is smaller than the value it is being compared with, interchange the two values. By the time you have gotten to the end of the array, position one will contain the highest value.

Next you compare the second position with each of the positions below it, and make any necessary interchanges.

You keep going in this manner until you are starting your compare and interchange with position N – 1, that is, the next to last position. Here there is only one comparison to make.

Check the code to make sure it does what we say it will:

```
80 FOR P=1 TO N-1
90 FOR I=P+1 TO N
100 IF A(P)>=A(I) THEN GOTO 120
110 T=A(I):A(I)=A(P):A(P)=T
120 NEXT
130 NEXT
140 FOR I=1 TO N:PRINT A(I):NEXT
```

Go ahead and enter and run this program. Keep at it until it works.

## EXERCISES

1. Write line 110 a different way, but have it accomplish the same thing.
2. Compress lines 80 to 140 above into the least number of numbered statements. Verify that your solution works.

3. Modify the bubble sort program so that the smallest number ends up on top and the largest number ends up on the bottom.

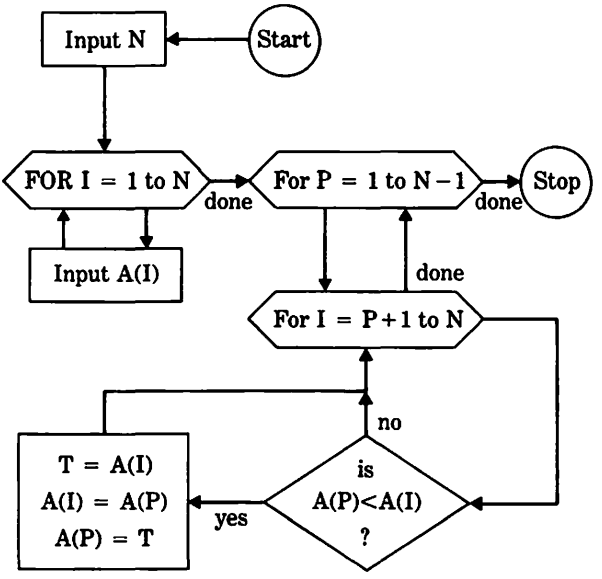
4. A slightly different bubble sort makes N − 1 passes through the array A, except it looks only at adjacent pairs, starting at the top. It put the smallest number at the bottom (into A(N)) after the first pass. On the second pass there is no need to go beyond A(N − 1). Finally on the N − 1 pass you need only compare A(1) and A(2). Rewrite the sort program so that it uses this method.

```
]LIST

10  REM   ***BUBBLE SORT***
20  DIM A(20)
30  INPUT "HOW MANY NUMBERS? ";N
40  IF N > 20 THEN  PRINT "TOO BI
    G": GOTO 30
50  FOR I = 1 TO N
60  INPUT "NUMBER PLEASE: ";A(I)
70  NEXT
80  FOR P = 1 TO N - 1
90  FOR I = P + 1 TO N
100  IF A(P) > = A(I) THEN  GOTO
     120
110  T = A(I):A(I) = A(P):A(P) = T
120  NEXT
130  NEXT
140  FOR I = 1 TO N: PRINT A(I): NEXT

]RUN
HOW MANY NUMBERS? 6
NUMBER PLEASE: 12.678
NUMBER PLEASE: -87.453
NUMBER PLEASE: 104.39
NUMBER PLEASE: -.00076
NUMBER PLEASE: -.0076
NUMBER PLEASE: 56.433
104.39
56.433
12.678
-7.6E-04
-7.6E-03
-87.453
```

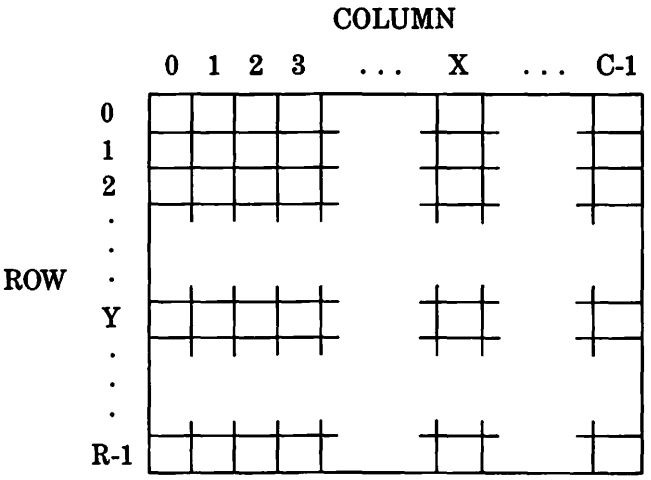
## FLOWCHART FOR BUBBLE SORT



# 19. TWO-DIMENSIONAL ARRAYS

We have seen that one-dimensional arrays naturally arise when we have a series of measurements on one item. We can think of a one-dimensional array as either a row of cells, or as a column of cells.

A two-dimensional array can be thought of as a rectangular array of cells, with R rows and C columns:




The row subscript, Y, goes from zero to R-1, and the column subscript goes from zero to C-1. To put a value of 8.7 into a certain cell, say row Y, column X, you would write

```
730 CE(X,Y) = 8.7
```

where CE is the name of the two-dimensional array.

As an example, we can create an array MULT with 10 rows and 10 columns, where the value in row Y, column X is the product X times Y.

```
10 REM   MULTIPLICATION TABLE
20 DIM  MULT(9,9)
30 FOR Y = 0 TO 9 :  FOR X = 0 TO 9
40 MULT(X,Y) = X*Y
50 NEXT : NEXT
```

This block of code will generate the array. Notice that the dimension statement uses one less than the number of rows or columns, because the numbering starts with zero. Notice also in line 50 that you don't have to specify the loop index immediately after NEXT.

In order to print out the above two-dimensional array we continue the code:

```
60 FOR Y = 0 TO 9
70 FOR X = 0 TO 9 : PRINT MULT(X,Y);"  "; :
   NEXT
80 PRINT : NEXT
```

Go ahead and enter this program and RUN it. You can achieve almost the same thing with one simple command, MAT PRINT MULT, but you will not get good screen alignment.

A good example where a two-dimensional array is needed is in frequency distribution analysis. Suppose, for example, that you give a test with 8 questions on it, each with a multiple choice answer 1 to 5. Let's say 100 students have taken the test, and their answers have been coded into a block of 100 data statements, each DATA statement containing 8 answers by an individual student.

If you want to discover what kind of test it was you will almost certainly want to know how many students answered each question in each of the possible ways. For example, how many students answered question number 7 with a 3? What was the most popular answer to question number 2?

What you want here is a two-dimensional array H (for how many) with say 5 rows (one row for each answer) and eight columns. When all the DATA statements have been read, H(X,Y) should contain the number of students who answered question X with answer Y.

How do we build H? The idea is to read one DATA statement at a time, see how that student answered the first question. If it was with a 3, say, the cell corresponding to question 1 answer 3 should be incremented by 1, or the same,

```
H(1,3) = H(1,3) + 1
```

Whatever the answer to the first question, we go to the appropriate cell and increment it by one.

We go through the remaining seven questions in a similar manner. When all DATA statements have been read, H should contain the required count.

See if you can write a program that calculates and prints H. Use the dimension H(8,5) and waste the extra row and column of the zero subscripts. Compare your program with the following solution.

```
 10 REM  FREQUENCY DISTRIBUTION OF TEST
    ANSWERS
 20 DIM H(8,5)
 30 FOR S = 1 TO 100   GOSUB 100 : NEXT
 40 GOSUB 200  REM  PRINT THE ARRAY H
100 REM  PROCESS ONE STUDENT
110 FOR C = 1 TO 8 :  READ A
120 H(C,A) = H(C,A) + 1   :  NEXT : RETURN
200 REM  PRINTOUT ROUTINE
210 FOR R = 1 TO 5 : FOR C = 1 TO 8
220 PRINT H(C,R); "  "; : NEXT : PRINT : NEXT

]PRINT CHR$(9) "80N"

]LIST

10  REM     TWO DIMENSIONAL ARRAY PROBLEM
20  DIM H(6,5)A(10,6)
30  FOR I = 1 TO 10: REM      FOR EACH OF TEN
    STUDENTS
40  REM                GET THEIR 6 EXAM SCORES
60  READ A(I,1),A(I,2),A(I,3),A(I,4),A(I,5),
    A(I,6)
70  REM                PROCESS THAT STUDENT
80  FOR Q = 1 TO 6:H(Q,A(I,Q)) = H(Q,A(I,Q))
    + 1: NEXT Q
90  NEXT I: REM    THAT STUDENT IS DONE
100  REM    NOW PRINT THE FREQUENCY
     DISTRIBUTION ARRAY H
110  FOR A = 1 TO 5: FOR Q = 1 TO 6: PRINT
     H(Q,A);"  ";: NEXT Q
```

```
120  PRINT : NEXT A: PRINT "THATS IT"
125  REM
130  DATA  1,2,3,2,5,5
140  DATA  2,2,3,2,5,3
150  DATA  1,2,4,5,1,3
160  DATA  1,4,3,5,5,3
170  DATA  2,2,3,5,1,4
180  DATA  5,2,1,5,4,4
190  DATA  1,2,3,5,4,3
200  DATA  1,2,4,2,2,2
210  DATA  2,1,3,5,1,3
220  DATA  2,2,2,2,2,2

]RUN
5  1  1  0  3  0
4  8  1  4  2  2
0  0  6  0  0  5
0  1  2  0  2  2
1  0  0  6  3  1
THATS IT
```

# 20. LOW-RESOLUTION COLOR GRAPHICS

Now that you know about two-dimensional arrays, using Applesoft color graphics will be easy. The screen that you see in low-resolution mode has 40 columns (0 to 39) and 40 rows (0 to 39). The "value" in cell (X,Y) can be one of 16 colors (shown later). It is called a *pixel*, short for *picture element*.

Just to get started, run this program with a color TV monitor. You should see all 16 colors (black should be no visible phosphor activity) as vertical stripes each two pixels wide.

```
10 REM   COLOR TV BAR GENERATOR
20 GR : REM  SET LOW RESOLUTION GRAPHIC MODE
30 FOR C = 0 TO 15 : REM   FOR EACH COLOR
40 COLOR = C     REM : SET THAT COLOR
50 FOR Y = 0 TO 39 : REM   PAINT THE STRIPE
60 PLOT 2*C,Y  :  PLOT 2*C+1,Y  : PLOT PAIR
   PIXELS
70 NEXT :  REM  FINISH STRIPE
80 NEXT :  REM  FINISHED ALL 16 COLORS
90 PRINT "ADJUST YOUR TV SET USING NEXT PAGE"
```

Below are listed some of the available commands.

## LOW-RESOLUTION GRAPHICS COMMANDS

| | |
|---|---|
| COLOR = *x* | Here *x* is a number from 0 to 15 as follows:<br>0 black<br>1 magenta<br>2 dark blue<br>3 purple<br>4 dark green<br>5 grey<br>6 medium blue<br>7 light blue<br>8 brown<br>9 orange<br>10 grey<br>11 pink<br>12 green<br>13 yellow<br>14 aqua<br>15 white<br>The selected color will be used for plotting until the color is changed by another COLOR command. |
| GR | Sets low-resolution graphics mode, clears screen, and moves cursor to move into |

| | |
|---|---|
| | text window. The color is automatically set to black (0). |
| HTAB $x$ | Moves cursor to column $x$, $x = 0,1,2, \ldots, 39$. |
| PLOT $x,y$ | Creates a pixel at $x,y$ of whatever color is currently set. |
| VLIN $y_1$, $y_2$ AT X<br><br>HLIN $x_1$, $x_2$ AT $y$<br><br>VTAB $y$ | Draws a vertical line from (X, $Y_1$) to (X, $Y_2$) |

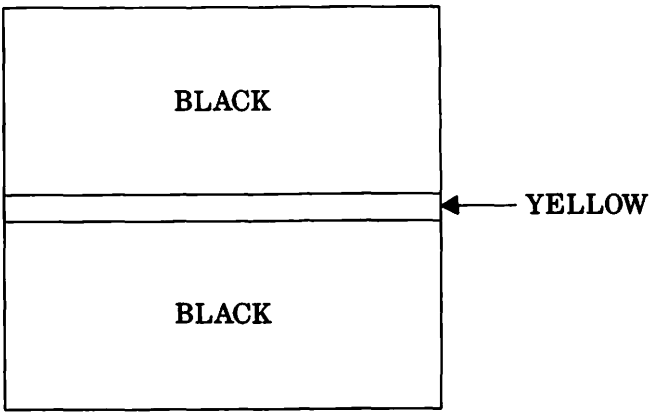
## HOW TO MAKE HORIZONTAL LINES

This particular task is done automatically by Applesoft, but it is instructive to see how you would have to do it on your own if the features were not available. The numbers you need are the leftmost horizontal coordinate, the rightmost horizontal coordinate, and the vertical coordinate. Let us assume these variables are named HL, HR and V, respectively. We should mention that Applesoft permits two characters worth of recognized variable name, and that second character can be very useful as a mnemonic.

Let us say we are in Applesoft low-resolution graphics (40 by 40) and we wish to draw a yellow line all the way across the screen, in its vertical center. The code that does this is easy to follow:

```
10 REM    HORIZONAL YELLOW LINE
20 GR
30 COLOR = 13
40 FOR H = 0 TO 39 : PLOT H,9  :  NEXT
50 PRINT "THERE YOU HAVE IT"
```

Go ahead and run this program. You should get this output:




THERE YOU HAVE IT

# 21. HIGH-RESOLUTION COLOR GRAPHICS

You have seen what you can do with low-resolution graphics. Although you have a wide color selection, there is not enough resolution to form detailed pictures. For this purpose you will want to use *high resolution graphics*, which features a screen 280 pixels wide by 160 pixels long. Each pixel can be 1 of 8 different colors (2 blacks, 2 whites, and 4 other colors).

Here are the commands you can use.

## HIGH-RESOLUTION GRAPHICS COMMANDS

| | |
|---|---|
| HGR | Sets high-resolution graphics page 1; clears top 280 × 160 area to black; bottom 4 lines text. |
| HGR2 | Sets high-resolution graphics page 2; clears entire 280 × 192 screen to black. |
| HCOLOR = *x* | Sets color (0 to 7) for next plotting. |
| HPLOT *x*,*y* | Places colored dot at horizontal coordinate *x* and vertical coordinate *y*. 0,0 is top left corner. |
| HPLOT *x*1,*y*1 TO *x*2,*y*2 | Draws a line from the point *x*1,*y*1 to the point *x*2,*y*2 in the selected color. |
| HPLOT *x*,*y* | Places colored dot at *x*,*y*. |
| HPLOT TO *x*,*y* | Draws a line from last point plotted to *x*,*y* in current color. |

As an example of a high-resolution program, we can show you how to make random lines on high-resolution screen 2.

The idea is quite simple and, indeed, so is the program. First we plot one point at the origin:

```
10 REM   ***RANDOM LINES***
20 HGR2:HCOLOR=7
30 HPLOT 0,0
```

You can go ahead and run this much if you want. You should get a small white dot at the upper left of your screen.

Next we get a random horizontal coordinate *x* and a random vertical coordinate *y* via the statements:

```
30 X=INT(279*RND(1))
40 Y=INT(189*RND(1))
```

The RND(1) argument will produce a random number between 0 and 1, and the INT function chops off any decimals.

Now all that remains to do is to plot a line from the last point plotted to the coordinates *x,y:*

```
50 HPLOT TO X,Y
60 GOTO 30
```

Go ahead and run this program. It is fun to watch. In this case the machine is doing quite a bit of work for you with very little instruction on your part. That is basically what you want from a good machine.

## EXERCISES

1. Modify the random lines program so that it plots the lines in random colors.

> *Hint:* Use HCOLOR = INT(7*RND(1)) at the right place.

2. Use this same technique to draw random squares on high resolution screen 2. Each square should be 50 pixels on a side. Don't worry if a square gets displayed partly off screen; it will automatically wrap around to the other side.
3. Draw squares of different colors and widths, one inside the other, getting smaller and smaller towards the center.

> *Hint:* Find someone that knows BASIC better than you and work on this project with them.

# 22. SYSTEM AND EDITOR COMMANDS

Now that you have tried a few short programs, and have made numerous typing and logical errors, it is time to take advantage of various system features that make life easier. These commands are divided into two groups: *system commands*, which operate on completed program text, and *editing commands*, which operate on individual characters within a developing program text.

Here is a partial list of available system commands that should get you going. If you want to see what is actually available, consult one of the references at the back of this Handy Guide.

## SOME SYSTEM COMMANDS

| | |
|---|---|
| RUN | Executes program starting at lowest line number. |
| RUN *x* | Executes program starting at line number *x*. |
| SAVE | Saves a program on cassette tape. Start the recorder, then type SAVE. Bleeps from the Apple speaker mark the start and stop of the recording. |
| LOAD | Loads a saved program from cassette tape into memory. Start the tape on a silent stretch before the start bleep, and then type LOAD. You will hear a bleep when the LOAD routine encounters the start of the program, and another bleep at the end. |
| NEW | Deletes current program. |
| LIST | Lists current program. |
| LIST *x-y* | Lists from line *x* to line *y*. |

# 23. DISK OPERATION

If the Apple // you are working with has a disk drive you are lucky indeed. Without a disk drive there is no convenient way to transfer work in and out of the machine.

You will notice a little door in the center of your disk drive. It pulls from the bottom and swings up. Go ahead and do it.

If there is a diskette in the drive you can remove it by holding it between your thumb and forefinger and pulling straight back. Find the jacket that protects the diskette and put the diskette in it. You can practice putting the diskette in the drive and closing the door. Always close the door before the system uses the disk drive.

You will need a SYSTEM MASTER diskette. We recommend DOS 3.3 for Apple // users, but DOS 3.2 will work fine.

Turn the power off, insert the system master into the disk drive, and turn the power on. This will cause the system master to load, or *boot*.

When you get the Applesoft prompt back again you can try out some of the DOS commands.

First type CATALOG and press return. The in-use light on the disk drive should come on, and the drive will make certain characteristic sounds. A list of files on the diskette will appear on the screen. A sample is shown below:

```
]CATALOG
DISK VOLUME 254

*A   006   HELLO
*I   018   ANIMALS
*T   003   APPLE PROMS
*I   006   APPLESOFT
*I   026   APPLEVISION
*I   017   BIORHYTHM
*B   010   BOOT13
*A   006   BRIAN'S THEME
*B   003   CHAIN
*I   009   COLOR DEMO
*A   009   COLOR DEMOSOFT
*I   009   COPY
*B   003   COPY.OBJ0
*A   009   COPYA
*A   010   EXEC DEMO
*B   020   FID
*B   050   FPBASIC
*B   050   INTBASIC
*A   028   LITTLE BRICK OUT
*A   003   MAKE TEXT
*B   009   MASTER CREATE
*B   027   MUFFIN
*A   051   PHONE LIST
*A   010   RANDOM
*A   013   RENUMBER
*A   039   RENUMBER INSTRUCTIONS
*A   003   RETRIEVE TEXT
```

The letters A, I, T, and B tell what type of file it is:

A = Applesoft BASIC Program
I = Integer BASIC Program
B = Binary object module
T = Text file

The three digit number to the right of the letter tells how much space, in *sectors*, the file takes up. One sector is one-sixteenth of a full circle on any one of the 35 concentric tracks around the read-write surface of the diskette. Such a sector holds 256 bytes, or characters.

If there is an Applesoft or Integer BASIC program (type A or I) on a diskette in the drive, then type LOAD followed by the filename. This will make the drive spin, and shortly a copy of the program will be available in main memory. You can LIST it as soon as the prompt appears.

As a shortcut, you can enter RUN followed by the filename, and the program will be loaded and run.

If you have a program that you want to save on diskette, you enter SAVE followed by the filename and press return. When the prompt reappears your file is saved. If you use the same name as a file that is already on the diskette, the program in memory will be put on the diskette under that name and the old program will be scratched, that is, erased.

You cannot save programs on a system master diskette because it is permanently write-protected. If you will pull a diskette out of the drive about two inches you will notice a small rectangular cut out on those diskettes that can be written to. No cut out, no write. On permanently write-protected diskettes the jacket has no such cut out.

So if you plan to save programs you will need a diskette with the cutout exposed. In addition the diskette needs to be *initialized*. If it responds successfully to a CATALOG command, then it has already been initialized.

To initialize a blank diskette, first boot DOS, then insert the blank diskette in the drive. Create a short greeting program, call it HELLO, that prints a few lines of identifying information. Then enter INIT HELLO, press return, and wait for the disk drive to stop spinning. It could take a minute or so.

Test that your diskette has been initialized by asking for a catalog.

| | |
|---|---|
| SAVE *x* | Save the program called *x* on the diskette. |
| LOAD *x* | Load the program called *x* on the diskette. |
| CATALOG | List the contents of the cassette. You will get a listing of each named file on the diskette, with each line reporting, in order:<br>• write protection (* means protected)<br>• type of file: A = Applesoft source; I = Integer BASIC source; B = binary object; T = text or data.<br>• number of sectors<br>• name of file<br>Always ask for a catalog when you mount a new diskette, to make sure everything is OK. |

# 24. SEQUENTIAL FILES

After you have used your Apple a while you will feel the need for some sort of way to store records of various kinds: phone numbers of friends, shopping expenses, business accounts, titles of hi-fi cassettes you own, income tax deductions, and on and on. You will want to store this data in a form that remains when you turn your Apple off.

There are only two convenient ways to store information outside the machine: cassette tape and floppy diskette. You can use either system with Applesoft, but the diskette system is to be preferred, for several reasons. For those of us who have done a lot of programming on microcomputers, it is impossible to imagine getting much work done without a disk drive available. On the other hand, if you have a long program on which you have worked many hours, and you don't want to lose it, and you don't have a disk drive, then any standard hi-fi cassette player and hi-fi tape will allow you to store and reload your program or data.

In this and subsequent sections we will assume that you have at least one disk drive connected to your Apple //. We will further assume that you have loaded the DOS System Master, version 3.2 or 3.3, or its equivalent.

Putting data into text (T) disk files on a diskette requires some forethought and attention to detail, primarily because access to the diskette for data storage purposes is via a rather roundabout way: you have to pretend that the diskette drive is some sort of a printer.

Let us begin with the task of storing one simple utterance on our diskette, the message GO RAIDERS. We will attempt to write it to a diskette, turn off the machine, remove the diskette, put the diskette back in, power up the machine, and recover the message we had stored.

Begin by entering NEW followed by:

```
10 REM   FIRST DISK DATA STORAGE PROGRAM
20 D$=CHR$(4)
```

This seemingly innocent program puts a *control D* character into the string variable D$. Thereafter, when we need a control D character somewhere in our program we can just use the string variable name D$.

In Applesoft BASIC we command the diskette file system using special PRINT statements, those of the form

```
PRINT D$;"message"
```

where message is a string of characters. You will see what kind of messages apply.

The first thing you must do is to OPEN the file. This involves giving it a filename and commanding the machine to OPEN it:

```
30 PRINT D$;"OPEN SESAME"
```

Among other things this command reserves space to send data back and forth from main memory to the diskette.

Next you will need to tell the machine to write to the diskette, rather than to the printer or some other device. This is accomplished by another PRINT statement,

```
40 PRINT D$;"WRITE SESAME"
```

Now then it will be a simple matter to put our message onto the diskette. We add the statement:

```
50 PRINT "GO RAIDERS"
```

and we shut down the diskette-write operation by issuing

```
60 PRINT D$
```

without any punctuation following the D$.

Finally we want to CLOSE our file, to make sure that everything will be OK when we go to read it:

```
70 PRINT D$;"CLOSE SESAME"
```

Go ahead and enter and RUN the program so far. The disk drive will light up and whir for a while and then the cursor will reappear. Take the diskette out. You will have created one text file, with one record in it containing the message "GO RAIDERS." Next we will see if we can get it back.

Type NEW to erase the program, and LIST to confirm that it has been erased. Press RETURN until everything is out of sight. You can even turn the power off and reboot DOS.

Enter this program:

```
10  REM   PROGRAM TO RECOVER GO R
      AIDERS
20  D$ =  CHR$  (4)
30  PRINT D$;"OPEN SESAME"
40  PRINT D$;"READ SESAME"
50  INPUT M$: PRINT M$
60  PRINT D$
70  PRINT D$;"CLOSE SESAME"

]RUN
GO RAIDERS

7  CLOS
```

If you managed to get back GO RAIDERS and another prompt cursor without any error messages you should congratulate yourself.

To summarize: all disk controls (for text or data) are issued using PRINT D$;"message" where D$ is the control D character, CHR$(4). Each file you work with must be given a filename, OPENED before using and CLOSED before exiting your program. When you write to the disk you use WRITE in a print statement. Thereafter all items appearing in PRINT statements without the D$ are sent serially to the disk file, exactly as they would be if they were sent to the printer, except that no tab function takes place with the disk file. The commas in a PRINT statement are treated like they were semicolons, as far as the disk file is concerned.

A PRINT (data items) statement with no comma or semicolon at the end will cause a RETURN char-

acter to be placed after the last print item. This RETURN is used to recover the printed field, as it signals to the INPUT statement that all items have been input. If you are printing more than one data item per print field you will want to separate them by commas, which you must insert as alphanumeric characters.

As an example of a more complex sequential file program we show a program to take the frequency distribution data on ten students and put it out to the disk.

```
LOAD WRITESTUDENT
]LIST

10  REM    DISK STORAGE OF STUDEN
     T DATA
20 D$ =  CHR$ (4)
30  PRINT D$;"OPEN SCORES"
40  PRINT D$;"WRITE SCORES"
50  FOR I = 1 TO 10
60  FOR Q = 1 TO 6
70  READ S: PRINT S: NEXT
90  NEXT I: PRINT D$
100  PRINT D$;"CLOSE SCORES"
200  DATA  1,2,3,2,5,5
210  DATA  2,2,3,2,5,3
220  DATA  1,2,4,5,1,3
230  DATA  1,4,3,5,5,3
240  DATA  2,2,3,5,1,4
250  DATA  5,2,1,5,4,4
260  DATA  1,2,3,5,4,3
270  DATA  1,2,4,2,2,2
280  DATA  2,1,3,5,1,3
290  DATA  2,2,2,2,2,2

]RUN

]LOAD READSTUDENT
]LIST

10  REM    DISK READ OF STUDENT D
     ATA
20 D$ =  CHR$ (4)
30  PRINT D$;"OPEN SCORES"
40  PRINT D$;"READ SCORES"
50  FOR I = 1 TO 10
60  INPUT S1,S2,S3,S4,S5,S6
70  PRINT S1;" ";S2;" ";S3;" ";S4
     ;" ";S5;" ";S6;" "
80  NEXT I

]RUN
1  2  3  2  5  5
2  2  3  2  5  3
1  2  4  5  1  3
1  4  3  5  5  3
2  2  3  5  1  4
5  2  1  5  4  4
1  2  3  5  4  3
1  2  4  2  2  2
2  1  3  5  1  3
2  2  2  2  2  2
```

# 25. WHERE TO GO FROM HERE

There are many features of Applesoft BASIC that we have not considered. The references listed in the Bibliography will introduce you to this additional material. In particular, the *Applesoft II Basic Programming Reference Manual* is a must if you plan to do extensive work with the Apple //.

A good text editor and a printer can be wonderful things to have as they fulfill an endless variety of purposes. The text for this Handy Guide was developed using a text editor called **Apple Writer** and the ever popular **Epson MX-80 printer.** It is much easier to write programs using an editor, because you can go back and change characters anywhere you want.

Several spread sheet type programs such as **VisiCalc** are available for the Apple. These are very powerful and easy to use tools for fiscal planning of any kind. You could even develop your own spreadsheet software, but it takes a lot of work to make such programs run smoothly.

All kinds of games are available for the Apple. Since there are thousands of games, you will have to sort your own way through them. There is a pirated version of **Pacman,** called **Taxman,** that is quite good, especially for beginners using only the keyboard.

If you want to do your own game programming then you will have to enter the elite group of Assembly Language Programmers. There is much you can do with Applesoft BASIC, but speed is very important for games, and assembly language is by far the fastest way to get things done. You can get a **Graphics Tablet** for your Apple //, and this is a good way to get graphic information quantized. Several routines let you make drawings with minimal effort and store them for later recall.

There are also interface boards that let you hook up a television camera to your Apple. Such systems are however quite demanding to design and program because of the high speed and large memory required.

You can get various synthesizer boards for your Apple which allow you to compose music. You can even program the self-contained speaker to play notes.

For business and data files type applications the reader is referred to Finkel and Brown (see Bibliography) for a detailed account of how to use sequential and random access disk files. The subject is quite complicated, and beyond the scope of this Handy Guide. Besides, it is difficult to improve upon their work in this area.

If you want lots of challenging exercises, consult the book by George Ledin. It has all kinds of them, at many different degrees of difficulty.

There is a version of LOGO that runs on the Apple. This language is very easy to use and very powerful graphically. It would be a good language to study in addition to BASIC.

Programming, like any skill, needs practice time. If you put in that time, you will always get better, and you will always know what to do next.

# BIBLIOGRAPHY


Roy Angeloff and Richard Mojena, *Applied BASIC Programming,* Wadsworth Publishing Company, 1980.

Apple Computer Inc., *The Applesoft Tutorial.*

Apple Computer Inc., *Applesoft II BASIC Programming Reference Manual.*

Apple Computer Inc., *The DOS Manual.*

LeRoy Finkel and Jerald R. Brown, *Apple BASIC: Data File Programming,* John Wiley & Sons, 1982.

George Ledin Jr., *A Structured Approach to General BASIC,* Boyd & Fraser Publishing Company, 1980.

Richard G. Peddicord, *Understanding BASIC,* Alfred Publishing Company, 1980.

Lon Poole and Mary Borchers, *Some Common BASIC Programs,* Adam Osborne & Associates, 1977.

# INDEX



*(Continued)*

## GET ON THE ALFRED COMPUTER MAILING LIST!
## KEEP UP-TO-DATE!

Send us your complete **name** and **address,** and we'll send you catalogs, newsletters, and new product listings, as they become available.

Or fill out and mail this coupon:

Name _____

Address _____

City _____ State _____ Zip _____

Handy Guide Titles You Own _____

Comments: _____

Send to: ALFRED PUBLISHING CO., INC.
Post Office Box 5964
Sherman Oaks, California 91413

5351

# Alfred Handy Guides

**Practical, economical, and concise**
Alfred Handy Guides tell you what you need to know quickly and easily—without a lot of reading!

*"A busy executive or professional who feels the need for a crash course in the subjects covered by the* **Handy Guides** *would do well to pick up a few on the way to the commuter train or the airport"* Personal Computing Magazine

## The Alfred Handy Guide Series to Computers

How to Buy a Personal Computer
How to Buy a Portable Computer
How to Buy a Word Processor
How to Choose a Computer Camp
How to Make Money with Your Personal Computer
How to Use Atari Computers
How to Use the Apple IIe
How to Use the Coleco Adam
How to Use the Commodore 64
How to Use the IBM PC
How to Use the IBM PCjr
How to Use the TRS-80 Model 100 Portable Computer
How to Use VisiCalc/SuperCalc
The Personal Computer Glossary
Quick and Easy dBase II
Quick and Easy Wordstar
Understanding APL
Understanding Apple Basic
Understanding Apple Graphics
Understanding Artificial Intelligence
Understanding Atari Graphics
Understanding BASIC
Understanding COBOL
Understanding Commodore 64 Basic
Understanding Commodore 64 Graphics
Understanding Computer Crime
Understanding Computer Graphics
Understanding Computer Information Networks
Understanding CP/M
Understanding Data Base Management
Understanding Data Communications
Understanding Electronic Mail
Understanding FORTH
Understanding FORTRAN
Understanding LISP
Understanding LOGO
Understanding Microcomputer Hardware
Understanding Pascal
Understanding Robots
Understanding Software Law

Look for new titles and new series.
For more information:

**Alfred Publishing Co., Inc.**
15335 Morrison St.
P.O. Box 5964
Sherman Oaks, CA 91413

ISBN 0-88284-246-3

0 38081 00295 00246